conference
proceedings

2013 USENIX Annual Technical Conference (USENIX ATC '13)

San Jose, CA, USA
June 26–28, 2013

Sponsored by
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Proceedings of the 2013 USENIX Annual Technical Conference
San Jose, CA, USA June 26–28, 2013

usenix ASSOCIATION

Thanks to Our USENIX ATC '13 Sponsors

Gold Sponsors

Silver Sponsors

Bronze Sponsors

Media Sponsors and Industry Partners

Computer
Computing in Science and Engineering
The Data Center Journal
Distributed Management Task Force (DMTF)
Free Software Magazine
HPCwire
IEEE Pervasive Computing
IEEE Security & Privacy
IEEE Software
InfoSec News
IT Professional
LXer
No Starch Press
O'Reilly Media
Server Fault
UserFriendly.org
Virus Bulletin

ISBN 978-1-931971-01-0

Thanks to Our USENIX and LISA Supporters

USENIX Patrons

Google InfoSys Microsoft Research NetApp VMware

USENIX Benefactors

Akamai EMC Hewlett-Packard *Linux Journal*
Linux Pro Magazine Oracle Puppet Labs

USENIX Partners

Nutanix

USENIX and LISA Partners

Cambridge Computer Google Meraki

USENIX Association

Proceedings of USENIX ATC '13:
2013 USENIX Annual Technical Conference

June 26–28, 2013
San Jose, CA

Conference Organizers

Program Co-Chairs

Andrew Birrell, *Microsoft Research Silicon Valley*
Emin Gün Sirer, *Cornell University*

Program Committee

Mustaque Ahamad, *Georgia Institute of Technology*
Lorenzo Alvisi, *The University of Texas at Austin*
Ozalp Babaoglu, *Università di Bologna*
Mike Burrows, *Google*
Manuel Costa, *Microsoft Research Cambridge*
Jason Flinn, *University of Michigan*
Phillipa Gill, *The Citizen Lab/Stony Brook University*
Robert Grimm, *New York University*
Hermann Härtig, *Technische Universität Dresden*
Jon Howell, *Microsoft Research Redmond*
Anthony Joseph, *University of California, Berkeley*
Terence Kelly, *HP Labs*
Steve Ko, *University of Buffalo*
Dejan Kostic, *Institute IMDEA Networks*
Paul Leach, *University of Washington*
Boon Loo, *University of Pennsylvania*
Vivek Pai, *Princeton University*
Dave Presotto, *Google*
Rama Ramasubramanian, *Microsoft Research Silicon Valley*
Karsten Schwan, *Georgia Institute of Technology*
Kai Shen, *University of Rochester*
Prashant Shenoy, *University Massachusetts Amherst*
Liuba Shrira, *Brandeis University*
Christopher Small, *Quanta Research*
Kobus van der Merwe, *University of Utah*
Jonathan Walpole, *Portland State University*
Meg Walraed-Sullivan, *Microsoft Research Redmond*
Alec Wolman, *Microsoft Research Redmond*
Bernard Wong, *University of Waterloo*
Yuanyuan Zhou, *University of California, San Diego*

External Reviewers

Deniz Altınbüken
Hans Boehm
Hyoun Kyu Cho
Björn Döbel
Ayush Dubey
Benjamin Engel
Robert Escriva
Goetz Graefe
Marcus Hähnel
Andy Hwang
Maciej Kuzniar
Adam Lackorzynski
Sandya Srivilliputtur Mannarswamy
Sang Lyul Min
Brad Morrey
Dejan Novakovic
Pradeep Padala
Stan Park
Michael Roitzsch
Jose Renato Santos
Julian Stecklina
Ioan Stefanovici
Tobias Stumpf
Yoshio Turner
Nedeljko Vasic
Marcus Völp
Carsten Weinhold

USENIX ATC '13:
2013 USENIX Annual Technical Conference
June 26–28, 2013
San Jose, CA

Message from the Program Co-Chairs vi

Wednesday, June 26, 2013

Virtual Machine Implementation

Optimizing VM Checkpointing for Restore Performance in VMware ESXi 1
Irene Zhang, *University of Washington and VMware;* Tyler Denniston, *MIT CSAIL and VMware;* Yury Baskakov, *VMware;* Alex Garthwaite, *CloudPhysics and VMware*

Hyper-Switch: A Scalable Software Virtual Switching Architecture 13
Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner, *Rice University*

MiG: Efficient Migration of Desktop VMs Using Semantic Compression 25
Anshul Rai and Ram Ramjee, *Microsoft Research India;* Ashok Anand, *Bell Labs India;* Venkata N. Padmanabhan, *Microsoft Research India;* George Varghese, *Microsoft Research US*

Computing in the Cloud

Copysets: Reducing the Frequency of Data Loss in Cloud Storage 37
Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum, *Stanford University*

TAO: Facebook's Distributed Data Store for the Social Graph 49
Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani, *Facebook, Inc.*

PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters 61
Rohan Gandhi, Di Xie, and Y. Charlie Hu, *Purdue University*

Flash-based Storage

FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs 67
Kai Shen and Stan Park, *University of Rochester*

The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs 79
Laura M. Grupp, *University of California, San Diego;* John D. Davis, *Microsoft Research;* Steven Swanson, *University of California, San Diego*

Janus: Optimal Flash Provisioning for Cloud Storage Workloads 91
Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock, *Google, Inc.*

(Wednesday, June 26, continues on p. iv)

Miscellanea #1

Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store 103
Christopher Mitchell, *New York University;* Yifeng Geng, *Tsinghua University;* Jinyang Li, *New York University*

Lightweight Memory Tracing 115
Mathias Payer, Enrico Kravina, and Thomas R. Gross, *ETH Zurich*

Flash Caching on the Storage Client. 127
David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer, *Harvard University*

Practical and Effective Sandboxing for Non-root Users 139
Taesoo Kim and Nickolai Zeldovich, *MIT CSAIL*

Thursday, June 27, 2013

Data Storage

TABLEFS: Enhancing Metadata Efficiency in the Local File System 145
Kai Ren and Garth Gibson, *Carnegie Mellon University*

Characterization of Incremental Data Changes for Efficient Data Protection 157
Hyong Shim, Philip Shilane, and Windsor Hsu, *EMC Corporation*

On the Efficiency of Durable State Machine Replication 169
Alysson Bessani, Marcel Santos, João Felix, and Nuno Neves, *FCUL/LaSIGE, University of Lisbon;* Miguel Correia, *INESC-ID, IST, University of Lisbon*

Estimating Duplication by Content-based Sampling 181
Fei Xie, Michael Condict, and Sandip Shete, *NetApp Inc*.

Miscellanea #2

MutantX-S: Scalable Malware Clustering Based on Static Features 187
Xin Hu, *IBM T.J. Watson Research Center;* Sandeep Bhatkar and Kent Griffin, *Symantec Research Labs;* Kang G. Shin, *University of Michigan*

Redundant State Detection for Dynamic Symbolic Execution. 199
Suhabe Bugrara and Dawson Engler, *Stanford University*

packetdrill: Scriptable Network Stack Testing, from Sockets to Packets 213
Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert, *Google*

Virtual Machine Performance

DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments 219
Dejan Novaković, Nedeljko Vasić, and Stanko Novaković, *École Polytechnique Fédérale de Lausanne (EPFL);* Dejan Kostić, *Institute IMDEA Networks;* Ricardo Bianchini, *Rutgers University*

Efficient and Scalable Paravirtual I/O System 231
Nadav Har'El, Abel Gordon, and Alex Landau, *IBM Research–Haifa;* Muli Ben-Yehuda, *Technion IIT and Hypervisor Consulting;* Avishay Traeger and Razya Ladelsky, *IBM Research–Haifa*

vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. 243
Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu, *Purdue University*

Managing Resources

When Slower Is Faster: On Heterogeneous Multicores for Reliable Systems 255
Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum, *VU University Amsterdam*

IAMEM: Interaction-Aware Memory Energy Management 267
Mingsong Bi, *Intel Corporation;* Srinivasan Chandrasekharan, and Chris Gniady, *University of Arizona*

XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints 279
Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa, *Karlsruhe Institute of Technology*

Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack 291
Konstantinos Menychtas, Kai Shen, and Michael L. Scott, *University of Rochester*

Friday, June 28, 2013

Small Applications

Mantis: Automatic Performance Prediction for Smartphone Applications 297
Yongin Kwon, *Seoul National University;* Sangmin Lee, *University of Texas at Austin;* Hayoon Yi, Donghyun Kwon, and Seungjun Yang, *Seoul National University;* Byung-Gon Chun, *Microsoft;* Ling Huang and Petros Maniatis, *Intel;* Mayur Naik, *Georgia Institute of Technology;* Yunheung Paek, *Seoul National University*

IO Stack Optimization for Smartphones 309
Sooman Jeong, *Hanyang University;* Kisung Lee, *Samsung Electronics;* Seongjin Lee, *Hanyang University;* Seoungbum Son, *Samsung Electronics;* Youjip Won, *Hanyang University*

How to Run POSIX Apps in a Minimal Picoprocess 321
Jon Howell, Bryan Parno, and John R. Douceur, *Microsoft Research*

Packets

Network Interface Design for Low Latency Request-Response Protocols 333
Mario Flajslik and Mendel Rosenblum, *Stanford University*

DEFINED: Deterministic Execution for Interactive Control-Plane Debugging 347
Chia-Chi Lin, Virajith Jalaparti, and Matthew Caesar, *University of Illinois at Urbana-Champaign;* Jacobus Van der Merwe, *University of Utah*

Improving Server Application Performance via Pure TCP ACK Receive Optimization 359
Michael Chan and David R. Cheriton, *Stanford University*

Message from the 2013 USENIX Annual Technical Conference Program Co-Chairs

Welcome to the 2013 USENIX Annual Technical Conference.

Once again, we received a record number of submissions to the conference. Authors registered 321 abstracts, of which 233 were submitted as complete papers. Of the submitted papers, 38 were submitted as short papers (no longer than six pages), and the rest were traditional full-length papers (up to 12 pages, including references). The program co-chairs rejected 10 papers without review for serious violations of the formatting rules (incorrect formatting that increased the effective available space by 5% or more).

Reviewing was single-blind, done almost entirely by the program committee, with some assistance from outsiders with special expertise. The reviewing was done in three rounds. In the first round, every paper received two reviews. Based on these reviews, 69 of the papers were tentatively rejected, because both reviews had overall merit scores of one or two (on a scale of one through five) with adequate confidence levels. In round two, the remaining 154 papers each received one more review. Finally, the 47 papers from round two that had at least one overall merit score less than three, and at least one higher than three, were each given two additional reviews in the third round. Altogether, we produced 700 reviews.

The program committee meeting was held in April in scenic Lombard, Illinois. Most of the committee was present in person. In a 10-hour session, we discussed 76 of the papers, including a few low-ranked ones that individual PC members thought merited more consideration. We accepted a total of 33 papers. Of these, six were accepted as short papers. Three of the short papers had originally been submitted as full-length papers. Each accepted paper was shepherded by a PC member in preparing revisions for the final published versions that you see here.

Our committee had 30 members, plus the two co-chairs. Nine of the committee and one co-chair were from industrial institutions. The committee members were allowed to submit papers; the chairs chose not to submit anything. We followed conventional rules for conflict of interest, with conflicted members (or co-chair) leaving the room during discussion of the conflicted papers.

The papers you see in this year's program represent a broad diversity of current systems work. In keeping with the goals and tradition of USENIX ATC, there is strong representation of papers with a very practical orientation, in addition to papers with pure novel research contributions.

Besides the paper authors and reviewers, we would like to thank the USENIX staff who actually do the organization here; without their support, our jobs would have been much harder. They made it possible for us to focus on creating the conference program, without worrying about the endless details of conference organization and proceedings publication.

Thank you for participating in the USENIX ATC community, and enjoy the conference.

Andrew Birrell, ***Microsoft Research Silicon Valley***
Emin Gün Sirer, ***Cornell University***
ATC '13 Program Co-Chairs

Optimizing VM Checkpointing for Restore Performance in VMware ESXi

Irene Zhang* University of Washington
Tyler Denniston* MIT CSAIL
Yury Baskakov VMware
Alex Garthwaite* CloudPhysics

Abstract

Cloud providers are increasingly looking to use virtual machine checkpointing for new applications beyond fault tolerance. Existing checkpointing systems designed for fault tolerance only optimize for saving checkpointed state, so they cannot support these new applications, which require better restore performance. Improving restore performance requires a predictive technique to reduce the number of disk accesses to bring in the VM's memory on restore. However, complex VM workloads can diverge at any time due to external inputs, background processes, and timing variation, so predicting which pages the VM will access on restore to reduce faults to disk is impossible. Instead, we focus on predicting which pages the VM will access *together* on restore to improve the efficiency of disk accesses.

To reduce the number of faults to disk on restore, we group memory pages likely to be accessed together into *locality blocks*. On each fault, we can load a block of pages that are likely to be accessed with the faulting page, eliminating future faults and increasing disk efficiency. We implement support for locality blocks, along with several other optimizations, in a new checkpointing system for VMware ESXi Server called Halite. Our experiments show that Halite reduces restore overhead by up to 94% for a range of workloads.

1 Overview

The ability to checkpoint and restore the state of a running virtual machine has been crucial for fault tolerance of virtualized workloads. Recently, cloud providers have been exploring new applications for VM checkpointing. For example, they want to use checkpointing to save and power off idle VMs to conserve energy. Restoring a checkpointed "template" VM could be used to clone new VMs on demand, which would enable fast, dynamic allocation of VMs for stateless workloads.

Unlike traditional fault tolerance applications, these new applications depend on efficient *restore* of checkpointed VMs. For example, using checkpointing for dynamic allocation of VMs depends on the ability to quickly start up a VM on demand. Checkpointing systems designed to support fault tolerance only restore on failures, so they optimize for *checkpoint save* performance instead. As a result, previous work rarely addresses restore beyond basic support, so existing systems would offer poor performance for these new applications.

Virtual machine checkpointing takes a snapshot of the state of a VM at a single point in time. The hypervisor writes any temporary VM state, like VM memory, to persistent storage and then reads it back into memory when restoring the checkpoint. Since memory images can be large, VMware ESXi uses a technique called *lazy restore* that loads the memory image from disk while the VM runs. While the VM's memory is partially on disk, any access to on-disk pages causes a fault that requires a disk synchronous access before the VM's execution can resume. Pauses in execution for faults to disk can quickly degrade the usability of the VM.

Improving lazy restore performance requires a predictive technique that reduces the number of faults to pages on disk. However, it is impossible to predict which pages the VM will access on restore; the VM's execution might diverge at any time due to timing differences or external inputs, particularly with complex workloads that have many background tasks and user applications. Previous work [24] based on predicting which pages the VM would access on restore could not cope with divergence, leading to poor performance for complex workloads like Windows desktop applications.

Rather than reducing the number of faults to disk by predicting the pages that the VM will access on restore, we instead predict the pages that the VM will access *together* on restore. On each fault to disk during lazy restore, we prefetch a few pages that are likely to be accessed with the faulting page, rather than prefetching before the VM's execution begins. This technique is more resilient to divergence since the prefetching decision is based directly on pages that have been accessed by the VM after the restore. There is a smaller penalty for incorrect predictions because only a few pages are prefetched at a time.

To allow for efficient prefetching on restore, we sort pages likely to be accessed together into *locality blocks* in the VM's checkpointed memory image. On restore,

*Work done while all authors were at VMware.

we load an entire locality block on each fault to disk. Since the other pages in the locality block are likely to be accessed with the faulting page, we eliminate faults to disk for those pages. We implement this technique in a new VM checkpointing system for VMware ESXi Server called *Halite*.

Halite uses two techniques to predict the access locality of memory pages. The first uses the VM's memory accesses during lazy save. The VM continues running past the checkpoint while its memory is written to disk, so the VM's execution during lazy restore is actually a re-execution of the VM's execution during lazy save. While the exact execution of the VM will vary on restore due to divergence, there is less change to the access locality. Pages accessed together during lazy save are likely to be accessed together again on restore.

Since the VM may not access all of its pages during checkpointing, we must use a second technique for predicting access locality. For the unaccessed pages, Halite uses locality in the guest operating system's *virtual address space* to predict access locality. Pages that are mapped together in the virtual address space are likely to be accessed together, so locality in the virtual address space is another good predictor of access locality.

We designed and implemented Halite as an improved VM memory checkpointing system for VMware ESXi 5.1 [22] and included other optimizations to the current system. Halite performs fine-grained compression of the checkpointed memory file, so that compression can be done in parallel on checkpoint save and only a small amount of decompression is required for each fault to disk on checkpoint restore. Compression increases the effectiveness of locality blocks because more pages can fit into each block. Unlike ESXi, Halite makes extensive use of threads to parallelize work during checkpoint save and restore, including threads for compression and I/O. Halite dynamically throttles background work during lazy save and restore to avoid disk contention.

The next section reviews some new applications for VM checkpointing that VMware has explored. Section 3 gives background on the current virtual machine memory checkpointing system in VMware ESX 5.1. Section 4 describes Halite's new memory file layout with locality blocks. Section 5 describes the algorithms that we use for predicting access locality. Section 6 details the other optimizations in Halite. Section 7 gives implementation details including the algorithm for saving and restoring VM memory in Halite. Section 8 presents our experimental results. Section 9 gives an overview of related work, including our previous work, and Section 10 concludes.

2 Checkpointing Workloads

The primary motivation for Halite is to improve the checkpoint restore performance of ESXi, enabling a variety of new and emerging use cases. In contrast to fault tolerance scenarios, where restore is uncommon and happens only on failure, these new use cases depend on efficient checkpoint restore.

2.1 Dynamic VM Provisioning

One of the advantages to cloud computing is the ability to allocate the appropriate amount of computing resources for any workload. This allocation does not have to be static; as a workload requires more or less resources, the number of allocated VMs can be increased or decreased. However, most cloud infrastructures are not able to quickly bring more VMs online. On Amazon EC2, it can take up to 10 minutes to bring up a VM [1]. Due to this delay, users must keep a buffer of unused VMs to handle spikes in requests. Running a number of idle VMs is both a waste of resources and still may not be sufficient to protect against severe spikes in usage.

Halite enables fast checkpoint restore from a template VM image, similar to VM fork supported by Snowflock [10], Kaleidoscope [2] or FlurryDB [14]. This feature allows users to better scale their resource allocation with usage. Using a checkpointed VM image with a running Apache server, a VM could be online and handling user requests in a few seconds. Using a checkpointed VM also offers advantages over quickly booting a VM; the applications in the VM benefit from a warm cache and several applications can be running in the VM without the overhead of application start up times, which can sometimes be long. Alternatively, for some workloads, Halite gives users the ability to allocate a single stateless VM for each incoming connection. Customers have requested this feature because it is an easy solution to ensure security between users.

2.2 Energy Conservation

Virtualization reduces energy usage with server consolidation, but conserving energy consumed by idle VMs is still a serious concern in cloud deployments. Some systems have explored turning off servers [3] or suspending idle VMs [5] to conserve power, but all of these systems struggle with restarting servers or VMs. They use predictive techniques to restart VMs in advance. When these techniques incorrectly predict usage patterns, either energy is wasted powering on VMs that are not needed, or users are forced to wait for the needed VM to restart.

Halite makes it much easier to turn off idle VMs without suffering from poor performance when the VM is needed again. Using Halite, the user can checkpoint VMs and power them off. Complex predictive models are not required with Halite because suspended VMs can be quickly restarted on demand.

2.3 Virtual Desktop Infrastructure

Large companies have started to move toward converting desktop PCs into VMs running in a datacenter. These virtual desktops are easier to maintain and reduce the amount of hardware needed. However, since there are more users sharing hardware, users can see performance degradation when there are spikes in usage. In particular, VMware has observed a "boot storm" problem, where all users arrive at work in the morning and attempt to boot their VMs close in time, leading to severe disk contention. VM checkpointing can be used to mitigate this problem. If users checkpoint their desktop VM before going home (or an energy conservation system checkpoints it for them), then they can simply restore their VM in the morning. Restoring a checkpointed VM requires reading much less from disk than a full boot, easing contention on the disk. In addition, Halite efficiently restores the VM in much less time than booting a VM, further reducing the disk usage and wait time for the user.

3 ESXi Checkpointing

In order to give some background and motivation to our work, we describe the state of the art in virtual machine checkpointing implemented in the current release of VMware ESXi. We primarily discuss the mechanism for checkpointing VM memory and not other VM state.

3.1 ESXi Save and Restore Algorithm

ESXi has used lazy checkpointing since VMware ESXi 4.0. Lazy checkpointing allows the VM to run while its memory is saved or restored, reducing the amount of downtime. ESXi implements a generic copy-on-write scheme similar to the one described here [20] and similar to Xen's implementation [4]. VMware's implementation depends on ESXi's memory tracing mechanism to track write accesses, which is also used for Halite.

On checkpoint save, ESXi pauses the VM's execution and saves its CPU and device state. ESXi installs memory traces on all of the VM's pages and resumes the VM. When the VM writes to an unsaved page, it triggers the trace on that page. ESXi saves the page and removes the memory trace before allowing the write to proceed.

While the VM runs, the hypervisor concurrently writes out memory pages using a background thread. This thread ensures that the checkpointing process finishes in a reasonable amount of time. The background thread walks the VM's physical address space, saving any pages that have not been already written. It removes the trace on any page that it saves to avoid triggering the trace later. The checkpoint save is complete when the background thread has walked the entire address space.

ESXi supports lazy checkpoint restore using the swap subsystem by treating a restoring VM like a VM with all of its memory swapped. This implementation was chosen for its simplicity and ease of deployment. To restore a VM, ESXi sets up the checkpointed memory file as the swap file and then restarts the VM at the checkpoint. On each access by the VM, a single memory page is swapped in from the memory file. Concurrently, a background thread touches swapped out pages to ensure that the restore finishes in a reasonable period of time.

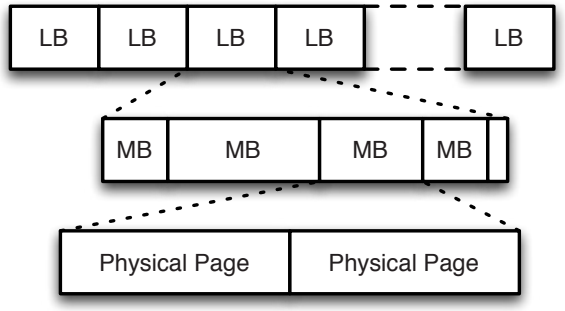


Figure 1: Block layout of checkpointed memory file. Memory blocks (MB) consist of several pages in physical address order. Locality blocks (LB) contain a variable number of compressed memory blocks. Memory blocks are grouped into locality blocks based on access locality.

3.2 Memory File Organization

ESXi saves the VM's memory in physical address order from physical address 0 to the VM's memory size. This file layout is simple and requires no metadata, but is not optimal for either checkpoint save or restore. On every write to an unsaved page, ESXi must save the page before allowing the VM to continue executing. Since these writes are to random memory pages, the disk accesses are random as well. ESXi avoids having to write to disk on each write access by buffering, but buffered pages still cannot be written out sequentially because of the file layout. These writes can degrade the VM's performance if the rate of writes to memory is high.

This organization is even worse for checkpoint restore. On every access to an unrestored memory page, the VM must pause while waiting for the page to be read from disk. Since physical memory is inherently random access, every access to an unrestored memory page requires the disk to seek and read a single page, leading to poor disk performance. Because ESXi treats the memory as swapped, the hypervisor only reads one 4K page from disk on each access, further degrading disk performance.

4 Halite Memory File Organization

This section describes Halite's memory file layout. Halite uses a significantly different memory file organization from ESXi, with locality blocks and fine-grained compression. Figure 1 shows the layout of the Halite memory file. Locality blocks are crucial for efficient prefetching on each fault to disk during restore.

4.1 Memory Blocks

For simplicity of implementation and to reduce the size of meta-data, Halite divides the VM's physical memory into fixed-size, aligned blocks of a few pages each called *memory blocks*. Halite uses memory blocks as the smallest unit of processing (i.e., compression, buffering, etc.), so memory blocks must be small enough that there is still some access locality in the physical address space.

We found using a memory block size of two pages was a good trade-off in our implementation; it halves the amount of meta-data, but remains small enough for workloads with poor access locality in the physical address space like Windows applications. Larger memory blocks sizes can be used for Linux because of its use of a buddy allocator[1], but performed poorly for Windows.

4.2 Locality Blocks

Halite groups a number of memory blocks into each locality block, based on access locality. Halite loads an entire locality block on each fault to disk, reducing disk accesses and increasing disk efficiency. For simplicity, locality blocks are fixed-size. Due to the fixed size of locality blocks, they contain a variable number of compressed memory blocks and some empty space. Locality blocks also reduce the size of meta-data needed for compression because only the byte offset within the locality block is needed. Larger locality blocks increase the efficiency of the disk, but also increase the latency of each fault to disk on restore. In our implementation, we used a locality block size of 64KB, which we found to be a good trade-off between efficiency and latency.

5 Access Locality Prediction

Halite uses two techniques to predict access locality for grouping memory pages into locality blocks. The first technique traces the execution of the VM past the checkpoint during the lazy save. The first technique only works for pages that are accessed during the lazy save, so we combine it with a second technique that uses guest virtual address locality to predict access locality. Both of these techniques are standard in CPU prefetching [19], although not at the 4KB page level.

5.1 Lazy Save Memory Accesses

In our previous work [24], we observed that the checkpoint restore period is a re-execution of the checkpoint save period since the VM restores back to the point in time at the beginning of the lazy save period. Unlike working set restore, Halite uses the VM's memory accesses during lazy save to predict access locality, rather than access ordering. If the VM accessed page X, followed by page Y during lazy save, then it is highly likely that if the VM accesses X or Y on restore, it will also access the other, even with divergence due to timing or different external inputs.

Halite groups the pages that were accessed together during lazy save into locality blocks. The first N pages accessed by the VM are stored in one locality block, the next N in another, where N is the size of a locality block. The number of pages in a locality block can vary due to compression. Halite does this sorting during the save process by writing pages out to locality blocks as they are accessed. Simply filling locality blocks in access order allows Halite to fill locality blocks without post-processing, to easily fill a locality block at a time, and to write locality blocks out sequentially to disk as they are filled.

For VMs with more than one virtual CPU (vCPU), Halite separates pages into locality blocks based on the vCPU. Since each vCPU is running a separate thread of execution, we believe that an access to page X on one vCPU, followed by an access to page Y on another vCPU is not a good predictor of access locality since differences in timing can easily cause divergence. Sorting based on vCPU simply requires Halite to fill one locality block per vCPU at a time.

5.2 Guest Virtual Address Space

Divergence from the VM's execution during lazy save on restore is unavoidable; there will be pages that weren't accessed during lazy save that are accessed on restore. For pages that are not accessed during lazy save, we use guest virtual address space locality to predict access locality. This technique assumes that if page X and Y are adjacent in the virtual address space, then an access to page X or Y is a good predictor that the VM will also access the other page. Previous work [16] has shown that the virtual address space is a better predictor of access locality than the physical address space.

Halite sorts pages not accessed by the VM during the checkpoint save into locality blocks based on virtual address. The first N mapped pages in a guest virtual address are stored in one locality block, the next N in another. Again, N may vary due to compression. Halite collects page table roots as the VM runs. The background thread in Halite walks the guest virtual address space using the guest page tables. As the background thread scans, it fills locality blocks in the order it encounters pages and writes them out sequentially to disk. We only save a single copy of each memory page. Memory pages that are mapped in more than one guest address are saved the first time we encounter them in a page table.

[1] Linux's buddy allocator increases access locality in the physical address space by mapping contiguous physical addresses to virtual addresses whenever possible.

6 Halite Checkpointing Optimizations

This section introduces other improvements made in Halite to ESXi's checkpointing system. These optimizations include dynamic background thread throttling, compression, zero page optimizations, and threading. Some of them take advantage of Halite's more sophisticated memory file organization, while some of them are just general improvements to the ESXi checkpointing infrastructure.

Dynamic Background Thread Throttling The background thread in ESXi is designed to ensure that the checkpointing process finishes, even if the VM does not access all of its memory pages. When the VM is rapidly touching pages, disk access to the checkpointing file becomes a bottleneck and the background thread begins to contend with the VM. However, we observed that if the VM is accessing pages rapidly, there is no reason for the background thread to run since the checkpointing process is clearly still making progress. Therefore, we only run the background thread in Halite if the checkpointing process is not progressing, which keeps the background thread from contending with the VM. We do this throttling for both checkpoint save and restore.

Compression Compression reduces the size of the checkpointing image, which reduces not only the size on disk of the image, but also the amount of data that needs to be moved to and from the disk for the checkpoint. Reducing the disk space required can be important, especially if the VM has a large memory size, but reducing the amount of I/O is even more important because the disk is a bottleneck during checkpointing. Compression also allows more memory to be prefetched on each page fault with the same amount of I/O. In Halite, each memory block in a locality block is separately compressed. We chose to compress memory blocks instead of whole locality blocks to allow more parallelization and to reduce the amount of decompression required on each page fault. We found that using smaller blocks for compression has minimal impact on the compression ratio.

Zero Pages ESXi scans guest memory for pages that are completely zero and does copy-on-write sharing of those pages. For these pages, there is no reason to read or write the page for checkpointing, so Halite tracks these pages and does not include them in the memory image. Halite also does this for pages that the VM has never touched, and therefore, are not backed in the hypervisor.

Threading Halite introduces several threads to allow more parallel processing of memory pages. On checkpoint save, these threads reduce the amount of time that the VM has to be paused on each page fault. Halite only needs to pause the VM long enough to copy the memory page to a buffer; threads perform the compression and writing the memory out to the checkpointing file. On restore, the faulting page must be read in from disk and decompressed synchronously, so threads cannot improve the performance. However, Halite decompresses and restores the other memory blocks in the locality block using threads in parallel. This minimizes the work for each prefetched page on a page fault and eliminates the work required if the VM later accesses one of the prefetched pages.

7 Implementation

We implemented Halite using VMware ESXi 5.1. Halite replaces ESXi's existing checkpointing mechanisms because they are not designed to asynchronously process memory and restore memory that is not organized in physical address order. In addition, ESXi does not save memory to a separate file by default; memory is normally stored in one file with other checkpointed state. Halite required a separate file because there is no way to anticipate the size of the region required for checkpointed memory due to compression. ESXi does not support compression of the memory image. Halite only replaces the VM memory checkpointing system, so ESXi still handles saving any other VM state.

7.1 Halite Save and Restore Algorithm

Like ESXi, Halite uses lazy checkpointing, but Halite does copy-on-access, rather than copy-on-write checkpointing to capture access locality. However, Halite buffers pages on checkpoint save and writes to disk sequentially, rather than randomly, so the overhead is small.

On both checkpoint save and restore, Halite tries to do as much work asynchronously as it can. During the lazy save period, when a trace triggers, Halite simply copies the memory block to a buffer and removes all of the traces on the pages in that block. Later the memory block is compressed by a thread and copied to a locality block. When the locality block fills up, another thread writes it out to disk. The thread also updates the mapping to record which locality block contains each memory block.

When the VM faults on an unrestored page, Halite consults the map to find which locality block it needs to fetch. The locality block might already be in memory if it was prefetched by a previous fault. If not, Halite reads the locality block, and decompresses and restores just the memory block of the faulting page. The other pages in the locality block will be decompressed by threads later.

The background thread in Halite works similarly to ESXi, except it is throttled as described in Section 6.

8 Evaluation

Our evaluation answers several questions about the performance of Halite:

- How does Halite compare to ESXi 5.1 for some representative workloads?

- How do locality blocks compare to other block organizations?
- How does compression impact the performance of Halite for workloads with differing amounts of compressibility?
- How much do locality blocks contribute to the performance benefits offered by Halite?
- How does restoring a checkpointed VM using Halite compare to a cold boot of the VM?

We evaluated Halite using a synthetic workload and three application benchmarks. pgbench [18] is a standard database benchmark for PostgreSQL. Worldbench is a Windows desktop application workload. We also designed a simulated Apache [6] web server benchmark. Our workloads represent a range of complexity from the simple synthetic workload to the complex Worldbench benchmark. We ran our experiments on a server with a 2.3GHz 8-core AMD Opteron processor and 24GB of RAM. All of the VMs and checkpoints are stored on a 15,000 RPM Seagate 1TB drive running VMFS-5.

8.1 Microbenchmark

First, to evaluate the performance benefit of Halite and its various optimizations in a controlled environment, we created a synthetic workload generator. The benefit of the workload generator is having control over every aspect of the workload. VM workloads tend to be very complex, making it difficult to isolate the source of performance differences.

The workload sequentially accesses memory using one thread per vCPU, with each thread accessing a separate region of memory. We ran our workload on Red Hat 6 Enterprise Server in a VM with 4 vCPUs and 2GB of RAM. The workload has a working set size of 256MB. It allocates 1GB of memory for the test and fills that memory with data that is 50% compressible. In order to increase physical memory fragmentation, the workload allocates memory in a 16 page stride. Workload performance is measured by the time needed to access 100,000 pages. We checkpointed the VM in the middle of the workload test run, then restored the checkpoint and recorded the time to complete the test. The VM restores back to the start of the checkpointing, so the result does not include any overhead from creating the checkpoint. We separately discuss the cost of checkpoint save in Section 8.3.

8.1.1 Checkpoint Restore Overhead

We tested the overhead imposed by checkpoint restore on the microbenchmark for several different test configurations. We tested the current implementation of VM checkpointing in ESXi against Halite with all of its optimizations. We also measured the impact of locality blocks compared to other block organization schemes. We used a version of Halite without any memory file optimizations, such as compression and the zero page optimization, but with different block organizations. We do not use compression in this test, so each file block holds 8 memory blocks or 16 pages. Thus, on each fault to disk, 15 other pages in the block are "prefetched" from disk. We tested three organizations: memory pages grouped into blocks randomly (random blocks), memory pages grouped by physical address (physical address blocks) and locality blocks.

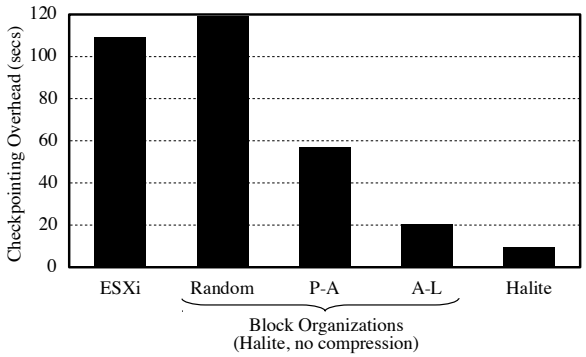


Figure 2: Synthetic workload performance for the current ESXi implementation, Halite with different memory file organizations and Halite with memory file optimizations. The middle bars give Halite performance for an uncompressed memory file using (from left to right) random blocks, physical address blocks (P-A) and access locality blocks (A-L). Performance is given as the increase in runtime caused by the checkpoint restore (lower is better).

Random blocks should be the lower bound worst-case performance; on each fault to disk, 15 random pages are loaded along with the faulting page. Physical address blocks use the same memory file organization as the current ESXi implementation, but group 16 pages into a block to read for each fault to disk instead of a single 4KB page. This organization simulates the performance of ESXi if we had added Halite's other optimizations, but kept the memory file organization. The test using locality blocks simulates Halite, but isolates the effects of locality blocks from Halite's other memory file organizations, including compression and the zero page optimization.

Figure 2 gives the performance overhead of checkpoint restore for each of our test configurations. Each test result is the average of 10 test runs. The baseline runtime of the workload generator is 54 seconds on average. The restore overhead is given as the increase in runtime of the synthetic workload due to the VM being restored in the middle of the run. We use the same snapshot for all test configurations by reformatting the same memory file, so the performance before the checkpoint is identical and the difference in runtime is only due to the restore process.

Comparing ESXi and Halite, Halite reduces the restore overhead by 100 seconds or more than 10x. The three

Table 1: Efficiency of each type of block organization (no compression) given by number of blocks faulted in from disk and percentage of other pages in the blocks later accessed.

	Disk Accesses	VM Access %
Random	22,975	19%
P-A	14,501	36%
A-L	6,908	83%

bars in the middle of the graph show the impact of varying just the block organization. It is important to isolate the performance impact of different block organizations to understand the benefit offered by locality blocks.

As expected, random blocks perform the worst. Random blocks perform worse than even ESXi, although reading blocks of pages from disk should increase disk efficiency. This result shows that reading blocks from disk only improves performance if the other pages in the block are useful for eliminating future faults to disk. Otherwise, using bigger blocks only increases the latency of each fault to disk without reducing the overall number of faults. Physical address blocks halve the overhead compared to the random organization because the other pages in a block are more likely to be accessed, eliminating some faults to disk. This improvement is due to access locality in the physical address space.

Locality blocks perform the best, improving performance by 6x over random blocks and almost 3x over physical address blocks. We see further improvement because locality blocks have better access locality than physical blocks. More of the other pages in the block are likely to be accessed after the faulting page, eliminating more faults to disk. These results show how crucial block organization is for restore performance.

8.1.2 Memory File Organization

We can see how different block organizations impact performance by looking at the checkpointing statistics collected by Halite. Table 1 gives the total number of faults to disk for each block organization and the percentage of prefetched pages the VM accessed after each fault. Prefetched pages are the pages other than the faulting page in a block of pages brought in from disk. Accesses to prefetched pages eliminate faults to disk, improving disk efficiency.

It is clear that locality blocks lead to more efficient disk access than the other block organizations. There are fewer faults to disk and more pages in each faulted block are eventually accessed, eliminating more faults and increasing disk efficiency.

We collected hypervisor statistics on the percentage of

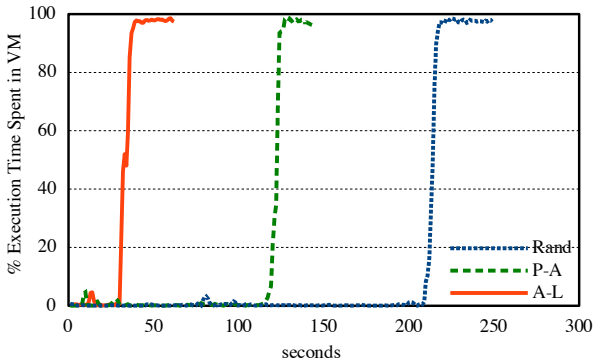


Figure 3: VM performance during lazy restore given as % of time spent executing in the VM (higher is better). Results are given for three block organizations: random, physical address blocks (P-A) and access locality blocks (A-L)

time spent running guest code to show the impact of faults to disk. Faults to disk pause the VM's execution, leading to less time spent running guest code and a reduction in performance for the guest. For the best performance, we want to return the guest to running almost 100% of the time as soon as possible.

Figure 3 shows the impact on guest execution during restore for the different block organizations. Locality blocks improve performance by reducing the period of time where the VM does not run much due to faults to disk. With locality blocks, the VM sees a large number of page faults for the first 30 seconds. That time increases to 120 seconds with a physical address blocks, and to 210 seconds with a random blocks. The total time to restore the VM also decreases from locality blocks to random blocks.

During the restore, the VM sees a large number of faults until the working set is entirely faulted in. The period where the VM sees performance degradation is not directly related to the test overhead given in Figure 2 because the VM is making some progress during that time. Locality blocks pack the VM's working set into fewer blocks and brings the working set in with fewer faults to disk.

For some workloads, we could prefetch these page before starting the VM as we previously proposed [24]. However, the working set cannot be determined before the VM restores for some workload, and in fact, changes while the VM runs. For those workloads, Halite provides better performance because locality blocks group pages that are likely to be accessed together into blocks, so Halite will still be able to efficiently fault in the working set, whereas working set restore would hurt performance by prefetching pages that are not needed before the VM starts. We saw this reduction in performance for working set restore with the Worldbench workload presented

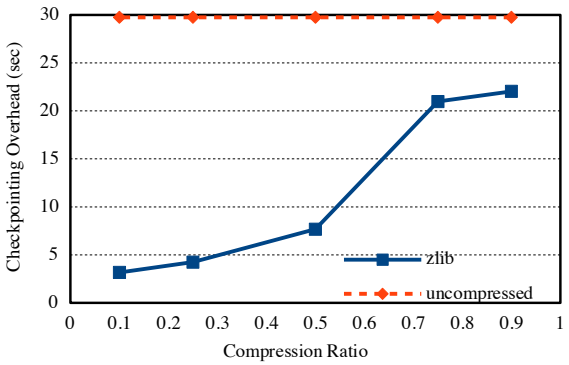


Figure 4: Restore overhead for Halite using compressed and uncompressed memory files for workloads with varying compression ratios. Smaller compression ratio means that the workload is more compressible. Performance is given as the increase in the synthetic workload runtime (lower is better).

in Section 8.2.1, whereas Halite provides a significant performance improvement.

8.1.3 Compression

The performance impact of including compression in Halite depends heavily on the compressibility of the workload. Thus, we compare Halite performance with a compressed and uncompressed memory file for workloads with different compression ratios. We varied the compression ratio of the memory allocated by the workload generator by packing some percentage of every page with already compressed data. These tests were performed using zlib [13]. Halite also supports LZRW [23]; however, we found that zlib performed similarly to or better than LZRW in most cases.

Figure 4 shows the performance results for workloads that compress to 10% of their original size to workloads that compress to 90% of their original size. The performance improvement due to compression varies from a 89% improvement to a 26% improvement depending on the compressibility of the VM's memory.

8.2 Application Benchmarks

In addition to our synthetic benchmark, we also evaluated the performance of Halite for a number of representative application benchmarks. These workloads vary in complexity, and therefore, the amount of divergence they exhibit. Worldbench is a simulated desktop application workload running in Windows XP. It is the most divergent; there are timing variances in the inputs from the workload generator, and Windows XP has many background processes that run at various times. In comparison, pgbench running on PostgreSQL in a server Linux install is more deterministic. The benchmark is deterministic and there are few background processes. Finally, our Apache server benchmark is designed to have divergence in the random selection of pages that we request from it, but it runs on top of Linux and has a small working set.

The workloads also vary in their compressibility, which affects the performance benefit of Halite, as shown in the previous section. The Worldbench checkpointed memory file only compresses to 67% of its original size due to a large number of media files in memory. The pgbench checkpointed memory file compresses to 10% of its original size due to pgbench filling the database with patterns.

8.2.1 Worldbench

Worldbench is a simulated desktop workload with typical desktop applications like word processing, web browsing and video editing. Worldbench closely simulates the expected workload of a VDI deployment described in Section 2.3. We ran Worldbench in Windows XP in a VM with 1GB of memory and 2 vCPUs. We used the multitasking test from the test suite that simulates a browser workload and media encoding. Worldbench reports the amount of time taken to run the test suite once. We checkpointed the VM 10 minutes into the test run, so the first 10 minutes are identical across runs. Each test is the average of 10 test runs.

We evaluated the performance of Worldbench on ESXi and three Halite configurations. The current ESXi implementation of checkpointing uses a memory file that is organized by physical address and reads one 4KB page on each fault to disk. The first Halite configuration is Halite (P-A), which gives the performance of Halite using physical blocks. Halite (P-A) uses Halite's checkpointing optimizations like threading and background thread throttling, but none of the memory file optimizations like locality blocks, compression and zero page optimization. This configuration simulates the performance of ESXi with the Halite optimizations that would be easy to add to ESXi, like increasing the block size faulted in from disk and throttling the background thread, but not changing the memory file layout.

The next Halite configuration is Halite (A-L), which gives the performance of Halite with locality blocks, which is the key contribution in Halite. This configuration isolates the performance impact of locality blocks, from other memory file optimizations like compression and zero page optimization. The last configuration is Halite with all optimizations including compression and zero page optimization. This configuration gives the total performance benefit of Halite over ESXi.

The baseline performance of Worldbench without checkpointing is 816 seconds. Figure 5 gives the Worldbench results as the average increase in the runtime due to checkpointing. Again, there is no performance impact from saving in these results because the checkpoint is

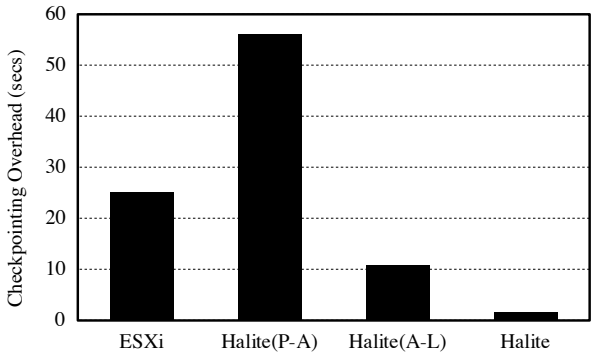


Figure 5: Checkpointing overhead for Worldbench given as number of seconds increase in runtime over baseline (lower is better).

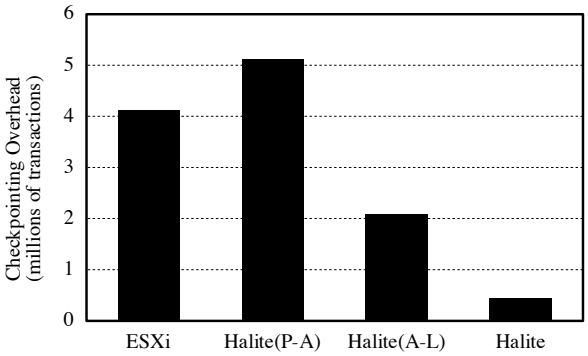


Figure 6: Checkpointing overhead for pgbench given as a reduction in transactions completed in 5 minutes over baseline (lower is better).

before the beginning of the save process, so the increase in runtime is only due to the lazy restore process. All configurations use the same memory file, re-organized into the appropriate block organization, so there is only one checkpoint across all of the configuration tests.

ESXi increases the runtime of Worldbench by more than 25 seconds due to faults to disk during the lazy restore period. Halite (P-A) adds a number of optimizations to ESXi, including physical blocks, which should increase disk efficiency. However, Worldbench has poor access locality in the physical address space, so physical blocks actually reduce performance like random blocks did in our microbenchmark test. Halite (P-A) actually increases the runtime overhead by almost 2x, up to 56 seconds on average.

Like Halite (P-A), Halite (A-L) also uses blocks, but locality blocks instead of physical blocks. Halite (A-L) reduces the runtime overhead by almost 6x compared to Halite (P-A), showing the importance of block organization based on access locality. Halite, which includes compression, further improves performance, reducing the checkpointing overhead to an average of 1.6 seconds. Compared to the current implementation of ESXi, Halite reduces restore overhead by 94%. This performance is a significant improvement over our previous work [24], which did not cope with divergence well and actually reduced performance for Worldbench.

8.2.2 pgbench

pgbench is a benchmarking tool, based on TPC-B, for the PostgreSQL database used to test the performance of a database installation. We used VMware's vFabric PostgreSQL [21] based on PostgreSQL 9.0. We ran pgbench and PostgreSQL in a Red Hat 6 Enterprise server in a VM with 2GB of memory and 4 vCPUs. pgbench measures database performance by recording the total number of transactions completed within a timed run.

We used a pgbench run of 5 minutes with 16 clients running on 4 threads. We ran pgbench with only select queries because we found the performance to be more consistent and it avoided disk contention. The default 85/15 read/write mix showed similar performance improvements, but with a larger range of performance over test runs. We checkpointed pgbench at the beginning of the run and collected the test results after restoring. vFabric PostgreSQL is designed to be an in-memory database, so we sized our database to 1.8GB.

We used ESXi with the same Halite configurations as the Worldbench experiments. pgbench shows the difference between the different configurations for a workload that has less divergence. The baseline performance of pgbench with no checkpoint taken is 6.9 million transactions. Figure 6 shows checkpointing overhead for pgbench as the reduction in number of transactions completed in 5 minutes. For example, pgbench completes 2.8 million transactions after restoring from a checkpoint on ESXi, a reduction of 4.1 million over the baseline. We chose to plot the overhead metric because it stays constant regardless of the length of the test.

For pgbench, Halite (P-A) only increases checkpointing overhead by 20% compared to ESXi. This increase is smaller than Worldbench because Linux workloads have better physical address locality due to Linux's buddy allocator. Still, Halite (A-L) reduces performance overhead by more than 2x for both ESXi and Halite (P-A). Halite with compression and the zero page optimization further reduces the overhead by 75%. Compared to ESXi, Halite reduces performance overhead by 89%.

8.2.3 Apache Webserver

To evaluate the performance of Halite for dynamic VM allocation that we described in Section 2.1, we created an experiment to simulate an Apache server application running in a VM. The test uses an Apache server with an

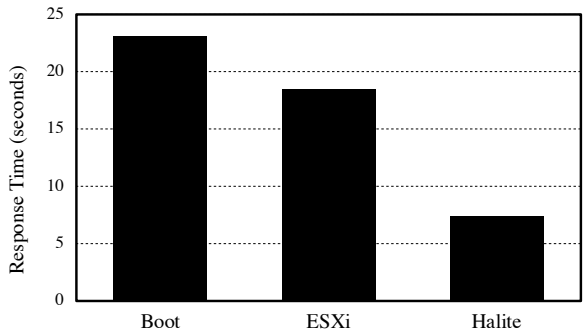


Figure 7: Apache Response Times. Time until first response (lower is better) from Apache server starting from boot of the VM, an ESXi checkpoint and a Halite checkpoint.

HTML dump of Spanish Wikipedia[2]. The test client requests pages randomly with a Gaussian distribution from a set of 10,000 pages from the dump. Before checkpointing, the client makes 10,000 requests to warm the cache. We ran Apache in Ubuntu 10.04 Server in a VM with 2 vCPUs and 2GB of memory.

We tested the performance of our server application with a few scenarios. First, we measured the response latency of the server when booting the VM and starting the web server on the first HTTP request. This scenario simulates dynamic allocation of VMs without using checkpointing. Next, we measured the response latency when restoring a checkpoint of a VM with a running Apache server using the current ESXi implementation. This scenario reflects the performance of using ESXi checkpointing for dynamic VM allocation. Finally, we tested the latency using Halite to restore the server VM.

Figure 7 gives the response times for each setup. Dynamically allocating a new VM for each connection requires 23 seconds on average to respond to the first HTTP request. Restoring a checkpointed VM with a running Apache server using the current ESXi checkpointing implementation gives response times of 18.4 seconds on average. Using Halite reduces response time to 7.3 seconds. Halite reduces the response time of a dynamically allocated web server VM by a factor of three compared to a cold boot of the VM and a factor of 2.5 compared to a restore using the current ESXi implementation, making it much more feasible to dynamically allocate VMs.

We also measured the response times for subsequent HTTP requests to the web server. Once the connection to the server has been established, our test client issues 10,000 random page requests, also with a Gaussian distribution. These measurements further show the benefits of Halite as well as showing the benefit of using a checkpoint, rather than a newly booted VM, for dynamic allocation. Table 2 shows the average and maximum response time for HTTP requests issued to the web server.

[2] `http://dumps.wikimedia.org/eswiki/`

Table 2: Average response time and maximum response time for the first 10,000 requests, excluding the first request.

	Avg. Response Time	Max Response Time
Boot	18 ms	25.168 s
ESXi	13 ms	9.333 s
Halite	3 ms	3.010 s

Table 3: Checkpoint save overhead for ESXi and Halite given as increase in runtime or reduction in transactions over baseline.

Workload	ESXi	Halite
Synthetic workload (sec)	1	4
pgbench (millions of trans.)	.31	.76
Worldbench (seconds)	11	5

Since the VM's cache is on disk at the beginning of all of these scenarios, the performance of the web server depends entirely on how efficiently the VM's cache can be filled from disk. Using a checkpointed VM reduces both average and maximum response times. The web server running in the VM already has a warmed cache, so the VM's working set just has to be restored, the cache does not have to be refilled. Restoring the checkpoint using ESXi reduces the response times, however, Halite performs the best because it is able to most efficiently restore the VM's working set from disk. Halite reduces the average response time of the web server by a factor of 6 and the maximum response time by a factor of more than 8 over booting the VM and starting the web server for each connection.

8.3 Checkpoint Save

Since most of the work for checkpoint save is done asynchronously in ESXi and Halite, the difference in performance between the two is minimal. Halite does copy-on-access checkpointing, which increases the checkpointing overhead, but writes out to disk sequentially, which reduces the overhead.

Table 3 gives performance results for our synthetic workload, pgbench and Worldbench. Performance was measured for each workload after a checkpoint was taken in the middle of the run. For the synthetic workload and pgbench, the additional read traces only reduced performance by 7-8%. These two workloads are both read-only

workloads, so the performance impact is higher than for a more balanced read-write workload. Halite performs better than ESXi for Worldbench due to the more write-heavy workload.

9 Related Work

Most previous checkpointing systems focused on checkpoint save performance for supporting fault tolerance, so there is a limited body of work on improving checkpoint restore performance. For systems that do not support lazy restore, the memory file organization is not important, so it is frequently not addressed. We also describe some techniques used by process and file system checkpointing systems to optimize checkpointing.

9.1 Virtual Machine Checkpointing

There is not a large body of work on virtual machine checkpointing and almost all of it focuses on checkpoint save performance. Many [15, 20] do not address the restore algorithm at all.

Commercial hypervisors all include support for checkpointing and restoring VMs, however not all support lazy checkpoint save or restore due to the complex implementation. For systems that do not support lazy checkpoint restore, the disk layout of checkpointed memory does not matter, although performance could be improved using compression. Xen supports lazy checkpointing [4], but it is not clear whether it supports lazy restore or what organization is used for checkpointed memory.

Our previous work [24] addressed the issue of lazy restore performance by prefetching the working set of the VM's memory before restarting the VM. However, we found that, while it was effective for simple workloads like MPlayer running on basic Linux, it offered little benefit for more complex workloads, like Windows desktop applications. These complex workloads have more divergence, and since working set restore depends on predicting which memory pages the VM will access on restore, it cannot cope with divergence. In contrast, Halite focuses on predicting which pages the VM will access *together* on restore, making it more effective for a wider range of workloads, including a 94% reduction in restore overhead for Windows workloads.

9.2 Other Virtualization

One related area of work is VM migration. Post-copy migration suffers from the same performance challenges as lazy restore due to the network latency while the VM is paused waiting for the page to be copied from the source. However, the organization of VM memory is not a factor because the VM's memory is not on disk on the source, so it can be accessed in any order with no performance penalty.

Hines et al. [7] implemented a background page walking thread that adaptively picks the order in which it walks depending on the last access. For each access, the page walker will try to push some of the other pages around that access in the physical address space. However, previous work [16] found that the guest virtual address space is a more reliable predictor of access locality and we found in our experiments that locality in the physical address space can be poor.

VM fork is another solution for dynamic allocation of virtual machines that requires restoring memory while the VM runs. Snowflock [10] depends on there being a small difference between forked VMs that the memory can be sent from the parent to the child with little performance degradation for the child VM. Kaleidoscope [2] groups pages based on what the page is used for as another way to predict access locality. Our approach is more general because it does not require paravirtualization to categorize pages.

9.3 Process Checkpointing

Previous work in optimizing checkpointing for individual or distributed processes has focused primarily on checkpoint save, but not checkpoint restore. Plank et al. [17] implemented the process checkpointing system *Ickp* using copy-on-write checkpointing as well as compression as an optimization. Li et al. [11] compare performance characteristics of four algorithms for checkpoint/restart of parallel programs. The work of Liao et al. [12], called Concurrent CKPT, aims to improve on the CLL algorithm by avoiding page table manipulation. However, all of this work focuses solely on checkpoint save performance, and does not discuss checkpoint restore.

9.4 Fast OS and Application boot

There has been some work on organizing operating systems and application files to improve boot times for both. Windows uses a mechanism called SuperFetch [8] that orders files on disk in the order that they are accessed during boot. SuperFetch uses an adaptive algorithm that tracks past boot processes to predict the order of accesses. Like Halite, SuperFetch addresses the performance of restoring some set of data by reordering the data on disk in a more optimal way. Unlike Halite, SuperFetch attempts to predict the order of all accesses, not just the locality, so performance suffers when there is divergence. However, divergence may be less of a concern for booting the OS.

Joo et al. [9] implemented a system that predicts and prefetches application data on application startup to optimize for interleaving application execution and I/O. Like other predictive techniques, theirs suffers from reduced performance on divergence, although divergence is less of a problem for applications. They do not address disk layout at all because their system is designed for SSDs. However, they could improve performance on SSDs by

reorganizing the data for prefetching into fewer blocks.

10 Conclusion

We presented a new checkpointing system, Halite, for VMware ESXi that reduces restore overhead for a range of workloads. Halite predicts which pages the VM will access together on restore and groups these pages into *locality blocks*. We showed that locality blocks offer significant performance benefits over other block organizations and copes well with divergence in complex VM workloads like Windows desktop applications. In particular, locality blocks outperform physical address blocks by 10x for Windows. Combining locality blocks with Halite's other optimizations, Halite reduces the overhead of checkpoint restore in VMware ESXi to 1.6 seconds for a Windows desktop workload, a reduction of 94%. This significant improvement in restore performance allows Halite to efficiently support new applications for VM checkpointing.

11 Acknowledgements

Many thanks to Karen Zee, Ron Mann and Dan Ports for their discussions on this work. Thanks to all of the members of the monitor group for their support throughout the project. Thanks to Steve Gribble and Pete Hornyack for their insightful evaluation of the evaluation. Thanks to the entire University of Washington systems lab and to our anonymous reviewers for their paper comments. Special thanks to our manager, Joyce Spencer, for her continued support and encouragement throughout this work.

References

[1] Amazon. Amazon EC2 FAQ. `aws.amazon.com/ec2/faqs/`.

[2] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys '11, pages 273–286, New York, NY, USA, April 2011. ACM.

[3] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating System Principles*, SOSP '01, pages 103–116, New York, NY, USA, October 2001. ACM.

[4] Patrick Colp, Chris Matthews, Bill Aiello, and Andrew Warfield. VM Snapshots, February 2009. `http://www.xen.org/files/xensummit_oracle09/VMSnapshots.pdf`.

[5] Tathagata Das, Pradeep Padala, Venkata N. Padmanabhan, Ramachandran Ramjee, and Kang G. Shin. Litegreen: saving energy in networked desktops using virtualization. In *Proceedings of the USENIX Annual Technical Conference*, USENIX'10, pages 3–3, Berkeley, CA, USA, June 2010. USENIX Association.

[6] Apache Software Foundation. Apache http server project, 2012. `http://httpd.apache.org/`.

[7] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 5th Conference on Virtual Execution Environments*, VEE '09, pages 51–60, Washington, DC, USA, March 2009. ACM.

[8] Thom Holwerda. SuperFetch: How it works & myths, May 2009. `http://www.osnews.com/story/21471/SuperFetch_How_it_Works_Myths`.

[9] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. Fast: quick application launch on solid-state drives. In *Proceedings of the 9th Conference on File and Storage Technologies*, FAST '11, Berkeley, CA, USA, February 2011. USENIX Association.

[10] Horacio Andrs Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th European Conference on Computer Systems*, EuroSys '09, pages 1–12, Nuremberg, Germany, April 2009. ACM.

[11] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Parallel & Distributed Systems*, pages 874–879, August 1994.

[12] Jianwei Liao and Yutaka Ishikawa. A new concurrent checkpoint mechanism for real-time and interactive processes. In *Proceedings of the 34th Computer Software and Applications Conference*, COMPSAC '10, pages 47–52, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Jean loup Gailly and Mark Adler. zlib. `zlib.net`.

[14] Michael J. Mior and Eyal de Lara. Flurrydb: a dynamically scalable relational database with virtual machine cloning. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 1–9, New York, NY, USA, 2011. ACM.

[15] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th Conference on Virtual Execution Environments*, VEE '11, pages 75–86, New York, NY, USA, March 2011. ACM.

[16] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: coalesced large-reach TLBs. In *Proceedings of the Conference on Microprogramming and Microarchitecture*, MICRO '12. IEEE, December 2012.

[17] James S. Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, June 1994.

[18] PostgreSQL. pgbench. `http://www.postgresql.org/docs/devel/static/pgbench.html`.

[19] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[20] Michael H. Sun and Douglas M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Georgia Institute of Technology, 2010.

[21] VMware. VMware vfabric postgres. `http://www.vmware.com/products/application-platform/vfabric-postgres/overview.html`.

[22] VMware. VMware vSphere Hypervisor. `www.vmware.com/products/vsphere-hypervisor/overview.html`.

[23] Ross N. Williams. `http://www.ross.net/compression/introduction.html`.

[24] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th Conference on Virtual Execution Environments*, VEE '11, pages 87–98, New York, NY, USA, March 2011. ACM.

Hyper-Switch: A Scalable Software Virtual Switching Architecture

Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha and Scott Rixner

Rice University

Abstract

In virtualized datacenters, the last hop switching happens inside a server. As the number of virtual machines hosted on the server goes up, the last hop switch can be a performance bottleneck. This paper presents the *Hyper-Switch*, a highly efficient and scalable software-based network switch for virtualization platforms that support driver domains. It combines the best of the existing I/O virtualization architectures by hosting device drivers in a driver domain to isolate faults and placing the packet switch in the hypervisor for efficiency. In addition, this paper presents several optimizations that enhance performance. They include virtual machine (VM) state-aware batching of packets to mitigate the costs of hypervisor entries and guest notifications, preemptive copying and immediate notification of blocked VMs to reduce packet arrival latency, and, whenever possible, packet processing is dynamically offloaded to idle processor cores. These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency.

We implemented a Hyper-Switch prototype in the Xen virtualization platform. This prototype's performance was then compared to Xen's default network I/O architecture and KVM's vhost-net architecture. The Hyper-Switch prototype performed better than both, especially for inter-VM network communication. For instance, in one scalability experiment measuring aggregate inter-VM network throughput, the Hyper-Switch achieved a peak of ∼81 Gbps as compared to only ∼31 Gbps under Xen and ∼47 Gbps under KVM.

1 Introduction

Machine virtualization is now used extensively in datacenters. This has led to considerable change to both the datacenter network and the I/O subsystem within virtualized servers. In particular, the communication endpoints within the datacenter are now virtual machines (VMs), not physical servers. Consequently, the datacenter network now *extends into the server* and *last hop switching* occurs within the physical server.

At the same time, thanks to increasing core counts on processors, server VM densities is on the rise. This trend is placing enormous pressure on the network I/O subsystem and the last-hop virtual switch to support efficient communication—especially between VMs—in virtualized servers.

There are many I/O architectures for network communication in virtualized systems. Of these, software device virtualization is the most widely used. This preference for software over specialized hardware devices is due in part to the rich set of features—including security, isolation, and mobility—that the software solutions offer.

The software solutions can be further divided into *driver domain* and *hypervisor-based* architectures. Driver domains are dedicated VMs that host the drivers that are used to access the physical devices. It provides a safe execution environment for the device drivers. Arguably, the hypervisors that support driver domains are more robust and fault tolerant, as compared to the alternate solutions that locate the device drivers within the hypervisor. However, on the flip side, they incur significant software overheads that not only reduce the achievable I/O performance but also severely limit I/O scalability [29, 31].

In existing I/O architectures, the virtual switch is implemented inside the same software domain where the virtual devices are implemented and the device drivers are hosted. For instance, all of these components are implemented inside a driver domain in Xen [13] and the hypervisor in KVM [26]. This collocation is purely a matter of convenience since packets must be switched when they are moved between the virtual devices and the device drivers.

In this paper, we introduce the *Hyper-Switch*, which challenges the existing convention by separating the virtual switch from the domain that hosts the device drivers. The Hyper-Switch is a highly efficient and scalable software-based switch for virtualization platforms that support driver domains. In particular, the hypervisor includes the data plane of a flow-based software switch, while the driver domain continues to safely host the device drivers. Since the data plane is small relative to the size of the switch control plane, it does not significantly increase the size of the hypervisor or the platform's overall trusted computing base (TCB). The *Hyper-Switch* explores a new point in the virtual switching design space.

Another contribution of this paper is a set of optimizations that increase performance. They enable the Hyper-Switch to efficiently support both bulk and latency sensitive network traffic. They include *VM state-aware batching* of packets to mitigate the costs of hypervisor entries on the transmit side and guest notifications on the receive side. *Preemptive copying* is employed at the receiving VM, when it is being notified of packet arrival, to reduce latency. Further, whenever possible, packet processing is *dynamically offloaded* to idle processor cores. The offloading is performed using a low-overhead mechanism that takes into account CPU cache locality, especially in NUMA systems.

These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. They take advantage of the Hyper-Switch data plane's integration within the hypervisor and its proximity to the scheduler. As a result, the Hyper-Switch enables much improved and scalable network performance, while maintaining the robustness and fault tolerance that derive from the use of driver domains. Further, we believe that these optimizations can and should be a part of any virtual switching solution that aims to deliver high performance.

We evaluated the Hyper-Switch using a prototype that was implemented in the Xen virtualization platform [4]. The prototype was built by modifying Open vSwitch [24], a multi-layer software switch for commodity servers. In this evaluation, the Hyper-Switch's performance was compared to that of KVM's vhost-net and Xen's default network I/O architectures. The results show that the Hyper-Switch enables much better scalability and peak throughput than both of these existing architectures.

The rest of this paper is organized as follows. Section 2 further motivates the Hyper-Switch by discussing some of the issues with existing architectures. Section 3 explains the Hyper-Switch's design. Section 4 describes the implementation of the Hyper-Switch prototype. Section 5 presents a detailed evaluation of the Hyper-Switch. Section 6 discusses related work. Finally, Section 7 summarizes the conclusions.

2 Motivation

The need for efficient and scalable network communication within virtualized servers is increasing. Intel already claims to have an architecture that can scale to 1000 cores on a single chip [15]. Furthermore, the number of cores on a chip is predicted to grow to 64 in a few years and to 256–512 by the end of the decade [12]. If this last prediction is borne out, then in 2020 a single 1U server will have as many cores as an entire rack of servers does today.

In addition, communication between servers within the same datacenter already accounts for a significant fraction of the datacenter's total network traffic [14]. Moreover, Benson *et al*.'s study of multiple datacenter networks reported that 80% of the traffic originating at servers in cloud datacenters never leaves the rack [5]. If the predictions for the growing number of cores come to pass, then a rack of servers may be replaced by VMs within a single physical server, and the network traffic that today never leaves the rack may instead never leave the server. Consequently, the Hyper-Switch has been heavily optimized to enable high performance inter-VM communication.

Modern multi-core systems enable concurrent processing of network packets. Under Xen's default network architecture, the driver domain can run in parallel to the transmitting and receiving VMs. Consequently, it is possible to perform packet switching in parallel with packet transmission and reception. However, there are several fundamental problems with traditional driver domain architectures that limit I/O performance scalability. Fundamentally, the driver domain must be scheduled to run whenever packets are waiting to be processed. This might involve scheduling multiple virtual processors depending on the number of threads used for packet processing in the driver domain. As a result, scheduling overheads are incurred while processing network packets. Further, the driver domain must be scheduled in a timely manner to avoid unpredictable delays in the processing of network packets.

Today, it is standard practice in real-world virtualization deployments to dedicate processor cores to the driver domain. This avoids scheduling delays, but often leaves cores idle. In fact, dedicating CPU resources for backend processing is not limited to just driver domain-based architectures. There have also been several proposals to offload some of the packet processing to dedicated cores—including Kumar *et al*.'s sidecore approach [17], and Landau *et al*.'s split execution (SplitX) model [18]. But, this can lead to underutilization of these cores. Further, this goes against one of the fundamental tenets of virtualization, which is to enable the most efficient utilization of the server resources. Hence the Hyper-Switch has been designed to smartly and dynamically utilize the available resources.

At the same time, reliability cannot be ignored, especially as servers in datacenters move toward multi-tenancy. Hypervisors that support driver domains are potentially more robust and fault tolerant. However, driver domains incur significant overheads. These overheads are due to the costs of moving packets between the guest VMs and the driver domain [21, 29, 31], because the driver domain cannot trivially access a packet in the guest VM's memory. The driver domain is just

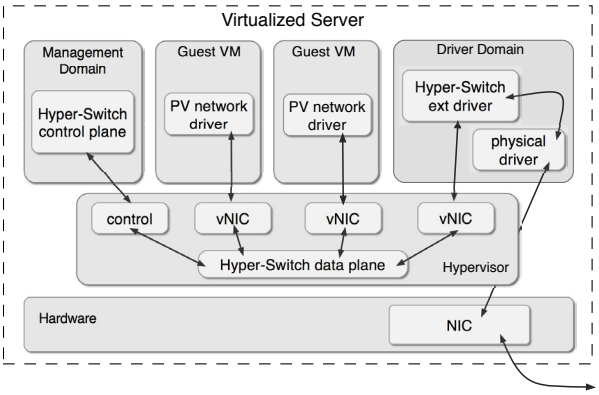


Figure 1: *The Hyper-switch architecture. The last hop virtual switch is implemented partly in the hypervisor (data plane) and partly in the management layer (control plane). The device drivers are hosted in the driver domain.*

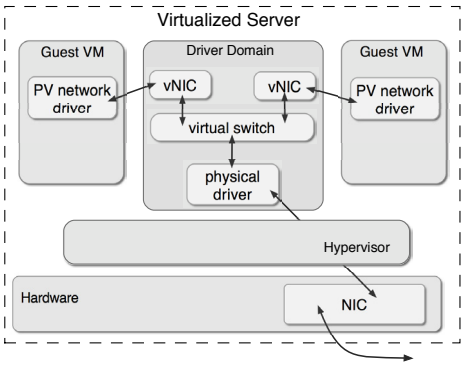


Figure 2: *Traditional driver domain architecture. The driver domain hosts the device drivers and the last hop virtual switch.*

another VM and the hypervisor maintains memory isolation between VMs. So, the driver domain must use an expensive memory sharing mechanism provided to access the packet. Hypervisor-based architectures do not incur these memory sharing overheads since the packets in the guest VMs' memory can be directly accessed from the hypervisor. The Hyper-Switch has been designed to take advantage of driver domains without incurring the associated memory sharing overheads.

3 Hyper-Switch Design

Figure 1 illustrates the Hyper-Switch architecture. There are two fundamental aspects to this architecture. First, unlike existing systems that use driver domains, the Hyper-Switch architecture—as the name implies—implements the virtual switch inside the hypervisor. So internal network traffic between virtual machines (VMs) that are collocated on the same server is handled entirely within the hypervisor. Incoming external network traffic is initially handled by the driver domain, since it hosts the device drivers, and then is forwarded to the destination VM through the Hyper-Switch. For outgoing external traffic, these two steps are simply reversed. In essence, from the Hyper-Switch's perspective, two guest VMs form the endpoints for internal network traffic, and the driver domain and a guest VM form the endpoints for external network traffic. Contrast this with the traditional driver domain architecture as illustrated in Figure 2.

Second, the hypervisor implements just the data-plane of the virtual switch that is used to forward network packets between VMs. The switch's control plane is implemented in the management layer. So the virtual switch implementation is distributed across virtualization software layers with only the bare essentials implemented inside the hypervisor. The separation of control and data planes is achieved using a flow-based switching approach. This approach has been previously used in other virtual switching solutions such as Open vSwitch [24]. However, Open vSwitch's control and data planes are both implemented inside the driver domain.

The rest of this section describes the Hyper-Switch's design in detail. First, the basics are explained by describing the path taken by a network packet. This is followed by several performance optimizations that improve upon the basic design.

3.1 Basic Design

Packet processing by the Hyper-Switch begins at the transmitting VM (or driver domain) where the packet originates and ends at the receiving VM (or driver domain) where the packet has to be delivered. It proceeds in four stages: (1) packet transmission, (2) packet switching, (3) packet copying, and (4) packet reception.

Packet Transmission. In the first stage, the transmitting VM pushes the packet to the Hyper-Switch for processing. Packet transmission begins when the guest VM's network stack forwards the packet to its paravirtualized network driver. Then the packet is queued for transmission by setting up descriptors in the transmit ring. A single packet can potentially span multiple descriptors depending on its size. Typically, packets are never segmented in the transmitting guest VM. So the packets belonging to internal network traffic can be forwarded as is. The external packets are segmented either in the driver domain or the network hardware. Today, segmentation offload is a standard feature in most network devices.

Packet Switching. In the second stage of packet processing, the packet is switched to determine its destination. Switching is triggered by a hypercall from the

transmitting VM. It begins with reading the transmit ring to find new packets. Each packet is then pushed to the Hyper-Switch's data plane where it is processed using flow-based packet switching. The data plane must be able to read the packet's headers in order to switch it. Since the data plane is located in the hypervisor, which has direct access to every VM's memory, it can read the headers directly from the transmitting VM's memory.

Packet switching by the data plane proceeds as follows: (1) The packet header fields are parsed to identify the corresponding packet flow. (2) The packet flow is used to lookup a matching flow rule in a software flow table. When this lookup fails, the packet is forwarded to the control plane in the management layer. (3) A successful flow table lookup identifies a flow rule, which specifies one or more actions to be performed. Typically, the action is to forward the packet to one or more destination ports or to drop the packet. Each port has an internal receive queue where the switched packet is temporarily placed. This port corresponds to a virtual network interface (vNIC) within the destination VM.

When the flow table lookup fails, the packet is forwarded to the control plane through a separate *control interface*. The control plane decides how the packet must be forwarded based on packet filtering rules, forwarding entries from an Ethernet address learning service, and/or other protocol specific tables. This is composed into a new flow rule that specifies the actions to be performed on packets belonging to this flow. Then the packet is re-injected into the hypervisor's data plane and the associated actions are executed. Finally, the control plane adds the new flow rule to the flow table. This allows the flow's subsequent packets to be handled entirely within the hypervisor's data plane.

Packet Copying. In the third stage of packet processing, the switched packet is copied into the receiving VM's memory. Empty buffers for receiving new packets are provided through the vNIC. Specifically, the descriptors in the receive ring provide the address of the empty buffers in the VM's memory.

By default, the destination VM is responsible for performing packet copies. Once switching is completed, the destination VM is notified via a virtual interrupt. Subsequently, that VM issues a hypercall. While in the hypervisor, it dequeues the packet from the port's internal receive queue, and copies the packet into the memory given by the next descriptor in the receive ring. The packet is copied directly from the transmitting VM's memory to the receiving VM's memory. After which, the memory that was allocated for this packet at various places—inside the hypervisor and in the transmitting VM—is released.

Packet Reception. In the fourth and final stage, the para-virtualized network driver in the receiving VM reconstructs the packet from the descriptors in the receive ring. Typically, the receiving OS is notified, through interrupts, that there are new packets to be processed in the receive ring. However, under the Hyper-Switch, the receiving VM was already notified in the previous stage. So packet reception can happen as soon as the hypercall for copying the packet is complete. The new packet is then pushed into the receiving VM's network stack.

3.2 Preemptive Packet Copying

Packet copies are performed by default in a receiving VM's context. When a packet is placed in the internal receive queue, after it has been switched, the receiving VM is notified. Eventually, the receiving VM calls into the hypervisor to copy the packet. However, delivering a notification to a VM already requires entry into the hypervisor. Under Xen, when there is a pending notification to a VM, the VM is interrupted and pulled inside the hypervisor. Since hypercalls are expensive operations, the Hyper-Switch tries to avoid them. In this case, it takes advantage of the hypervisor entry upon event notification to avoid a separate hypercall to perform the packet copy. Instead, the packet copy is performed *preemptively* when the receiving VM is being notified. In essence, the packet copy operation is combined with the notification to the receiving VM. This optimization avoids one hypervisor entry for every packet that is delivered to a VM.

3.3 Batching Hypervisor Entries

In the Hyper-Switch architecture, as described thus far, the transmitting VM enters the hypervisor every time there is a packet to send. Moreover, the receiving VM is notified every time there is a packet pending in the internal receive queue. As mentioned before, even this notification requires hypervisor intervention.[1] Therefore, despite the preemptive packet copy optimization, the overhead of entering the hypervisor is incurred multiple times on every packet.

To mitigate this overhead, we use *VM state-aware batching*, which amortizes the cost of entering the hypervisor across several packets. This approach to batching shares some features with the interrupt coalescing mechanisms of modern network devices. Typically, in network devices, the interrupts are coalesced irrespective

[1] In Xen, notifying a running guest VM involves two entries into the hypervisor. First, the running VM is interrupted via an IPI and forced to enter the hypervisor. Then the hypervisor runs a special exception context where the guest VM handles all pending notifications. Finally, the guest VM again enters the hypervisor to return from the exception context.

of whether the host processor is busy or not. But, unlike those devices the Hyper-Switch is integrated within the hypervisor, where it can easily access the scheduler to determine when and where a VM is running. So a blocked VM can be notified immediately when there are packets pending to be received by that VM. This enables the VM to wake up and process the new packets without delay. On the other hand, the notification to a running VM may be delayed if it was recently interrupted.

3.4 Offloading Packet Processing

In Hyper-Switch, by default, packet switching is performed in the transmitting VM's context and packet copying is performed in the receiving VM's context. As a result, asynchronous packet switching does not occur with respect to the transmitting VM, and asynchronous packet copying does not occur with respect to the receiving VM. However, concurrent and asynchronous packet processing can significantly improve performance.

Concurrent packet processing can be achieved by polling: (1) all the internal receive queues, looking for packets waiting to be copied, and (2) all the transmit rings, looking for packets waiting to be switched. This can be performed by processor cores that are currently idle. Packet copying is prioritized over switching because packet copying is typically the more expensive operation and the receiving VM is more likely to be performance bottlenecked than a transmitting VM.

Instead, the idle cores are woken up just when there is work to be done. On the receive side, this can be ascertained precisely when packets are placed in an internal receive queue of a vNIC. Then one of the idle cores is chosen and woken up to perform the packet copy. A low-overhead mechanism is used to offload work to the idle cores. It uses a simple interprocessor messaging facility to request a specific idle core to copy packets at a specific vNIC. Further, this mechanism attempts to spread the work across many idle cores. Otherwise, if all the work is offloaded to a single idle core, it might become a bottleneck.

The offloading to idle cores is delayed if the receiving VM is going to be notified immediately. As explained previously, this typically happens when the receiving VM is not running. Subsequently, the receiving VM copies a bounded number of packets sufficient to keep it busy, and then if packets are still pending in the internal receive queue, the remaining copies are offloaded to an idle core. The rationale is to immediately copy some packets so that the receiver can start processing them, while the remaining packets are concurrently copied at an idle core.

Unfortunately, offloading packet switching to idle cores is not trivial. In the common case, packets are asynchronously queued by the transmitting VM without entering the hypervisor. So it is not possible to offload the switching tasks precisely when packets are queued. Therefore, packet switching is performed at the idle cores only as a *side effect* of offloading packet copies. In other words, when an idle core is woken up to perform packet copies, it also polls all the transmit rings looking for packets pending to be switched.

Further, when packets are being processed by an idle core, the Hyper-Switch checks for any other work that might need that core. If so, it aborts the packet processing. This ensures that the offloaded packet processing happens at the lowest possible priority and does not prevent other tasks from using that processor.

CPU Cache Awareness. CPU cache locality can have a significant impact on the cost of packet copying under Hyper-Switch. Essentially, the packet data is accessed in three places:[2] (1) The transmitting VM, (2) the packet copier, and (3) the receiving VM. So the packet data can be potentially brought into three different CPU caches depending on the system's cache hierarchy and where the two VMs and the packet copier are run.

If the receiving VM is also the packet copier, then the packet data is brought into the receiving VM's CPU cache while the copy is performed. Subsequently, when the packet is accessed in the receiving VM, it can be read with low latency from the cache. But if the packet copier runs on an idle core, the access latency will depend on whether the idle core shares any cache with the receiving VM's core. Therefore, under Hyper-Switch, the offload mechanism for packet processing is optimized to take advantage of CPU cache locality. At the same time, it ensures that the offloaded work does not unfairly affect the performance of other VMs running on cores that share their CPU cache with the idle cores.

Hysteresis. Waking up an idle core takes a non-trivial amount of time, particularly when the idle core is using deeper sleep states to save power. Further, the interprocessor interrupts (IPIs) that are used to wake up cores are not cheap. Therefore, a small hysteresis period is introduced to ensure that the idle cores stay awake longer than they normally would. The idea is to keep the cores running, after they are woken up, until there is a period—the hysteresis time period—during which no packets are processed. In other words, the idle cores are kept running as long as there is a steady stream of packets to process.

3.5 More Packet Processing Opportunities

A packet that is queued in the transmit ring at a vNIC will *eventually* be switched by either the transmitting VM or

[2] Packet switching is ignored here since it only accesses the packet headers.

an idle core. This might happen immediately if an idle core polls this interface looking for packets waiting to be switched or it might happen only when the transmit timer period that is implemented by VM-state aware batching elapses.[3]

Consider a VM that queues some packets for transmission at its vNIC and then blocks. Let's assume that there are no other idle cores. If another VM is scheduled to run on this core, then the queued packets are not going to be switched until the blocked VM is scheduled to run again. But this might happen only at the end of the transmit timer period. Even if the core becomes idle after the VM blocks, there is no guarantee that the blocked VM's packets will be switched at that idle core. In fact, the idle core can end up copying packets destined for other VMs. In essence, a VM can block despite its packets waiting to be switched.

When a VM's virtual processor blocks, it has to enter the hypervisor to give up its core. Since the VM is already inside the hypervisor, it might as well as check if there are packets pending to be switched or copied. This allows any packet processing work to be completed before the VM stops running. Also, new packet copies result in a notification to the VM. Consequently, instead of blocking, the VM returns to process the packets that were just received.

4 Implementation Details

We implemented a prototype of the Hyper-Switch architecture, which is depicted in Figure 3. We implemented the switch's data plane by porting parts of Open vSwitch to the Xen hypervisor. Open vSwitch's control plane was used without modification. We also developed a new para-virtualized (PV) network interface for the guest VMs to communicate with the data plane. The same interface was also used by the driver domain to forward external network traffic. The rest of this section describes each part of the Hyper-Switch prototype in detail.

Open vSwitch Overview. Open vSwitch [24] is an OpenFlow compatible, multi-layer software switch for commodity servers. The control and data planes are separated. While the data plane is implemented inside the OS kernel, the control plane is implemented in user space. It uses the flow-based approach for switching packets in its data plane. In a typical deployment of Open vSwitch as a last hop virtual switch, it is implemented entirely inside a driver domain (Xen) or the hypervisor (KVM). In the common case, the network traffic between the guest VMs is directly switched by Open vSwitch's data plane within the kernel. Open vSwitch provides a *vport* abstraction that can be bound to any network interface in the driver domain. In addition, there is one vport for every vNIC in the system.

[3]The maximum delay is bounded by the transmit timer period.

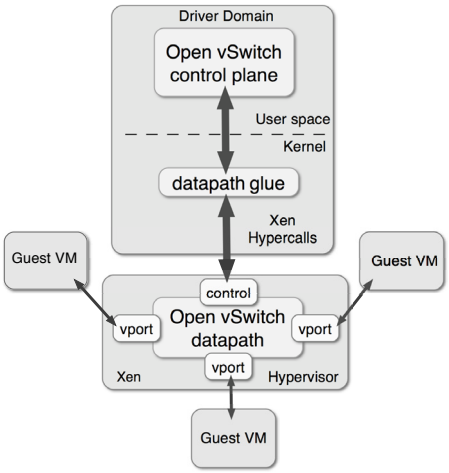


Figure 3: *Hyper-Switch prototype. It was built by porting essential parts of Open vSwitch's datapath to the Xen hypervisor.*

Porting Open vSwitch's Datapath. We implemented the Hyper-Switch's data plane by porting Open vSwitch's datapath to the Xen hypervisor. The vports on the datapath were bound to a newly developed para-virtualized network interface that allowed guest VMs to communicate with the Hyper-Switch's data plane.

The driver domain kernel also included a datapath glue layer to enable communication between the control and data planes. This layer converted the commands from Open vSwitch's control plane into a new set of Xen hypercalls to manipulate the flow tables in the datapath. The glue layer also transferred the packets that are punted to the control plane.

Para-virtualized Network Interface. The guest VMs and the driver domain communicated with the Hyper-Switch through a para-virtualized network interface (vNIC). The interface included two transmit rings—one for queueing packets for transmission and another for receiving transmission completion notifications—and one receive ring to deliver incoming packets. The rings were fixed circular buffers where the producer and consumer(s) could access the ring descriptors concurrently. The interface also included an internal receive queue that contained packets that were yet to be copied into the receiving VM's memory.

Hypervisor Integration. As explained in Section 3.2, packet copying was preemptively performed by combining it with the notification to the receiving VM. We implemented this by checking for packets to copy when the associated virtual interrupt was delivered by the Xen hypervisor to a VM. Further, packet switching and copying were also performed when a VM *voluntarily* blocked.

Thus the VM's vNIC was polled for packets to be copied or switched, just before the scheduler was invoked to yield the processor and find another VM to run.

Offloading Packet Processing. We implemented the offloading of packet processing inside Xen's *idle domain*. The idle domain contains one *idle vCPU* for every physical CPU core in the system. The idle vCPUs have the lowest priority among all the vCPUs and therefore, they are scheduled to run on a physical CPU core only when none of the VMs' vCPUs are runnable on that core. The idle vCPUs execute an *idle loop* that checks for pending softirqs and tasklets, and executes the corresponding handlers. Finally, when there is no more work to be done, it enters one of the sleep states to save power.

In the Hyper-Switch architecture, we extended Xen's idle loop to copy and switch packets. A simple, low-overhead mechanism was used to offload packet processing to idle cores. The mechanism identified a suitable idle core based on an *offload criteria*. The criteria were chosen to select an idle core that made the best use of the CPU caches. Further, this mechanism also ensured that the offloaded work was distributed across multiple idle cores using a simple hash function. The mechanism included a lightweight interprocessor messaging facility that was implemented using small fixed circular buffers. There was one buffer for every processor core in the system. It was used to communicate the vNICs that were being offloaded to a specific idle core. The Hyper-Switch-related packet processing was performed only at the lowest priority. The pending softirqs and tasklets were checked after each packet was processed. If there was ever higher priority work to be done, then the offloaded packet processing was aborted.

5 Evaluation

This section presents a detailed evaluation of the Hyper-Switch architecture. The evaluation was performed using the Hyper-Switch prototype in Xen. The primary goal of this evaluation was to compare Hyper-Switch with existing architectures that implement the virtual switch either entirely within the driver domain or entirely within the hypervisor. Toward this end, the end-to-end performance under Hyper-Switch was compared to that under Xen's default driver domain-based architecture and KVM's hypervisor-based architecture.

5.1 Experimental Setup and Methodology

The experiments were run on a 32-core server with two 2.2 GHz AMD Opteron 6274 processors and 64 GB of memory. This processor is based on AMD's Bulldozer micro-architecture where two cores (called a *module*) share the second level data cache (L2) and the instruction caches (L1i and L2i). Further, four modules (called a *node*) share the unified third level cache (L3). And each Opteron 6274 processor includes two such nodes. Under Xen,[4] the server was configured to run up to 32 para-virtualized (PV) Linux guest VMs (v2.6.38 pvops) and one PV Linux driver domain (v2.6.38 pvops), in addition to the privileged management domain 0 (Linux v3.4.4 pvops). The PV linux guests use a specialized network driver which is optimized for the virtual network interface that the hypervisor provides to the VMs. The guest VMs were each configured with a single virtual CPU (vCPU) and 1 GB of memory. The driver domain was configured with up to 8 vCPUs and 2 GB of memory. But under Hyper-Switch the driver domain was given only a single vCPU since it only handled external network traffic. The server was directly connected to an external client using a 10 Gbps Ethernet link. The client consisted of a 2.67 GHz Intel Xeon W3520 quad-core CPU and 6 GB of memory. It ran an Ubuntu distribution of native Linux kernel v2.6.32. The CPUs at the external client were never a performance bottleneck in any of the experiments.

The netperf microbenchmark [2] was used in all the experiments to generate network traffic. In particular, netperf was used to create two types of network traffic: (1) TCP stream and (2) UDP request/response traffic. The TCP stream traffic was used to measure the achievable throughput. The UDP request/response traffic was used to measure the packet processing latency. Unless otherwise specified, the `sendfile` option was used on the transmit side in all experiments. The performance of Hyper-Switch was compared to the performance of Open vSwitch under both Xen [13] and KVM [26]. Para-virtualized network interfaces were used in all these systems. In the rest of this section, we use "KVM" to refer to the performance of Open vSwitch under KVM. Similarly, we use "Xen" to refer to the performance of Open vSwitch under Xen's default network I/O architecture. This should not be confused with the Hyper-Switch prototype that is also implemented in Xen.

Open vSwitch under Xen. In Xen, Open vSwitch is implemented entirely in the driver domain. Under Xen, all network packets are forwarded to the driver domain, where they are switched. Xen's backend driver called *netback* acts as an intermediary between the guest VMs and the virtual switching module in the driver domain. Netback is multi-threaded, and there is one netback (kernel) thread for every vCPU in the driver domain. Each guest VM's vNIC is bound to one of these threads. The packets associated with a specific vNIC are processed only by the thread to which it is bound. The recom-

[4] Xen v4.2 - mainline git repository (`xen-unstable.git`) May 2012.

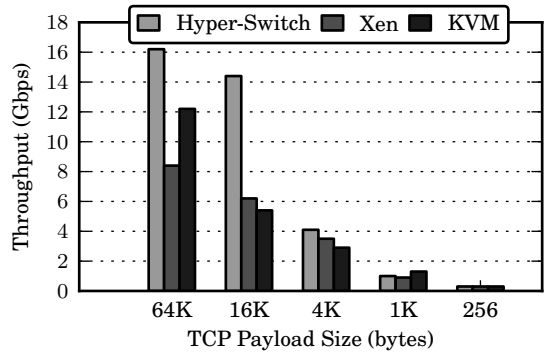


Figure 4: *Throughput results from TCP stream traffic between a single pair of VMs under different payload sizes.*

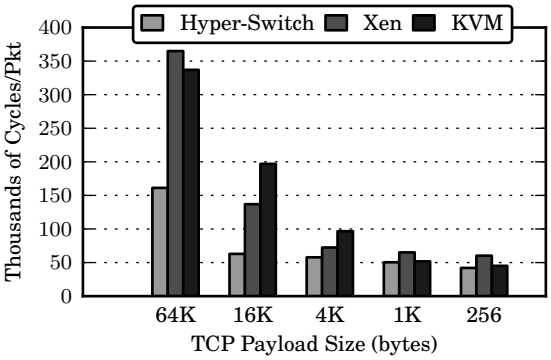


Figure 5: *CPU load results from TCP stream traffic between a single pair of VMs under different payload sizes.*

mended practice is to dedicate cores for running the driver domain's vCPUs. In this evaluation, the driver domain was configured with up to 8 vCPUs.

Open vSwitch under KVM. In KVM, Open vSwitch is implemented entirely in the hypervisor (also referred to as the *KVM host*). Under KVM's *vhost-net* architecture, all network packets are forwarded to the vhost-net driver in the host, which is similar to Xen's netback. But unlike netback, there is a separate vhost-net (kernel) thread for every vNIC in the system. The vhost-net's threads can also be run on dedicated cores.

5.2 Experimental Results

5.2.1 Inter-VM Performance and Scalability

In these experiments, network performance was studied under different loads by setting up network traffic between VMs collocated on the same server.

Single VM Pair. In the first set of experiments, traffic was set up between just a single pair of VMs. Each guest VM's vCPU was pinned to a separate core within the same processor node to avoid any potential VM scheduling effects. Xen's driver domain was configured with 2 vCPUs. Recall that there is one netback kernel thread for every vCPU in Xen's driver domain. The driver domain's vCPUs were also pinned to separate processor cores, but on the same processor node where the corresponding guest VMs' vCPUs were pinned. Similarly, under KVM, the two vhost-net kernel threads (one per guest VM) were also pinned.

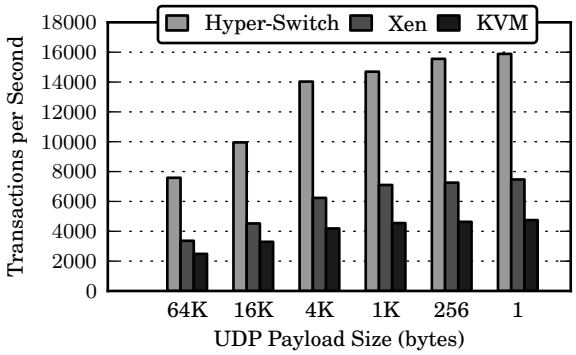


Figure 6: *Latency results from UDP request/response traffic between a single pair of VMs under different payload sizes.*

First, as shown in Figure 4, higher throughput was achieved under Hyper-Switch than under both the existing architectures in the experiments where the TCP payload was between 4 KB and 64 KB, with stream-based traffic. On average, the throughput under Hyper-Switch, in these cases, was ∼56% higher than that under Xen and ∼61% higher than that under KVM. But there was not much performance difference at smaller packet sizes since in those experiments the transmitting VM was the performance bottleneck. Figure 5 shows the average CPU load (cycles/packet) in each of these experiments. Clearly, the Hyper-Switch is more efficient in processing packets than both the existing architectures in KVM and Xen.

Second, as shown in Figure 6, higher transactions per second was achieved under Hyper-Switch, across all UDP payload sizes, with request-response traffic. A transaction comprises of a single request followed by a single response in the opposite direction. So these results indicate that the round-trip packet latencies were the lowest under the Hyper-Switch among all the three architectures. On average, the transactions per second under Hyper-Switch was ∼117% higher than that under Xen and ∼222% higher than that under KVM. So the Hyper-Switch architecture is suited for both bulk as well as latency sensitive network traffic. Further, these results show the benefit from optimizations such as preemptive copying and immediate notification of blocked VMs that enable timely delivery of packets.

Pairwise Scalability Experiments. In the next set of experiments, the performance scalability of the three architectures was studied by setting up TCP stream-based traffic flows between 1–16 pairs of VMs in one direction. TCP payload size of 64 KB was used in all the sub-

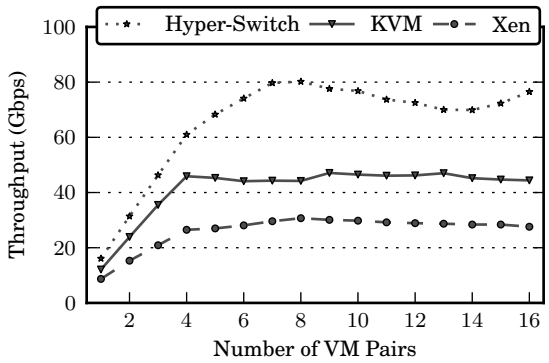


Figure 7: *Pairwise performance scalability results. Multiple concurrent TCP streams set up between pairs of VMs. Figure shows aggregate inter-VM throughput as the number of VM pairs is increased.*

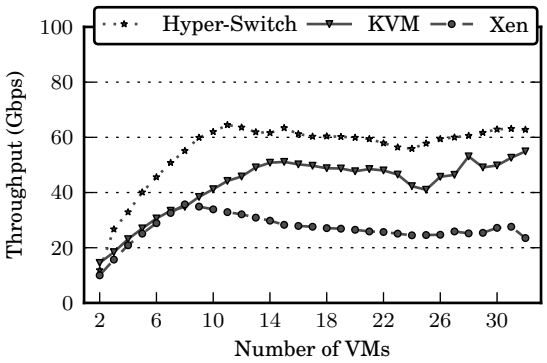


Figure 8: *All-to-all performance scalability results. Multiple concurrent TCP streams set up between all VMs. Figure shows aggregate inter-VM throughput as the number of VMs is increased.*

sequent experiments. Again, the guest VMs' vCPUs, the vhost-net kernel threads (KVM), and the driver domains' vCPUs (Xen) were pinned to specific processor cores. Also, the VMs that were communicating with each other were always pinned to the same processor node. The pinning was done such that, in each experiment, the load was uniformly distributed across all the processor modules and nodes in the system. For instance, one core from each module was used for pinning VMs across the system, before the other cores were used. Under KVM, the guest VMs' vCPUs and the vhost-net kernel threads were pinned. As a result, when the system was scaled beyond 8 pairs of VMs, each processor core had to run one of the guest VM's vCPUs *and* one of the vhost-net threads. Under Xen, the driver domain was configured with 8 vCPUs. The driver domain's vCPUs were distributed by pinning two of them to each processor node in the system. Then the guest VMs' vCPUs were evenly distributed across the remaining processor cores.

The results in Figure 7 show that the Hyper-Switch architecture exhibited much better performance scalability than both the existing architectures. Specifically, under Hyper-Switch, the performance reached a peak throughput of ∼81 Gbps before it started to flatten out. But the peak throughput was only ∼47 Gbps and ∼31 Gbps under KVM and Xen respectively. Further, the performance under these existing architectures did not scale beyond 4 pairs of VMs. On average, the throughput under Hyper-Switch was ∼55% higher than that under KVM and ∼146% higher than that under Xen. Figure 7 also shows three distinct regions in Hyper-Switch's performance curve: (1) The performance scaled almost linearly, from ∼16.2 Gbps to ∼62.7 Gbps, between 1 and 4 pairs of VMs. (2) The performance continued to scale linearly but at a lower rate, from ∼62.7 Gbps to ∼81 Gbps, between 5 and 7 pairs of VMs. (3) The performance did not scale beyond 8 pairs of VMs.

Fundamentally, the network performance is determined by the number of packets that can be transferred between the source and destination VMs in a given time. A typical packet transfer involves switching and packet copying overheads. But there are limits to how many packets that can be processed by a single processor. This is determined in part by the underlying hardware architecture. The hardware determines how efficiently the available processor time is used to process—switch and copy—packets. Today's processors are incredibly complex and therefore, there are several factors that impact this efficiency. In particular, the structure of the memory subsystem can have a significant impact on performance [25]. This includes the size and levels of the CPU caches, the maximum number of outstanding reads/writes/cache misses, the available memory bandwidth, the number of channels to the system memory, and so on. One can scale the performance beyond the limits imposed by a single processor core by increasing concurrency, *i.e.* by using multiple processor cores. But some of the system resources could be shared between processor cores—such as CPU caches, memory channels, etc.—that could potentially reduce the available concurrency. When additional VMs are added to the system, there is a natural increase in concurrency since many of the switching tasks can be concurrently performed under each VM's context. Further, under Hyper-Switch, the offloading of packet processing adds to this concurrency. When choosing an idle core for offloading packet processing, preference is given to idle processor cores that are on the same node as the receiving VM's vCPU, to take advantage of any CPU cache locality. Further, the packet processing is offloaded only to a processor core in an idle module, i.e. a module where both the processor cores are idle, to avoid potential cache interference effects [3].

In the first region of the curve (Figure 7), each pair of VMs were run in a separate processor node. So the packet processing could be offloaded to other cores

within the same node. As a result, under these conditions, the best scalability was achieved. In the second region, some of the packet processing had to be offloaded to idle modules on other nodes in the system. This was not as efficient since packets had to be copied across processor nodes. Hence the performance scalability was reduced. In the third region, the performance stopped scaling in part due to the reduction in the offloading of packet processing since most of the processor modules were busy. Also, some of the VMs' vCPUs were running on two cores within the same processor module. So the cache interference effects also came into effect. Finally, as more VMs were added to the system, there was increased contention for the system resources such as CPU caches. So, effectively, all these factors offset the increase in packet processing concurrency and hence the performance stopped scaling.

All-to-all Scalability Experiments. In the second set of scalability experiments, TCP stream-based traffic was set up between every pair of VMs in the system in both directions. These experiments were designed to generate significant load on the network by having tens of VMs concurrently communicating with each other. For instance, when there were 30 VMs in the system, there were as many as 870 concurrent TCP flows. The configuration and setup was similar to the previous set of experiments.

Figure 8 shows the results from these experiments. The performance again scaled much better under Hyper-Switch than under KVM or Xen. Specifically, under Hyper-Switch, the performance reached a peak throughput of ∼65 Gbps as compared to ∼55 Gbps and ∼31 Gbps under KVM and Xen respectively. Similar to the previous set of experiments, the performance curve under Hyper-Switch scaled up very well at the beginning before tapering off. The performance analysis presented with the previous results is applicable here as well. In fact, the contention for system resources is even higher in this case, due to the significant load placed on the system.

5.2.2 External Performance

In the external experiments, the network traffic was set up between guest VM(s) and the external client. The driver domain, under Hyper-Switch and Xen, was configured with only a single vCPU. In the TX and RX experiments, there were one or two guest VMs (concurrently) sending and receiving packets respectively. The guest VMs' vCPUs and the driver domain's vCPU were again pinned.

The results from these experiments showed that the Hyper-Switch's performance was comparable and in some cases even better than the performance under KVM and Xen. In the TX experiments, with a single guest VM transmitting packets, line rate of ∼9.4 Gbps was achieved under both Hyper-Switch and Xen. But under KVM, the TX VM's vCPU was a performance bottleneck. Therefore, only ∼7.8 Gbps was possible in this case. In the RX experiments, with one guest VM receiving packets, the CPU at the guest VM was the bottleneck. So line rate was not achieved under any of the architectures. But the performance was better under Hyper-Switch (7.5 Gbps) and KVM (7.8 Gbps) than Xen (4.1 Gbps). But with two guest VMs receiving packets, line rate of ∼9.4 Gbps was achieved under both Hyper-Switch and KVM. Under Xen, the driver domain's vCPU was the performance bottleneck. Therefore, having a second guest VM receive packets had no positive impact on the aggregate throughput. These results show that the driver domain under Hyper-Switch can send and receive packets at 10 GbE line rate using a single CPU core. So the driver domain consumes minimal resources.

TCP request-response traffic was also set up between a single guest VM and the external client. In these experiments, the Hyper-Switch achieved 13,243 transactions per second of as compared to 10,721 and 11,342 under KVM and Xen respectively. As explained before, higher transactions per second indicate lower round trip latency. Therefore, despite the "longer" route taken by packets under Hyper-Switch due to their forwarding through both the hypervisor and the driver domain, the packet latencies were still the lowest under Hyper-Switch.

5.2.3 Design Evaluation

Experiments were also run to determine the *offload criteria* under Hyper-Switch. In these experiments, the packet processing was offloaded to different CPU cores relative to where the transmitting and receiving VMs' vCPUs were running. The results from these experiments[5] indicated that, for best performance, packet processing must be offloaded to a processor module where both the cores were idle. Further, while searching for idle modules, first the processor node on which the receiving VM's vCPU was running must be searched, before searching the other nodes in the system. However, the offload criteria could vary depending on a processor's cache hierarchy. So the exact criteria must be determined based on the particular hardware platform on which Hyper-Switch is run.

6 Related Work

The current state-of-the-art network subsystem architectures for virtualized servers can be broadly classified

[5]Due to lack of space, the results from these experiments are not presented here.

into three categories. The first category of systems includes a simple network card (NIC) that is virtualized by a software intermediary, either the hypervisor (e.g. KVM [26], VMware ESX server [10]) or a driver domain (e.g. Xen [13]). Today, this category of systems is most commonly used in virtualized servers since it offers a rich set of features, including security, isolation, and mobility. There are several software virtual switches—such as Linux bridge [1], VMware vswitch [32], Cisco Nexus 1000v [8], Open vSwitch [24], etc.—that are used in these systems. Recently, Rizzo *et al*. [30] also proposed a new virtual switching solution based on their netmap API. They use memory-mapped buffers to avoid data copies inside the host. It will be interesting to see if the netmap API can be exported all the way to the VMs. Unlike Hyper-Switch, all these existing systems implement the entire virtual switch within a single software domain—either the hypervisor or the driver domain. But we believe that the optimizations proposed in this paper are applicable to many of these solutions. Further, in this paper, the Hyper-Switch's performance was only compared to the performance under Xen and KVM. A recent report from VMware has shown an impressive performance of 27 Gbps between two VMs running on their vSphere architecture [33]. Unfortunately, it is hard to compare this to Hyper-Switch's performance since the hardware platforms used in the evaluations are vastly different.

The second category of systems employ more sophisticated NICs (direct-access NICs) with multiple *contexts* that present a vNIC interface directly to each VM [20, 27, 34]. Today, there exists an industry-wide standard called SR-IOV, which has been adopted by several network interface vendors to implement this solution [6, 16, 22]. These NICs also implement a virtual switch internally within the hardware. However, today most of them only implement a rudimentary form of switch. The sNICh [28] architecture explores the idea of switch/server integration. It implements a full-fledged switch while enabling a low cost NIC solution, by exploiting its tight integration with the server internals. This makes sNICh more valuable than simply a combination of a network interface and a datacenter switch. Luo *et al*. [19] propose offloading Open vSwitch's in-kernel data path to programmable NICs. Similarly, one can also imagine offloading Hyper-Switch's data plane to the NIC hardware. These solutions can enable high-performance since the VMs directly communicate with the NIC. But, in general, they lack the flexibility that pure software solutions offer.

The third category of switches attempt to leverage the functionality that already exist in today's datacenter switches. This approach uses an external switch for switching *all* network packets [9, 23]. But, fundamentally, this approach results in a waste of network bandwidth since even packets from inter-VM traffic must travel all the way to the external switch and back again.

There have also been proposals to distribute virtual networking across all endpoints within a data center [7, 11]. Here the software-based components reside on all servers that collaborate with each other and implement network virtualization and access control for VMs, while network switches are completely unaware of the individual VMs on the end-points. All these architectures are aimed at solving the network management problem, which is not the focus of this paper. But the Hyper-Switch can easily be a part of these solutions.

7 Conclusions

This paper presented the Hyper-Switch architecture that combines the best of the existing last hop virtual switching architectures. It hosts the device drivers in a driver domain to isolate any faults and the last hop virtual switch in the hypervisor to perform efficient packet switching. In particular, the hypervisor implements just the fast, efficient data plane of a flow-based software switch. The driver domain is needed only for handling external network traffic.

Further, this paper also presented several carefully designed optimizations that enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. The optimizations take advantage of the Hyper-Switch data plane's integration within the hypervisor. As a result, the Hyper-Switch enables much improved and scalable network performance, while maintaining the robustness and fault tolerance that derive from the use of driver domains. Moreover, these optimizations should be a part of any virtual switching solution that aims to deliver high performance.

This paper also presented an evaluation of the Hyper-Switch architecture using a prototype implemented in the Xen platform. The evaluation showed that, for inter-VM network communication, the Hyper-Switch achieved higher performance and exhibited better scalability than both Xen's default network I/O architecture and KVM's vhost-net architecture. Further, the external network performance under Hyper-Switch was comparable and in some cases even better than the performance under Xen and KVM.

References

[1] Linux Ethernet bridge. `http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge`.

[2] Netperf: A network performance benchmark. `http://www.netperf.org`, 1995. Revision 2.5.

[3] AMD CORPORATION. Shared level-1 instruction-cache performance on AMD family 15h CPUs.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (October 2003), pp. 164–177.

[5] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[6] BROADCOM CORPORATION. BCM57712 product brief. `http://www.broadcom.com/collateral/pb/57712-PB00-R.pdf`, January 2010.

[7] CABUK, S., DALTON, C. I., RAMASAMY, H., AND SCHUNTER, M. Towards automated provisioning of secure virtualized networks. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security* (October 2007), pp. 235–245.

[8] CISCO SYSTEMS, INC. Cisco Nexus 1000V series switches. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/data_sheet_c78-492971.pdf`, August 2011.

[9] CONGDON, P. Virtual Ethernet port aggregator. `http://www.ieee802.org/1/files/public/docs2008/new-congdon-vepa-1108-v01.pdf`, November 2008.

[10] DEVINE, S., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent #6,397,242* (October 1998).

[11] EDWARDS, A., FISCHER, A., AND LAIN, A. Diverter: A new approach to networking within virtualized infrastructures. In *WREN '09: Proceedings of the ACM SIGCOMM Workshop: Research on Enterprise Networking* (August 2009).

[12] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALĪNGAM, K., AND BURGER, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Compute r architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

[13] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. Safe hardware access with the Xen virtual machine monitor. In *OASIS '04: Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT Infrastructure* (October 2004).

[14] GREENBERG, A., HAMILTON, J., MALTZ, D. A., AND PATEL, P. The cost of a cloud: Research problems in data center networ ks. *SIGCOMM Computer Communcation Review 39*, 1 (2009), 68–73.

[15] INTEL. `http://goo.gl/5lpY8`, 2010. "Intel 1000 Core Chip".

[16] INTEL CORPORATION. Intel 82599 10 GbE controller datasheet. `http://download.intel.com/design/network/datashts/82599_datasheet.pdf`, October 2011. Revision 2.72.

[17] KUMAR, S., RAJ, H., SCHWAN, K., AND GANEV, I. Re-architecting VMMs for multicore systems: The sidecore approach. In *WIOSCA '07: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture* (June 2007).

[18] LANDAU, A., BEN-YEHUDA, M., AND GORDON, A. SplitX: Split guest/hypervisor execution on multi-core. In *WIOV '11: Proceedings of the 4th Workshop on I/O Virtualization* (May 2011).

[19] LUO, Y., MURRAY, E., AND FICARRA, T. Accelerated virtual switching with programmable NICs for scalable data center networking. In *VISA '10: Proceedings of the 2nd ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures* (September 2010).

[20] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In *Proceedings of the Euro-Par Workshop on Parallel Processing* (August 2007), pp. 224–233.

[21] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (June 2005), pp. 13–23.

[22] PCI-SIG. Single Root I/O Virtualization. `http://www.pcisig.com/specifications/iov/single_root`.

[23] PELISSIER, J. VNTag 101. `http://www.ieee802.org/1/files/public/docs2009/new-pelissier-vntag-seminar-0508.pdf`, 2009.

[24] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending networking into the virtualization layer. In *HotNets-VIII: Proceedings of the Workshop on Hot Topics in Networks* (October 2009).

[25] PORTERFIELD, A., FOWLER, R., MANDAL, A., AND LIM, M. Y. Empirical evaluation of multi-core memory concurrency. Tech. rep., RENCI, January 2009. `www.renci.org/publications/techreports/TR-09-01.pdf`.

[26] QUMRANET. KVM: Kernel-based virtualization driver. `http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf`.

[27] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the IEEE International Symposium on High Performance Distributed Computing* (June 2007).

[28] RAM, K. K., MUDIGONDA, J., COX, A. L., RIXNER, S., RANGANATHAN, P., AND SANTOS, J. R. sNICh: Efficient last hop networking in the data center. In *ANCS '10: Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (October 2010), pp. 1–12.

[29] RAM, K. K., SANTOS, J. R., TURNER, Y., COX, A. L., AND RIXNER, S. Achieving 10 Gb/s using safe and transparent network interface virtualization. In *VEE '09: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (March 2009), pp. 61–70.

[30] RIZZO, L., AND LETTIERI, G. VALE, a switched ethernet for virtual machines. In *CoNEXT '12: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (Decemeber 2012), pp. 61–72.

[31] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *ATC '08: Proceedings of the USENIX Annual Technical Conference* (June 2008), pp. 29–42.

[32] VMWARE, INC. VMware virtual networking concepts. `http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf`, 2007.

[33] VMWARE, INC. VMware vSphere 4.1 networking performance. `http://www.vmware.com/files/pdf/techpaper/Performance-Networking-vSphere4-1-WP.pdf`, April 2011.

[34] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture* (February 2007).

MiG: Efficient Migration of Desktop VMs using Semantic Compression

Anshul Rai[†], Ramachandran Ramjee[†], Ashok Anand[‡*],
Venkata N. Padmanabhan[†], and George Varghese[§]
[†]Microsoft Research India [‡]Bell Labs India [§]Microsoft Research US

ABSTRACT

We consider the problem of efficiently migrating desktop virtual machines. The key challenge is to migrate the desktop VM quickly and in a bandwidth-efficient manner. The idea of replaying computation to reconstruct state seems appealing. However, our detailed analysis shows that the match between the source memory and the memory reconstructed via replay at the destination is poor, even at the sub-page level; the ability to reconstruct memory state is stymied because modern OSes use address space layout randomization (ASLR) to improve security, and page prefetching to improve performance.

Despite these challenges, we show that desktop VM memory state can be efficiently compressed for transfer without relying on replay, using a suite of *semantic* techniques – collectively dubbed as MiG – that are tailored to the type of each memory page. Our evaluation on Windows and Linux desktop VMs shows that MiG is able to compress the VM state effectively, requiring on average 51-65% fewer bytes to be transferred during migration compared to standard compression, and halving the migration time in a typical setting.

1. INTRODUCTION

Efficient migration of desktop virtual machines (VM) is important in a variety of scenarios. First, consider the vision of a desktop PC environment that is always available and local to the user [15, 16, 25, 27]. In these systems, the user's desktop environment is encapsulated in a VM, so that it can be moved flexibly between, say, the user's office workstation, home PC, and laptop, providing a *seamless computing experience, without sacrificing interactive responsiveness of local execution*. Second, consider the desktop as a service model where desktop VMs execute in the cloud and are accessible from any local device. A key requirement in this scenario is ensuring low response times [24]. This necessitates migrating the VM over WAN links so that the VM executes in a data center that is always close to the user. Finally, desktop VM migration has also been utilized for saving energy [20]. In these systems, when the user is not engaged in computing, the VM is migrated to a server in the cloud so that the local machine can go to sleep and save energy.

A key challenge common to the above scenarios is efficient migration of VMs, both in terms of migration time and the amount of data transferred, especially over links of modest bandwidth. For instance, transferring a 4 GB VM over a 10 Mbps connection would take nearly an hour, which can be frustrating for a user who wants to transfer the VM from workplace or cloud to her home for better interactivity. Further, many ISPs worldwide offer tiered service plans with bandwidth caps ranging from 1GB to 250GB per month, with higher cost for higher limits [4]; apart from *transfer time*, a home user would also care equally about the *amount of bytes* transferred.

In this paper, we consider the problem of efficiently migrating desktop VMs. We start by revisiting the idea of replaying input to speed up migration [28] and show its limitations in practice. We then present *MiG*, which categorizes memory pages based on type (e.g., free page, code (i.e., image) page, heap page, etc.) and then employs a page-type-specific technique to perform effective compression. This paper only considers migration of memory state; while migration of disk state could be important in certain settings, measurements presented in prior work show that the amount of dirty disk state to be migrated is an order or magnitude smaller than the VM's memory size (e.g., [20] reports dirtying of disk blocks at an uncompressed rate of 40-100 MB per hour).

Input replay has been proposed as a technique to speed up desktop VM migration [28], by trading computation for byte savings. By replaying user input (e.g., keyboard/mouse events), the "same" computation is performed on the destination machine. The hope is to recreate much of the source's memory state on the destination, thereby reducing the state to be transferred during VM migration. Our study reveals that mechanisms employed by modern OSes pose many practical difficulties in benefiting from input replay.

First, for improving interactive performance, modern desktop OSes prefetch pages into memory based on user actions, application behavior, etc. (e.g., SuperFetch [9] in Windows and preload in Linux). Thus, a long running workload might result in a certain set of pages prefetched into memory while the same workload, replayed in an accelerated fashion for fast migration, might result in a different set of prefetched pages at the replayed VM.

*The author was an intern at MSR India during part of this work.

Second, for security reasons, modern OSes (e.g., Windows Vista/7 and recent versions of Linux) employ Address Space Layout Randomization (ASLR), wherein the layout of code segments is randomized, which in turn impacts the values of the embedded pointers. Therefore, the "same" pages, or even sub-pages, at the source and the destination will not match with input replay.

Third, managed code runtimes (e.g., the .NET Common Language Runtime) actively manage memory using mechanisms such as garbage collection. The invocation of these mechanisms during replay on the destination machine will typically *not* match that on the source machine, resulting in poor matches for heap pages. We find that even matching at the level of heap allocation units yields little benefit.

Our first contribution in this paper is an evaluation of the impact of the above mechanisms through extensive measurements of VMs running multiple flavours of Windows and Linux, spanning the evolution in the prevalence of ASLR and page prefetching. Our findings go beyond a recent study of memory similarity in VMs [13] by showing that even identical VMs with identical input can have dramatic differences in memory, even at the sub-page level. For example, while zero pages account for 72% of the pages in a Windows XP VM that is left running for several hours, the corresponding figure for the newer Windows 7 OS is only 4%, because of SuperFetch implemented by the latter. Likewise, the fraction of non-zero pages that match across two freshly booted VMs goes from 66% in the case of Windows XP to 33% with Windows 7, on account of ASLR. We also see a corresponding, though less pronounced, trend with Linux.

Despite the above findings, we show that we do not have to turn off prefetching and ASLR (which could have undesirable performance and security implications) for efficient VM migration. *We present MiG, our second contribution, which leverages observations from our measurement study to tailor compression to the semantics of memory pages, thereby obtaining significant gains in the context of VM migration.* The page-semantics-dependent techniques including identifying and suppressing free pages, eliminating significant intra-VM redundancy in heap pages and compressing image and SuperFetch pages using a novel approach that uses file system data as a primer dictionary for a dictionary-based redundancy elimination [12]. Our experiments bear out the effectiveness of MiG, which yields average byte savings of 51% and 65% over a `gzip`-compressed VM image, for Windows and Linux desktop VMs, respectively. These byte savings translate into a significant speedup in migration time; e.g., for a 2GB Windows 7 VM being migrated over a 10Mbps link, MiG halves the migration time (including computing overhead) to 275s from 558s with `gzip`-only compression.

Our third contribution is a reality check on the gains achievable through replay. To this end, we develop MiG-Replay, which uses MiG as the starting point but additionally exploits full page and heap matches with respect to the memory state of a replayed VM. We find that MiG-Replay can provide 15% gains over MiG but only in specific cases where either the workload is short or the pace of replay is identical to the original; for long workloads and where replay is accelerated in time to be practical, MiG-Replay even underperforms MiG, because of ASLR and SuperFetch.

Type	WinXP	Win7	Debian 6d	Debian 6
Blank VM	85%	66%	89%	80%
Short workload	72%	47%	68%	56%
Long Workload	72%	4%	63%	55%

Table 1: **Zero pages in Win XP, Win 7, Debian 6 with preload/ASLR disabled (Debian 6d) and Debian 6**

Type	WinXP	Win7	Debian 6d	Debian 6
Blank VM	66%	33%	76%	61%
Short+Paced	42%	34%	66%	48%
Long+Accelerated	41%	14%	62%	43%

Table 2: **Identical non-zero pages in OSes with replay**

2. MEMORY SIMILARITY

A high degree of full and partial page similarity were reported [23] in Windows XP and older Linux VMs (Debian 3.1/Slackware 10.2). A recent study [13] of memory similarity among VMs shows that page similarity has reduced to 15%. However, these studies [13, 23] were in the context of a server hosting disparate VMs. In this section, we characterize memory similarity between *two identical VMs provided with identical input*. In particular, we seek to answer the following questions:

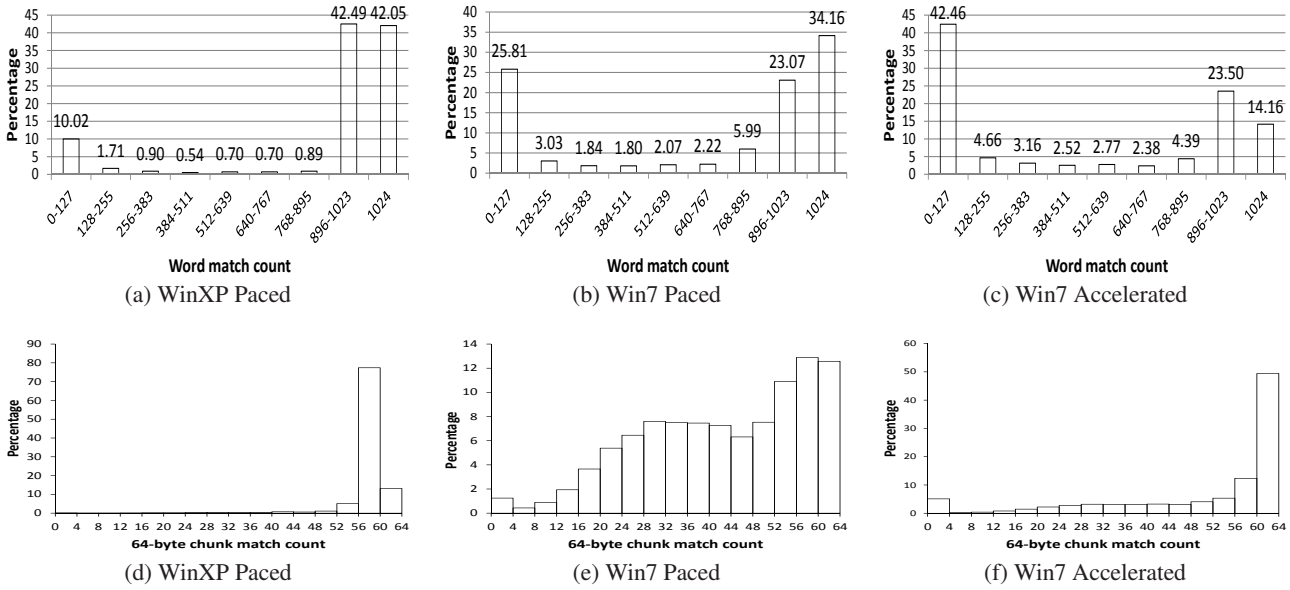
- How similar are two VMs at the full page level? At the sub-page level?
- How effective are existing techniques, like `rsync` [29], in leveraging inter-VM similarity?
- How do these similarities vary for different page types (e.g., Heap, Image, etc.)?
- How much redundancy exists intra-VM? How effective are existing compression techniques (`gzip`, `bzip2`, `7zip`) on intra-VM redundancy?

Understanding these issues is crucial for designing an efficient migration scheme. We now briefly describe the workload and the replay techniques used in our experiments before presenting the results of our analysis.

2.1 VM Workloads and Input Replay

Workload. Our workload consists of VMs running various versions of Windows and Linux for three cases: i) Freshly booted blank VM, ii) short workload of running applications for 30 minutes and iii) long workload of running applications over several hours (workload is typical desktop office applications, detailed in Section 5).

Replay. Replay on VMs can be accomplished in a number of ways. Instruction-level replay with strict adherence to timing as in the ReVirt system [21] will ensure that the destination VM is identical to the source VM in all respects. However, accomplishing instruction-level replay on a multiprocessor system has large overheads [22]. In this paper, we



(a) WinXP Paced (b) Win7 Paced (c) Win7 Accelerated

(d) WinXP Paced (e) Win7 Paced (f) Win7 Accelerated

Figure 1: **Distribution of sub-page matches**

consider input replay, which involves simply replaying the user inputs to the system (e.g., keyboard and mouse events), detailed in Section 5). While input replay cannot guarantee that the destination VM memory is identical to the source VM due to non-determinism and network interactions, the hope is to recreate similar memory so that creating an identical version is efficient. In this section, we evaluate the *similarity* of VMs that are created using this input replay mechanism.

We consider 3 scenarios: i) *Blank VM:* two freshly booted VMs, ii) *short workload, paced replay:* two VMs with identical apps executing for 30 minutes with paced input replay (same keyboard and mouse events paced identically at both VMs), iii) *long workload, accelerated replay:* two VMs with identical apps/replay but one is a long running VM where input is spread over a period of several hours representing typical usage, while, in the other VM the input is accelerated in time (e.g., ten minutes), representing a practical scenario of using replay for fast migration.

2.2 Page-level Similarity

We start with Table 1 that lists the percentage of zero pages in VMs running various OSes that have each been allocated 2 GB of memory. We see that the fraction of zero pages in Windows XP starts at 85% and reduces to 72%; *in the case of Windows 7, the fraction of zero pages starts at 66% but goes down to 4% for the long workload case*.

The dramatic reduction in zero pages in Windows 7 is due to a new feature that was first introduced in Windows Vista called SuperFetch [9]. SuperFetch is a user-customized prefetching technique that tracks application usage and selectively preloads applications or data into memory in order to improve interactive responsiveness. Linux has a similar feature called *preload* available in Debian 6 that preloads pages to improve performance. While it does not appear to be as aggressive as SuperFetch, it also reduces the number of zero pages. For a blank VM, Debian with preload disabled had 89% zero pages which reduces to 80% with preload enabled, and for a long workload, the corresponding numbers are 63% and 55%, respectively.

Next, in Table 2, we consider the number of identical non-zero pages when replay is used. Consider the case of freshly booted Windows XP and Windows 7 VMs. While 66% of the non-zero pages are identical in two XP VMs, only 33% are identical in Windows 7. Next consider paced replay which represents an ideal scenario for recreating similar memory; the percentage of identical non-zero pages in Windows XP reduces to 42%, while, for Windows 7 it is 34%.

However, in the more practical accelerated replay scenario, we notice an interesting divergence. While the numbers for Windows XP do not change significantly, we notice *a drastic reduction in the percentage of identical non-zero pages in Windows 7 to 14%*. This reduction is due to a combination of SuperFetch (acceleration of input has significant impact on SuperFetch's pre-fetching) and ASLR, that we will discuss in the next sub-section.

In the case of Linux, we verified that Debian 3.1 used in Difference Engine [23] does not have ASLR while Debian 6 has a weaker form of ASLR with less randomization, resulting in higher non-zero page matches than Windows 7. For the long workload, accelerated replay scenario, Debian 6 with ASLR and preload disabled had 62% non-zero identical page matches which reduced to 43% when ASLR and preload was enabled.

Two observations follow from these results:

- **O1:** Fraction of zero pages is dramatically reduced to 4% in Windows 7 and significantly reduced to 55% in Debian 6 compared to 70+% in Windows XP due to prefetching.

- **O2:** Fraction of identical non-zero pages between two Windows 7 VMs running identical applications with identical input ranges between 14-33% compared to 41-66% for Windows XP and 43-61% for Debian 6.

2.3 Sub-page-level Similarity

We now investigate partial-page similarity between two 2GB identical VMs provided with same input. A brute-force way to identify a page in the second VM most similar to a page in the source VM would entail 2GB*2GB or 10^{18} comparisons! Instead, we adapt Min-wise hashing [14] and compute hashes of each 4-byte word using 16 hash functions. For each hash function, we store the minimum hash value of all words in the page as a 16-tuple that succinctly represents the page. For each page in the source VM, we find the page in the second VM with the largest number of matching hash values in the 16-tuple. Tuple similarity is an unbiased estimator [14] of page similarity, a fact we verified by brute-force calculation on a small sample of source VM. After finding the most similar page in the destination VM, we compute a similarity measure as the number of corresponding words that match between the two similar pages

The distribution of word match count between non-zero pages of two identical Windows XP and Windows 7 VMs are shown in Figures 1a and 1b, respectively. Note that a page is 4KB in size and thus there are 1024 4-byte words in a page. Consider Windows XP with paced replay (accelerated replay is similar). We find that, in 42% of pages, the word-level match count is 1024, i.e., these are identical pages. We also find that, for another 42.5% of the pages, there is a very high degree of similarity (896-1023 word matches). Now consider Windows 7 with paced replay. While 34% of pages are identical, only 23% of pages have a high degree of similarity.

The presence of highly similar pages is not sufficient for reducing the amount of bytes transferred; the word differences in these similar pages must also be clustered to have long sequences of continuous word matches, which can be efficiently removed. To examine this issue, we segment each 4 KB page into sixty four 64-byte chunks and study the distribution of differences between the highly similar pages. Figure 1d shows that the differences are indeed clustered in the case of Windows XP (56 or more out of 64 chunks match in over 90% of the cases) while Figures 1e and 1f shows that the differences are spread throughout the page in the case of Windows 7, resulting in far fewer chunk-level matches.

The difference between Windows XP and Windows 7 is due to Address Space Layout Randomization (ASLR) [2], a security feature where the start addresses of executables, the heap, etc. are placed at random locations to make it difficult for an attacker to guess. The randomization, performed at the granularity of 64 KB chunks, can result in pointer references in code/heap pages being different in executions in two VMs. This results in differences between similar Windows 7 pages being spread throughout the page. For Linux, a minimal version of ASLR was enabled only in 2.6.12 while both Linux versions studied in [23] used older kernels. Thus, while the authors in [23] found a high degree of partial page matches ($>$ 2KB) across VMs, our findings corroborate the diminished page sharing found in [13].

Finally, Figure 1c shows the distribution of word match count for the Windows 7 VM with accelerated replay. The fraction of pages that have very low match (0-127) has increased to 42.5% for the Windows 7 VM with accelerated replay, up from 26% in the case of Windows 7 VM with paced replay and just 10% in the case of Windows XP. SuperFetch is the primary reason for this increase in the prevalence of low matches (as elaborated further in Section 2.5), since SuperFetch customized to the first VM is unlikely to make matching decisions regarding prefetching in the second VM, where the input replay is accelerated in time.

Summarizing sub-page-level similarity results:

- **O3:** Even among pages that are highly similar (896-1023 word matches), the locations of differences in the page are *not* clustered in Windows 7 due to ASLR. Thus, partial page sharing opportunities, as identified in [23], are significantly diminished.
- **O4:** Fraction of pages with little match is significant (42%) in Windows 7, primarily, due to SuperFetch.

2.4 Chunk-level Matches using rsync

While pages are a natural way of segmenting physical memory, an alternative is finer-grained chunk-level matching between two VMs. In this section, we consider synchronizing two VM memory dumps using `rsync`[29], a file synchronization application that leverages a similar, remote version of the file for compression. `rsync` computes sliding window chunk hashes over the remote version (replayed VM in our case) and uses these hashes to identify and compress identical chunks in the local version (current VM) for efficient migration.

We perform a parameter sweep, in steps of 32 bytes, to determine the optimal chunk-size for `rsync` that maximizes compression for the VM dumps. Using this optimal chunk size (128 bytes), `rsync` yields compression savings of 69.7% and 40.4%, respectively, for the Windows 7 with paced and accelerated replay.. These savings correspond roughly to the sum of the last three bars in Figures 1b and 1c, respectively. Applying `gzip` in addition to `rsync` yields a total savings of 72.5% with accelerated replay.

In the context of VM migration, the on-the-wire traffic goes from 100-66.5 = 33.5% with gzip compression alone to 100-72.5 = 27.5% of the VM size with rsync (plus gzip). Thus, `rsync`, which relies on replay, provides only a modest 18% relative byte savings over gzip. Furthermore, it takes 840s, 10X slower than gzip.

- **O5:** Applying a fine-grained chunk-matching technique like `rsync` on two VMs with identical applications and identical replay, only yields about an 18% reduction over conventional gzip compression.

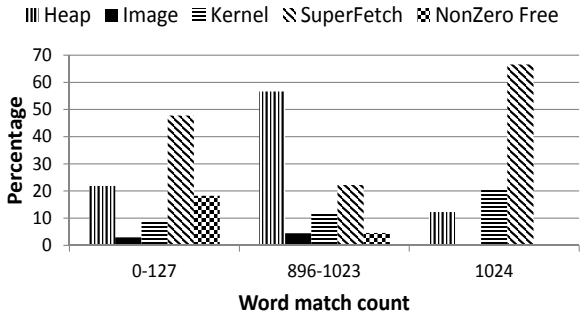


Figure 2: **Page Type Distribution**

2.5 Semantic Analysis

To gain a deeper understanding, we now parse the page similarity results by page type. We consider the accelerated replay case. We classify the pages into five categories: heap, image (i.e., code), kernel, SuperFetch, and free. Figure 2 shows the relative distribution of different page types, corresponding to a few cases in Figure 1c, namely, pages with very low matches (0-127), highly similar pages (896-1023), and identical pages (1024).

For pages with very low match (0-127), SuperFetch pages are dominant, (50%); this is caused due to the difference in the prefetching decisions in the long-running VM and the replayed VM where input is accelerated in time. In contrast, for pages with high similarity (896-1023), heap is dominant (over 50%) while SuperFetch is second (over 20%). ASLR converts what might have been identical pages in Windows XP into pages that are highly similar in Windows 7. Finally, for identical page matches, SuperFetch constitutes over 60%, followed by kernel pages at 20%.

Full VM	Heap	Image	Kernel	SFetch	Free
66.5%	80.0%	69.2%	67.6%	47.3%	81.4%

Table 3: **Savings by page type using gzip**

Intra-VM redundancy: We applied `gzip` on the entire VM and also on different collection of pages collated by their type (Table 3). While the entire VM can be reduced by 66.5% using `gzip`, we see that compression savings vary significantly across page types. Heap and free pages can be reduced by 80% (due to a predominance of zero bytes; e.g. 66% of bytes in heap pages were zeros compared to 45% for the entire VM), while SuperFetch pages can be reduced by only 47%.

gzip vs bzip2 vs 7zip: We also examined other well-known compression utilities such as `bzip2` and `7zip` that have been shown to be better than gzip in other contexts [18]. However, these utilities were all *significantly* slower than `gzip`, significantly inflating overall VM migration time, another metric of interest in our setting. For example, on a 2.2 GHz CPU core, `gzip` takes 65s when optimized for speed (with compression savings of 66.5%) and 117s when optimized for compression (savings increases to 68.5%). In contrast, using default settings, `bzip2` [5] takes 390s to reduce the VM by 68.4% and `7zip` [1] takes 810s to reduce the VM by 77.5%.

Summarizing, semantic analysis of memory pages helps inform our design of efficient migration:

- **O6:** Free pages can be compressed by almost 100% since these need not be transferred.
- **O7:** Heap pages are highly compressible using gzip.
- **O8:** SuperFetch pages constitute a significant fraction of low-match pages and are not highly compressible using gzip. Hence, we need an efficient technique for transferring these pages. Many SuperFetch pages are also image pages; a technique that works for these SuperFetch pages will also work for image pages.
- **O9:** Full page matches can benefit kernel and SuperFetch pages.

3. MIG DESIGN

We now present the design of MiG, our solution for efficient migration of desktop VMs, which does *not* resort to input replay. As a point of comparison, we also design MiG-Replay, which leverages a replayed VM's memory state where appropriate.

Design Constraints: In order to ensure correct operation post migration, we need to ensure that source memory is replicated fully and identically at destination. Even the goal of input replay is only to create similar memory at the destination so that creating an identical version is efficient, since as mentioned earlier, due to non-determinism and network interactions, input replay cannot guarantee a semantically identical VM.

The only exception to the above is that Free pages need not be identical since the OS does not rely on its contents. Note that even SuperFetch pages have to be identical at the two ends. This is because the SuperFetch service may "activate" these pages at any time based on its internal representation (e.g, SuperFetch page 1 is image page for process x), without the knowledge of the hypervisor.

Another constraint we impose is that the design should not require changes to the guest OS. For example, requiring that the guest OS implement a mechanism to get/set the ASLR random seed via the hypervisor is out of scope. This is to ensure that the migration solution will work for existing versions of guest OSes that are deployed today. Note that this constraint does not preclude the design from using any publicly documented information of the guest OSes for its operation, since this does not affect its deployability.

3.1 Overview

At a high level, MiG and MiG-Replay operate as follows. When a desktop machine is to be migrated from a source machine *S* to a destination machine *D*, we examine each memory page on *S* and apply techniques tailored to the type of the page. MiG relies only on local state at *S*, including disk state that has been synced previously, MiG-Replay, in addition, also leverages the memory state of the VM at *D* that has been constructed via input replay. The set of techniques applied derives from the observations **O6** through **O9**:

- **Free/Zero pages**: In both MiG and MiG-Replay, these pages are identified on *S* and not transferred.
- **Full-page matches:** In MiG, memory image of a freshly booted VM is pre-provisioned at both *S* and *D*. Full-page matches with respect to this "blank VM" helps reduce the bytes transferred. In MiG-Replay, full-page matches are computed against the replayed VM at *D* instead of a blank VM.
- **Image/SuperFetch pages:** For image pages, whether active or prefetched, both MiG and MiG-Replay employ a novel approach involving statically precomputing a common, primer dictionary at both *S* and *D*. This dictionary comprises the contents of commonly-accessed executable and library files, and is used as a reference for computing a diff of the memory state.
- **Heap pages:** In MiG, we employ a combination of history-based redundancy elimination [12] and gzip to identify and eliminate redundancy within heap pages. In MiG-Replay, where possible, we parse the heap to identify the chunks that match between *S* and *D*.
- **Other:** For both MiG and MiG-Replay, the remaining pages are compressed using a combination of dictionary-based redundancy elimination [12] and gzip.

Next, we discuss each technique in greater detail.

3.2 Free/Zero Pages

MiG and MiG-Replay identify free/zero pages by parsing the page allocation table on *S* (Section 4) and only convey their indices to *D*, thereby achieving nearly 100% compression for these pages. Since free pages can have non-zero content, conventional compression schemes achieve less savings on these pages (e.g., only 81% savings when gzip is applied on free pages – Table 3).

3.3 Full-page Matches

As seen in Figure 2, kernel pages constitute a good percentage of full-page matches, in large part because ASLR is typically not applied to kernel pages. Thus, MiG preprovisions the memory state of a freshly booted "blank VM" and the corresponding page hashes at both *S* and *D*. At transfer time, MiG simply computes a fast 4-byte hash [6] for each page at *S*, matches it against the hash list of the blank VM, verifies using a byte-by-byte comparison with the local copy (to neutralize hash collision risk), and sends across the index and location of the matched page to *D*, which then reads in the corresponding page from its local copy to reconstruct the memory state.

In MiG-Replay, we look for full-page matches between *S* and the replayed VM, *D*. A 4-byte hash is computed for each page at *S* and these are sent across to *D* as a list of *(page index, hash)* pairs. *D* then compares these hashes from *S* with those computed locally on its own pages. When a hash from *S* matches one at *D*, the corresponding page need not be transferred from *S* to *D*. To reduce hash collision risk in MiG-Replay, we also send a full 20-byte SHA1 hash [11] of just the matched pages.

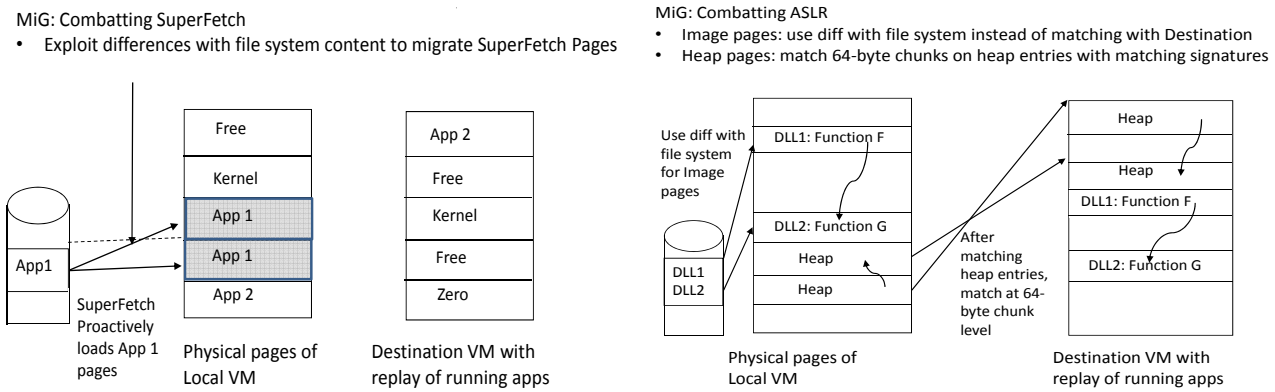
3.4 Image/SuperFetch Pages

Image pages comprise active pages that are in the address space of a process as well as SuperFetch pages that are prefetched in anticipation of future use. ASLR impacts both active and SuperFetch pages by impeding even sub-page level matching. Indeed, as reported in Section 2, even if the page were divided into 64-byte chunks, 40-75% of the pages have 8 or more chunks that do *not* match (Figures 1e and 1f). The fine-grained nature of the matches, interspersed with non-matching pointers, means that two pages would ideally need to be compared side-by-side, defeating the goal of efficient transfer.

3.4.1 Using Precomputed File System Context

For the reasons noted above, neither MiG nor MiG-Replay relies on *D* for compressing image pages. Instead, as shown in Figure 3, they employ a novel approach that builds on the observation that the image pages in memory are derived from the file system content. Indeed, the OS loader reads binaries from the file system and places these in memory, albeit with modifications because of ASLR. The content of these binary files provides context both similar to the target pages (image/SuperFetch) and also locally available at both *S* and *D*. Thus, the pre-computed context built using file system content offers the prospect of good compression savings without any overhead incurred in establishing the shared context.

To realize *pre-computed context based compression*, we prime the dictionary in an existing redundancy elimination algorithm, EndRE [12]. EndRE works as follows. Given a cache/dictionary of past packets that have been transferred from a source to a destination, EndRE identifies contiguous strings of bytes in the current packet that are also present in the cache. This is accomplished by 1) identifying a set of representative "fingerprints" for each packet 2) looking up these fingerprints in a "fingerprints store" that holds the fingerprints of all the past packets in the cache' and 3) for each fingerprint of the packet that is found in the store, the matching packet is retrieved and the matching region is expanded byte-by-byte in both directions to obtain the maximal region of redundant bytes. Once all matches are identified, the matched segments are replaced with fixed-size pointers into the cache, thereby suppressing redundancy. In the original EndRE, the cache starts empty and is dynamically built up as packets are transferred between source and destination. In the primed version, the cache at both ends is primed with 2 GB worth of file system content, comprising commonly-used binary and library files. The priming is done by passing these files to EndRE which builds its internal data structures for identifying redundancy. Subsequently, when the SuperFetch/image pages are passed to EndRE, contiguous byte strings that are redundant with the bytes in the primed context are identified and replaced with pointers, which are then restored at the destination using its primed cache.



(a) SuperFetch (shaded pages) impacts the ability of replay to recreate matching pages at the destination VM; MiG/MiG-Replay exploit differences with (local) file system to migrate these pages

(b) Due to ASLR, function and data pointers in image and heap pages may not match across two VMs; MiG relies on intra-VM redundancy for heap while MiG-Replay matches 64-byte heap chunks across VMs

Figure 3: **Illustration depicting how MiG and MiG-Replay combat SuperFetch and ASLR**

While we have not performed any optimization or customization of the context, we believe user-specific personalization could yield both reduction in context size as well as potentially higher compression savings.

3.5 Heap Pages

Now we turn to heap pages. Since MiG does not rely on replay, it exploits intra-VM redundancy for heap pages. We already saw that gzip was able to compress heap pages by 80% due to predominance of zero-bytes. Further, examining the heap pages of a VM instance, we found that out of the total heap of 670MB, about 170MB (non-zero bytes) was redundant. Even though over 91% of this redundancy was from within the heaps belonging to the same process, the vast majority of the redundancy was between byte strings located in different memory pages. So compression techniques such as gzip, which look for redundancy over a small window (64KB), will often not be able to identify such redundancy across disparate locations. Hence, to compress heap pages, MiG uses an intra-VM redundancy technique based on EndRE [12], that identifies redundancy over a large history (e.g., 2GB), coupled with gzip.

MiG-Replay, on the other hand, has the advantage of access to the replayed VM to eke out additional gains over MiG. Conceptually, replay should create a heap at D similar or identical to the one at S. However, in practice there is a distinction between heap *content* and heap *structure*. Replay could, in fact, help make the heap content similar. Yet, the structure of the heap could be very different across the two ends because of garbage collection and compaction, which kick in asynchronously.

Therefore, to match the process heaps across S and D, MiG-Replay looks deeper. Heaps in Windows 7 come in two forms: *managed heap*, whose structure can be parsed, and *unmanaged heap* which are private to a process. For managed heap, MiG-Replay performs a heap walk on the heap of each process at S and D, to produce the list of heap entries at each end. For each heap entry at S, MiG-Replay computes a hash of its used portion (parts of the heap entry might be unused), and sends it across to D. D looks through its heap entries for hash matches. If a matching heap entry is found, the corresponding content need not be transferred from S to D.

As shown in Figure 3b, ASLR can again result in differing pointers inside the heap entries of S and D, that make an exact match of the full heap entry less likely. To mitigate this effect of ASLR, we chunk the heap entry into n-byte blocks and compute 4-byte hashes to identify potential matches. Based on our evaluation (Section 5), we find that blocks of size $n = 64$ bytes provided us with the highest compression savings. Finally, for unmanaged heaps, since the heap structure cannot be parsed, MiG-Replay simply chunks them into 64-byte chunks at S and looks for chunk matches at D on corresponding heap pages that belong to the same process.

To keep the processing overhead manageable, we only apply the above procedure for heap entries that are larger than 1KB in allocated size. Our measurements show that heap entries that qualify as being "large" per the above criterion account for an overwhelming 80+% of the bytes in the heap.

We stress again that all of the above complexity associated with replay and parsing the heap is only for the case of MiG-Replay, which we designed solely for the purpose of comparison with the much simpler MiG scheme.

3.6 Other Pages

The pages that remain include stack pages, and also kernel pages that did not benefit from a full-page match. For such pages, both with MiG and MiG-Replay, we employ intra-VM redundancy elimination using EndRE [12].

4. IMPLEMENTATION

We now briefly discuss the implementation of MiG on Windows. MiG runs in the root partition of the Microsoft Windows Server 2008 Hyper-V system [8]. While our current prototype is targeted towards the quick migration feature of Hyper-V (suspend-migrate-resume), wherein the VM

state is saved, moved, and restored at the destination, we believe MiG can be effectively applied to the VM live migration [17] scenario as well.

To initiate migration, we use existing Hyper-V mechanisms to save the state of the VM, which yields a VM physical memory file and two small (few MB) configuration files containing the VM configuration information and the saved state of devices. MiG reads in the saved physical memory and extracts semantic information in two steps. First, it consults a system-wide data structure, the Page Frame Number (PFN) database, which has metadata information for each page (zero or free, allocated to a process or kernel, etc.), and allows the reverse mapping of a physical memory address to the virtual address of a process, where applicable. Second, MiG consults the Virtual Address Descriptor (VAD) tree process-specific data structure to determine the type of the page (`MEM_PRIVATE` for heap, `MEM_IMAGE` for code and `MEM_MAPPED` for memory mapped file pages) in the virtual address space. MiG then applies the appropriate technique from Section 3 to each of the pages and, thus, creates a compressed version of the memory file, which is migrated to the destination. Note that both PFN and VAD are publicly documented, so while MiG is intrusive in having to look into the memory state of the VM, it does not depend on access to any proprietary information.

In case of MiG-Replay, our prototype takes in two saved memory image files — one each corresponding to the original and the replayed VM — and simply performs an analysis of the compression gains of having a replayed VM. In addition to extracting the above semantic information, MiG-Replay also performs a heap walk by parsing the heap structure of each process on the two VMs, to identify the allocated heap chunks for heap compression.

For Linux, we use the libVMI tool [7] for introspection into VM memory.

5. EVALUATION

Metrics. We evaluate the performance of MiG, MiG-Replay and `rsync` primarily in terms of volume of bytes transferred relative to using gzip as the compression scheme.[1] Let gzip yield a total byte transfer requirement of b_{gzip}. For any other scheme x (e.g., MiG), let the byte requirement be b_x. The byte savings, or compression *gains*, of x over the baseline is then $g_x = \frac{(b_{gzip}-b_x)}{b_{gzip}} \times 100$. This relative savings metric captures the bytes saved compared to the scheme, namely `gzip`, that is commonly used in commercial systems such as Windows Server 2008 Hyper-V. Further, if compression processing is faster than the link speed, this relative byte savings would translate into an equivalent reduction in migration time. Thus, we also evaluate the *migration transfer time* for the various schemes. Finally, we do not present *absolute byte savings* as a separate metric since it is already captured as part of the migration transfer time metric. In general, absolute byte savings for MiG ranges between 80-95% for the various scenarios.

Workloads. We evaluate MiG performance for both Windows and Linux OSes. For Windows, we collect memory dumps from 10 desktops with real user workloads, running the *32-bit* version of Windows 7 with *2-4 GB* of RAM, 8 of which were from desktops used by researchers and 2 by admin staff. For Linux, we use *64-bit* VMs running Debian squeeze (2.6.32.5-amd64) and workload consists of a mix of document editing (openoffice word/ presentation, gedit), image manipulation (gimp, inkscape, photo manager), and web-browsing (firefox, epiphany) applications, reflecting common desktop usage.

In order to evaluate MiG-Replay, we use Windows VMs with artificially generated workloads that emulate a Windows desktop computing environment, with applications such as Outlook (email), Internet Explorer (browser), Word (document editor), Excel (spreadsheet), etc. running. We use the AutoIt scripting language [3] to automate the Windows GUI and design scripts to feed keyboard input into Word or Excel interspersed with random think-time, sync email, download pages from different websites, etc. We perform 5 runs of this emulation, with different combinations of applications used in each instance, with each experiment lasting between 30 minutes and four hours to mimic a user work session. For each of these experiments, we also performed paced and accelerated replay (same script without think-times).

5.1 MiG Byte Savings

Figures 4 and 5 show the percentage byte savings achieved by MiG relative to gzip for each of the individual Windows and Linux desktop VMs, respectively. First, we see that *MiG delivers consistent byte savings of 40%-60% for the ten Windows VMs (average 51%) and 58%-68% for the five Linux VMs (average 65%) over gzip*.

It is interesting to observe the contribution of the different MiG techniques towards achieving the overall gains. For Linux VMs, the bulk of the gains come from the use of precomputed context (30-38%), followed by Full page matches (18-25%) and intra-VM redundancy elimination (6-13%). In contrast, for Windows VMs, the majority of the gains come from Intra-VM redundancy elimination (35-44%), followed by precomputed context (3-15%), Full page matches (2-7%), and Free pages (0-7%).

The surprising finding in the above results is that while the use of precomputed context (Section 3.4) provides substantial benefits for Linux VMs (30-38%), its contribution to savings in Windows VMs is modest (3-15%). Upon examining this in more detail, we find that while precomputed context in Windows had indeed full or partial matches with over 80% of image pages and 45% of SuperFetch pages[2], many of these matches were also captured by intra-VM redundancy elimination. These intra-VM redundant matches were

[1] We do not use `rsync or 7zip` as a baseline to compare against since these are an order of magnitude slower than gzip at the settings that provide savings.

[2] The lower cache hit rate for SuperFetch pages is because SuperFetch pages can also be non-image pages.

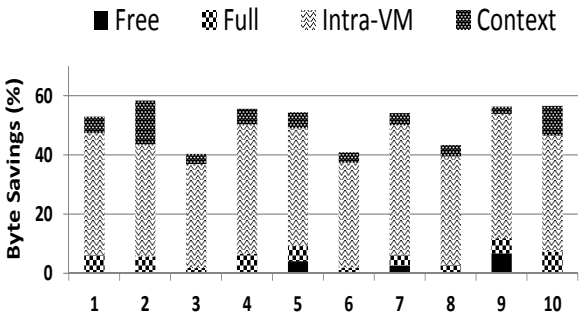


Figure 4: **MiG byte savings on Windows VMs**

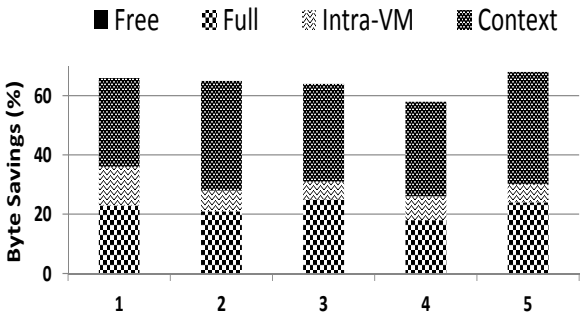


Figure 5: **MiG byte savings on Linux VMs**

between pages belonging to the same DLL that had been loaded by different processes (e.g., user32.dll was loaded by almost 50 out of the 100 processes in one Windows desktop), though, vast majority of these matches were for only small portions of a page (average match length of only 107 and 83 bytes for image and SuperFetch pages, respectively). Thus, in Windows, a substantial portion of the savings that would have accrued from using a precomputed context is already obtained by intra-VM redundancy elimination.

The other notable difference is the higher contribution of Full page matches in Linux (18-25%) versus Windows (2-7%). This is explained by the fact that Windows has much more extensive ASLR support turned on by default than Linux and agrees with the higher full page sharing numbers for Linux (Section 2).

In summary, MiG delivers significant byte savings of 51% and 65% over gzip for Windows and Linux desktop VMs, respectively. The different techniques in MiG each contribute towards achieving these savings, though, the significance of each technique's contribution varies between Windows and Linux.

5.2 Replay

Figure 6 depicts the byte savings relative to gzip for the non-semantic scheme `rsync`, and the two semantic schemes (MiG and MiG-replay) for four Windows desktop VM workloads, viz., different combinations of short/long workloads and paced/accelerated replays. In this case, short/long workloads lasted 30 mins/four hours of automated use of office applications and while paced replay took the same time as the original workload, accelerated replay took under ten minutes to complete for both workloads.

From the figure, we see that MiG provides average savings relative to gzip of 38-48% for all these workloads without relying on any replay. Using the replayed VM memory, we find that `rsync` provides about 18-34% relative savings while MiG-Replay provides 39-63% relative savings. Note that `rsync` gains over gzip are modest when the replay is accelerated, indicating that the memory image created with accelerated replay is not as close to the source image as in paced replay.

Interestingly, MiG-Replay delivers about 15% additional savings compared to MiG in cases where either the replay is paced or the workload is short; however, when the workload is long and replay is accelerated in time (as would be necessary for fast migration), we *find that MiG-Replay surprisingly performs worse than MiG by 8%*. The reason is twofold. First, the gains due to matching of SuperFetch pages disappear because accelerated replay fails to evoke the same prefetching pattern as the actual execution of the VM. Second, the degree of similarity in the heap also diminishes, so the overhead of performing heap matching (e.g., sending hash values across from *S* to *D*) overwhelms the gains obtained from the actual matches. We examine this second reason next.

Type	Excel	Outlook	Powerpoint	OneNote
Managed pages	131	212	205	347
Unmanaged pages	893	1045	1249	860
Bytes (%) (heap size $> 1KB$)	84.5	84.9	83.8	89.8
Bytes % match	84	80.6	77.4	73.3
MiG-Replay savings %	50	44	36	42

Table 4: **Heap characteristics of some office apps**

Table 4 shows some important statistics for the heap pages of a few Office applications. The heap comprises of *managed* and *unmanaged* heap and we can see that managed heap is only 13-29% of total heap for these applications. The ability of replay to recreate the source's memory state at the destination is highlighted in the next row that lists the percentage of bytes that match heap entries between source and destination VMs. For these applications, we see that between 73 to 84% of bytes do indeed match between source and destination VM. However, since the match is not exact (because of pointers affected by ASLR), we resort to dividing the heap into chunks and perform matching at the smaller granularity of chunks rather than matching at larger full heap entries. We evaluated compression savings for the entire managed heap size using the replay mechanism for different chunk sizes (not shown) and found that *64-byte chunks provide the highest savings of 40-50%*, balancing the overhead of sending 4-byte hashes for each of the chunks and the cost of losing a chunk match due to a small difference (e.g., a pointer value change) between source and destination chunks. For unmanaged heap, we use the same chunk size to divide up the heap pages and try to identify matches at the replayed VM; again, the 4-byte hash overhead for each 64-byte chunk results in decreasing the savings. Additional protection against hash collisions will further reduce these savings.

To summarize, while replay does indeed create similar

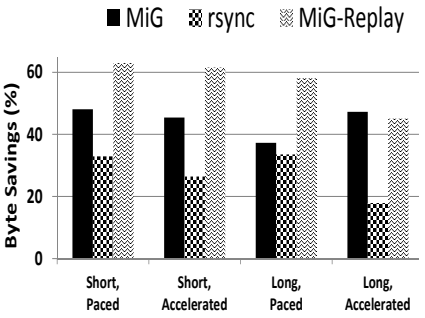


Figure 6: **Replay workloads**

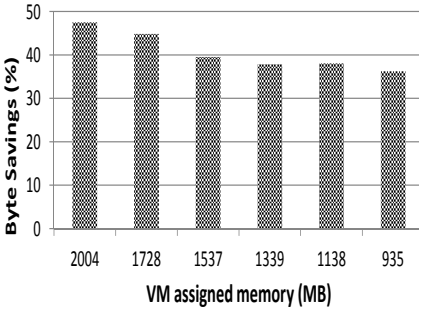


Figure 7: **Ballooning**

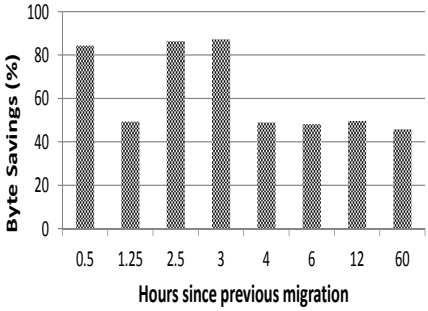


Figure 8: **Repeated migrations**

state at the destination, MiG-Replay's ability to compress heap pages effectively using replay is impeded by the reality that (a) only a small fraction (13-29%) of the heap is parseable, (b) the reconstruction of the heap using replay is far from perfect (only 73-84% of bytes match), and (c) we have to resort to matching small, 64-byte chunks to get maximum compression savings, incurring high overhead and limiting savings. Since MiG is able to perform effective intra-VM redundancy elimination for heap pages, MiG-Replay is unable to gain over MiG.

5.3 Ballooning

Ballooning is a technique to artificially introduce memory pressure in a VM, leading it to evict less important pages [30]. One application of ballooning proposed in the literature is in the context of efficient migration, wherein unnecessary pages are shed from the VM memory prior to migration [26].

In general, it is hard to estimate the amount of memory to be ballooned out; overdoing it can cause the eviction of important pages and thus adversely impact user-perceived performance. Thus, *migrating the entire memory is desirable*. However, since memory ballooning can be applied independently of MiG, in this section, we investigate the impact of memory ballooning on MiG byte savings by using Hyper-V dynamic memory feature to create different amounts of memory pressure on the VM before applying MiG.

Figure 7 shows MiG's relative savings over gzip for different amounts of assigned memory for a 2GB VM, corresponding to a memory reduction of 5-55% through ballooning. While the savings decreases as the assigned memory is reduced, MiG is still able to deliver 36% relative savings even with 55% of VM memory pages evicted. This is because while ballooning evicts low priority pages like free pages, it does not evict all SuperFetch or heap pages[3] and, thus, a substantial portion of the MiG byte savings remains.

Further, at high memory pressure, many of the evicted pages are paged out into a *pagefile (swap)* in disk which, of course, also needs to be migrated. MiG can be directly applied to the pages sitting in the pagefile just as to the pages in memory. MiG's savings on pagefile was similar to the savings achieved for in-memory pages.

[3]SuperFetch pages retain the priority of the original page.

5.4 Repeated Migrations

One of the scenarios targeted by MiG is the desktop that is always-on and is migrated repeatedly between work and home or work and cloud. In this section, we evaluate the benefit of using the memory from previously migrated state for byte savings. In these cases, both source and destination save the previously transferred VM memory and MiG uses this for its full page matches instead of a blank VM as before (all other techniques remain the same).

We had a user use a 2GB Windows 7 VM for several days; applications used included the browser and several office applications. Every once in a while, sometimes after short intervals of 30 minutes to few hours, and sometimes after long intervals of several hours to even days, we took snapshots of the VM memory, representing a checkpoint of VM state that needs to be migrated. We then used MiG with the benefit of the previous snapshot for computing byte savings.

Figure 8 depicts MiG's relative byte savings over gzip for the different migration cases corresponding to workloads that last from 30 minutes to 60 hours. From the figure, we see that short workloads of up to a couple of hours in general significantly benefit from using the previous snapshot by delivering relative savings over gzip of up to 87% (corresponding to absolute reduction of VM size by 95%). Of course, not all short workloads result in such gains (for example the 1.25 hour data point), due to windows update or other system activity that can potentially induce large change in memory, effectively reducing the effectiveness of the previous snapshot. Finally, we see that for workloads beyond a few hours, the previous snapshot is not as useful and MiG's relative gains drop down to about 50%.

5.5 Migration Time

We now evaluate the time for VM migration for the various schemes. Let us first consider the computational cost of the different MiG techniques on a 2.2 GHz CPU core for a typical 2GB Windows 7 VM.[4] In the following analysis, MiG's context cache is preloaded in memory and the VM image is also in memory.

Parsing the PFN database to extract semantic informa-

[4]Most of these numbers scale proportionately with VM size but can vary depending on the amount of compression achieved.

	Compressed Size	Compute time	Transfer time
gzip	670MB	65s	558s
MiG	330MB	67s	275s

Table 5: **Migration time for 2GB VM on 10Mbps link**

	Off	Default	Aggressive
Full	26%	23%	22%
Full+Intra-VM	32%	32%	30%
Full+Intra-VM+context	62%	69%	70%

Table 6: **Impact of preload on MiG savings (Linux)**

tion takes about 20s.[5]. MiG also creates a 4-byte hash of each page using Jenkins Hash [6] and compares these hashes against the local precomputed hashes of a blank VM for identifying full page matches. This process takes around 8s. The processing of SuperFetch and image pages using EndRE [12] with primed context takes about 15s. Finally, all remaining pages are compressed using EndRE+gzip, which takes about 24s. Thus, our MiG prototype implementation is able to reduce a 2GB VM to about 330 MB in 67s. For the same VM, gzip reduces it to 670 MB in 65s.

Migration time (Table 5) is determined by the maximum of transfer time and compression processing time. Transfer time is directly proportional to (compressed) VM size and inversely proportional to link speed. Thus, on slow links and/or for large VMs, MiG's migration is significantly faster than gzip. For example, *on a 10 Mbps link, MiG transfers the 2GB VM in about 275s, halving the transfer time of gzip (558s)*. For comparison, it takes 810s for `7zip`, 840s for `rsync` and 1665s for uncompressed transfer.

We can also take advantage of multi-core CPUs to perform many of the above operations in parallel to optimize MiG (and gzip). Using 4 cores, processing for an optimized version of MiG can easily be limited to less than 28s (and optimized gzip to less than 55s), thereby allowing *MiG to retain the 50% reduction in both bytes and migration time compared to gzip at 100 Mbps speeds*. Of course, on 1 Gbps or faster links, transferring the raw VM may be faster than using either gzip or MiG.

6. DISCUSSION

Turning off page prefetching prior to migration: Turning off page prefetching mechanisms such as SuperFetch could aid VM migration by cutting out the prefetched pages from the set that needs to be moved. However, doing so can have an adverse impact on user-perceived performance [10]. Nevertheless, it is interesting to ask how much there is to be gained from varying the amount of prefetching, in terms of the byte savings achieved by MiG.

To answer this question, we used a Linux desktop VM loaded with a few applications and tested it under three different settings for prefetching: off (preload removed), default, and aggressive (increased free memory to be used for prefetching from default of 50% to 90%). The byte savings relative to gzip is shown in Table 6. While increasing the degree of prefetching results in a reduction in the number of full-page matches relative to a blank VM, this loss is more than offset by leveraging pre-computed context to compress the prefetched (and other) pages, resulting in similar absolute byte savings for all these cases. This suggests that we do not have to turn off prefetching to obtain byte savings for migration.

Influencing randomization in ASLR: While turning off ASLR would adversely impact security, one could arguably influence ASLR's randomization policy more subtly, to make it more migration friendly while not compromising security. For instance, when a VM is migrated, the randomization seed used for ASLR at the destination could be set to be the same as that at the source, which might then make the memory state of a replayed instance match more closely with the source. However, to our knowledge, for security reasons, OSes do not make the seed available through an API or document the location of the seed in memory so that, for instance, it could be read from the hypervisor.

Nevertheless, to get a sense for the gains to be had if the same seed were used at the source and the destination, consider the evaluation presented in Section 5.4. Since a single VM instance was snapshoted repeatedly, the randomization seed remained unchanged across the snapshots. When the interval between two snapshots is short (under 2 hours), there is a high degree of match between the snapshots. However, when the interval is longer, the snapshots tend to diverge, even though the randomization seed is the same across the snapshots. The divergence is because of SuperFetch, garbage collection, etc., factors which exist independent of ASLR. Thus, any short-term gains arising from maintaining the same ASLR seed get overshadowed over longer durations by other sources of non-determinism.

7. RELATED WORK

ISR, Collective, Transient PCs. The vision of a desktop PC environment that is mobile and available anywhere was articulated by Chen and Noble [16] and in the Internet Suspend Resume (ISR) project [25]. The Collective [15] is another system that provides users with a consistent desktop environment at a computer nearby as users move. Recently, this paradigm of having the desktop environment stored in the cloud but executed on a PC close to the user has been dubbed the transient PC [27].

Efficient Migration. Prior work closest to MiG is the work on optimizing the migration of virtual computers [26]. Their system uses copy-on-write disks in order to migrate disk changes, supplanted with demand paging to fetch needed blocks, memory ballooning to zero out unused memory, and page hashing to suppress identical memory blocks. While MiG can benefit from the disk migration techniques in [26], migrating memory state is much more challenging today due to new OS features. The idea of using replay in order to migrate VMs efficiently was proposed in [28]. However, as we

[5]This is primarily due to our prototype using windbg APIs which makes disk accesses; an optimized version should take under 10s.

show in this paper, replay provides only small gains.

CloudNet [31] supports efficient Live WAN migration of VMs. It implements smart stop and copy to reduce the number of iterations/copies for Live migration which can be useful for live migration support in MiG. It also implements redundancy elimination by computing sub page-level hashes (1 KB in size) and comparing this to previously sent data. MiG's intra-VM redundancy elimination eliminates redundant chunks that are as small as 32 bytes. Remus [19] is a system that replicates VMs asynchronously. Remus uses page compression including delta and gzip compression for efficient checkpointing. Since Remus checkpoints state every 25*ms*, memory page delta-based approach works well for them. For durations comprising several hours, typical for VM migration, we find that previous memory state is not useful.

Similarity in VM memory. Looking beyond migration, the recent study by Barker et al. [13] reports that page sharing in Linux and Windows VMs running at a server is diminished because of ASLR. Our study differs from and goes beyond this prior work in several ways. First, since our goal is efficient migration, we compare two VMs running the same OSes/applications and provided with the same input, which is not a scenario considered in [13]. Second, while [13] focuses mostly on page-level sharing, we show that even at 64-byte chunk level, changes due to ASLR render sharing ineffective. Third, going beyond ASLR, we also evaluate and show the significant impact of OS prefetching (e.g., SuperFetch) on memory redundancy.

8. CONCLUSION

When we started our investigation into efficient migration of desktop VMs, we had assumed that replay and memory similarity would lead to efficiency. However, we were puzzled by the lack of similarity even in blank VMs. The culprit, as explained in this paper, is randomness and nondeterminism due to mechanisms such as ASLR and page prefetching in modern OSes. Through extensive experiments on both Windows and Linux, we have characterized and quantified the impact of these mechanisms.

Despite these hurdles, our migration solution, MiG, yields compression gains of 51% and 65% over gzip on Windows and Linux VMs, respectively, and halving of the overall migration time. Central to MiG is the idea of tailoring the compression technique to the semantics of memory pages, an approach which we believe could transcend the specific OS mechanisms and compression techniques considered here.

9. ACKNOWLEDGEMENTS

We thank our shepherd, Jason Flinn, and the anonymous reviewers for their constructive comments.

10. REFERENCES

[1] 7zip. http://www.7-zip.org.

[2] ASLR. http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx.

[3] AutoIT. http://www.autoitscript.com/site/.

[4] Bandwidth caps around the world. http://www.maximumpc.com/article/features/how_bad_do_we_really_have_it_bandwidth_caps_around_world.

[5] bzip2. http://www.bzip.org.

[6] Jenkins Hash. http://burtleburtle.net/bob/c/lookup3.c.

[7] LibVMI tool. http://code.google.com/p/vmitools/.

[8] Microsoft Hyper-V. http://www.microsoft.com/en-us/server-cloud/windows-server/hyper-vaspx.

[9] SuperFetch. http://msdn.microsoft.com/en-us/library/bb188739.aspx.

[10] SuperFetch performance. http://everythingexpress.wordpress.com/2011/11/13/how-to-adjusting-windows-7-superfetch/.

[11] US Secure Hash Algorithm 1 (SHA1). RFC 3174, September 2001.

[12] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *USENIX NSDI*, April 2010.

[13] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman. An Empirical Study of Memory Sharing in Virtual Machines. In *USENIX ATC*, June 2012.

[14] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of IEEE Compression and Complexity of Sequences*, June 1997.

[15] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A Cache-Based System Management Architecture. In *NSDI*, May 2005.

[16] P. Chen and B. D. Noble. When virtual is better than real. In *8th IEEE Workshop on Hot Topics on Operating Systems*, May 2001.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, May 2005.

[18] L. Collin. A quick benchmark: Gzip vs. Bzip2 vs. LZMA, 2005. http://tukaani.org/lzma/benchmarks.html.

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *USENIX NSDI*, April 2008.

[20] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. Shin. LiteGreen: Saving Energy in Networked Desktops using Virtualization. In *USENIX ATC*, June 2010.

[21] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[22] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[23] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines . In *OSDI*, December 2008.

[24] A. Kochut and H. Shaikh. Desktop to cloud transformation planning. In *IEEE IPDPS*, May 2009.

[25] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *IEEE WMCSA*, June 2002.

[26] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, 2002.

[27] M. Satyanarayanan, S. Smaldone, B. Gilbert, J. Harkes1, and L. Iftode. Bringing the Cloud Down to Earth: Transient PCs Everywhere. In *MobiCloud 2010, Santa Clara, CA*, October 2010.

[28] A. Surie, H. A. Lagar-Cavilla, E. de Lara, and M. Satyanarayanan. Low-Bandwidth VM Migration via Opportunistic Replay. In *HotMobile*, February 2008.

[29] A. Tridgell. Efficient Algorithms for Sorting and Synchronization, 2000. PhD thesis, Australian National University.

[30] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, December 2002.

[31] T. Wood, K. Ramakrishnan, P. Shenoy, and J. V. der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Virtual Execution Environments (VEE)*, March 2011.

Copysets: Reducing the Frequency of Data Loss in Cloud Storage

Asaf Cidon, Stephen M. Rumble, Ryan Stutsman,
Sachin Katti, John Ousterhout and Mendel Rosenblum
Stanford University
cidon@stanford.edu, {rumble,stutsman,skatti,ouster,mendel}@cs.stanford.edu

ABSTRACT

Random replication is widely used in data center storage systems to prevent data loss. However, random replication is almost guaranteed to lose data in the common scenario of simultaneous node failures due to cluster-wide power outages. Due to the high fixed cost of each incident of data loss, many data center operators prefer to minimize the frequency of such events at the expense of losing more data in each event.

We present Copyset Replication, a novel general-purpose replication technique that significantly reduces the frequency of data loss events. We implemented and evaluated Copyset Replication on two open source data center storage systems, HDFS and RAMCloud, and show it incurs a low overhead on all operations. Such systems require that each node's data be scattered across several nodes for parallel data recovery and access. Copyset Replication presents a near optimal trade-off between the number of nodes on which the data is scattered and the probability of data loss. For example, in a 5000-node RAMCloud cluster under a power outage, Copyset Replication reduces the probability of data loss from 99.99% to 0.15%. For Facebook's HDFS cluster, it reduces the probability from 22.8% to 0.78%.

1. INTRODUCTION

Random replication is used as a common technique by data center storage systems, such as Hadoop Distributed File System (HDFS) [25], RAMCloud [24], Google File System (GFS) [14] and Windows Azure [6] to ensure durability and availability. These systems partition their data into chunks that are replicated several times (we use R to denote the replication factor) on randomly selected nodes on different racks. When a node fails, its data is restored by reading its chunks from their replicated copies.

However, large-scale correlated failures such as cluster power outages, a common type of data center failure scenario [7, 10, 13, 25], are handled poorly by random replication. This scenario stresses the availability of the system because a non-negligible percentage of nodes (0.5%-1%) [7, 25] do not come back to life after power has been restored. When a large number of nodes do not power up there is a high probability that all replicas of at least one chunk in the system will not be available.

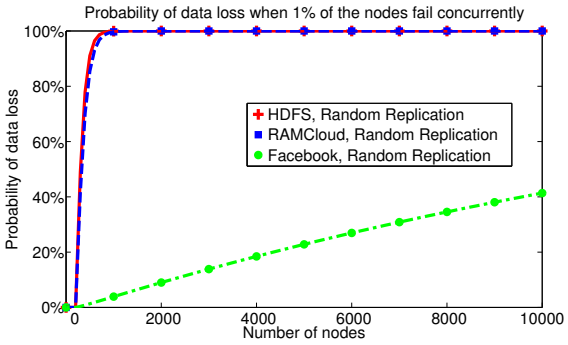


Figure 1: Computed probability of data loss with $R = 3$ when 1% of the nodes do not survive a power outage. The parameters are based on publicly available sources [5, 7, 24, 25] (see Table 1).

Figure 1 shows that once the size of the cluster scales beyond 300 nodes, this scenario is nearly guaranteed to cause a data loss event in some of these systems. Such data loss events have been documented in practice by Yahoo! [25], LinkedIn [7] and Facebook [5]. Each event reportedly incurs a high fixed cost that is not proportional to the amount of data lost. This cost is due to the time it takes to locate the unavailable chunks in backup or recompute the data set that contains these chunks. In the words of Kannan Muthukkaruppan, Tech Lead of Facebook's HBase engineering team: "Even losing a single block of data incurs a high fixed cost, due to the overhead of locating and recovering the unavailable data. Therefore, given a fixed amount of unavailable data each year, it is much better to have fewer incidents of data loss with more data each than more incidents with less data. We would like to optimize for minimizing the probability of incurring *any* data loss" [22]. Other data center operators have reported similar experiences [8].

Another point of view about this trade-off was expressed by Luiz André Barroso, Google Fellow: "Having a framework that allows a storage system provider to manage the profile of frequency vs. size of data losses is very useful, as different systems prefer different policies. For example, some providers might prefer frequent, small losses since they are less likely to tax storage nodes and fabric with spikes in data reconstruction traffic. Other services may not work well when even a small fraction of the data is unavailable. Those will prefer to have all or nothing, and would opt for fewer events even if they come at a larger loss penalty." [3]

Random replication sits on one end of the trade-off between the frequency of data loss events and the amount lost at each event. In this paper we introduce Copyset Replication, an alternative general-purpose replication scheme with the same performance of random replication, which sits at the other end of the spectrum.

Copyset Replication splits the nodes into $copysets$, which are sets of R nodes. The replicas of a single chunk can only be stored on one copyset. This means that data loss events occur only when all the nodes of some copyset fail simultaneously.

The probability of data loss is minimized when each node is a member of exactly one copyset. For example, assume our system has 9 nodes with $R = 3$ that are split into three copysets: $\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}$. Our system would only lose data if nodes 1, 2 and 3, nodes 4, 5 and 6 or nodes 7, 8 and 9 fail simultaneously.

In contrast, with random replication and a sufficient number of chunks, any combination of 3 nodes would be a copyset, and any combination of 3 nodes that fail simultaneously would cause data loss.

The scheme above provides the lowest possible probability of data loss under correlated failures, at the expense of the largest amount of data loss per event. However, the copyset selection above constrains the replication of every chunk to a single copyset, and therefore impacts other operational parameters of the system. Notably, when a single node fails there are only $R - 1$ other nodes that contain its data. For certain systems (like HDFS), this limits the node's recovery time, because there are only $R - 1$ other nodes that can be used to restore the lost chunks. This can also create a high load on a small number of nodes.

To this end, we define the $scatter\ width$ (S) as the number of nodes that store copies for each node's data.

Using a low scatter width may slow recovery time from independent node failures, while using a high scatter width increases the frequency of data loss from correlated failures. In the 9-node system example above, the following copyset construction will yield $S = 4$: $\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}, \{1, 4, 7\}, \{2, 5, 8\}, \{3, 6, 9\}$. In this example, chunks of node 5 would be replicated either at nodes 4 and 6, or nodes 2 and 8. The increased scatter width creates more copyset failure opportunities.

The goal of Copyset Replication is to minimize the probability of data loss, *given any scatter width* by using the smallest number of copysets. We demonstrate that Copyset Replication provides a near optimal solution to this problem. We also show that this problem has been partly explored in a different context in the field of combinatorial design theory, which was originally used to design agricultural experiments [26].

Copyset Replication transforms the profile of data loss events: assuming a power outage occurs once a year, it would take on average a 5000-node RAMCloud cluster 625 years to lose data. The system would lose an average of 64 GB (an entire server's worth of data) in this rare event. With random replication, data loss events occur frequently (during every power failure), and several chunks of data are lost in each event. For example, a 5000-node RAMCloud cluster would lose about 344 MB in each power outage.

To demonstrate the general applicability of Copyset Replication, we implemented it on two open source data center storage systems: HDFS and RAMCloud. We show that Copyset Replication incurs a low overhead on both systems. It reduces the probability of data loss in RAMCloud from 99.99% to 0.15%. In addition, Copyset Replication with 3 replicas achieves a lower data loss probability than the random replication scheme does with 5 replicas. For Facebook's HDFS deployment, Copyset Replication reduces the probability of data loss from 22.8% to 0.78%.

The paper is split into the following sections. Section 2 presents the problem. Section 3 provides the intuition for our solution. Section 4 discusses the design of Copyset Replication. Section 5 provides details on the implementation of Copyset Replication in HDFS and RAMCloud and its performance overhead. Additional applications of Copyset Replication are presented in in Section 6, while Section 7 analyzes related work.

2. THE PROBLEM

In this section we examine the replication schemes of three data center storage systems (RAMCloud, the default HDFS and Facebook's HDFS), and analyze their vulnerability to data loss under correlated failures.

2.1 Definitions

The replication schemes of these systems are defined by several parameters. R is defined as the number of replicas of each chunk. The default value of R is 3 in these systems. N is the number of nodes in the system. The three systems we investigate typically have hundreds to thousands of nodes. We assume nodes are

System	Chunks per Node	Cluster Size	Scatter Width	Replication Scheme
Facebook	10000	1000-5000	10	Random replication on a small group of nodes, second and third replica reside on the same rack
RAMCloud	8000	100-10000	N-1	Random replication across all nodes
HDFS	10000	100-10000	200	Random replication on a large group of nodes, second and third replica reside on the same rack

Table 1: Replication schemes of data center storage systems. These parameters are estimated based on publicly available data [2, 5, 7, 24, 25]. For simplicity, we fix the HDFS scatter width to 200, since its value varies depending on the cluster and rack size.

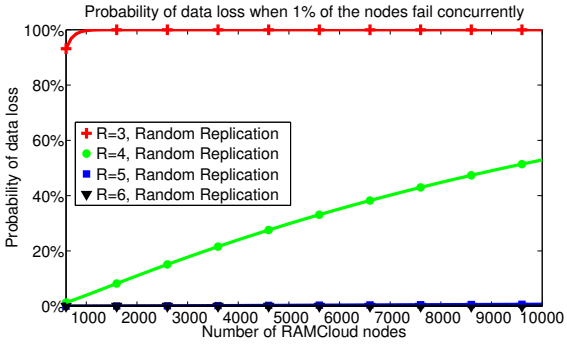


Figure 2: Simulation of the data loss probabilities of a RAMCloud cluster, varying the number of replicas per chunk.

indexed from 1 to N. S is defined as the scatter width. If a system has a scatter width of S, each node's data is split uniformly across a group of S other nodes. That is, whenever a particular node fails, S other nodes can participate in restoring the replicas that were lost. Table 1 contains the parameters of the three systems.

We define a *set*, as a group of R distinct nodes. A *copyset* is a set that stores all of the copies of a chunk. For example, if a chunk is replicated on nodes $\{7, 12, 15\}$, then these nodes form a copyset. We will show that a large number of distinct copysets increases the probability of losing data under a massive correlated failure. Throughout the paper, we will investigate the relationship between the number of copysets and the system's scatter width.

We define a *permutation* as an ordered list of all nodes in the cluster. For example, $\{4, 1, 3, 6, 2, 7, 5\}$ is a permutation of a cluster with $N = 7$ nodes.

Finally, random replication is defined as the following algorithm. The first, or primary replica is placed on a random node from the entire cluster. Assuming the primary replica is placed on node i, the remaining $R - 1$ secondary replicas are placed on random machines chosen from nodes $\{i + 1, i + 2, ..., i + S\}$. If $S = N - 1$, the secondary replicas' nodes are chosen uniformly from all the nodes in the cluster [1].

2.2 Random Replication

The primary reason most large scale storage systems use random replication is that it is a simple replication technique that provides strong protection against uncorrelated failures like individual server or disk failures [13, 25] [2]. These failures happen frequently (thousands of times a year on a large cluster [7, 10, 13]), and are caused by a variety of reasons, including software, hardware and disk failures. Random replication across failure domains (e.g., placing the copies of a chunk on different racks) protects against concurrent failures that happen within a certain domain of nodes, such as racks or network segments. Such failures are quite common and typically occur dozens of times a year [7, 10, 13].

However, multiple groups, including researchers from Yahoo! and LinkedIn, have observed that when clusters with random replication lose power, several chunks of data become unavailable [7, 25], i.e., all three replicas of these chunks are lost. In these events, the entire cluster loses power, and typically 0.5-1% of the nodes fail to reboot [7, 25]. Such failures are not uncommon; they occur once or twice per year in a given data center [7].

Figure 1 shows the probability of losing data in the event of a power outage in the three systems. The figure shows that RAMCloud and HDFS are almost guaranteed to lose data in this event, once the cluster size grows beyond a few hundred nodes. Facebook has a lower data loss probability of about 20% for clusters of 5000 nodes.

Multiple groups have expressed interest in reducing the incidence of data loss, at the expense of losing a larger amount of data at each incident [3, 8, 22]. For example, the Facebook HDFS team has modified the default HDFS implementation to constrain the replication in their deployment to significantly reduce the probability of data loss at the expense of losing more data during each incident [2, 5]. Facebook's Tech Lead of the HBase engineering team has confirmed this point, as cited above [22]. Robert Chansler, Senior Manager of

[1] Our definition of random replication is based on Facebook's design, which selects the replication candidates from a window of nodes around the primary node.

[2] For simplicity's sake, we assume random replication for all three systems, even though the actual schemes are slightly different (e.g., HDFS replicates the second and third replicas on the same rack [25].). We have found there is little difference in terms of data loss probabilities between the different schemes.

Hadoop Infrastructure at Linkedin has also confirmed the importance of addressing this issue: "A power-on restart of HDFS nodes is a real problem, since it introduces a moment of correlated failure of nodes and the attendant threat that data becomes unavailable. Due to this issue, our policy is to not turn off Hadoop clusters. Administrators must understand how to restore the integrity of the file system as fast as possible, and *an option to reduce the number of instances when data is unavailable–at the cost of increasing the number of blocks recovered at such instances–can be a useful tool since it lowers the overall total down time*" [8].

The main reason some data center operators prefer to minimize the frequency of data loss events, is that there is a fixed cost to each incident of data loss that is not proportional to the amount of data lost in each event. The cost of locating and retrieving the data from secondary storage can cause a whole data center operations team to spend a significant amount of time that is unrelated to the amount of data lost [22]. There are also other fixed costs associated with data loss events. In the words of Robert Chansler: "In the case of data loss... [frequently] the data may be recomputed. For re-computation an application typically recomputes its entire data set whenever any data is lost. *This causes a fixed computational cost that is not proportional with the amount of data lost"*. [8]

One trivial alternative for decreasing the probability of data loss is to increase R. In Figure 2 we computed the probability of data loss under different replication factors in RAMCloud. As we would expect, increasing the replication factor increases the durability of the system against correlated failures. However, increasing the replication factor from 3 to 4 does not seem to provide sufficient durability in this scenario. In order to reliably support thousands of nodes in current systems, the replication factor would have to be at least 5. Using $R = 5$ significantly hurts the system's performance and almost doubles the cost of storage.

Our goal in this paper is to decrease the probability of data loss under power outages, without changing the underlying parameters of the system.

3. INTUITION

If we consider each chunk individually, random replication provides high durability even in the face of a power outage. For example, suppose we are trying to replicate a single chunk three times. We randomly select three different machines to store our replicas. If a power outage causes 1% of the nodes in the data center to fail, the probability that the crash caused the exact three machines that store our chunk to fail is only 0.0001%.

However, assume now that instead of replicating just one chunk, the system replicates millions of chunks (each node has 10,000 chunks or more), and needs to ensure that every single one of these chunks will survive the failure. Even though each individual chunk is very safe, in aggregate across the entire cluster, some chunk is expected to be lost. Figure 1 demonstrates this effect: in practical data center configurations, data loss is nearly guaranteed if *any combination of three nodes* fail simultaneously.

We define a copyset as a distinct set of nodes that contain all copies of a given chunk. *Each copyset is a single unit of failure*, i.e., when a copyset fails at least one data chunk is irretrievably lost. Increasing the number of copysets will increase the probability of data loss under a correlated failure, because there is a higher probability that the failed nodes will include at least one copyset. With random replication, almost every new replicated chunk creates a distinct copyset, up to a certain point.

3.1 Minimizing the Number of Copysets

In order to minimize the number of copysets a replication scheme can statically assign each node to a single copyset, and constrain the replication to these preassigned copysets. The first or primary replica would be placed randomly on any node (for load-balancing purposes), and the other secondary replicas would be placed deterministically on the first node's copyset.

With this scheme, we will only lose data if all the nodes in a copyset fail simultaneously. For example, with 5000 nodes, this reduces the data loss probabilities when 1% of the nodes fail simultaneously from 99.99% to 0.15%.

However, the downside of this scheme is that it severely limits the system's scatter width. This may cause serious problems for certain storage systems. For example, if we use this scheme in HDFS with $R = 3$, each node's data will only be placed on two other nodes. This means that in case of a node failure, the system will be able to recover its data from only two other nodes, which would significantly increase the recovery time. In addition, such a low scatter width impairs load balancing and may cause the two nodes to be overloaded with client requests.

3.2 Scatter Width

Our challenge is to design replication schemes that minimize the number of copysets given the required scatter width set by the system designer.

To understand how to generate such schemes, consider the following example. Assume our storage system has the following parameters: $R = 3$, $N = 9$ and $S = 4$. If we use random replication, each chunk will be replicated on another node chosen randomly from a group of S nodes following the first node. E.g., if the primary replica is placed on node 1, the secondary replica will be

randomly placed either on node 2, 3, 4 or 5.

Therefore, if our system has a large number of chunks, it will create 54 distinct copysets.

In the case of a simultaneous failure of three nodes, the probability of data loss is the number of copysets divided by the maximum number of sets:

$$\frac{\#\ copysets}{\binom{N}{R}} = \frac{54}{\binom{9}{3}} = 0.64$$

Now, examine an alternative scheme using the same parameters. Assume we only allow our system to replicate its data on the following copysets:

$$\{1,2,3\},\{4,5,6\},\{7,8,9\}\{1,4,7\},\{2,5,8\},\{3,6,9\}$$

That is, if the primary replica is placed on node 3, the two secondary replicas can only be randomly on nodes 1 and 2 or 6 and 9. Note that with this scheme, each node's data will be split uniformly on four other nodes.

The new scheme created only 6 copysets. Now, if three nodes fail, the probability of data loss is:

$$\frac{\#\ copysets}{84} = 0.07.$$

As we increase N, the relative advantage of creating the minimal number of copysets increases significantly. For example, if we choose a system with $N = 5000$, $R = 3$, $S = 10$ (like Facebook's HDFS deployment), we can design a replication scheme that creates about 8,300 copysets, while random replication would create about 275,000 copysets.

The scheme illustrated above has two important properties that form the basis for the design of Copyset Replication. First, each copyset overlaps with each other copyset by at most one node (e.g., the only overlapping node of copysets $\{4,5,6\}$ and $\{3,6,9\}$ is node 6). This ensures that each copyset increases the scatter width for its nodes by exactly $R-1$. Second, the scheme ensures that the copysets cover all the nodes equally.

Our scheme creates two permutations, and divides them into copysets. Since each permutation increases the scatter width by $R-1$, the overall scatter width will be:

$$S = P(R-1)$$

Where P is the number of permutations. This scheme will create $P\frac{N}{R}$ copysets, which is equal to: $\frac{S}{R-1}\frac{N}{R}$.

The number of copysets created by random replication for values of $S < \frac{N}{2}$ is: $N\binom{S}{R-1}$. This number is equal to the number of primary replica nodes times $R-1$ combinations of secondary replica nodes chosen from a group of S nodes. When S approaches N, the number of copysets approaches the total number of sets, which is equal to $\binom{N}{R}$.

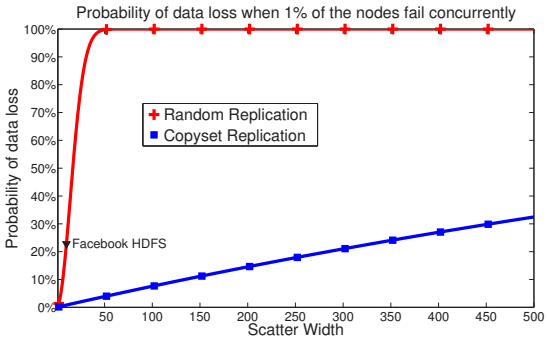


Figure 3: Data loss probability when 1% of the nodes fail simultaneously as a function of S, using $N = 5000$, $R = 3$.

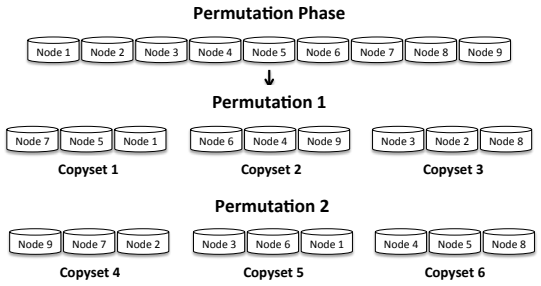


Figure 4: Illustration of the Copyset Replication Permutation phase.

In summary, in a minimal copyset scheme, the number of copysets grows linearly with S, while random replication creates $O(S^{R-1})$ copysets. Figure 3 demonstrates the difference in data loss probabilities as a function of S, between random replication and Copyset Replication, the scheme we develop in the paper.

4. DESIGN

In this section we describe the design of a novel replication technique, Copyset Replication, that provides a near optimal trade-off between the scatter width and the number of copysets.

As we saw in the previous section, there exist replication schemes that achieve a linear increase in copysets for a linear increase in S. However, it is not always simple to design the optimal scheme that creates non-overlapping copysets that cover all the nodes. In some cases, with specific values of N, R and S, it has even been shown that no such non-overlapping schemes exist [18, 19]. For a more detailed theoretical discussion see Section 7.1.

Therefore, instead of using an optimal scheme, we propose Copyset Replication, which is close to optimal in practical settings and very simple to implement. Copyset Replication randomly generates permutations

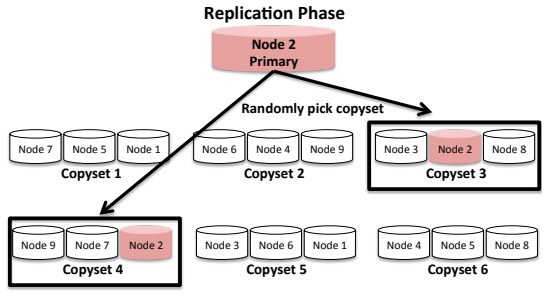


Figure 5: Illustration of the Copyset Replication Replication phase.

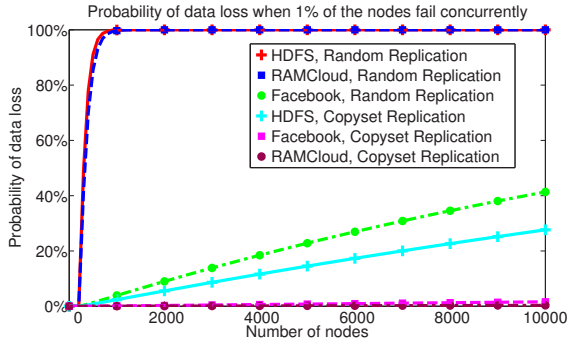


Figure 6: Data loss probability of random replication and Copyset Replication with $R = 3$, using the parameters from Table 1. HDFS has higher data loss probabilities because it uses a larger scatter width ($S = 200$).

and splits each permutation into copysets. We will show that as long as S is much smaller then the number of nodes in the system, this scheme is likely to generate copysets with at most one overlapping node.

Copyset Replication has two phases: Permutation and Replication. The permutation phase is conducted offline, while the replication phase is executed every time a chunk needs to be replicated.

Figure 4 illustrates the permutation phase. In this phase we create several permutations, by randomly permuting the nodes in the system. The number of permutations we create depends on S, and is equal to $P = \frac{S}{R-1}$. If this number is not an integer, we choose its ceiling. Each permutation is split consecutively into copysets, as shown in the illustration. The permutations can be generated completely randomly, or we can add additional constraints, limiting nodes from the same rack in the same copyset, or adding network and capacity constraints. In our implementation, we prevented nodes from the same rack from being placed in the same copyset by simply reshuffling the permutation until all the constraints were met.

In the replication phase (depicted by Figure 5) the system places the replicas on one of the copysets generated in the permutation phase. The first or primary replica can be placed on any node of the system, while the other replicas (the secondary replicas) are placed on the nodes of a randomly chosen copyset that contains the first node.

Copyset Replication is agnostic to the data placement policy of the first replica. Different storage systems have certain constraints when choosing their primary replica nodes. For instance, in HDFS, if the local machine has enough capacity, it stores the primary replica locally, while RAMCloud uses an algorithm for selecting its primary replica based on Mitzenmacher's randomized load balancing [23]. The only requirement made by Copyset Replication is that the secondary replicas of a chunk are always placed on one of the copysets that contains the primary replica's node. This constrains the number of copysets created by Copyset Replication.

4.1 Durability of Copyset Replication

Figure 6 is the central figure of the paper. It compares the data loss probabilities of Copyset Replication and random replication using 3 replicas with RAMCloud, HDFS and Facebook. For HDFS and Facebook, we plotted the same S values for Copyset Replication and random replication. In the special case of RAMCloud, the recovery time of nodes is not related to the number of permutations in our scheme, because disk nodes are recovered from the memory across all the nodes in the cluster and not from other disks. Therefore, Copyset Replication with with a minimal $S = R - 1$ (using $P = 1$) actually provides the same node recovery time as using a larger value of S. Therefore, we plot the data probabilities for Copyset Replication using $P = 1$.

We can make several interesting observations. Copyset Replication reduces the probability of data loss under power outages for RAMCloud and Facebook to close to zero, but does not improve HDFS as significantly. For a 5000 node cluster under a power outage, Copyset Replication reduces RAMCloud's probability of data loss from 99.99% to 0.15%. For Facebook, that probability is reduced from 22.8% to 0.78%. In the case of HDFS, since the scatter width is large ($S = 200$), Copyset Replication significantly improves the data loss probability, but not enough so that the probability of data loss becomes close to zero.

Figure 7 depicts the data loss probabilities of 5000 node RAMCloud, HDFS and Facebook clusters. We can observe that the reduction of data loss caused by Copyset Replication is equivalent to increasing the number of replicas. For example, in the case of RAMCloud, if the system uses Copyset Replication with 3 replicas, it has lower data loss probabilities than random replication

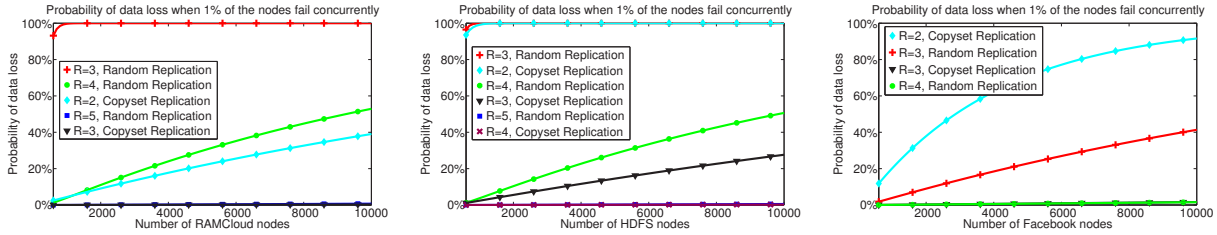


Figure 7: Data loss probability of random replication and Copyset Replication in different systems.

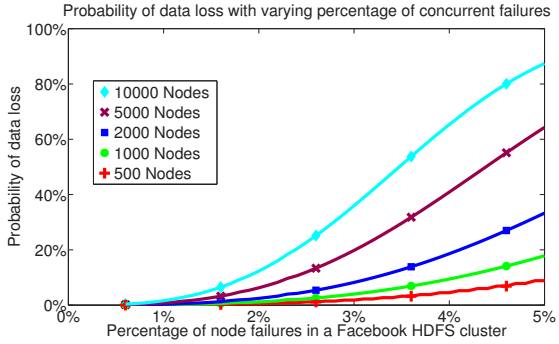


Figure 8: Data loss probability on Facebook's HDFS cluster, with a varying percentage of the nodes failing simultaneously.

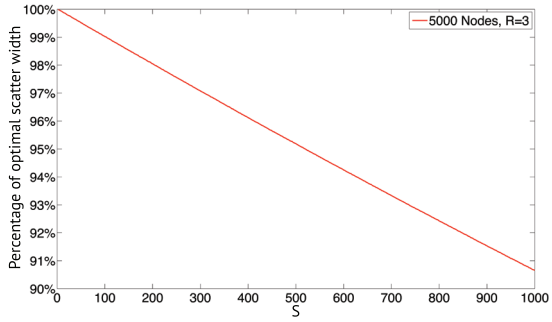


Figure 9: Comparison of the average scatter width ofCopyset Replication to the optimal scatter width in a 5000-node cluster.

with 5 replicas. Similarly, Copyset Replication with 3 replicas has the same the data loss probability as random replication with 4 replicas in a Facebook cluster.

The typical number of simultaneous failures observed in data centers is 0.5-1% of the nodes in the cluster [25]. Figure 8 depicts the probability of data loss in Facebook's HDFS system as we increase the percentage of simultaneous failures much beyond the reported 1%. Note that Facebook commonly operates in the range of 1000-5000 nodes per cluster (e.g., see Table 1). For these cluster sizes Copyset Replication prevents data loss with a high probability, even in the scenario where 2% of the nodes fail simultaneously.

4.2 Optimality of Copyset Replication

Copyset Replication is not optimal, because it doesn't guarantee that all of its copysets will have at most one overlapping node. In other words, it doesn't guarantee that each node's data will be replicated across exactly S different nodes. Figure 9 depicts a monte-carlo simulation that compares the average scatter width achieved by Copyset Replication as a function of the maximum S if all the copysets were non-overlapping for a cluster of 5000 nodes.

The plot demonstrates that when S is much smaller than N, Copyset Replication is more than 90% optimal. For RAMCloud and Facebook, which respectively use $S = 2$ and $S = 10$, Copyset Replication is nearly optimal. For HDFS we used $S = 200$, and in this case Copyset Replication provides each node an average of 98% of the optimal bandwidth, which translates to $S = 192$.

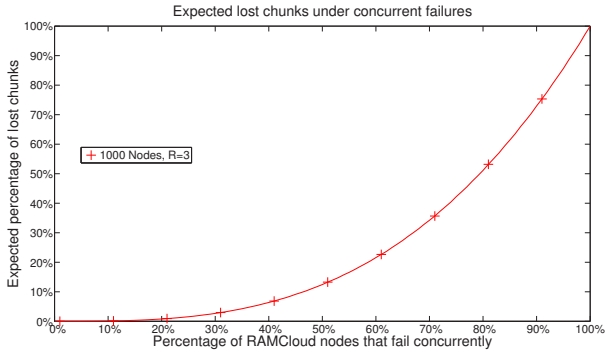


Figure 10: Expected amount of data lost as a percentage of the data in the cluster.

4.3 Expected Amount of Data Lost

Copyset Replication trades off the probability of data loss with the amount of data lost in each incident. The expected amount of data lost remains constant regardless of the replication policy. Figure 10 shows the amount of data lost as a percentage of the data in the cluster.

Therefore, a system designer that deploys Copyset

Replication should expect to experience much fewer events of data loss. However, each one of these events will lose a larger amount of data. In the extreme case, if we are using Copyset Replication with $S = 2$ like in RAMCloud, we would lose a whole node's worth of data at every data loss event.

5. EVALUATION

Copyset Replication is a general-purpose, scalable replication scheme that can be implemented on a wide range of data center storage systems and can be tuned to any scatter width. In this section, we describe our implementation of Copyset Replication in HDFS and RAMCloud. We also provide the results of our experiments on the impact of Copyset Replication on both systems' performance.

5.1 HDFS Implementation

The implementation of Copyset Replication on HDFS was relatively straightforward, since the existing HDFS replication code is well-abstracted. Copyset Replication is implemented entirely on the HDFS NameNode, which serves as a central directory and manages replication for the entire cluster.

The permutation phase of Copyset Replication is run when the cluster is created. The user specifies the scatter width and the number of nodes in the system. After all the nodes have been added to the cluster, the NameNode creates the copysets by randomly permuting the list of nodes. If a generated permutation violates any rack or network constraints, the algorithm randomly reshuffles a new permutation.

In the replication phase, the primary replica is picked using the default HDFS replication.

5.1.1 Nodes Joining and Failing

In HDFS nodes can spontaneously join the cluster or crash. Our implementation needs to deal with both cases.

When a new node joins the cluster, the NameNode randomly creates $\frac{S}{R-1}$ new copysets that contain it. As long as the scatter width is much smaller than the number of nodes in the system, this scheme will still be close to optimal (almost all of the copysets will be non-overlapping). The downside is that some of the other nodes may have a slightly higher than required scatter width, which creates more copysets than necessary.

When a node fails, for each of its copysets we replace it with a randomly selected node. For example, if the original copyset contained nodes $\{1, 2, 3\}$, and node 1 failed, we re-replicate a copy of the data in the original copyset to a new randomly selected node. As before, as long as the scatter width is significantly smaller than the number of nodes, this approach creates non-overlapping copysets.

Replication	Recovery Time (s)	Minimal Scatter Width	Average Scatter Width	# Copysets
Random Replication	600.4	2	4	234
Copyset Replication	642.3	2	4	13
Random Replication	221.7	8	11.3	2145
Copyset Replication	235	8	11.3	77
Random Replication	139	14	17.8	5967
Copyset Replication	176.6	14	17.8	147
Random Replication	108	20	23.9	9867
Copyset Replication	127.7	20	23.9	240

Table 2: Comparison of recovery time of a 100 GB node on a 39 node cluster. Recovery time is measured after the moment of failure detection.

5.2 HDFS Evaluation

We evaluated the Copyset Replication implementation on a cluster of 39 HDFS nodes with 100 GB of SSD storage and a 1 GB ethernet network. Table 2 compares the recovery time of a single node using Copyset Replication and random replication. We ran each recovery five times.

As we showed in previous sections, Copyset Replication has few overlapping copysets as long as S is significantly smaller than N. However, since our experiment uses a small value of N, some of the nodes did not have sufficient scatter width due to a large number of overlapping copysets. In order to address this issue, our Copyset Replication implementation generates additional permutations until the system reached the minimal desired scatter width for all its nodes. The additional permutations created more copysets. We counted the average number of distinct copysets. As the results show, even with the extra permutations, Copyset Replication still has orders of magnitude fewer copysets than random replication.

To normalize the scatter width between Copyset Replication and random replication, when we recovered the data with random replication we used the average scatter width obtained by Copyset Replication.

The results show that Copyset Replication has an overhead of about 5-20% in recovery time compared to random replication. This is an artifact of our small cluster size. The small size of the cluster causes some nodes to be members of more copysets than others, which means they have more data to recover and delay the overall recovery time. This problem would not occur if we used

Scatter Width	Mean Load	75th % Load	99th % Load	Max Load
10	10%	10%	10%	20%
20	5%	5%	5%	10%
50	2%	2%	2%	6%
100	1%	1%	2%	3%
200	0.5%	0.5%	1%	1.5%
500	0.2%	0.2%	0.4%	0.8%

Table 3: The simulated load in a 5000-node HDFS cluster with $R = 3$, using Copyset Replication. With Random Replication, the average load is identical to the maximum load.

a realistic large-scale HDFS cluster (hundreds to thousands of nodes).

5.2.1 Hot Spots

One of the main advantages of random replication is that it can prevent a particular node from becoming a 'hot spot', by scattering its data uniformly across a random set of nodes. If the primary node gets overwhelmed by read requests, clients can read its data from the nodes that store the secondary replicas.

We define the load $L(i, j)$ as the percentage of node i's data that is stored as a secondary replica in node j. For example, if $S = 2$ and node 1 replicates all of its data to nodes 2 and 3, then $L(1, 2) = L(1, 3) = 0.5$, i.e., node 1's data is split evenly between nodes 2 and 3.

The more we spread the load evenly across the nodes in the system, the more the system will be immune to hot spots. Note that the load is a function of the scatter width; if we increase the scatter width, the load will be spread out more evenly. We expect that the load of the nodes that belong to node i's copysets will be $dfrac1S$. Since Copyset Replication guarantees the same scatter width of random replication, it should also spread the load uniformly and be immune to hot spots with a sufficiently high scatter width.

In order to test the load with Copyset Replication, we ran a monte carlo simulation of data replication in a 5000-node HDFS cluster with $R = 3$.

Table 3 shows the load we measured in our monte carlo experiment. Since we have a very large number of chunks with random replication, the mean load is almost identical to the worst-case load. With Copyset Replication, the simulation shows that the 99th percentile loads are 1-2 times and the maximum loads 1.5-4 times higher than the mean load. Copyset Replication incurs higher worst-case loads because the permutation phase can produce some copysets with overlaps.

Therefore, if the system's goal is to prevent hot spots even in a worst case scenario with Copyset Replication, the system designer should increase the system's scatter width accordingly.

5.3 Implementation of Copyset Replication in RAMCloud

The implementation of Copyset Replication on RAMCloud was similar to HDFS, with a few small differences. Similar to the HDFS implementation, most of the code was implemented on RAMCloud's coordinator, which serves as a main directory node and also assigns nodes to replicas.

In RAMCloud, the main copy of the data is kept in a master server, which keeps the data in memory. Each master replicates its chunks on three different backup servers, which store the data persistently on disk.

The Copyset Replication implementation on RAMCloud only supports a minimal scatter width ($S = R - 1 = 2$). We chose a minimal scatter width, because it doesn't affect RAMCloud's node recovery times, since the backup data is recovered from the master nodes, which are spread across the cluster.

Another difference between the RAMCloud and HDFS implementations is how we handle new backups joining the cluster and backup failures. Since each node is a member of a single copyset, if the coordinator doesn't find three nodes to form a complete copyset, the new nodes will remain idle until there are enough nodes to form a copyset.

When a new backup joins the cluster, the coordinator checks whether there are three backups that are not assigned to a copyset. If there are, the coordinator assigns these three backups to a copyset.

In order to preserve $S = 2$, every time a backup node fails, we re-replicate its entire copyset. Since backups don't service normal reads and writes, this doesn't affect the sytem's latency. In addition, due to the fact that backups are recovered in parallel from the masters, re-replicating the entire group doesn't significantly affect the recovery latency. However, this approach does increase the disk and network bandwidth during recovery.

5.4 Evaluation of Copyset Replication on RAMCloud

We compared the performance of Copyset Replication with random replication under three scenarios: normal RAMCloud client operations, a single master recovery and a single backup recovery.

As expected, we could not measure any overhead of using Copyset Replication on normal RAMCloud operations. We also found that it does not impact master recovery, while the overhead of backup recovery was higher as we expected. We provide the results below.

5.4.1 Master Recovery

One of the main goals of RAMCloud is to fully re-

Replication	Recovery Data	Recovery Time
Random Replication	1256 MB	0.73 s
Copyset Replication	3648 MB	1.10 s

Table 4: Comparison of backup recovery performance on RAMCloud with Copyset Replication. Recovery time is measured after the moment of failure detection.

cover a master in about 1-2 seconds so that applications experience minimal interruptions. In order to test master recovery, we ran a cluster with 39 backup nodes and 5 master nodes. We manually crashed one of the master servers, and measured the time it took RAMCloud to recover its data. We ran this test 100 times, both with Copyset Replication and random replication. As expected, we didn't observe any difference in the time it took to recover the master node in both schemes.

However, when we ran the benchmark again using 10 backups instead of 39, we observed Copyset Replication took 11% more time to recover the master node than the random replication scheme. Due to the fact that Copyset Replication divides backups into groups of three, it only takes advantage of 9 out of the 10 nodes in the cluster. This overhead occurs only when we use a number of backups that is not a multiple of three on a very small cluster. Since we assume that RAMCloud is typically deployed on large scale clusters, the master recovery overhead is negligible.

5.4.2 Backup Recovery

In order to evaluate the overhead of Copyset Replication on backup recovery, we ran an experiment in which a single backup crashes on a RAMCloud cluster with 39 masters and 72 backups, storing a total of 33 GB of data. Table 4 presents the results. Since masters re-replicate data in parallel, recovery from a backup failure only takes 51% longer using Copyset Replication, compared to random replication. As expected, our implementation approximately triples the amount of data that is re-replicated during recovery. Note that this additional overhead is not inherent to Copyset Replication, and results from our design choice to strictly preserve a minimal scatter width at the expense of higher backup recovery overhead.

6. DISCUSSION

This section discusses how coding schemes relate to the number of copysets, and how Copyset Replication can simplify graceful power downs of storage clusters.

6.1 Copysets and Coding

Some storage systems, such as GFS, Azure and HDFS, use coding techniques to reduce storage costs. These techniques generally do not impact the probability of data loss due to simultaneous failures.

Codes are typically designed to compress the data rather than increase its durability. If the coded data is distributed on a very large number of copysets, multiple simultaneous failures will still cause data loss.

In practice, existing storage system parity code implementations do not significantly reduce the number of copysets, and therefore do not impact the profile of data loss. For example, the HDFS-RAID [1, 11] implementation encodes groups of 5 chunks in a RAID 5 and mirroring scheme, which reduces the number of distinct copysets by a factor of 5. While reducing the number of copysets by a factor of 5 reduces the probability of data loss, Copyset Replication still creates two orders of magnitude fewer copysets than this scheme. Therefore, HDFS-RAID with random replication is still very likely lose data in the case of power outages.

6.2 Graceful Power Downs

Data center operators periodically need to gracefully power down parts of a cluster [4, 10, 13]. Power downs are used for saving energy in off-peak hours, or to conduct controlled software and hardware upgrades.

When part of a storage cluster is powered down, it is expected that at least one replica of each chunk will stay online. However, random replication considerably complicates controlled power downs, since if we power down a large group of machines, there is a very high probability that all the replicas of a given chunk will be taken offline. In fact, these are exactly the same probabilities that we use to calculate data loss. Several previous studies have explored data center power down in depth [17, 21, 27].

If we constrain Copyset Replication to use the minimal number of copysets (i.e., use Copyset Replication with $S = R - 1$), it is simple to conduct controlled cluster power downs. Since this version of Copyset Replication assigns a single copyset to each node, as long as one member of each copyset is kept online, we can safely power down the remaining nodes. For example, a cluster using three replicas with this version of Copyset Replication can effectively power down two-thirds of the nodes.

7. RELATED WORK

The related work is split into three categories. First, replication schemes that achieve optimal scatter width are related to a field in mathematics called combinatorial design theory, which dates back to the 19th century. We will give a brief overview and some examples of such designs. Second, replica placement has been studied in the context of DHT systems. Third, several data center storage systems have employed various solutions to mitigate data loss due to concurrent node failures.

7.1 Combinatorial Design Theory

The special case of trying to minimize the number of copysets when $S = N - 1$ is related to combinatorial design theory. Combinatorial design theory tries to answer questions about whether elements of a discrete finite set can be arranged into subsets, which satisfy certain "balance" properties. The theory has its roots in recreational mathematical puzzles or brain teasers in the 18th and 19th century. The field emerged as a formal area of mathematics in the 1930s for the design of agricultural experiments [12]. Stinson provides a comprehensive survey of combinatorial design theory and its applications. In this subsection we borrow several of the book's definitions and examples [26].

The problem of trying to minimize the number of copysets with a scatter width of $S = N - 1$ can be expressed a Balanced Incomplete Block Design (BIBD), a type of combinatorial design. Designs that try to minimize the number of copysets for any scatter width, such as Copyset Replication, are called unbalanced designs.

A combinatorial design is defined a pair (X, A), such that X is a set of all the nodes in the system (i.e., $X = \{1, 2, 3, ..., N\}$) and A is a collection of nonempty subsets of X. In our terminology, A is a collection of all the copysets in the system.

Let N, R and λ be positive integers such that $N > R \geq 2$. A (N, R, λ) BIBD satisfies the following properties:

1. $|A| = N$
2. Each copyset contains exactly R nodes
3. Every pair of nodes is contained in exactly λ copysets

When $\lambda = 1$, the BIBD provides an optimal design for minimizing the number of copysets for $S = N - 1$.

For example, a $(7, 3, 1)BIBD$ is defined as:

$X = \{1, 2, 3, 4, 5, 6, 7\}$
$A = \{123, 145, 167, 246, 257, 347, 356\}$

Note that each one of the nodes in the example has a recovery bandwidth of 6, because it appears in exactly three non-overlapping copysets.

Another example is the $(9, 3, 1)BIBD$:

$X = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$A = \{123, 456, 789, 147, 258, 369, 159, 267, 348, 168, 249, 357\}$

There are many different methods for constructing new BIBDs. New designs can be constructed by combining other known designs, using results from graph and coding theory or in other methods [20]. The Experimental Design Handbook has an extensive selection of design examples [9].

However, there is no single technique that can produce optimal BIBDs for any combination of N and R. Moreover, there are many negative results, i.e., researchers that prove that no optimal designs exists for a certain combination of N and R [18, 19].

Due to these reasons, and due to the fact that BIBDs do not solve the copyset minimization problem for any scatter width that is not equal to $N - 1$, it is not practical to use BIBDs for creating copysets in data center storage systems. This is why we chose to utilize Copyset Replication, a non-optimal design based on random permutations that can accommodate any scatter width. However, BIBDs do serve as a useful benchmark to measure how optimal Copyset Replication in relationship to the optimal scheme for specific values of S, and the novel formulation of the problem for any scatter width is a potentially interesting future research topic.

7.2 DHT Systems

There are several prior systems that explore the impact of data placement on data availability in the context of DHT systems.

Chun et al. [15] identify that randomly replicating data across a large "scope" of nodes increases the probability of data loss under simultaneous failures. They investigate the effect of different scope sizes using Carbonite, their DHT replication scheme. Yu et al. [28] analyze the performance of different replication strategies when a client requests multiple objects from servers that may fail simultaneously. They propose a DHT replication scheme called "Group", which constrains the placement of replicas on certain groups, by placing the secondary replicas in a particular order based on the key of the primary replica. Similarly, Glacier [16] constrains the random spread of replicas, by limiting each replica to equidistant points in the keys' hash space.

None of these studies focus on the relationship between the probability of data loss and scatter width, or provide optimal schemes for different scatter width constraints.

7.3 Data Center Storage Systems

Facebook's proprietary HDFS implementation constrains the placement of replicas to smaller groups, to protect against concurrent failures [2, 5]. Similarly, Sierra randomly places chunks within constrained groups in order to support flexible node power downs and data center power proportionality [27]. As we discussed previously, both of these schemes, which use random replication within a constrained group of nodes, generate orders of magnitude more copysets than Copyset Replica-

tion with the same scatter width, and hence have a much higher probability of data loss under correlated failures.

Ford et al. from Google [13] analyze different failure loss scenarios on GFS clusters, and have proposed geo-replication as an effective technique to prevent data loss under large scale concurrent node failures. Geo-replication across geographically dispersed sites is a fail-safe way to ensure data durability under a power outage. However, not all storage providers have the capability to support geo-replication. In addition, even for data center operators that have geo-replication (like Facebook and LinkedIn), losing data at a single site still incurs a high fixed cost due to the need to locate or recompute the data. This fixed cost is not proportional to the amount of data lost [8, 22].

8. ACKNOWLEDGEMENTS

We would like to thank David Gal, Diego Ongaro, Israel Cidon, K.V. Rashmi and Shankar Pasupathy for their valuable feedback. We would also like to thank our shepherd, Bernard Wong, and the anonymous reviewers for their comments. Asaf Cidon is supported by the Leonard J. Shustek Stanford Graduate Fellowship. This work was supported by the National Science Foundation under Grant No. 0963859 and by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] HDFS RAID. `http://wiki.apache.org/hadoop/HDFS-RAID`.

[2] Intelligent block placement policy to decrease probability of data loss. `https://issues.apache.org/jira/browse/HDFS-1094`.

[3] L. A. Barroso. Personal Communication, 2013.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[5] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at Facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[7] R. J. Chansler. Data Availability and Durability with the Hadoop Distributed File System. *;login: The USENIX Magazine*, 37(1), February 2012.

[8] R. J. Chansler. Personal Communication, 2013.

[9] W. Cochran and G. Cox. Experimental designs . 1957.

[10] J. Dean. Evolution and future directions of large-scale storage and computation systems at Google. In *SoCC*, page 1, 2010.

[11] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: Replication as a prelude to erasure coding in data-intensive scalable computing, 2011.

[12] R. Fisher. An examination of the different possible solutions of a problem in incomplete blocks. *Annals of Human Genetics*, 10(1):52–75, 1940.

[13] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[15] B. gon Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *IN PROC. OF NSDI*, pages 45–58, 2006.

[16] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *IN PROC. OF NSDI*, 2005.

[17] D. Harnik, D. Naor, and I. Segall. Low power mode in cloud storage systems.

[18] S. Houghten, L. Thiel, J. Janssen, and C. Lam. There is no (46, 6, 1) block design*. *Journal of Combinatorial Designs*, 9(1):60–71, 2001.

[19] P. Kaski and P. Östergård. There exists no (15, 5, 4) RBIBD. *Journal of Combinatorial Designs*, 9(3):227–232, 2001.

[20] J. Koo and J. Gill. Scalable constructions of fractional repetition codes in distributed storage systems. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*, pages 1366–1373. IEEE, 2011.

[21] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, Mar. 2010.

[22] K. Mathukkaruppan. Personal Communication, 2012.

[23] M. D. Mitzenmacher. The power of two choices in randomized load balancing. Technical report, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 1996.

[24] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, pages 29–41, 2011.

[25] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–10, 2010.

[26] D. Stinson. *Combinatorial designs: construction and analysis*. Springer, 2003.

[27] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. *Proceedings of Eurosys 11*, pages 169–182, 2011.

[28] H. Yu, P. B. Gibbons, and S. Nath. Availability of multi-object operations. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 16–16, Berkeley, CA, USA, 2006. USENIX Association.

TAO: Facebook's Distributed Data Store for the Social Graph

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov
Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov
Dmitri Petrov, Lovro Puzar, Yee Jiun Song, Venkat Venkataramani
Facebook, Inc.

Abstract

We introduce a simple data model and API tailored for serving the social graph, and TAO, an implementation of this model. TAO is a geographically distributed data store that provides efficient and timely access to the social graph for Facebook's demanding workload using a fixed set of queries. It is deployed at Facebook, replacing memcache for many data types that fit its model. The system runs on thousands of machines, is widely distributed, and provides access to many petabytes of data. TAO can process a billion reads and millions of writes each second.

1 Introduction

Facebook has more than a billion active users who record their relationships, share their interests, upload text, images, and video, and curate semantic information about their data [2]. The personalized experience of social applications comes from timely, efficient, and scalable access to this flood of data, the *social graph*. In this paper we introduce TAO, a read-optimized graph data store we have built to handle a demanding Facebook workload.

Before TAO, Facebook's web servers directly accessed MySQL to read or write the social graph, aggressively using memcache [21] as a lookaside cache. TAO implements a graph abstraction directly, allowing it to avoid some of the fundamental shortcomings of a lookaside cache architecture. TAO continues to use MySQL for persistent storage, but mediates access to the database and uses its own graph-aware cache.

TAO is deployed at Facebook as a single geographically distributed instance. It has a minimal API and explicitly favors availability and per-machine efficiency over strong consistency; its novelty is its scale: TAO can sustain a billion reads per second on a changing data set of many petabytes.

Overall, this paper makes three contributions. We motivate (§ 2) and characterize (§ 7) a challenging workload: efficient and available read-mostly access to a changing graph. We describe objects and associations, a data model and API that we use to access the graph (§ 3). Lastly, we detail TAO, a geographically distributed system that implements this API (§§ 4–6), and evaluate its performance on our workload (§ 8).

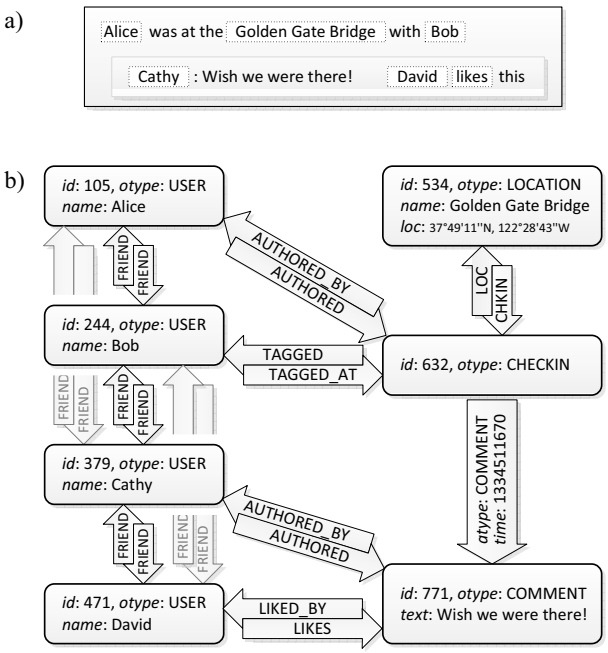


Figure 1: A running example of how a user's checkin might be mapped to objects and associations.

2 Background

A single Facebook page may aggregate and filter hundreds of items from the social graph. We present each user with content tailored to them, and we filter every item with privacy checks that take into account the current viewer. This extreme customization makes it infeasible to perform most aggregation and filtering when content is created; instead we resolve data dependencies and check privacy each time the content is viewed. As much as possible we pull the social graph, rather than pushing it. This implementation strategy places extreme read demands on the graph data store; it must be efficient, highly available, and scale to high query rates.

2.1 Serving the Graph from Memcache

Facebook was originally built by storing the social graph in MySQL, querying it from PHP, and caching results in memcache [21]. This lookaside cache architecture is well suited to Facebook's rapid iteration cycles, since all

of the data mapping and cache-invalidation computations are in client code that is deployed frequently. Over time a PHP abstraction was developed that allowed developers to read and write the objects (nodes) and associations (edges) in the graph, and direct access to MySQL was deprecated for data types that fit the model.

TAO is a service we constructed that directly implements the objects and associations model. We were motivated by encapsulation failures in the PHP API, by the opportunity to access the graph easily from non-PHP services, and by several fundamental problems with the lookaside cache architecture:

Inefficient edge lists: A key-value cache is not a good semantic fit for lists of edges; queries must always fetch the entire edge list and changes to a single edge require the entire list to be reloaded. Basic list support in a lookaside cache would only address the first problem; something much more complicated is required to coordinate concurrent incremental updates to cached lists.

Distributed control logic: In a lookaside cache architecture the control logic is run on clients that don't communicate with each other. This increases the number of failure modes, and makes it difficult to avoid thundering herds. Nishtala et al. provide an in-depth discussion of the problems and present *leases*, a general solution [21]. For objects and associations the fixed API allows us to move the control logic into the cache itself, where the problem can be solved more efficiently.

Expensive read-after-write consistency: Facebook uses asynchronous master/slave replication for MySQL, which poses a problem for caches in data centers using a replica. Writes are forwarded to the master, but some time will elapse before they are reflected in the local replica. Nishtala et al.'s *remote markers* [21] track keys that are known to be stale, forwarding reads for those keys to the master region. By restricting the data model to objects and associations we can update the replica's cache at write time, then use graph semantics to interpret cache maintenance messages from concurrent updates. This provides (in the absence of multiple failures) read-after-write consistency for all clients that share a cache, without requiring inter-regional communication.

2.2 TAO's Goal

TAO provides basic access to the nodes and edges of a constantly changing graph in data centers across multiple regions. It is optimized heavily for reads, and explicitly favors efficiency and availability over consistency.

A system like TAO is likely to be useful for any application domain that needs to efficiently generate fine-grained customized content from highly interconnected data. The application should not expect the data to be stale in the common case, but should be able to tolerate it. Many social networks fit in this category.

3 TAO Data Model and API

Facebook focuses on people, actions, and relationships. We model these entities and connections as nodes and edges in a graph. This representation is very flexible; it directly models real-life objects, and can also be used to store an application's internal implementation-specific data. TAO's goal is not to support a complete set of graph queries, but to provide sufficient expressiveness to handle most application needs while allowing a scalable and efficient implementation.

Consider the social networking example in Figure 1a, in which Alice used her mobile phone to record her visit to a famous landmark with Bob. She 'checked in' to the Golden Gate Bridge and 'tagged' Bob to indicate that he is with her. Cathy added a comment that David has 'liked.' The social graph includes the users (Alice, Bob, Cathy, and David), their relationships, their actions (checking in, commenting, and liking), and a physical location (the Golden Gate Bridge).

Facebook's application servers would query this event's underlying nodes and edges every time it is rendered. Fine-grained privacy controls mean that each user may see a different view of the checkin: the individual nodes and edges that encode the activity can be reused for all of these views, but the aggregated content and the results of privacy checks cannot.

3.1 Objects and Associations

TAO *objects* are typed nodes, and TAO *associations* are typed directed edges between objects. Objects are identified by a 64-bit integer (id) that is unique across all objects, regardless of object type (otype). Associations are identified by the source object (id1), association type (atype) and destination object (id2). At most one association of a given type can exist between any two objects. Both objects and associations may contain data as key→value pairs. A per-type schema lists the possible keys, the value type, and a default value. Each association has a 32-bit time field, which plays a central role in queries[1].

Object: (id) → (otype, (key → value)∗)
Assoc.: (id1, atype, id2) → (time, (key → value)∗)

Figure 1b shows how TAO objects and associations might encode the example, with some data and times omitted for clarity. The example's users are represented by objects, as are the checkin, the landmark, and Cathy's comment. Associations capture the users' friendships, authorship of the checkin and comment, and the binding between the checkin and its location and comments.

[1]The time field is actually a generic application-assigned integer.

Actions may be encoded either as objects or associations. Both Cathy's comment and David's 'like' represent actions taken by a user, but only the comment results in a new object. Associations naturally model actions that can happen at most once or record state transitions, such as the acceptance of an event invitation, while repeatable actions are better represented as objects.

Although associations are directed, it is common for an association to be tightly coupled with an inverse edge. In this example all of the associations have an inverse except for the link of type COMMENT. No inverse edge is required here since the application does not traverse from the comment to the CHECKIN object. Once the checkin's id is known, rendering Figure 1a only requires traversing outbound associations. Discovering the checkin object, however, requires the inbound edges or that an id is stored in another Facebook system.

The schemas for object and association types describe only the data contained in instances. They do not impose any restrictions on the edge types that can connect to a particular node type, or the node types that can terminate an edge type. The same atype is used to represent authorship of the checkin object and the comment object in Figure 1, for example. Self-edges are allowed.

3.2 Object API

TAO's object API provides operations to allocate a new object and id, and to retrieve, update, or delete the object associated with an id. A notable omission is a compare-and-set functionality, whose usefulness is substantially reduced by TAO's eventual consistency semantics. The update operation can be applied to a subset of the fields.

3.3 Association API

Many edges in the social graph are bidirectional, either symmetrically like the example's FRIEND relationship or asymmetrically like AUTHORED and AUTHORED_BY. Bidirectional edges are modeled as two separate associations. TAO provides support for keeping associations in sync with their inverses, by allowing association types to be configured with an *inverse type*. For such associations, creations, updates, and deletions are automatically coupled with an operation on the inverse association. Symmetric bidirectional types are their own inverses. The association write operations are:

- **assoc_add(id1, atype, id2, time, (k→v)*)** – Adds or overwrites the association (id1, atype,id2), and its inverse (id1, inv(atype), id2) if defined.
- **assoc_delete(id1, atype, id2)** – Deletes the association (id1, atype, id2) and the inverse if it exists.
- **assoc_change_type(id1, atype, id2, newtype)** – Changes the association (id1, atype, id2) to (id1, newtype, id2), if (id1, atype, id2) exists.

3.4 Association Query API

The starting point for any TAO association query is an originating object and an association type. This is the natural result of searching for a specific type of information about a particular object. Consider the example in Figure 1. In order to display the CHECKIN object, the application needs to enumerate all tagged users and the most recently added comments.

A characteristic of the social graph is that most of the data is old, but many of the queries are for the newest subset. This *creation-time locality* arises whenever an application focuses on recent items. If the Alice in Figure 1 is a famous celebrity then there might be thousands of comments attached to her checkin, but only the most recent ones will be rendered by default.

TAO's association queries are organized around *association lists*. We define an association list to be the list of all associations with a particular id1 and atype, arranged in descending order by the time field:

Association List: $(\mathsf{id1}, \mathsf{atype}) \rightarrow [a_{\text{new}} \ldots a_{\text{old}}]$

For example, the list $(i,$ COMMENT$)$ has edges to the example's comments about i, most recent first.

TAO's queries on associations lists:

- **assoc_get(id1, atype, id2set, high?, low?)** – returns all of the associations (id1, atype, id2) and their time and data, where id2 $\in$ id2set and high $\geq$ time $\geq$ low (if specified). The optional time bounds are to improve cacheability for large association lists (see § 5).
- **assoc_count(id1, atype)** – returns the size of the association list for (id1, atype), which is the number of edges of type atype that originate at id1.
- **assoc_range(id1, atype, pos, limit)** – returns elements of the (id1, atype) association list with index $i \in [\mathsf{pos}, \mathsf{pos}+\mathsf{limit})$.
- **assoc_time_range(id1, atype, high, low, limit)** – returns elements from the (id1, atype) association list, starting with the first association where time $\leq$ high, returning only edges where time $\geq$ low.

TAO enforces a per-atype upper bound (typically 6,000) on the actual limit used for an association query. To enumerate the elements of a longer association list the client must issue multiple queries, using pos or high to specify a starting point.

For the example shown in Figure 1 we can map some possible queries to the TAO API as follows:

- "50 most recent comments on Alice's checkin" ⇒ assoc_range(632, COMMENT, 0, 50)
- "How many checkins at the GG Bridge?" ⇒ assoc_count(534, CHECKIN)

4 TAO Architecture

In this section we describe the units that make up TAO, and the multiple layers of aggregation that allow it to scale across data centers and geographic regions. TAO is separated into two caching layers and a storage layer.

4.1 Storage Layer

Objects and associations were stored in MySQL at Facebook even before TAO was built; it was the backing store for the original PHP implementation of the API. This made it the natural choice for TAO's persistent storage.

The TAO API is mapped to a small set of simple SQL queries, but it could also be mapped efficiently to range scans in a non-SQL data storage system such as LevelDB [3] by explicitly maintaining the required indexes. When evaluating the suitability of a backing store for TAO, however, it is important to consider the data accesses that don't use the API. These include backups, bulk import and deletion of data, bulk migrations from one data format to another, replica creation, asynchronous replication, consistency monitoring tools, and operational debugging. An alternate store would also have to provide atomic write transactions, efficient granular writes, and few latency outliers.

Given that TAO needs to handle a far larger volume of data than can be stored on a single MySQL server, we divide data into logical *shards*. Each shard is contained in a logical database. Database servers are responsible for one or more shards. In practice, the number of shards far exceeds the number of servers; we tune the shard to server mapping to balance load across different hosts. By default all object types are stored in one table, and all association types in another.

Each object id contains an embedded shard_id that identifies its hosting shard. Objects are bound to a shard for their entire lifetime. An association is stored on the shard of its id1, so that every association query can be served from a single server. Two ids are unlikely to map to the same server unless they were explicitly colocated at creation time.

4.2 Caching Layer

TAO's cache implements the complete API for clients, handling all communication with databases. The caching layer consists of multiple *cache servers* that together form a *tier*. A tier is collectively capable of responding to any TAO request. (We also refer to the set of databases in one region as a tier.) Each request maps to a single cache server using a sharding scheme similar to the one described in § 4.1. There is no requirement that tiers have the same number of hosts.

Clients issue requests directly to the appropriate cache server, which is then responsible for completing the read or write. For cache misses and write requests, the server contacts other caches and/or databases.

The TAO in-memory cache contains objects, association lists, and association counts. We fill the cache on demand and evict items using a least recently used (LRU) policy. Cache servers understand the semantics of their contents and use them to answer queries even if the exact query has not been previously processed, e.g. a cached count of zero is sufficient to answer a range query.

Write operations on an association with an inverse may involve two shards, since the forward edge is stored on the shard for id1 and the inverse edge is on the shard for id2. The tier member that receives the query from the client issues an RPC call to the member hosting id2, which will contact the database to create the inverse association. Once the inverse write is complete, the caching server issues a write to the database for id1. TAO does not provide atomicity between the two updates. If a failure occurs the forward may exist without an inverse; these *hanging* associations are scheduled for repair by an asynchronous job.

4.3 Client Communication Stack

It is common for hundreds of objects and associations to be queried while rendering a Facebook page, which is likely to require communication with many cache servers in a short period of time. The challenges of the resulting all-to-all communication are similar to those faced by our memcache pools. TAO and memcache share most of the client stack described by Nishtala et al. [21]. The latency of TAO requests can be much higher than those of memcache, because TAO requests may access the database, so to avoid head-of-line blocking on multiplexed connections we use a protocol with out-of-order responses.

4.4 Leaders and Followers

In theory a single cache tier could be scaled to handle any foreseeable aggregate request rate, so long as shards are small enough. In practice, though, large tiers are problematic because they are more prone to hot spots and they have a quadratic growth in all-to-all connections.

To add servers while limiting the maximum tier size we split the cache into two levels: a *leader* tier and multiple *follower* tiers. Some of TAO's advantages over a lookaside cache architecture (as described in § 2.1) rely on having a single cache coordinator per database; this split allows us to keep the coordinators in a single tier per region. As in the single-tier configuration, each tier contains a set of cache servers that together are capable of responding to any TAO query; that is, every shard in the system maps to one caching server in each tier. Leaders (members of the leader tier) behave as described in § 4.2, reading from and writing to the storage layer. Fol-

lowers (members of follower tiers) will instead forward read misses and writes to a leader. Clients communicate with the closest follower tier and never contact leaders directly; if the closest follower is unavailable they fail over to another nearby follower tier.

Given this two-level caching hierarchy, care must be taken to keep TAO caches consistent. Each shard is hosted by one leader, and all writes to the shard go through that leader, so it is naturally consistent. Followers, on the other hand, must be explicitly notified of updates made via other follower tiers.

TAO provides eventual consistency [33, 35] by asynchronously sending cache maintenance messages from the leader to the followers. An object update in the leader enqueues invalidation messages to each corresponding follower. The follower that issued the write is updated synchronously on reply from the leader; a version number in the cache maintenance message allows it to be ignored when it arrives later. Since we cache only contiguous prefixes of association lists, invalidating an association might truncate the list and discard many edges. Instead, the leader sends a *refill* message to notify followers about an association write. If a follower has cached the association, then the refill request triggers a query to the leader to update the follower's now-stale association list. § 6.1 discusses the consistency of this design and also how it tolerates failures.

Leaders serialize concurrent writes that arrive from followers. Because a single leader mediates all of the requests for an id1, it is also ideally positioned to protect the database from thundering herds. The leader ensures that it does not issue concurrent overlapping queries to the database and also enforces a limit on the maximum number of pending queries to a shard.

4.5 Scaling Geographically

The leader and followers configuration allows TAO to scale to handle a high workload, since read throughput scales with the total number of follower servers in all tiers. Implicit in the design, however, is the assumption that the network latencies from follower to leader and leader to database are low. This assumption is reasonable if clients are restricted to a single data center, or even to a set of data centers in close proximity. It is not true, however, in our production environment.

As our social networking application's computing and network requirements have grown, we have had to expand beyond a single geographical location: today, follower tiers can be thousands of miles apart. In this configuration, network round trip times can quickly become the bottleneck of the overall architecture. Since read misses by followers are 25 times as frequent as writes in our workloads, we chose a master/slave architecture that requires writes to be sent to the master, but that allows

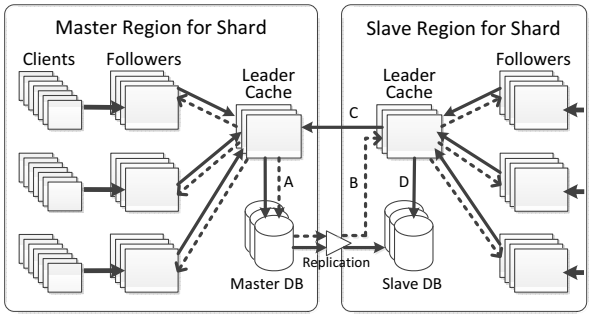


Figure 2: Multi-region TAO configuration. The master region sends read misses, writes, and embedded consistency messages to the master database (A). Consistency messages are delivered to the slave leader (B) as the replication stream updates the slave database. Slave leader sends writes to the master leader (C) and read misses to the replica DB (D). The choice of master and slave is made separately for each shard.

read misses to be serviced locally. As with the leader/-follower design, we propagate update notifications asynchronously to maximize performance and availability, at the expense of data freshness.

The social graph is tightly interconnected; it is not possible to group users so that cross-partition requests are rare. This means that each TAO follower must be local to a tier of databases holding a complete multi-petabyte copy of the social graph. It would be prohibitively expensive to provide full replicas in every data center.

Our solution to this problem is to choose data center locations that are clustered into only a few *regions*, where the intra-region latency is small (typically less than 1 millisecond). It is then sufficient to store one complete copy of the social graph per region. Figure 2 shows the overall architecture of the master/slave TAO system.

Followers behave identically in all regions, forwarding read misses and writes to the local region's leader tier. Leaders query the local region's database regardless of whether it is the master or slave. Writes, however, are forwarded by the local leader to the leader that is in the region with the master database. This means that read latency is independent of inter-region latency.

The master region is controlled separately for each shard, and is automatically switched to recover from the failure of a database. Writes that fail during the switch are reported to the client as failed, and are not retried. Note that since each cache hosts multiple shards, a server may be both a master and a slave at the same time. We prefer to locate all of the master databases in a single region. When an inverse association is mastered in a different region, TAO must traverse an extra inter-region link to forward the inverse write.

TAO embeds invalidation and refill messages in the database replication stream. These messages are delivered in a region immediately after a transaction has been replicated to a slave database. Delivering such messages earlier would create cache inconsistencies, as reading from the local database would provide stale data. At Facebook TAO and memcache use the same pipeline for delivery of invalidations and refills [21].

If a forwarded write is successful then the local leader will update its cache with the fresh value, even though the local slave database probably has not yet been updated by the asynchronous replication stream. In this case followers will receive two invalidates or refills from the write, one that is sent when the write succeeds and one that is sent when the write's transaction is replicated to the local slave database.

TAO's master/slave design ensures that all reads can be satisfied within a single region, at the expense of potentially returning stale data to clients. As long as a user consistently queries the same follower tier, the user will typically have a consistent view of TAO state. We discuss exceptions to this in the next section.

5 Implementation

Previous sections describe how TAO servers are aggregated to handle large volumes of data and query rates. This section details important optimizations for performance and storage efficiency.

5.1 Caching Servers

TAO's caching layer serves as an intermediary between clients and the databases. It aggressively caches objects and associations to provide good read performance.

TAO's memory management is based on Facebook's customized memcached, as described by Nishtala et al. [21]. TAO has a slab allocator that manages slabs of equal size items, a thread-safe hash table, LRU eviction among items of equal size, and a dynamic slab rebalancer that keeps the LRU eviction ages similar across all types of slabs. A slab item can hold one node or one edge list.

To provide better isolation, TAO partitions the available RAM into *arenas*, selecting the arena by the object or association type. This allows us to extend the cache lifetime of important types, or to prevent poor cache citizens from evicting the data of better-behaved types. So far we have only manually configured arenas to address specific problems, but it should be possible to automatically size arenas to improve TAO's overall hit rate.

For small fixed-size items, such as association counts, the memory overhead of the pointers for bucket items in the main hash table becomes significant. We store these items separately, using direct-mapped 8-way associative caches that require no pointers. LRU order within each bucket is tracked by simply sliding the entries down. We achieve additional memory efficiency by adding a table that maps the each active atype to a 16 bit value. This lets us map (id1, atype) to a 32-bit count in 14 bytes; a negative entry, which records the absence of any id2 for an (id1, atype), takes only 10 bytes. This optimization allows us to hold about 20% more items in cache for a given system configuration.

5.2 MySQL Mapping

Recall that we divide the space of objects and associations into shards. Each shard is assigned to a logical MySQL database that has a table for objects and a table for associations. All of the fields of an object are serialized into a single 'data' column. This approach allows us to store objects of different types within the same table, Objects that benefit from separate data management polices are stored in separate custom tables.

Associations are stored similarly to objects, but to support range queries, their tables have an additional index based on id1, atype, and time. To avoid potentially expensive SELECT COUNT queries, association counts are stored in a separate table.

5.3 Cache Sharding and Hot Spots

Shards are mapped onto cache servers within a tier using consistent hashing [15]. This simplifies tier expansions and request routing. However, this semi-random assignment of shards to cache servers can lead to load imbalance: some followers will shoulder a larger portion of the request load than others. TAO rebalances load among followers with *shard cloning*, in which reads to a shard are served by multiple followers in a tier. Consistency management messages for a cloned shard are sent to all followers hosting that shard.

In our workloads, it is not uncommon for a popular object to be queried orders of magnitude more often than other objects. Cloning can distribute this load across many followers, but the high hit rate for these objects makes it worthwhile to place them in a small client-side cache. When a follower responds to a query for a hot item, it includes the object or association's access rate. If the access rate exceeds a certain threshold, the TAO client caches the data and version. By including the version number in subsequent queries, the follower can omit the data in replies if the data has not changed since the previous version. The access rate can also be used to throttle client requests for very hot objects.

5.4 High-Degree Objects

Many objects have more than 6,000 associations with the same atype emanating from them, so TAO does not cache

the complete association list. It is also common that assoc_get queries are performed that have an empty result (no edge exists between the specified id1 and id2). Unfortunately, for high-degree objects these queries will always go to the database, because the queried id2 could be in the uncached tail of the association list.

We have addressed this inefficiency in the cache implementation by modifying client code that is observed to issue problematic queries. One solution to this problem is to use assoc_count to choose the query direction, since checking for the inverse edge is equivalent. In some cases where both ends of an edges are high-degree nodes, we can also leverage application-domain knowledge to improve cacheability. Many associations set the time field to their creation time, and many objects include their creation time as a field. Since an edge to a node can only be created after the node has been created, we can limit the id2 search to associations whose time is $\geq$ than the object's creation time. So long as an edge older than the object is present in cache then this query can be answered directly by a TAO follower.

6 Consistency and Fault Tolerance

Two of the most important requirements for TAO are availability and performance. When failures occur we would like to continue to render Facebook, even if the data is stale. In this section, we describe the consistency model of TAO under normal operation, and how TAO sacrifices consistency under failure modes.

6.1 Consistency

Under normal operation, objects and associations in TAO are eventually consistent [33, 35]; after a write, TAO guarantees the eventual delivery of an invalidation or refill to all tiers. Given a sufficient period of time during which external inputs have quiesced, all copies of data in TAO will be consistent and reflect all successful write operations to all objects and associations. Replication lag is usually less than one second.

In normal operation (at most one failure encountered by a request) TAO provides read-after-write consistency within a single tier. TAO synchronously updates the cache with locally written values by having the master leader return a *changeset* when the write is successful. This changeset is propagated through the slave leader (if any) to the follower tier that originated the write query. If an inverse type is configured for an association, then writes to associations of that type may affect both the id1's and the id2's shard. In these cases, the changeset returned by the master leader contains both updates, and the slave leader (if any) and the follower that forwarded the write must each send the changeset to the id2's shard in their respective tiers, before returning to the caller.

The changeset cannot always be safely applied to the follower's cache contents, because the follower's cache may be stale if the refill or invalidate from a second follower's update has not yet been delivered. We resolve this race condition in most cases with a version number that is present in the persistent store and the cache. The version number is incremented during each update, so the follower can safely invalidate its local copy of the data if the changeset indicates that its pre-update value was stale. Version numbers are not exposed to the TAO clients. In slave regions, this scheme is vulnerable to a rare race condition between cache eviction and storage server update propagation. The slave storage server may hold an older version of a piece of data than what is cached by the caching server, so if the post-changeset entry is evicted from cache and then reloaded from the database, a client may observe a value go back in time in a single follower tier. Such a situation can only occur if it takes longer for the slave region's storage server to receive an update than it does for a cached item to be evicted from cache, which is rare in practice.

Although TAO does not provide strong consistency for its clients, because it writes to MySQL synchronously the master database is a consistent source of truth. This allows us to provide stronger consistency for the small subset of requests that need it. TAO reads may be marked *critical*, in which case they will be proxied to the master region. We could use critical reads during an authentication process, for example, so that replication lag doesn't allow use of stale credentials.

6.2 Failure Detection and Handling

TAO scales to thousands of machines over multiple geographical locations, so transient and permanent failures are commonplace. Therefore, it is important that TAO detect potential failures and route around them. TAO servers employ aggressive network timeouts so as not to continue waiting on responses that may never arrive. Each TAO server maintains per-destination timeouts, marking hosts as down if there are several consecutive timeouts, and remembering downed hosts so that subsequent requests can be proactively aborted. This simple failure detector works well, although it does not always preserve full capacity in a brown-out scenario, such as bursty packet drops that limit TCP throughput. Upon detection of a failed server, TAO routes around the failures in a best effort fashion in order to preserve availability and performance at the cost of consistency. We actively probe failed machines to discover when (if) they recover.

Database failures: Databases are marked down in a global configuration if they crash, if they are taken offline for maintenance, or if they are replicating from a master database and they get too far behind. When a

master database is down, one of its slaves is automatically promoted to be the new master.

When a region's slave database is down, cache misses are redirected to the TAO leaders in the region hosting the database master. Since cache consistency messages are embedded in the database's replication stream, however, they can't be delivered by the primary mechanism. During the time that a slave database is down an additional binlog tailer is run on the master database, and the refills and invalidates are delivered inter-regionally. When the slave database comes back up, invalidation and refill messages from the outage will be delivered again.

Leader failures: When a leader cache server fails, followers automatically route read and write requests around it. Followers reroute read misses directly to the database. Writes to a failed leader, in contrast, are rerouted to a random member of the leader's tier. This replacement leader performs the write and associated actions, such as modifying the inverse association and sending invalidations to followers. The replacement leader also enqueues an asynchronous invalidation to the original leader that will restore its consistency. These asynchronous invalidates are recorded both on the coordinating node and inserted into the replication stream, where they are spooled until the leader becomes available. If the failing leader is partially available then followers may see a stale value until the leader's consistency is restored.

Refill and invalidation failures: Leaders send refills and invalidations asynchronously. If a follower is unreachable, the leader queues the message to disk to be delivered at a later time. Note that a follower may be left with stale data if these messages are lost due to permanent leader failure. This problem is solved by a bulk invalidation operation that invalidates all objects and associations from a shard_id. After a failed leader box is replaced, all of the shards that map to it must be invalidated in the followers, to restore consistency.

Follower failures: In the event that a TAO follower fails, followers in other tiers share the responsibility of serving the failed host's shards. We configure each TAO client with a primary and backup follower tier. In normal operations requests are sent only to the primary. If the server that hosts the shard for a particular request has been marked down due to timeouts, then the request is sent instead to that shard's server in the backup tier. Because failover requests still go to a server that hosts the corresponding shard, they are fully cacheable and do not require extra consistency work. Read and write requests from the client are failed over in the same way. Note that failing over between different tiers may cause read-after-write consistency to be violated if the read reaches the failover target before the write's refill or invalidate.

read requests	**99.8 %**	**write requests**	**0.2 %**
assoc_get	15.7 %	assoc_add	52.5 %
assoc_range	40.9 %	assoc_del	8.3 %
assoc_time_range	2.8 %	assoc_change_type	0.9 %
assoc_count	11.7 %	obj_add	16.5 %
obj_get	28.9 %	obj_update	20.7 %
		obj_delete	2.0 %

Figure 3: Relative frequencies for client requests to TAO from all Facebook products. Reads account for almost all of the calls to the API.

7 Production Workload

Facebook has a single instance of TAO in production. Multi-tenancy in a system such as TAO allows us to amortize operational costs and share excess capacity among clients. It is also an important enabler for rapid product innovation, because new applications can link to existing data and there is no need to move data or provision servers as an application grows from one user to hundreds of millions. Multi-tenancy is especially important for objects, because it allows the entire 64-bit id space to be handled uniformly without an extra step to resolve the otype.

The TAO system contains many follower tiers spread across several geographic regions. Each region has one complete set of databases, one leader cache tier, and at least two follower tiers. Our TAO deployment continuously processes a billion reads and millions of writes per second. We are not aware of another geographically distributed graph data store at this scale.

To characterize the workload that is seen by TAO, we captured a random sample of 6.5 million requests over a 40 day period. In this section, we describe the results of an analysis of that sample.

At a high level, our workload shows the following characteristics:

- reads are much more frequent than writes;
- most edge queries have empty results; and
- query frequency, node connectivity, and data size have distributions with long tails.

Figure 3 breaks down the load on TAO. Reads dominate, with only 0.2% of requests involving a write. The majority of association reads resulted in empty association lists. Calls to assoc_get found an association only 19.6% of the time, 31.0% of the calls to assoc_range in our trace had a non-empty result, and only 1.9% of the calls to assoc_time_range returned any edges.

Figure 4 shows the distribution of the return values from assoc_count. 45% of calls return zero. Among the non-zero values, although small values are the most common, 1% of the return values were $> 500,000$.

Figure 5 shows the distribution of the number of asso-

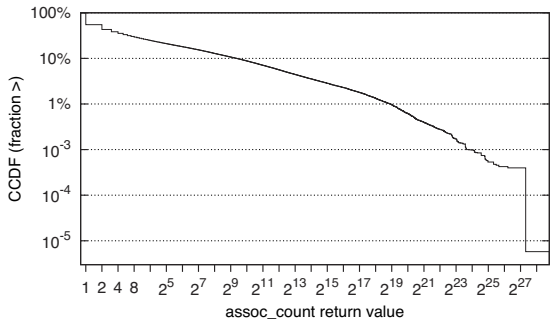


Figure 4: assoc_count frequency in our production environment. 1% of returned counts were ≥512K.

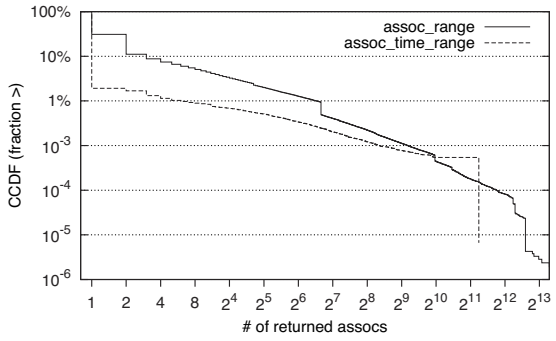


Figure 5: The number of edges returned by assoc_range and assoc_time_range queries. 64% of the non-empty results had 1 edge, 13% of which had a limit of 1.

ciations returned for range and time-range queries, and the subset that hit the limit for returned associations. Most range and time range queries had large client-supplied limits. 12% of the queries had limit $= 1$, but 95% of the remaining queries had limit ≥ 1000. Less than 1% of the return values for queries with a limit ≥ 1 actually reached the limit.

Although queries for non-existent associations were common, this is not the case for objects. A valid id is only produced during object creation, so obj_get can only return an empty result if the object has been removed or if the object's creation has not yet been replicated to the current region. Neither of these cases occurred in our trace; every object read was successful. This doesn't mean that objects were never deleted – it just means that there was never an attempt to read a deleted object.

Figure 6 shows the distribution of the data sizes for TAO query results. 39.5% of the associations queried by clients contained no data. Our implementation allows objects to store 1MB of data and associations to store 64K of data (although a custom table must be configured for associations that store more than 255 bytes of data). The actual size of most objects and associations is much

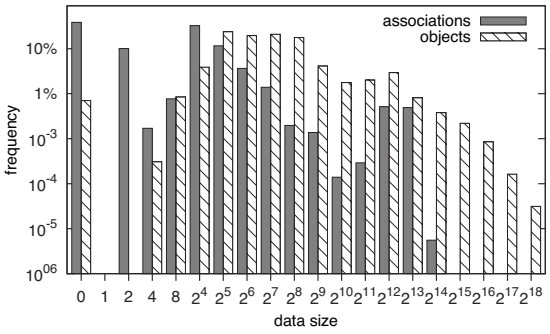


Figure 6: The size of the data stored in associations and objects that were returned by the TAO API. Associations typically store much less data than objects. The average association data size was 97.8 bytes for the 60.5% of returned associations that had some data. The average object data size was 673 bytes.

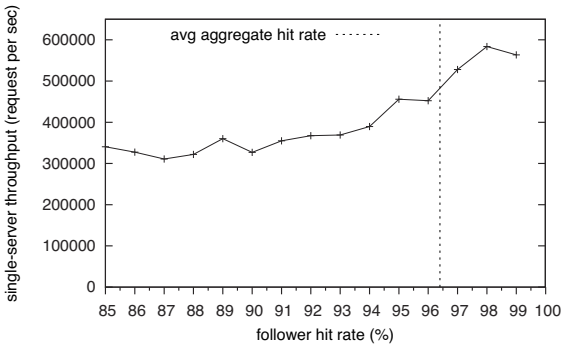


Figure 7: Throughput of an individual follower in our production environment. Cache misses and writes are more expensive than cache hits, so the peak query rate rises with hit rate. Writes are included in this graph as non-hit requests.

smaller. However, large values are frequent enough that the system must deal with them efficiently.

8 Performance

Running a single TAO deployment for all of Facebook allows us to benefit from economies of scale, and makes it easy for new products to integrate with existing portions of the social graph. In this section, we report on the performance of TAO under a real workload.

Availability: Over a period of 90 days, the fraction of failed TAO queries as measured from the web server was 4.9×10^{-6}. Care must be taken when interpreting this number, since the failure of one TAO query might prevent the client from issuing another query with a dynamic data dependence on the first. TAO's failures may also be correlated with those of other dependent systems.

	hit lat. (msec)			miss lat. (msec)		
operation	*50%*	*avg*	*99%*	*50%*	*avg*	*99%*
assoc_count	1.1	2.5	28.9	5.0	26.2	186.8
assoc_get	1.0	2.4	25.9	5.8	14.5	143.1
assoc_range	1.1	2.3	24.8	5.4	11.2	93.6
assoc_time_range	1.3	3.2	32.8	5.8	11.9	47.2
obj_get	1.0	2.4	27.0	8.2	75.3	186.4

Figure 8: Client-observed TAO latency in milliseconds for read requests, including client API overheads and network traversal, separated by cache hits and cache misses.

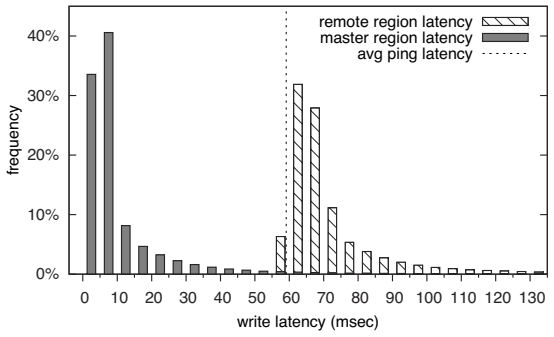


Figure 9: Write latency from clients in the same region as database masters, and from a region 58 msec away.

Follower capacity: The peak throughput of a follower depends on its hit rate. Figure 7 shows the highest 15-minute average throughput we observe in production for our current hardware configuration, which has 144GB of RAM, 2 Intel Xeon 8 core E5-2660 CPUs running at 2.2Ghz with Hyperthreading, and 10 Gigabit ethernet.

Hit rates and latency: As part of the data collection process that was described in § 7, we measured latencies in the client application; these measurements include all network latencies and the time taken to traverse the PHP TAO client stack. Requests were sampled at the same rate in all regions. TAO's overall hit rate for reads was 96.4%. Figure 8 shows the client-observed latencies for reads. obj_get has higher miss latencies than the other reads because objects typically have more data (see Figure 6). assoc_count requests to the persistent store have a larger id1 working set than other association queries, and hence make poorer use of the database's buffer cache.

TAO's writes are performed synchronously to the master database, so writes from other regions include an inter-region round trip. Figure 9 compares the latency in two data centers that are 58.1 milliseconds away from each other (average round trip). Average write latency in the same region as the master was 12.1 msec; in the remote region it was 74.4 = 58.1 + 16.3 msec.

Replication lag: TAO's asynchronous replication of writes between regions is a design trade-off that favors read performance and throughput over consistency. We observed that TAO's slave storage servers lag their master by less than 1 second during 85% of the tracing window, by less than 3 seconds 99% of the time, and by less than 10 seconds 99.8% of the time.

Failover: Follower caches directly contact the database when a leader is unavailable; this failover path was used on 0.15% of follower cache misses over our sample. Failover for write requests involves delegating those requests to a random leader, which occurred for 0.045% of association and object writes. Slave databases were promoted to be the master 0.25% of the time due to planned maintenance or unplanned downtime.

9 Related Work

TAO is a geographically distributed eventually consistent graph store optimized for reads. Previous distributed systems works have explored relaxed consistency, graph databases, and read-optimized storage. To our knowledge, TAO is the first to combine all of these techniques in a single system at large scale.

Eventual consistency: Terry et al. [33] describe eventual consistency, the relaxed consistency model which is used by TAO. Werner describes read-after-write consistency as a property of some variants of eventual consistency [35].

Geographically distributed data stores: The Coda file system uses data replication to improve performance and availability in the face of slow or unreliable networks [29]. Unlike Coda, TAO does not allow writes in portions of the system that are disconnected.

Megastore is a storage system that uses Paxos across geographically distributed data centers to provide strong consistency guarantees and high availability [5]. Spanner, the next generation globally distributed database developed at Google after Megastore, introduces the concept of a time API that exposes time uncertainty and leverages that to improve commit throughput and provide snapshot isolation for reads [8]. TAO addresses a very different use case, providing no consistency guarantees but handling many orders of magnitude more requests.

Distributed hash tables and key-value systems: Unstructured key-value systems are an attractive approach to scaling distributed storage because data can be easily partitioned and little communication is needed between partitions. Amazon's Dynamo [10] demonstrates how they can be used in building flexible and robust commercial systems. Drawing inspiration from Dynamo, LinkedIn's Voldemort [4] also implements a distributed key-value store but for a social network. TAO accepts lower write availability than Dynamo in exchange for avoiding the programming complexities that arise from multi-master conflict resolution. The simplicity of key-

value stores also allows for aggressive performance optimizations, as seen in Facebook's use of memcache [21].

Many contributions in distributed hash tables have focused on routing [28, 32, 25, 24]. Li et al. [16] characterize the performance of DHTs under churn while Dabek et al. [9] focus on designing DHTs in a wide-area network. TAO exploits the hierarchy of inter-cluster latencies afforded by our data center placement and assumes a controlled environment that has few membership or cluster topology changes.

Many other works have focused on the consistency semantics provided by key-value stores. Gribble et al. [13] provide a coherent view of cached data by leveraging two-phase commit. Glendenning et al. [12] built a linearizable key-value store tolerant of churn. Sovran et al. [31] implement geo-replicated transactions.

The COPS system [17] provides causal consistency in a highly available key-value store by tracking all dependencies for all keys accessed by a client context. Eiger [18] improves on COPS by tracking conflicts between pending operations in a column-family database. The techniques used in Eiger may be applicable TAO if the per-machine efficiency can be improved.

Hierarchical connectivity: Nygren et al. [22] describe how the Akamai content cache optimizes latency by grouping edge clusters into regional groups that share a more powerful 'parent' cluster, which are similar to TAO's follower and leader tiers.

Structured storage: TAO follows the recent trend of shifting away from relational databases towards structured storage approaches. While loosely defined, these systems typically provide weaker guarantees than the traditional ACID properties. Google's BigTable [6], Yahoo!'s PNUTS [7], Amazon's SimpleDB [1], and Apache's HBase [34] are examples of this more scalable approach. These systems all provide consistency and transactions at the per-record or row level similar to TAO's semantics for objects and associations, but do not provide TAO's read efficiency or graph semantics. Escriva et al. [27] describe a *searchable* key-value store. Redis [26] is an in-memory storage system providing a range of data types and an expressive API for data sets that fit entirely in memory.

Graph serving: Since TAO was designed specifically to serve the social graph, it is unsurprising that it shares features with existing works on graph databases. Shao and Wang's Trinity effort [30] stores its graph structures in-memory. Neo4j [20] is a popular open-source graph database that provides ACID semantics and the ability to shard data across several machines. Twitter uses its FlockDB [11] to store parts of its social graph, as well. To the best our knowledge, none of these systems scale to support Facebook's workload.

Redis [26] is a key-value store with a rich selection of value types sufficient to efficiently implement the objects and associations API. Unlike TAO, however, it requires that the data set fit entirely in memory. Redis replicas are read-only, so they don't provide read-after-write consistency without a higher-level system like Nishtala et al.'s remote markers [21].

Graph processing: TAO does not currently support an advanced graph processing API. There are several systems that try to support such operations but they are not designed to receive workloads directly from client applications. PEGASUS [14] and Yahoo's Pig Latin [23] are systems to do data mining and analysis of graphs on top of Hadoop, with PEGASUS being focused on peta-scale graphs and Pig Latin focusing on a more-expressive query language. Similarly, Google's Pregel [19] tackles a lot of the same graph analysis issues but uses its own more-expressive job distribution model. These systems focus on throughput for large tasks, rather than a high volume of updates and simple queries. Facebook has similar large-scale offline graph-processing systems that operate on data copied from TAO's databases, but these analysis jobs do not execute within TAO itself.

10 Conclusion

Overall, this paper makes three contributions. First, we characterize a challenging Facebook workload: queries that require high throughput, low latency read access to the large, changing social graph. Second, we describe the objects and associations data model for Facebook's social graph, and the API that serves it. Lastly, we detail TAO, our geographically distributed system that implements this API.

TAO is deployed at scale inside Facebook. Its separation of cache and persistent store has allowed those layers to be independently designed, scaled, and operated, and maximizes the reuse of components across our organization. This separation also allows us to choose different tradeoffs for efficiency and consistency at the two layers, and to use an idempotent cache invalidation strategy. TAO's restricted data and consistency model has proven to be usable for our application developers while allowing an efficient and highly available implementation.

Acknowledgements

We would like to thank Rajesh Nishtala, Tony Savor, and Barnaby Thieme for reading earlier versions of this paper and contributing many improvements. We thank the many Facebook engineers who built, used, and scaled the original implementation of the objects and associations API for providing us with the design insights and workload that led to TAO. Thanks also to our reviewers and our shepherd Phillipa Gill for their detailed comments.

References

[1] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[2] Facebook – Company Info. http://newsroom.fb.com.

[3] LevelDB. https://code.google.com/p/leveldb.

[4] Project Voldemort. http://project-voldemort.com/.

[5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research*, CIDR, 2011.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, OSDI. USENIX Assoc., 2006.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI, Berkeley, CA, USA, 2012.

[9] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2004.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 2007.

[11] FlockDB. http://engineering.twitter.com/2010/05/introducing-flockdb.html.

[12] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 2011.

[13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, OSDI, Berkeley, CA, USA, 2000.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowledge Information Systems*, 27(2), 2011.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing*, STOC, 1997.

[16] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the Third International Conference on Peer-to-Peer Systems*, IPTPS, Berlin, Heidelberg, 2004.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In T. Wobber and P. Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating System Design and Implementation*, SOSP. ACM, 2011.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2013.

[19] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2010.

[20] Neo4j. http://neo4j.org/.

[21] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2013.

[22] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: a platform for high-performance internet applications. *SIGOPS Operating Systems Review*, 44(3), Aug. 2010.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In J. T.-L. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2008.

[24] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2004.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM, New York, NY, USA, 2001.

[26] Redis. http://redis.io/.

[27] E. G. S. Robert Escriva, Bernard Wong. Hyperdex: A distributed, searchable key-value store for cloud computing. Technical report, Department of Computer Science, Cornell University, Ithaca, New York, December 2011.

[28] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware, London, UK, UK, 2001.

[29] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems*, 20(2), May 2002.

[30] B. Shao and H. Wang. Trinity. http://research.microsoft.com/en-us/projects/trinity/.

[31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 2011.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM, New York, NY, USA, 2001.

[33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 1995.

[34] The Apache Software Foundation. http://hbase.apache.org, 2010.

[35] W. Vogels. Eventually consistent. *Queue*, 6(6), Oct. 2008.

PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters

Rohan Gandhi, Di Xie, Y. Charlie Hu
Purdue University

Abstract

For power, cost, and pricing reasons, datacenters are evolving towards heterogeneous hardware. However, MapReduce implementations, which power a representative class of datacenter applications, were originally designed for homogeneous clusters and performed poorly on heterogeneous clusters. The natural solution, rebalancing load among the reducers running on heterogeneous nodes has been explored in Tarazu, but shown to be only mildly effective.

In this paper, we revisit the key design challenge in this important optimization for MapReduce on heterogeneous clusters and make three contributions. (1) We show that Tarazu estimates the target load distribution too early into MapReduce job execution, which results in the rebalanced load far from the optimal. (2) We articulate the delicate tradeoff between the estimation accuracy versus wasted work from delayed load adjustment, and propose a load rebalancing scheme that strikes a balance between the tradeoff. (3) We implement our design in the PIKACHU task scheduler, which outperforms Hadoop by up to 42% and Tarazu by up to 23%.

1 Introduction

For power, cost, and pricing reasons, datacenters have evolved towards heterogeneous hardware. For example, different hardware generations exist in Amazon EC2 [1] due to phased hardware upgrades over the years. Heterogeneity also arises due to other factors including special hardware such as GPUs, unequal creation of instances, and background load variation [18, 11, 15].

MapReduce, a high-level programming model for data-intensive applications [10], has been widely adopted in cloud datacenters such as Google, Yahoo, Microsoft, and Facebook [7, 8, 17, 9], to power a significant portion of applications. However, the numerous MapReduce implementations have been designed and optimized for homogeneous clusters. A recent study [6] has shown that contemporary MapReduce implementations can perform extremely poorly on heterogeneous clusters.

The same study characterized how heterogeneous hardware, *i.e.,* mix of fast and slow nodes, adversely affects the performance of MapReduce frameworks into two primary effects. (1) Map-side effect: The built-in load balance of map tasks leads to faster nodes stealing tasks from slow nodes, which can greatly increase the network load which in turn can coincide with and slow down the subsequent network-intensive shuffle phase. (2) Reduce-side effect: MapReduce implementations assume homogeneous nodes and distribute the keys equally among reduce tasks. Such distribution leads to disparate progress on fast and slow nodes in heterogeneous clusters, and contributes to prolonged job completion time.

In [6], the authors proposed Tarazu, a suite of optimizations for heterogeneous clusters. For map-side effect, it adaptively allows task stealing from slow nodes and interleaving map tasks with shuffling on fast nodes. For reduce-side effect, it explores the natural solution, *i.e.,* rebalancing load between reducers running on fast and slow nodes. In particular, it estimates the target load split between fast and slow nodes, *i.e.,* key range partitions, right before the start of the reduce tasks, based on the relative progress rates of map tasks running on the fast and slow nodes so far. Evaluation results in [6] however show the simple load rebalancing scheme is only mildly effective, and can even degrade job performance from inaccurate key distribution estimation.

In this paper, we revisit the key design challenge in this important optimization for MapReduce on heterogeneous clusters: load rebalancing among reduce tasks to even out their completion time. We make three concrete contributions. First, we show that the relative progress rates of map tasks on fast and slow nodes often do not give a good indication of the relative progress rates of reduce tasks on heterogeneous nodes due to different resource requirement, and hence estimating the target reducer load distribution before reduce tasks start can result in the adjusted load being far from well-balanced.

Second, we explore the design space and articulate the tradeoff between the estimation accuracy versus wasted work from delayed load adjustment, and propose a load rebalancing scheme that strikes a balance between the two factors. We show an estimator that simply peeks into the initial relative progress rates of reduce tasks can still incur estimator error, because reducers on fast and slow nodes can have different room for increased resource utilization. Our final design captures this additional intricacy using observed reducer CPU utilization on fast and slow nodes to accurately estimate the target load split.

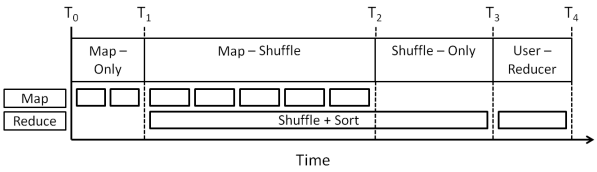


Figure 1: Four stages and their concurrency in MapReduce job execution.

Finally, we implement our new load rebalancing scheme in the PIKACHU task scheduler, and experimentally show it substantially outperforms Tarazu and Hadoop, reducing the job completion time by up to 23% and 42%, respectively, for a diverse set of benchmarks and cluster configurations.

2 Background

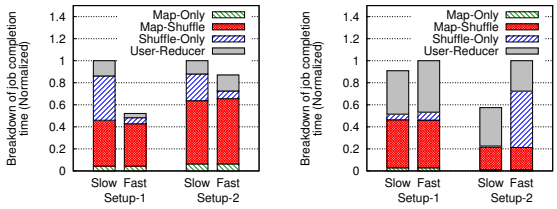
The execution of a MapReduce job is broken down by the runtime system into many Map tasks and Reduce tasks (called reducers hereafter) running in parallel on different nodes of the cluster. A reducer consists of three types of subtasks: (1) shuffle, (2) sort, and (3) user-defined reduce function. Every node in the cluster has a fixed number of map and reduce slots, and the scheduler assigns a task whenever a slot frees up.

Four stages of MapReduce execution in Hadoop. To help illustrate of the impact of heterogeneous hardware on MapReduce performance, we divide the execution of a MapReduce job in Hadoop into four distinct stages in the time dimension, as shown in Figure 1. (1) **Map-Only**: In this stage, only map tasks are running across the nodes in the cluster; the reducer is yet to begin. (2) **Map-Shuffle**: This stage starts when the reduce tasks start to run (T_1 in Figure 1). The start time for reducers is configurable, but is typically set to be when the first wave of map tasks is finished, *i.e.,* at least one map task is finished on all nodes. In this stage, the reduce task continuously performs staggered shuffle and sort [1] (or simply shuffle-sort hereafter) to digest the output of each wave of map tasks. Effective, map tasks and shuffle-sort are running concurrently on all nodes. (3) **Shuffle-Only**: This stage begins when all map tasks are finished (time T_2) but the shuffle-sort phases of the reducers are yet to be finished. In this stage, only the shuffle and sort tasks are running concurrently. (4) **User-Reducer**: This stage begins when all the data have been shuffled and sorted, and only the user-defined reducer function executes. Finally, the job is said to be finished when the user-defined reducer function is finished on all the nodes.

3 Impact of Heterogeneity

The scheduler of MapReduce implementations, *e.g.,* Hadoop, however, does not consider heterogeneity, which results in poor application performance on heterogeneous clusters. Using testbed measurement, we dissect the impact of hardware heterogeneity on the four stages of MapReduce execution.

[1] They do not have to be strictly inter-leaved as each sort task can begin before the corresponding shuffle task is over, when sufficient amount of data has been shuffled.



(a) Wordcount (b) Sort

Figure 2: Job completion time breakdown (normalized to total time) for Wordcount and Sort.

3.1 Setup

Our heterogeneous cluster consists of 5 Xeon (slow) nodes and 2 Opteron (fast) nodes, all of which are connected to a 1Gbps switch. Each Opteron node has 8 cores and 16GB RAM, and each Xeon node has 2 cores and 2GB RAM. We run Hadoop Wordcount (CPU-intensive) and Sort (IO-intensive) benchmarks and analyze their job completion time. The total job size consists of 40GB input data, *i.e.,* 680 map tasks each with 64 MB data size.

We use two configurations in our experiments. Both have 8 and 2 map slots each on fast and slow nodes, proportional to their numbers of cores, as in Tarazu [6]. They differ in reduce slots per node. Config-1 uses 2 reduce slots on both fast and slow nodes, as in Tarazu, while Config-2 uses 4 and 1 reduce slots on fast and slow nodes, *i.e.,* proportional to their numbers of cores.

3.2 Impact of Heterogeneous Nodes

Figure 2 shows the execution time and their breakdown into the four stages discussed in §2, of the two benchmarks on the fast and slow nodes, respectively, under the two configurations. We make the following observations.

(1) **Map-Only:** We observe the duration of Map-Only stage is short. For Wordcount, this stage ends when 1 wave of map tasks is over on the slow nodes, and 2 waves of map tasks are completed on the fast nodes.

(2) **Map-Shuffle:** The Map-Shuffle stage always finishes at almost the same time on the fast and slow nodes. This is due to the inherent load balancing feature of the task scheduler: whenever a Map slot is freed on a node, a new map task is scheduled. For example, in Config-1, each fast node processes far more map tasks (41%) than slow nodes (3.6%) for Wordcount. This imbalanced map task processing has two consequences. First, after the fast nodes finish map tasks on local data first (a locality feature of the Hadoop scheduler), they will execute remote map tasks (stealing data from the slow nodes). In Config-1, about 9% of the total map tasks (of the whole job) executed by a fast node in Wordcount are remote

map tasks. Such remote map tasks generate extra network traffic from fetching data remotely. Second, since a fast node performed more map tasks, it will shuffle much more intermediate data out to other nodes than slow nodes. In Figure 2(a) Config-1, each fast node in total shuffles out 7 times more data than each slow node.

(3) **Shuffle-Only:** Figure 2 shows the duration of the Shuffle-Only stage can vary significantly on fast and slow nodes. The gap results from the difference between shuffle-sort speeds on fast and slow nodes, which results in different total shuffle-sort durations – the shuffle-only stage is the leftover shuffle-sort beyond the time all map tasks are finished. Figure 2(a) shows the stage is 7.4 times shorter on fast nodes than on slow nodes for Wordcount, but completes at about the same on fast and slows for Sort, under Config-1.

(4) **User-Reducer:** Since the default scheduler equally partitions the key range across reducers, each reducer processes equal amount of data in the user-reducer phase. The execution time, however, can differ among different nodes due to the difference in their processing speed. Figure 2(a) shows under Config-1, this stage is 3.51 times slower on slow nodes than on fast nodes for Wordcount but finishes at about the same time for Sort, whereas under under Config-2, it finishes at about the same time for Wordcount, but is 1.26 times slower on fast nodes than on slow nodes.

(5) **Diversity of impact:** Overall, Figure 2 shows the impact of hardware heterogeneity on different stages differ for different applications under different configurations, suggesting it cannot be easily solved by any static map/reduce slot configuration.

4 Dynamic Load Rebalancing

We revisit the key design challenge in dynamic load rebalancing, a potentially effective technique to optimize MapReduce execution on heterogeneous clusters.

4.1 General Approach

The idea of load rebalancing is straight-forward: faster reducer gets more data; the task scheduler calculates the key range partition for fast and slow nodes that results in the reducers on them finishing at about the same time.

One can potentially derive an analytic model to capture the effects of all contributing factors to the reducer completion time on fast and slow nodes [6]. However, the extensive information needed in such a model are application and hardware specific, which requires extensive profiling and makes it infeasible to use in practice [6]. This motivates the practical approach of *dynamic load rebalancing*, *i.e.,* the task scheduler starts with the default even split policy, estimates the key range partitions for fast and slow nodes at runtime, and instructs the reduce tasks to carry their new workload accordingly.

Dynamic load rebalancing faces two conflicting challenges. (1) The new load split estimate needs to be *accurate*, to maximally even out the reducer completion time on fast and slow nodes. (2) The new load split estimate needs to be calculated as *early* as possible, to minimize the wasted (and hence extra) data movement and processing. In particular, when the assignment of a bin changes from one reduce task to another, the data associated with the bin needs to be reshuffled to the newly assigned reduce task and re-processed thereafter. Conceptually, the two challenges are at odds with each other: the longer the task scheduler waits to estimate the new load split, the more information it can collect and estimate the split more accurately, but also the more wasted (and hence extra) data movement and processing due to the default even load split before rebalancing takes place.

4.2 Design Space

We define the target ratio of key partition sizes assigned to each reducer on a fast node to each reducer on a slow node as the *partition ratio — P*. The challenge is to calculate *P* accurately to balance the completion time of the reducers. We now explore the design space for when and how the task scheduler should attempt to estimate *P*.

D_1: At start of the Map-Only stage (T_0)[2]. At the beginning of job execution, since no information is available about the progress rates of map and reduce tasks, *P* can only be set to the default value 1. This is the default even-split policy which is oblivious to cluster heterogeneity. [3]

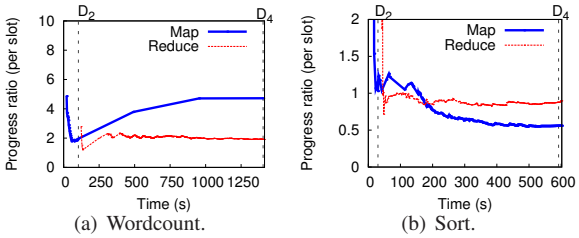
D_2: At start of the Map-Shuffle stage (T_1). At T_1, since the reduce tasks have not started, *P* can only be estimated using the relative progress rates of map tasks (so far) on fast and slow nodes, *i.e.,* $P = \frac{S_{fa,m}}{S_{sl,m}}$, where $S_{fa,m}$ and $S_{sl,m}$ are the progress rates for map tasks on fast and slow nodes. This method is used in Tarazu [6].

The main advantage of this method is that, since shuffle-sort has not started, there is no need to reshuffle any data after the load rebalancing act. However, it can give a poor estimate of *P*. Map and reduce tasks are known to have very different resource requirements, *e.g.,* a map task is CPU-intensive in the first half and I/O-intensive in the second half, whereas shuffle-sort has interleaved network-intensive and CPU- and I/O-intensive phases. As a result, the relative speed of map tasks can be a poor approximation to the relative speed of shuffle-sort. Figure 3(b) shows for Sort, the ratio of map task progress rates at T_1 is 1.25 , which would be a poor approximation to the steady-state ratio of shuffle-sort progress rates 0.7.

D_3: during the Map-shuffle stage (between T_1 and T_2). Between T_1 and T_2, *P* can be estimated as $\frac{S_{fa}}{S_{sl}}$ where S_{fa} and S_{sl} denote the actual progress rates of Shuffle-Sort so far. The ratio, however, may not be a good ap-

[2] T_0 to T_4 are marked in Figure 1.

[3] Although conceptually *P* can be set to a biased value based on the prior knowledge about the node heterogeneity, picking a suitable value is hard as the progress rate varies significantly for different phases and jobs on the same node.



(a) Wordcount. (b) Sort.

Figure 3: Ratio of progress rates of map and reduce tasks on fast and slow nodes.

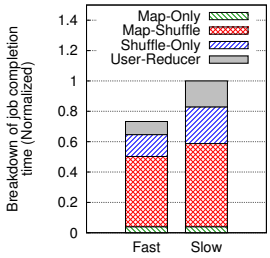


Figure 4: Job execution time and breakdown of Wordcount under D_3 (P=2).

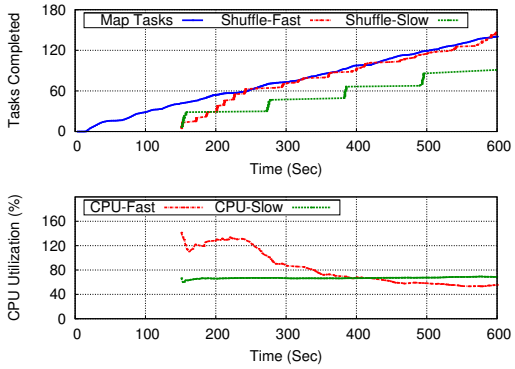


Figure 5: Shuffling progress and CPU utilization by each reducer on fast and slow nodes in D_3 (calculated P=2).

proximation to the ratio of progress rates of user-reducers on fast and slow nodes, which perform different operations from shuffle-sort subtasks.

D_3 achieves better estimate of P at the cost of the penalty associated with adjusting load in the middle of the Shuffle-Sort stage in two ways: (1) Re-Shuffling: the reducer on a fast node needs to shuffle in some data already shuffled to slow nodes; (2) Dropping data: the reducer on a slow node needs to drop some data shuffled in and sorted under even split.

To strike a balance between accuracy and penalty, the progress rates and their ratio of reducers on fast and slow nodes can be measured once they are observed to stabilize, typically after shuffling in one wave of map tasks.

D_4: during the Shuffle-Only stage (after T_2). Estimation of P can be further delayed till the shuffle-only or even user-reducer stage has started. At this point, the relative progress rates of these stages on fast and slow nodes can be measured accurately; but this design choice suffers a major disadvantage in terms of reshuffling costs as slow and fast nodes have fetched substantial amount of data, ranging from 30-100%. Thus, rebalancing load at this stage would result in too high data reshuffling penalty which is likely to erase the gain from rebalancing the data. We do not consider this option further.

4.3 Design Refinement

We implemented D_3 (details in §5, the calculated P=2 from Figure 3(a)) and reran Wordcount. The new execution time breakdowns, shown in Figure 4, show that the Shuffle-Only stage still finishes at different time on fast and slow nodes! To understand the reason, we plot the (total) map task completion rate and the rate map tasks are shuffled in by fast nodes and slow nodes for Wordcount in Figure 5. We make two observations. (1) The fast node is able to match the rate at which map tasks are completed, which shows that fast node is able to get enough CPU and network resources to fetch the map outputs. (2) The shuffle on slow nodes never catches up with the total number of map tasks completed, possibly due to lack of resources, *i.e.,* slow nodes are overloaded.

The CPU utilization shown in Figure 5 further confirms this explanation. We see the CPU utilization of the reducer on slow nodes is stable between 59-66%. Since the reducer on slow nodes always lags behind the map tasks completed, we can conclude that 66% is the maximum CPU a reducer can get on slow nodes. In contrast, the reducer CPU utilization on fast nodes reaches 120% (the multi-threaded process uses multiple cores) at the start of reducers, then gradually decreases and stabilizes at 55%, at which moment it has caught up with map task completion. This suggests at the steady state, the reducer on fast nodes just needs 55% of CPU, but it can get as much as 120% of CPU if needed.

The above finding suggests D_3 needs to be adjusted to use the potential progress rate of the reducer on fast nodes, as opposed to the progress rate observed (so far). The partition ratio P is now calculated as

$$P = \frac{S_{fa}}{S_{sl}} * \frac{1}{E_{fa}} \quad (1)$$

where E_{fa} denotes the CPU efficiency ($<$1) of the reducer on fast nodes, defined as the ratio of the CPU utilization in the steady state (T_w in Figure 6 bottom) to the CPU utilization when shuffle (on fast nodes) has caught up with map tasks completed (T_s in Figure 6 top). In practice, we observe the steady state T_w on fast nodes is typically reached when 1 wave of map-tasks are completed after T_s. Note the CPU utilization on slow nodes is fairly stable. Figure 6 shows the CPU utilization and shuffle when the partition ratio is adjusted at time T_w using the refined scheme, denoted by D_3'. The calculated partition ratio was 4.34. It can be seen that the fast node regains CPU utilization and both slow and fast nodes shuffle data at the same rate.

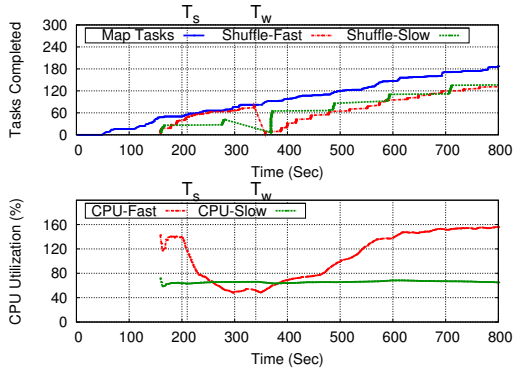


Figure 6: Shuffling progress and CPU utilization by each reducer on fast and slow nodes under D_3'.

Lastly, D_3' can be easily extended to more than two types of nodes. We skip the details due to page limit.

5 Implementation

We implemented our new load rebalancing scheme D_3' in Hadoop v0.20.203.0 [4] by adding ≈2KLOC. We name the new system PIKACHU. Partition ratio P is calculated at JobTracker based on Hadoop progress rates and CPU efficiency of the reducer processes, which are reported by TaskTracker on each node every 3 seconds.

We use virtual bins to dynamically change the load between fast nodes and slow nodes based on partition ratio P. Each map task output is partitioned into $10 \cdot N$ splits, where N is the total number of reducers. Initially each reducer is mapped with 10 virtual bins. Once the JobTracker determines the ratio P, it translates the ratio to the target number of virtual bins for reducers on fast and slow nodes. Let N_s and N_f be the total number of reducer slots on all slow nodes and all fast nodes, respectively. The numbers of virtual bins for a reducer on a slow node V_s and on a fast node V_f are calculated as

$$V_s = \frac{10 \cdot (N_s + N_f)}{N_s + P \cdot N_f}, V_f = \frac{10 \cdot (N_s + N_f) \cdot P}{N_s + P \cdot N_f} \quad (2)$$

Following this, the JobTracker assigns a new virtual bin mapping to each reducer. Upon receiving the new mapping, the reducers on fast nodes need to fetch the newly added virtual bins, while the reducers on slow nodes will drop the existing sorted data corresponding to the dropped virtual bins.

6 Evaluation

We also implemented Tarazu [6] in Hadoop (version 0.2.203.0). We compare job completion time under PIKACHU, Tarazu, and Hadoop. We also measure the overhead incurred in PIKACHU due to re-shuffling and re-sorting. We use five benchmark applications: *Wordcount, Sort, Multi-Wordcount, Inverted-index and Self-join* [6]. *Wordcount* counts the occurrences of every word. *Sort* sorts the given dataset. *Multi-Wordcount* counts all unique sets of 3 consecutive words. *Inverted-index* generates words-to-file indexing. *Self-join* generates association among *k+1* fields given the set of *k*-field association. *Sort* and *Selfjoin* are shuffle-intensive, whereas the other 3 applications are compute-intensive.

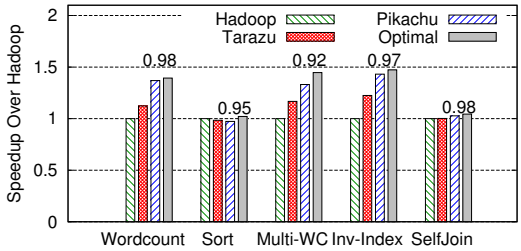


Figure 7: Speedup of of Tarazu and PIKACHU over Hadoop, under Config-1.

Performance on Local Cluster. Figure 7 shows the speedup (in terms of job completion time) achieved by PIKACHU and Tarazu against Hadoop for 5 different applications using Config-1. In addition to Hadoop, Tarazu and PIKACHU, we also measure the job completion time at the optimal partition ratio found using trial-and-error method. The numbers above the bars denote the percentage of the optimal performance PIKACHU achieves. For Sort and Selfjoin applications, the initial configuration was close to optimal (the difference between the job completion time of Hadoop and Optimal was <4%) and there was little room for improvement. For the remaining applications, PIKACHU outperforms Hadoop by 33-42% and Tarazu by 14-22% because of better accuracy in calculating P. Furthermore, PIKACHU achieves 92-98% of the optimal job completion time, showing there is not much room to improve over PIKACHU.

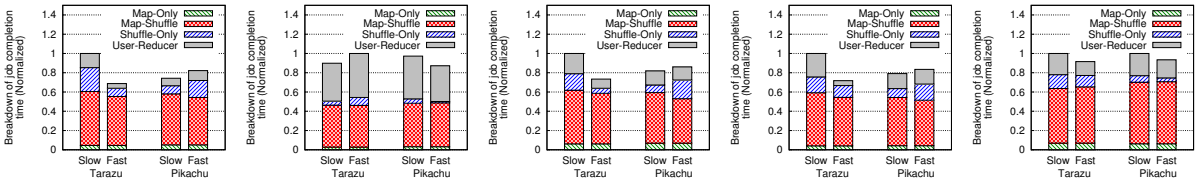
Table 1 summarizes the partition ratios calculated by Tarazu (T), PIKACHU (P) and Optimal (O). The partition ratio calculated using PIKACHU is closer to Optimal compared to Tarazu. Table 1 also shows the overhead incurred by PIKACHU, measured as the extra data shuffled by all the nodes in PIKACHU compared to Hadoop. We see PIKACHU incurs a low overhead 0.96-4.75% in re-shuffling and re-sorting.

Figure 8 shows the breakdown of the job completion time under PIKACHU normalized to the job completion time under Tarazu for all 5 applications on slow and fast nodes. It can be seen that in all 5 cases, the difference between the shuffle-only execution time, and more importantly the difference between the reducer task completion time, on the nodes are within 10% on PIKACHU and 31% on Tarazu.

Performance on EC2 Cluster. Finally, we compared PIKACHU with Tarazu and Hadoop on a 60-node heterogeneous cluster in EC2, consisting of 40 `m1.small` (slow) and 20 `m1.xlarge` (fast) nodes. We evaluated the performance using 3 applications, Wordcount, Sort and Multi-Wordcount under Config-1 and Config-2 for

Table 1: Partitioning ratio P and overhead under Tarazu, PIKACHU, and optimal partition for the five applications.

Observation	Wordcount			Sort			Multi-Wordcount			Inverted-Index			Self-join		
	T	P	O	T	P	O	T	P	O	T	P	O	T	P	O
Calculated P	2	4.5	4	1.1	0.67	0.9	2.37	3.7	3.5	2.44	3.4	3.3	1	1.1	1.2
Shuffle-Overhead	3.86%			4.13%			4.58%			4.75%			0.96%		



(a) Wordcount (b) Sort (c) Multi-Wordcount (d) Inverted Index (e) Selfjoin

Figure 8: Job completion time breakdown on fast and slow nodes (Normalized to Tarazu).

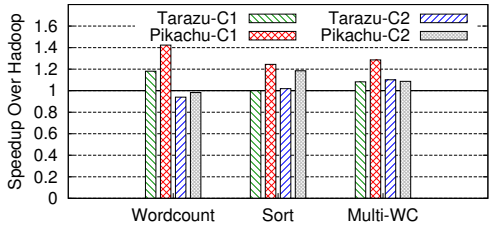


Figure 9: Speedup of Tarazu and PIKACHU over Hadoop on the 60-node EC2 cluster on 2 configurations.

180 GB data (2880 map tasks). Figure 9 shows the speedup achieved by Tarazu and PIKACHU over Hadoop for the 2 configurations.

In Config-1, Tarazu and PIKACHU outperform Hadoop by 0-18% and 25-42%, respectively. Config-2 was optimal configuration for Hadoop for Wordcount without much scope for improvement. Tarazu and PIKACHU performance was lower than Hadoop by 6% and 2%. For the other 2 applications, Tarazu and PIKACHU outperformed Hadoop by up to 10% and 18%, respectively.

7 Related Work

Many implementations, extensions, and domain specific libraries of MapReduce have been developed to support large-scale data processing [3, 4, 14, 2, 16, 5]. None of them explicitly study optimizing MapReduce execution on heterogeneous hardware. LATE [18] was one of the first work to show the shortcomings of MapReduce on heterogeneous clusters. However, it focused on straggler detection and mitigation. Mantri [8] further explores the causes of stragglers/outliers. Such designs treat the symptoms of heterogeneity, *i.e.,* stragglers, as opposed to the root cause, and speculatively re-execute tasks on fast nodes, wasting utilization of slow nodes.

Lee *et al.* also considered heterogeneity in the MapReduce scheduler [13, 12] and proposed a fair scheduler [12] for a multi-tenant heterogeneous cluster. This work is orthogonal to ours as it improves the performance of multiple jobs rather than a single job. Finally, Tarazu [6] has already been discussed previously.

8 Conclusion

We showed that the prior-art MapReduce scheduler for heterogeneous clusters, Tarazu, poorly balances the load among reducers on fast and slow nodes. We proposed PIKACHU, which strikes a balance between accuracy and overhead in estimating the load adjustment and doubles Tarazu's improvement over Hadoop. We have released PIKACHU at `http://github.com/mapreduce-pikachu`.

Acknowledgment. This work was supported in part by NSF grant CNS-1065456.

References

[1] Amazon ec2. `aws.amazon.com/ec2/`.
[2] Apache mahout: Scalable machine learning and data mining. `http://mahout.apache.org`.
[3] Facebook hive. `hadoop.apache.org/hive`.
[4] Hadoop. `http://lucene.apache.org/hadoop`.
[5] X-rime: Hadoop based large scale social network analysis. `http://xrime.sourceforge.net/`.
[6] F. Ahmad, *et al.* Tarazu: optimizing mapreduce on heterogeneous clusters. In *ASPLOS '12*.
[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why let resources idle? aggressive cloning of jobs with dolly. In *HotCloud'12*.
[8] G. Ananthanarayanan, *et al.* Reining in the outliers in map-reduce clusters using mantri. In *OSDI'10*.
[9] E. Bortnikov, *et al.* Predicting execution bottlenecks in map-reduce clusters. In *HotCloud'12*.
[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*.
[11] B. Farley, *et al.* More for your money: exploiting performance heterogeneity in public clouds. In *SoCC '12*.
[12] G. Lee, *et al.* Heterogeneity-aware resource allocation and scheduling in the cloud. In *HotCloud'11*.
[13] G. Lee, *et al.* Topology-aware resource allocation for data-intensive workloads. In *APSys '10*.
[14] C. Olston, *et al.* Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*.
[15] C. Reiss, *et al.* Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC '12*.
[16] Y. Yu, *et al.* Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08*.
[17] M. Zaharia, *et al.* Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*.
[18] M. Zaharia, *et al.* Improving mapreduce performance in heterogeneous environments. In *OSDI'08*.

FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs

Kai Shen Stan Park
Department of Computer Science, University of Rochester

Abstract

On Flash-based solid-state disks (SSDs), different I/O operations (reads vs. writes, operations of different sizes) incur substantially different resource usage. This presents challenges for fair resource management in multi-programmed computer systems and multi-tenant cloud systems. Existing timeslice-based I/O schedulers achieve fairness at the cost of poor responsiveness, particularly when a large number of tasks compete for I/O simultaneously. At the same time, the diminished benefits of I/O spatial proximity on SSDs motivate fine-grained fair queueing approaches that do not enforce task-specific timeslices. This paper develops a new Flash I/O scheduler called FlashFQ. It enhances the start-time fair queueing schedulers with throttled dispatch to exploit restricted Flash I/O parallelism without losing fairness. It also employs I/O anticipation to minimize fairness violation due to deceptive idleness. We implemented FlashFQ in Linux and compared it with several existing I/O schedulers—Linux CFQ [2], an Argon [19]-inspired quanta scheduler, FIOS timeslice scheduler [17], FIOS with short timeslices, and 4-Tag SFQ(D) [11]. Results on synthetic I/O benchmarks, the Apache web server, and Kyoto Cabinet key-value store demonstrate that only FlashFQ can achieve both fairness and high responsiveness on Flash-based SSDs.

1 Introduction

NAND Flash devices are increasingly used as solid-state disks (SSDs) in computer systems. Compared to traditional secondary storage, Flash-based SSDs deliver much higher I/O performance which can alleviate the I/O bottlenecks in critical data-intensive applications. At the same time, the SSD resource management must recognize unique Flash characteristics and work with increasingly sophisticated firmware management. For instance, while the raw Flash device desires sequential writes, the write-order-based block mapping on modern SSD firmware can translate random write patterns into sequential writes on Flash and thereby relieve this burden for the software I/O scheduler. On the other hand, different I/O operations on Flash-based SSDs may exhibit large resource usage discrepancy. For instance, a write can consume much longer device time than a read due to the erase-before-write limitation on Flash. In addition, a larger I/O operation can take much longer than a small request does (unlike on a mechanical disk when both are dominated by mechanical seek/rotation delays). Without careful regulation, heavy resource-consuming I/O operations can unfairly block light operations.

Fair I/O resource management is desirable in a multi-programmed computer system or a multi-tenant cloud platform. Existing I/O schedulers including Linux CFQ [2], Argon [19], and our own FIOS [17] achieve fairness by assigning timeslices to tasks that simultaneously compete for the I/O resource. One critical drawback for this approach is that the tasks that complete their timeslices early may experience long periods of unresponsiveness before their timeslices are replenished in the next epoch. Such unresponsiveness is particularly severe when one must wait for a large number of co-running tasks in the system to complete their timeslices. Poor responsiveness is harmful but unnecessary on Flash-based SSDs that often complete an I/O operation in a fraction of a millisecond.

High responsiveness is supported by classic fair queueing approaches that originated from network packet switching [6, 8, 9, 16] but were also used in storage systems [3, 11, 18]. They allow fine-grained interleaving of requests from multiple tasks / flows as long as fair resource utilization is maintained through balanced virtual time progression. The lagging virtual time for an inactive task / flow is brought forward to avoid a large burst of requests from one task / flow and prolonged unresponsiveness for others. One drawback for fine-grained fair queueing on mechanical disks is that frequent task switches induce high seek and rotation costs. Fortunately, this is only a minor concern for Flash-based SSDs due to diminished benefits of I/O spatial proximity on modern SSD firmware.

This paper presents a new operating system I/O scheduler (called *FlashFQ*) that achieves fairness and high responsiveness at the same time. FlashFQ enhances the start-time fair queueing scheduler SFQ(D) [11] with two new mechanisms to support I/O on Flash-based SSDs. First, while SFQ(D) allows concurrent dispatch of requests (called *depth*) to exploit I/O parallelism, it violates fairness when parallel I/O operations on a Flash device interfere with each other. We introduce a throttled dispatch technique to exploit restricted Flash I/O par-

allelism without losing fairness. Second, existing fair queueing schedulers are work-conserving—they never idle the device when there is pending work to do. However, work-conserving I/O schedulers are susceptible to deceptive idleness [10] that causes fairness violation. We propose anticipatory fair queueing to mitigate the effects of deceptive idleness. We have implemented FlashFQ with the throttled dispatch and anticipatory fair queueing mechanisms in Linux.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 characterizes key motivations and challenges for our fair queueing scheduler on Flash-based SSDs. Sections 4 and 5 present the design techniques and implementation issues in our FlashFQ scheduler. Section 6 illustrates our experimental evaluation and comparison with several alternative I/O schedulers. Section 7 concludes this paper with a summary of our findings.

2 Related Work

Flash I/O characterization and operating system support have been recognized in research. Agrawal *et al.* [1] discussed the impact of block erasure (before writes) and parallelism on the performance of Flash-based SSDs. Work by Chen *et al.* [4] further examined strided access patterns and identified abnormal performance issues like those caused by storage fragmentation. File system work [5,14,15] attempted to improve the sequential write patterns through the use of log-structured file systems. These efforts are orthogonal to our research on Flash I/O scheduling.

New I/O scheduling heuristics were proposed to improve Flash I/O performance. In particular, write bundling [12], write block preferential [7], and page-aligned request merging/splitting [13] help match I/O requests with the underlying Flash device data layout. The effectiveness of these write alignment techniques, however, is limited on modern SSDs with write-order-based block mapping. Further, these Flash I/O schedulers have paid little attention to the issue of fairness.

Fairness-oriented resource scheduling has been extensively studied. Fairness can be realized through per-task timeslices (as in Linux CFQ [2], Argon [19], and FIOS [17]) and credits (as in the SARC rate controller [20]). The original fair queueing approaches, including Weighted Fair Queueing (WFQ) [6], Packet-by-Packet Generalized Processor Sharing (PGPS) [16], and Start-time Fair Queueing (SFQ) [8, 9], take virtual time-controlled request ordering over several task queues to maintain fairness. While they are designed for network packet scheduling, later fair queueing approaches like Cello's proportionate class-independent scheduler [18], YFQ [3] and SFQ(D) [11] are adapted to support I/O resources. In particular, they allow the flexibility to reorder and parallelize I/O requests for better efficiency. Most of these fair-share schedulers (with the only exception of FIOS) do not address unique characteristics on Flash-based SSDs and many (including FIOS) do not support high responsiveness.

3 Motivations and Challenges

Timeslice Scheduling vs. Fair Queueing Timeslice-based I/O schedulers such as Linux CFQ, Argon, and FIOS achieve fairness by assigning timeslices to co-running tasks. A task that completes its timeslice early would have to wait for others to finish before its timeslice is replenished in the next epoch, leading to a period of unresponsiveness at the end of each epoch. Figure 1(A) illustrates this effect in timeslice scheduling. While some schedulers allow request interleaving (as shown in Figure 1(B)), the period of unresponsiveness still exists at the end of an epoch. This unresponsiveness is particularly severe in a highly loaded system where one must wait for a large number of co-running tasks to complete their timeslices. One may shorten the per-task timeslices to improve responsiveness. However, outstanding requests at the end of a timeslice may consume resources at the next timeslice that belongs to some other task. Such resource overuse leads to unfairness and this problem is particularly pronounced when each timeslice is short (Figure 1(C)).

In fine-grained fair queueing (as shown in Figure 1(D)), requests from multiple tasks are interleaved in a fine-grained fashion to enable fair progress by all tasks. It achieves fairness and high responsiveness at the same time. Furthermore, since fine-grained fair queueing does not restrict the request-issuing task in each timeslice, it works well with I/O devices possessing internal parallelism (Figure 1(E)).

Finally, due to substantial background maintenance such as Flash garbage collection, Flash-based SSDs provide time-varying capacities (more I/O capacity at one moment and less capacity at a later time). The timeslice scheduling that focuses on the equal allocation of device time may not provide fair shares of time-varying resource capacities to concurrent tasks. In contrast, the fair queueing scheduling targets equal progress of completed work and therefore it can achieve fairness even for resources with time-varying capacities.

Restricted Parallelism Flash-based SSDs have some built-in parallelism through the use of multiple channels. Within each channel, the Flash package may have multiple planes which are also parallel. We run experiments to understand such parallelism. We utilize the following Flash-based storage devices—

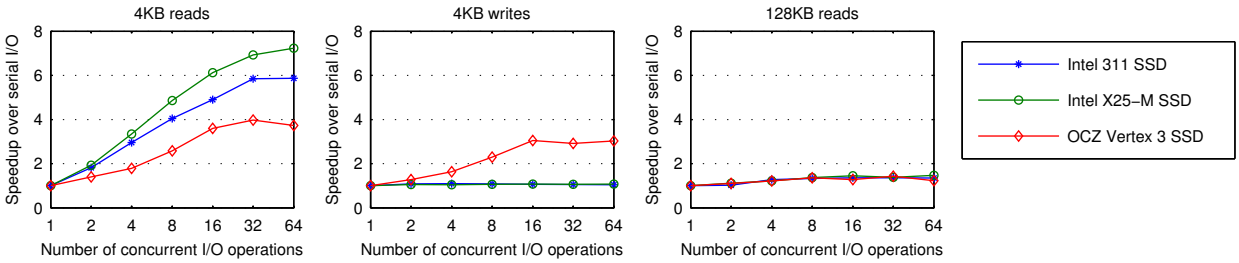


Figure 2: Efficiency of I/O parallelism (throughput speedup over serial I/O) for 4 KB reads, 4 KB writes, and 128 KB reads on three Flash-based SSDs.

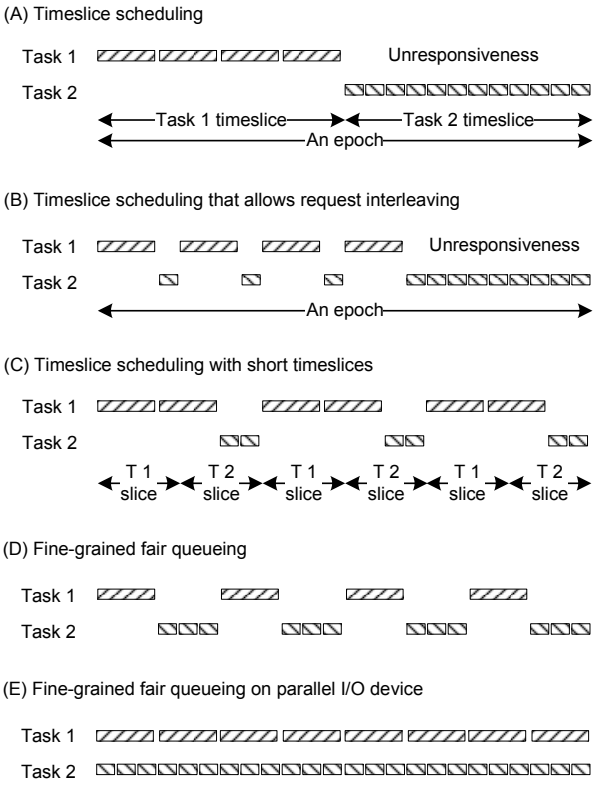


Figure 1: Fairness and responsiveness of timeslice scheduling and fine-grained fair queueing.

- An Intel 311 Flash-based SSD, released in 2011, using single-level cells (SLC) in which a particular cell stores a single bit of information.
- An Intel X25-M Flash-based SSD, released in 2009, using multi-level cells (MLC).
- An OCZ Vertex 3 Flash-based SSD, released in 2011, using MLC.

To acquire the native device properties, we bypass the memory buffer (through direct I/O) and operating system I/O scheduler (configuring Linux `noop` scheduler) in these measurements. We also disable the device write cache so that all writes reach the durable storage medium.

Figure 2 shows the efficiency of Flash I/O parallelism for 4 KB reads, 4 KB writes, and 128 KB reads on our SSDs. We observe that the parallel dispatch of multiple 4 KB reads to an SSD lead to substantial throughput enhancement (up to 4-fold, 6-fold, and 7-fold for the three SSDs respectively). However, the parallelism-induced speedup is diminished by writes and large reads. We observe significant write parallelism only on the Vertex 3 SSD. Large reads suppress the parallel efficiency because a large read already uses the multiple channels in a Flash device.

Such restricted parallelism leads to new challenges for a fair queueing I/O scheduler. On one hand, the scheduler should allow the simultaneous dispatch of multiple I/O requests to exploit the Flash parallel efficiency when available. On the other hand, it must recognize the unfairness resulting from the interference of concurrently dispatched I/O requests and mitigate it when necessary.

Diminished Benefits of Spatial Proximity One drawback for fine-grained fair queueing on mechanical disks is that frequent task switches lead to poor spatial proximity and consequently high seek and rotation costs. Fortunately, at the absence of such mechanical overhead, Flash I/O performance is not as dependent on the I/O spatial proximity. This is particularly the case for modern SSDs with write-order-based block mapping where random writes become spatially contiguous on Flash due to block remapping.

We run experiments to demonstrate such diminished benefits of spatial proximity. Besides the three Flash-based SSDs, we also include a conventional mechanical disk (a 10 KRPM Fujitsu SCSI drive) for the purpose of comparison. Figure 3 shows the performance discrepancies between random and sequential I/O on the storage devices. The random I/O performance is measured when each I/O operation is applied to a randomized offset address in a 256 MB file.

We observe that the sequential I/O for small (4 KB)

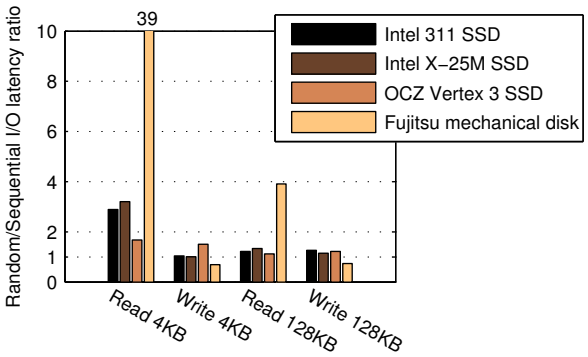


Figure 3: The ratios of random I/O latency over sequential I/O latency for reads / writes at two I/O request granularities (4 KB and 128 KB) on three Flash-based SSDs.

reads are still beneficial for some SSDs (3-fold speedup for the two Intel SSDs). However, such benefits are much diminished compared to the 39-fold sequential read speedup on the mechanical disk. The performance difference between random and sequential I/O is further diminished for writes and large-grained (128 KB) I/O requests on SSDs. These results also match the findings in Chen *et al.*'s 2009 paper [4] for mid- and high-end SSDs at the time. The diminished benefits of I/O spatial proximity mitigates a critical drawback for adopting fine-grained fair queueing on Flash-based SSDs. Particularly in the case of small reads, while the fine-grained fair queueing loses some sequential I/O efficiency, it gains the much larger benefit of I/O parallelism on small reads.

Note that the poor sequential write performance on the rotating mechanical disk in our measurement results is due to the disabling of its write cache, in which case all writes must reach the durable disk. Therefore the disk head has typically rotated beyond the desired location when a new write request arrives after a small delay (software system processing) from the completion of the previous operation. This effect is particularly pronounced in our measurement setup where the disk rotation time dominates the seek time since our random I/O addresses are limited in a small disk area (a 256 MB file).

Deceptive Idleness Fair queueing I/O schedulers like SFQ(D) [11] are work-conserving—they never idle the device when there is pending work to do. However, work-conserving I/O schedulers are susceptible to deceptive idleness [10]—an active task that issues the next request a short time after receiving the result of the previous one may temporarily appear to be idle. For fair queueing schedulers, the deceptive idleness may let an active task to be mistakenly considered as being "inactive". This can result in the premature advance of virtual time for such "inactive tasks" and therefore unfairness.

Consider the simple example of a concurrent run involving two tasks—one continuously issues I/O requests with heavy resource usage (*heavy task*) while the other continuously issues light I/O requests (*light task*). We further assume that the I/O scheduler issues one request to the device at a time (no parallelism). At the moment when a request from the light task completes, the only queued request is from the heavy task and therefore a work-conserving I/O scheduler will dispatch it to the device. This effectively results in one-request-at-a-time alternation between the two tasks and therefore unfairness favoring the heavy task.

4 FlashFQ Design

In a concurrent system, many resource principals simultaneously compete for a shared I/O resource. The scheduler should regulate I/O in such a way that accesses are fair. When the storage device time is the bottleneck resource in the system, fairness is the case that each resource principal acquires an equal amount of device time. At the same time, responsiveness requires that each user does not experience prolonged periods with no response to its I/O requests. We present the design of our FlashFQ I/O scheduler that achieves fairness and high responsiveness for Flash-based SSDs. It enhances the classic fair queueing approach with new techniques to address the problems of restricted parallelism and deceptive idleness described in the last section.

Practical systems may desire fairness and responsiveness for different kinds of resource principals. For example, a general-purpose operating system desires the support of fairness and responsiveness for concurrent applications. A server system wants such support for simultaneously running requests from multiple user classes. A shared hosting platform needs fairness and responsiveness for multiple active cloud services (possibly encapsulated in virtual machines). Our I/O scheduling design and much of our implementation can be generally applied to supporting arbitrary resource principals. When describing the FlashFQ design, we use the term *task* to represent the resource principal that receives the fairness and responsiveness support in a concurrent system.

4.1 Fair Queueing Preliminaries

As described in Section 2, a considerable number of fair queueing schedulers have been proposed in the past. Our FlashFQ design is specifically based on SFQ(D) [11] for two reasons. First, it inherits the advantage of Start-time Fair Queueing (SFQ) [8, 9] that the virtual time can be computed efficiently. Second, it allows the simultaneous dispatch of multiple requests which is necessary for exploiting the internal parallelism on Flash devices.

SFQ(D) maintains a system-wide virtual time $v(t)$. It

uses the virtual time to assign a start and finish tag to each arriving request. The start tag is the larger of the current system virtual time (at the request arrival time) and the finish tag of the last request by the same task. The finish tag is the start tag plus the expected resource usage of the request. Request dispatch is ordered by each pending request's start tag. Multiple requests (up to the depth D) can be dispatched to the device at the same time.

A key issue with SFQ(D) is the way the virtual time $v(t)$ is advanced and the related treatment of *lagging tasks*—those that are slower than others in utilizing allotted resources. If $v(t)$ advances too quickly, it could artificially bring forward the request start tags of lagging tasks such that their unused resources are forfeited which leads to unfairness. On the other hand, if the virtual time advances too slowly, it could allow a lagging task to build up its unused resources and utilize them in a sudden burst of request arrivals that cause prolonged unresponsiveness to others. Three versions of the scheduler were proposed [11], with different ways of maintaining the system virtual time—

- **Min-SFQ(D)** assigns the virtual time $v(t)$ to be the minimum start tag of any outstanding request at time t. A request is *outstanding* if it has arrived but not yet completed. A key problem with Min-SFQ(D) is that its virtual time advances too slowly which makes it susceptible to unresponsiveness caused by a sudden burst of requests from a lagging task described above.
- **Max-SFQ(D)**[1] assigns the virtual time $v(t)$ to be the maximum start tag of dispatched requests on or before time t. A drawback with this approach is that its virtual time may advance too quickly and result in unfairness as described above.
- **4-Tag SFQ(D)** attempts to combine the above two approaches to mitigate each's problem. Specifically, it maintains two pairs of start / finish tags for each request according to Min-SFQ(D) and Max-SFQ(D) respectively. The request dispatch ordering is primarily based on the Max-SFQ(D) start tags while ties are broken using Min-SFQ(D) start tags.

4.2 Min-SFQ(D) with Throttled Dispatch

The characterization in Section 3 shows that Flash-based SSDs exhibit restricted parallelism—while parallel executions can sometimes produce higher throughput, simultaneously dispatched requests may also interfere with each other on the Flash device. Utilizing such restricted parallelism may lead to uncontrolled resource usage under any version of the SFQ(D) schedulers described above. Consider two tasks running together in the system and each task issues no more than one request at a time. If the I/O scheduler depth $D \geq 2$, then requests of both tasks will be dispatched to the device without delay at the scheduler. Interference at the Flash device often results in unbalanced resource utilization between the two tasks.

While such unbalanced resource utilization affects all three SFQ(D) versions, it is particularly problematic for Max-SFQ(D) and 4-Tag SFQ(D) who advance the system virtual time too quickly—any request dispatch from an aggressive task leads to an advance of the system virtual time, and consequently the forfeiture of unused resources by the lagging tasks. In comparison, Min-SFQ(D) properly accounts for the unbalanced resource utilization for all active tasks. Therefore we employ Min-SFQ(D) as the foundation of our scheduler.

Proper resource accounting alone is insufficient for fairness, we need an additional control to mitigate the imbalance of resource utilization between concurrent tasks. Our solution is a new *throttled dispatch* mechanism. Specifically, we monitor the relative progresses of concurrently active tasks and block a request dispatch if the progress of its issuing task is excessively ahead of the most lagging task in the system (i.e, the difference between those tasks' progress exceeds a threshold). Under SFQ(D) schedulers, the progress of a task is represented by its last dispatched start tag—the start tag of its most recently dispatched request. When requests from aggressive tasks (using more resources relative to their shares) are blocked, lagging tasks can catch up with less interference at the device. The blocking is relieved as soon as the imbalance of resource utilization falls below the triggering threshold.

4.3 Anticipation for Fairness

A basic principle of fair queueing scheduling is that when a task becomes inactive (it has no I/O requests to issue), its resource share is not allowed to accumulate. The rationale is simple—one has no claim to resources when it has no intention of using them. Even Min-SFQ(D)—which, among the three SFQ(D) versions, advances the system virtual time most conservatively—ignores tasks that do not have any outstanding I/O requests. As explained in Section 3, this approach may mistakenly consider an active task to be "inactive" due to deceptive idleness in I/O—an active task that issues the next request a short time after receiving the result of the previous one may temporarily appear to be idle to the I/O scheduler. Even during a very short period of deceptive idleness, the system virtual time may advance with no regard to the deceptively "inactive" task, leading to the forfeiture of its unused resources.

[1] This version is called SFQ(D) in the original paper [11]. We use a different name to avoid the confusion with the general reference to all three SFQ(D) versions.

The deceptive idleness was first recognized to cause undesirable task switches on mechanical disks that result in high seek and rotation delays. It was addressed by anticipatory I/O [10] which temporarily idles the disk (despite the existence of pending requests) to hope for a soon-arriving new request (typically issued by the process that is receiving the result of the just completed request) with better locality. We adopt anticipatory I/O for a different purpose—ensuring the continuity of a task's "active" status when deceptive idleness appears between its two consecutive requests. Specifically, when a synchronous I/O request completes, the task that will be receiving the result of the just completed request is allowed to stay "active" for a certain period of time. During this period, we adjust Min-SFQ(D) to consider the anticipated next request from the task as a hypothetical outstanding request in its virtual time maintenance. The start tag for the anticipated request, if arriving before the anticipation expires, should be the finish tag of the last request by the task.

The "active" status anticipation ensures that an active task's unused resources are not forfeited during deceptive idleness. It also enables the dispatch-blocking of excessively aggressive tasks (described in Section 4.2) when the lagging task is deceptively idle for a short amount of time. While both goals are important, anticipation for these two cases have different implications. Specifically, the anticipation that blocks the request dispatch from aggressive tasks is not work-conserving—it may leave the device idle while there is pending work—and therefore may waste resources. We distinguish these two anticipation purposes and allow a shorter timeout for the non-work-conserving anticipation that blocks aggressive tasks.

4.4 Knowledge of Request Cost

Recall that the request finish tag assignment requires knowledge of the resource usage of a request (or its cost). The determination of a request's cost is an important problem for realizing our fair queueing scheduler in practice and it deserves a careful discussion.

A basic question on this problem is by what time a request's cost must be known. This question is relevant because it may be easier to estimate a request's cost after its completion. According to our design, this is when the request's finish tag is assigned. In theory, for fair queueing schedulers that schedule requests based on their start tag ordering [8, 9, 11] (including ours), only the start tag assignments are directly needed for scheduling. A request's finish tag assignment can be delayed to when it is needed to compute some other request's start tag. In particular, one request (r_1)'s finish tag is needed to compute the start tag of the next arriving request (r_2) by the same task. Since the two requests may be dispatched in parallel, r_2's start tag (and consequently r_1's finish tag) might be needed before r_1's completion.

Given the potential need of knowing request costs early, our system estimates a request's cost at the time of its arrival. Specifically, we model the cost of a Flash I/O request based on its access type (read / write) and its data size. For reads and writes respectively, we assume a linear model (typically with a substantial nonzero offset) between the cost and data size of an I/O request. Our estimation model requires the offline calibration of the Flash I/O time for only four data access cases—read 4 KB, read 128 KB, write 4 KB, and write 128 KB. In general, such calibration is performed once for each device. Additional (but infrequent) calibrations can be performed for devices whose gradual wearout affects their I/O performance characteristics.

5 Implementation Issues

FlashFQ can be implemented in an operating system to regulate I/O resource usage by concurrent applications. It can also be implemented in a virtual machine monitor to allocate I/O resources among active virtual machines. As a prototype, we have implemented our FlashFQ scheduler in Linux 2.6.33.4. Below we describe several notable implementation issues.

Implementation in Linux An important feature of Linux I/O schedulers is the support of plugging and request merging—request queue is plugged (blocking request dispatches) temporarily to allow physically contiguous requests to merge into a larger request before dispatch. This is beneficial since serving a single large request is much more efficient than serving multiple small requests. Request merging, however, is challenging for our FlashFQ scheduler due to the need of re-computing request tags and task virtual time when two requests merge. For simplicity, we only implemented the most common case of request back-merging—merging a new arriving request (r_2) to an existing queued request (r_1) if r_2 contiguously follows (on the back of) r_1.

While the original anticipatory I/O [10] requires a single timer, our anticipation support may require multiple outstanding timers due to the nature of parallelism in our scheduler. Specifically, we may need to track deceptive idleness of multiple parallel tasks. To minimize the cost of parallel timer management, our implementation maintains a list of pending timers ranked by their fire time (we call them *logical timers*). Only the first logical timer (with the soonest fire time) is supported by a physical system timer. Most logical timer manipulations (add / delete timers) do not involve the physical system timer unless the first logical timer is changed.

Our prototype implementation runs on the ext4 file system. We mount the file system with the `noatime` option to avoid metadata updates on file reads. Note that the metadata updates (on modification timestamps) are still necessary for file writes. The original ext4 file system uses very fine-grained file timestamps (in nanoseconds) so that each file write always leads to a new modification time and thus triggers an additional metadata write. This is unnecessarily burdensome to many write-intensive applications. We revert back to file timestamps in the granularity of seconds (which is the default in Linux file systems that do not make customized settings). In this case, at most one timestamp metadata write per second is needed regardless how often the file is modified.

Parameter Settings We describe important parameter settings and their tuning guidelines in FlashFQ. The depth D in SFQ(D) represents the maximum device dispatch parallelism. A higher depth allows the exploitation of more parallel efficiency (if supported on the device) while large parallel dispatches weaken the scheduler's ability to regulate I/O resources in a fine-grained fashion. Our basic principle is to set a minimum depth that can exploit most of the device-level I/O parallelism. According to the parallel efficiency of the three SSDs in Figure 2, we set the depth D to 16 for all three SSDs.

For throttled dispatch, we set the task progress difference threshold that triggers the dispatch-blocking to be 100 millisecs. This threshold represents a tradeoff between fairness and efficiency—how much temporary resource utilization imbalance is tolerated to utilize restricted device parallelism?

The "active" status anticipation timeout is set to 20 millisecs—a task is considered to be continuously active as long as its inter-request thinktime does not exceed 20 millisecs. We set a shorter timeout (2 millisecs) for the anticipation that blocks aggressive tasks while leaving the device idle. The latter anticipation timeout is shorter because it may waste resources (as explained in Section 4.3).

I/O Context Our FlashFQ design in Section 4 uses a *task* to represent a resource principal that receives fairness support. In Linux I/O schedulers, each resource principal is represented by an *I/O context*. By default, a unique I/O context is created for each process or thread. However, it is sometimes more desirable to group a number of related processes as a single resource principal—for instance, all `httpd` processes in an Apache web server. In Linux, such grouping is accomplished for a set of processes created by the `fork()` / `clone()` system call with the `CLONE_IO` flag. We added the `CLONE_IO` flag to relevant `fork()` system calls in the Apache web server so that all `httpd` processes in a web server share a unified I/O context. We also fixed a problem in the original Linux that fails to unify the I/O context if `fork(CLONE_IO)` is called when the parent process has not yet initialized its I/O context.

One problem we observed in our Linux/ext4-based prototyping and experimentation is that the journaling-related I/O requests are issued from the I/O context of the JBD2 journaling daemon and they compete for I/O resources as if they represent a separate resource principal. However, since journaling I/O are by-products of higher-layer I/O requests originated from applications, ideally they should be accounted in the I/O contexts of respective original applications. We have not yet implemented this accounting in our current prototype. To avoid resource mis-management due to the JBD2 I/O context in Linux, we disabled ext4 journaling in our experimental evaluation.

6 Experimental Evaluation

We compare FlashFQ against alternative fairness-oriented I/O schedulers. One alternative is Linux CFQ. The second alternative (*Quanta*) is our implementation of a quanta-based I/O scheduler that follows the basic principles in Argon [19]. Quanta puts a high priority on achieving fair resource use (even if some tasks only have partial I/O load). All tasks take round robin turns of I/O quanta. Each task has exclusive access to the storage device within its quantum. Once an I/O quantum begins, it will last to its end, regardless of how few requests are issued by the corresponding task. However, a quantum will not begin, if no request from the corresponding task is pending. The third alternative is the FIOS I/O scheduler developed in our earlier work [17]. FIOS allows simultaneous request dispatches from multiple tasks to exploit Flash I/O parallelism, as long as the per-task timeslice constraint is maintained. FIOS also prioritizes reads over writes and it reclaims unused resources by inactive tasks. The fourth alternative is 4-Tag SFQ(D) [11]. Finally, we compare against the raw device I/O in which requests are always dispatched immediately (without delay) to the storage device.

Three of the alternative schedulers (Linux CFQ, Quanta, and FIOS) are timeslice-based. Timeslice parameters for these schedulers follow the default settings for synchronous I/O operations in Linux. Specifically, Linux tries to limit the epoch size and the maximum unresponsiveness at 300 millisecs. Therefore when multiple (n) tasks compete for I/O simultaneously, the per-task timeslice is set at $\frac{300}{n}$ millisecs. This setting is subject to the lower bound of 16 millisecs and the upper bound of 100 millisecs in Linux. To assess the effect of timeslice scheduling with short timeslices, we include a new setting that configures the per-task timeslice at $\frac{60}{n}$ millisecs when n tasks compete for I/O simultane-

ously (with the goal of limiting the maximum unresponsiveness at 60 millisecs). We also shorten the timeslice lower bound to 1 millisec. We include FIOS with such short timeslice setting in our evaluation and we call it *FIOS-ShortTS*.

Our experiments utilize the three Flash-based storage devices (Intel 311, Intel X25-M, and OCZ Vertex 3 SSDs) that were described earlier in Section 3. On both Intel SSDs, writes are substantially slower than reads (by about 4-fold and 6-fold on Intel 311 and Intel X25-M respectively). The Vertex drive employs a SandForce controller which supports new write acceleration techniques such as online compression. The Vertex write performance only moderately lags behind the read performance. For instance, a 4 KB read and a 4 KB write take 0.18 and 0.22 millisec respectively on the drive.

6.1 Evaluation on Task Fairness

Fairness is defined as the case that each task gains its share of resources in concurrent execution. When n tasks compete for I/O simultaneously, equal resource sharing suggests that each task should experience a factor of n slowdown compared to running-alone, or *proportional slowdown*. This is our first fairness measure. We further note that better performance for some tasks may be achieved when others do not utilize all of their allotted resource shares. Some tasks may also gain better I/O efficiency during concurrent runs by exploiting the device-level I/O parallelism. When all tasks experience better performance than the proportional slowdown, we further measure fairness according to the slowdown of the slowest task. Specifically, scheduler $\mathcal{S}_1$ achieves better fairness than scheduler $\mathcal{S}_2$ if the slowest task under $\mathcal{S}_1$ makes more progress than the slowest task does under $\mathcal{S}_2$.

We use a variety of synthetic I/O benchmarks to evaluate the scheduling fairness in different resource competition scenarios. Each benchmark contains a number of tasks issuing I/O requests of different types and sizes—

- a concurrent run with a reader continuously issuing 4 KB reads and a writer continuously issuing 4 KB writes;
- a concurrent run with sixteen 4 KB readers and sixteen 4 KB writers;
- a concurrent run with sixteen 4 KB readers and sixteen 128 KB readers;
- a concurrent run with sixteen 4 KB writers and sixteen 128 KB writers.

In order for these I/O patterns to reach the I/O scheduler at the block device layer, we perform direct I/O to bypass the memory buffer in these tests.

Figure 4 shows the fairness and performance under different schedulers. The raw device I/O, Linux CFQ, and 4-Tag SFQ(D) fail to achieve fairness by substantially missing the proportional slowdown in many cases. Specifically, lighter tasks (issuing reads instead of writes, issuing smaller I/O operations instead of larger ones) experience many times the proportional slowdown while heavy tasks experience much less slowdown in concurrent runs. Because raw device I/O makes no scheduling attempt, I/O operations are interleaved as they are issued by applications, severely affecting the response of light requests. The Linux CFQ does not perform much better because it disables I/O anticipation for non-rotating storage devices like Flash. For instance, without anticipation, two-task executions degenerate to one-request-at-a-time alternation between the two tasks and therefore poor fairness. 4-Tag SFQ(D) also suffers from poor fairness since its unthrottled parallel dispatches make it behave like the raw device I/O in many cases.

Under the Quanta scheduler, tasks generally experience similar slowdown in most cases. But such "fairness" is attained at substantial degradation of I/O efficiency due to its aggressive maintenance of per-task quantum. Specifically, its strict quanta enforcement throws away unused resources by some tasks. It also fails to exploit device I/O parallelism, as demonstrated by its poor performance in cases with large numbers of concurrent tasks.

Both FIOS and FlashFQ maintain fairness (approximately at or below proportional slowdown) in all the evaluation cases. Furthermore, both FIOS and FlashFQ can exploit the device I/O parallelism when available and achieve the best performance in all evaluation cases.

FIOS-ShortTS achieves good fairness for the single reader, single writer case (first row in Figure 4). However, it exhibits degraded fairness (compared to the original FIOS and FlashFQ) in cases with large numbers of concurrent tasks due to very short timeslices. In particular, it fails to maintain proportional slowdown for 16 4KB-writers, 16 128KB-writers on the two Intel SSDs (substantially so on Intel X25-M). It also produces relatively poor worst-task-slowdown compared to the original FIOS and FlashFQ in some other cases (particularly tests with 16 4KB-readers, 16 128KB-readers).

6.2 Evaluation on Responsiveness

The fairness evaluation shows that only FIOS and FlashFQ consistently achieve fairness for a variety of workload scenarios on the three SSDs. As a timeslice scheduler, however, FIOS achieves fairness at the cost of poor responsiveness. Even though FIOS allows simultaneous request dispatches from multiple tasks, the timeslice constraint at the end of each epoch still leads to long unresponsiveness for light tasks who complete their timeslices early.

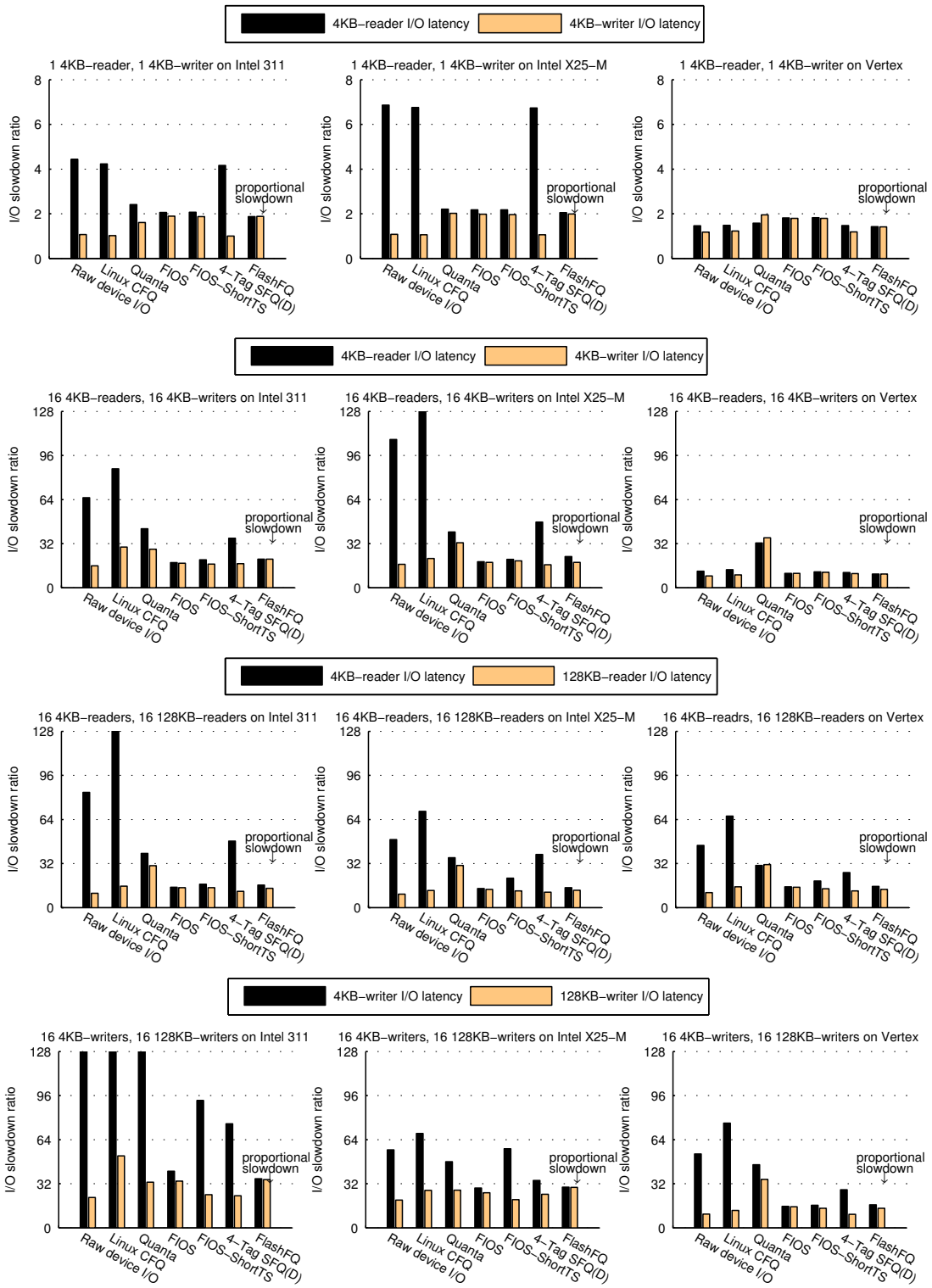


Figure 4: Fairness and performance of synthetic benchmarks under different I/O schedulers. The *I/O slowdown ratio* for a task is its average I/O latency normalized to that when running alone. For a run with multiple tasks per class (e.g., 16 readers and 16 writers), we only show the performance of the slowest task per class (e.g., the slowest reader and slowest writer). Results cover four workload scenarios (corresponding to the four rows) and three Flash-based SSDs (corresponding to the three columns). For each case, we mark the slowdown ratio that is proportional to the total number of tasks in the system.

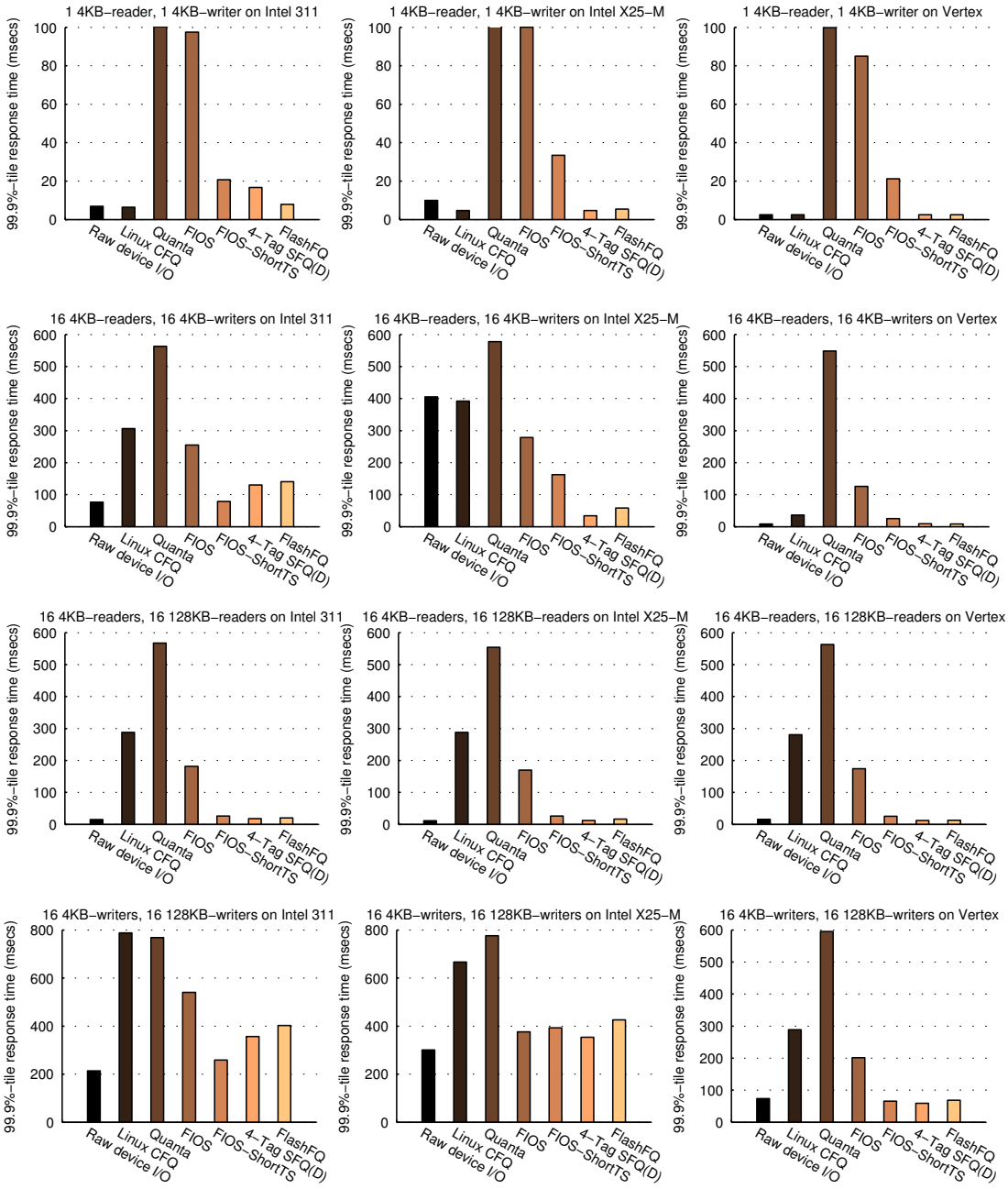


Figure 5: Worst-case (99.9-percentile) response time for four workload scenarios (rows) on three SSDs (columns) under different I/O schedulers.

In a system with high responsiveness, no task should experience prolonged periods of no response to its outstanding requests. We use the worst-case (99.9-percentile) I/O request response time during the execution as a measure of the system responsiveness. Figure 5 shows the responsiveness for our four workload scenarios on the three SSDs. Results clearly show poor responsiveness for the three timeslice schedulers (Linux CFQ, Quanta, and FIOS) in many of the test scenarios. In particular, they exhibit worst-case response time at half a second or more in some highly concurrent executions.

In comparison, FlashFQ shows much better responsiveness than these approaches (reaching an order of magnitude response time reduction in many cases). At the same time, we observe that FlashFQ's worst-case response time is quite long for the case of 16 4KB-writers and 16 128KB-writers on the two Intel drives (left two plots in the bottom row). This is due to the long write

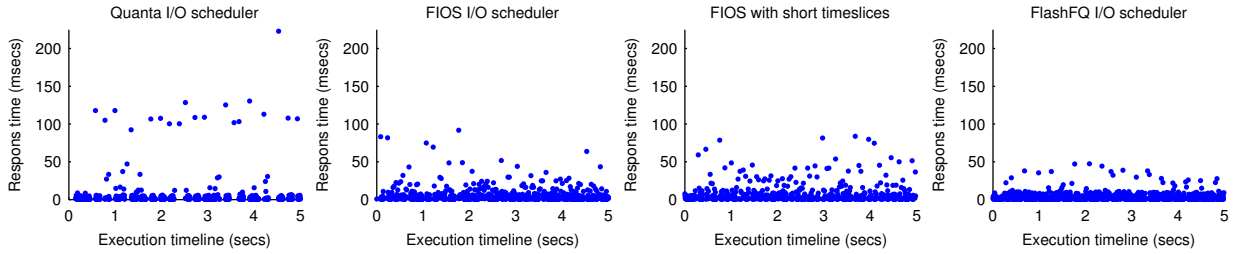


Figure 7: Time of Apache request responses under Quanta, FIOS, FIOS-ShortTS, and FlashFQ I/O schedulers. Each dot represents a request, whose X-coordinate indicates its timestamp in the execution while its Y-coordinate indicates its response time.

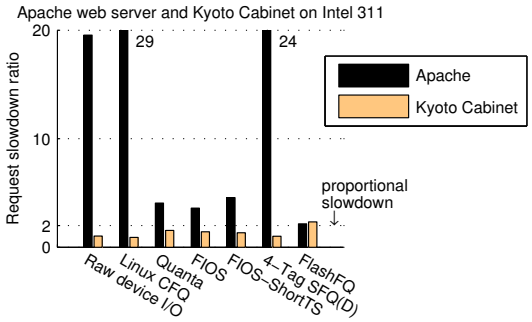


Figure 6: Fairness and performance of the read-only Apache web server workload running with a write-mostly Kyoto Cabinet key-value workload. The slowdown ratio for an application is its average request response time normalized to that when running alone.

time on these drives and the sheer amount of time to simply iterate through all 32 tasks while processing at least one request from each. This is evidenced by the long response time even under raw device I/O.

FIOS-ShortTS indeed exhibits much better responsiveness than the original FIOS. But this comes at the cost of degraded fairness (as shown in Section 6.1). Furthermore, FlashFQ still achieves better responsiveness than FIOS-ShortTS as any timeslice maintenance (even for very short timeslices) adds some scheduling constraint that impedes the system responsiveness.

6.3 Evaluation with the Apache Web Server and Kyoto Cabinet

Beyond the synthetic benchmarks, we evaluate the effect of I/O schedulers using realistic data-intensive applications. We run the Apache 2.2.3 web server over a set of HTTP objects according to the size distribution in the SPECweb99 specification. The total data size is 15 GB and the workload is I/O-intensive on a machine with 2 GB memory. As explained in Section 5, we attached the `CLONE_IO` flag to relevant `fork()` system calls in the Apache web server so that all `httpd` processes in the web server share a unified I/O context. Our web server is driven by a client that issues requests back-to-back (i.e., issuing a new request as soon as the previous one returns). The client runs on a different machine in a low-latency local area network.

Together with the read-only web server, we run a write-intensive workload on the Kyoto Cabinet 1.2.76 key-value store. In our workload, the value field of each key-value record is 128 KB. We pre-populate 1000 records in a database and our test workload issues "replace" requests each of which updates the value of a randomly chosen existing record. Each record replace is performed in a synchronous transaction supported by Kyoto Cabinet. In our workload, eight back-to-back clients operate on eight separate Kyoto Cabinet databases. All databases belong to a single I/O context that competes with the Apache I/O context.

Figure 6 illustrates the fairness under different I/O schedulers on the Intel 311 SSD. Since the Kyoto Cabinet workload consists of large write requests at high concurrency, it tends to be an aggressive I/O resource consumer and the Apache workload is naturally susceptible to more slowdown. Among the seven scheduling approaches, only FlashFQ can approximately meet the fairness goal of proportional slowdown for both applications. Among the alternatives, Quanta, FIOS and FIOS-ShortTS exhibit better fairness than others. Specifically, the Apache slowdown under Quanta, FIOS and FIOS-ShortTS are 4.1×, 3.6×, and 4.6× respectively.

Among the four schedulers with best fairness (Quanta, FIOS, FIOS-ShortTS, and FlashFQ), we illustrate the timeline of Apache request responses in Figure 7. Under Quanta, we observe periodic long responses (up to 200 millisecs) due to its timeslice management. The worst-case responses are around 100 millisecs under FIOS and FIOS-ShortTS. In comparison, FlashFQ achieves the best responsiveness with all requests responded within 50 millisecs.

7 Conclusion

This paper presents FlashFQ—a new Flash I/O scheduler that attains fairness and high responsiveness at the same time. The design of FlashFQ is motivated by unique characteristics on Flash-based SSDs—1) restricted parallelism with interference on SSDs presents a tension between efficiency and fairness, and 2) the diminished benefits of I/O spatial proximity on SSDs allow fine-grained task interleaving without much loss of I/O performance. FlashFQ enhances the start-time fair queueing schedulers with throttled dispatch to exploit restricted Flash I/O parallelism without losing fairness. It also employs I/O anticipation to minimize fairness violation due to deceptive idleness. We evaluated FlashFQ's fairness and responsiveness and compared against several alternative schedulers. Only FIOS [17] achieves fairness as well as FlashFQ does but it exhibits much worse responsiveness. FIOS with short timeslices can improve its responsiveness, but it does so at the cost of degraded fairness.

Acknowledgments This work was supported in part by the National Science Foundation grants CCF-0937571, CNS-1217372, and CNS-1239423. Kai Shen was also supported by a Google Research Award. We thank Jeff Chase for clarifying the design of the SFQ(D) scheduler. We also thank the anonymous USENIX ATC reviewers and our shepherd Prashant Shenoy for comments that helped improve this paper.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conf.*, pages 57–70, Boston, MA, June 2008.

[2] J. Axboe. Linux block IO — present and future. In *Ottawa Linux Symp.*, pages 51–61, Ottawa, Canada, July 2004.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE Int'l Conf. on Multimedia Computing and Systems*, pages 400–405, Florence , Italy, June 1999.

[4] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of Flash memory based solid state drives. In *ACM SIGMETRICS*, pages 181–192, Seattle, WA, June 2009.

[5] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured Flash file system for micro sensor nodes. In *SenSys'04: Second ACM Conf. on Embedded Networked Sensor Systems*, pages 176–187, Baltimore, MD, Nov. 2004.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM*, pages 1–12, Austin, TX, Sept. 1989.

[7] M. Dunn and A. L. N. Reddy. A new I/O scheduler for solid state devices. Technical Report TAMU-ECE-2009-02, Dept. of Electrical and Computer Engineering, Texas A&M Univ., Apr. 2009.

[8] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. on Networking*, 5(5):690–704, Oct. 1997.

[9] A. G. Greenberg and N. Madras. How fair is fair queuing. *Journal of the ACM*, 39(3):568–598, July 1992.

[10] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP'01: 18th ACM Symp. on Operating Systems Principles*, pages 117–130, Banff, Canada, Oct. 2001.

[11] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS*, pages 37–48, New York, NY, June 2004.

[12] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drives. In *EMSOFT'09: 7th ACM Conf. on Embedded Software*, pages 295–304, Grenoble, France, Oct. 2009.

[13] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Trans. on Computers*, Apr. 2011.

[14] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, July 2006.

[15] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, July 2008.

[16] A. K. Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Dept. Elec. Eng. Comput. Sci., MIT, 1992.

[17] S. Park and K. Shen. FIOS: A fair, efficient Flash I/O scheduler. In *FAST'12: 10th USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2012.

[18] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS*, pages 44–55, Madison, WI, June 1998.

[19] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST'07: 5th USENIX Conf. on File and Storage Technologies*, pages 61–76, San Jose, CA, Feb. 2007.

[20] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *ACM Trans. on Storage*, 2(3):283–308, Aug. 2006.

The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs

Laura M. Grupp[†], John D. Davis[‡], Steven Swanson[†]
[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Microsoft Research, Mountain View

Abstract

Recent years have witnessed significant gains in the adoption of flash technology due to increases in bit density, enabling higher capacities and lower prices. Unfortunately, these improvements come at a significant cost to performance with trends pointing toward worst-case flash program latencies on par with disk writes.

We extend a conventional flash translation layer to schedule flash program operations to flash pages based on the operations' performance needs and the pages' performance characteristics. We then develop policies to improve performance in two scenarios: First, we improve peak performance for latency-critical operations of short bursts of intensive activity by 36%. Second, we realize steady-state bandwidth improvements of up to 95% by rate-matching garbage collection performance and external access performance.

1 Introduction

NAND flash memory can provide orders-of-magnitude faster performance than traditional rotating media (HDDs), albeit at the cost of reduced capacity. Pushing flash to higher densities, causes significant decline in other metrics – like performance, endurance, and reliability. Increasing flash's capacity by storing an additional bit per memory cell (1 to 2 bits, or 2 to 3 for example) reduces the chip's lifetime by 5-10%, shrinks throughput by 22% to 98% (55% on average) and increases latency by 1.3× to 4.0× (2.3× on average) [14]. Increasing density via scaling leads to smaller, but still significant declines.

Despite the disturbing trends resulting from increasing the density of the underlying flash technology, flash systems remain very promising. The chip-level trends are driving the development of increasingly sophisticated flash management techniques. For example, sophisticated error coding techniques based on a deep understanding of flash's behavior [12, 5] can bring triple-level cell (TLC) bit error rates and performance in line with multi-level cell (MLC 2-bit/cell) technology [1], and aggressively exploiting parallelism can partially compensate for increasing latencies.

This paper exploits another characteristic of high-density flash devices to improve SSD performance. The dominance of MLC over SLC devices leads to systematic variation in the program latency of different pages. We have developed a flash translation layer (FTL) that schedules programs to pages according to the program operation's purpose (e.g., internal garbage collection vs. storing user data) and the speed of the page (i.e., faster or slower). Our scheduling algorithm improves performance without sacrificing capacity or endurance, providing speed of the hare (high performance) *and* the endurance of the tortoise (increased capacity and reduced write amplification). In particular, we make the following contributions:

- A flexible FTL which is aware of different page types and can direct operations accordingly.
- A *Many Write Point* mechanism for increasing scheduler flexibility and thereby enhancing the effect of scheduling policies.
- A scheduling policy that provides SLC performance on an MLC device for performance-critical operations and bursty workloads.
- An analytical model of steady state SSD performance that guides our access scheduler and suggests some non-intuitive scheduling algorithms.

Our FTL architecture and multi-write point mechanism allow the system to more readily access the array's variability. With this improved access and our policies, our FTL improves burst bandwidth by up to 36% (equal to the performance of an SLC array) with no increase in wear, and improves performance of sustained traffic by up to 95%.

First, we provide some background information on NAND flash and SSDs. Section 3 follows with a description of our baseline architecture, simulation infrastructure and our methodology. Next, Section 4 describes our

enhancements to the FTL which efficiently leverage page latency variation. We follow this with our evaluation in Section 5, suggestions for applying the mechanisms in Section 6, related work in Section 7, and conclusions in Section 8.

2 Background

NAND flash memory is the driving force behind the ongoing success of solid-state drives (SSDs). This section describes the basics of flash chip operation and the source, magnitude and patterns of page latency variation.

2.1 Flash memory

The packages composing the flash array in an SSD each contain one or more flash dies. Within a flash die, multiple (typically two) "planes" each contain several thousand 128 kB to 3 MB blocks that, in turn, contain 64 to 384 2-8 kB pages. The chips perform reads and writes on pages. However, before the chip can program (write) new data to a page, it must first erase the parent block. Further complicating writes, FTLs must write pages in order within each block. The FTL may skip over a page, but after doing so cannot write to it until after erasing it.

To represent the data, each memory element uses charge stored on a floating gate between the control gate and channel of a transistor. Varying amounts of charge on the floating gate determine the effective threshold voltage (V_{TH}) of the transistor, creating an analog range which the chip interprets as two regions for a single bit. Physically, a block comprises an array of "flash chains" that each contain 32-128 floating gate transistors in series with each other. To a first order, the n^{th} page in the block comprises the n^{th} bit in each of the block's chains (we discuss this more detail in Section 2.2).

Multi-level cell (MLC) flash stores multiple bits per floating gate (usually 2 bits) to improve density by interpreting the range of possible V_{TH} as 4 regions. This improved density (i.e., lower cost) makes MLC the dominant type of flash. Single-level cell (SLC) devices are less-dense, faster, and more expensive. TLC is in production systems and Macronix recently demonstrated 6-bit-per-cell technology [16]. We focus on the performance of the write operation in MLC devices in this study, and we discuss it in more detail in the next section.

Flash memories exhibit a well-known wear-out behavior which causes their data retention time to degrade with increasing program-erase (PE) cycle counts. Manufacturers rate current MLC devices for between 5,000 and 10,000 PE cycles, after which the data may become unrecoverable without very aggressive ECC protection. While wear-out remains a first-class concern, large over-provisioned flash arrays, common wear management techniques and recent advances in chip-level technology [11] help.

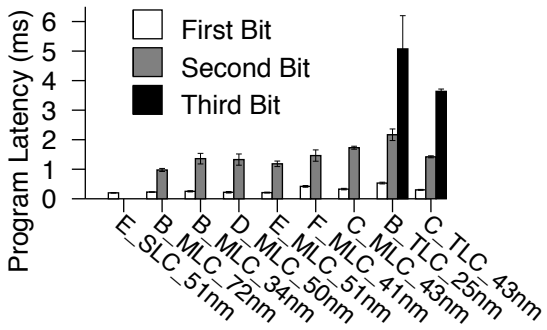


Figure 1: **Chip Program Latency** Multi-bit flash chips retain single-bit performance in their fast pages. The increase in latency is confined to the chips' added capacity.

2.2 Flash Chip Performance Variability

The techniques we propose exploit systematic page-level variation in write performance. This section describes the source of this variation, magnitude of variation we have measured in flash chips, the architectural lay-out of fast and slow pages within each flash chip, and how the FTL can non-destructively detect this pattern. Each of the 30 chip models (from 6 manufacturers) we have characterized show distinct groups of latencies in proportion with the number of bits stored in each memory element.

The variation arises because, although MLC devices store multiple bits on a single floating gate, those bits map into different pages. As a result, the programming operation for the first *fast* bit stored on the gate is much faster than the programming operation for the second *slow* bit, and so on for all additional bits stored in the cell. We refer to individual pages as fast or slow depending on which kind of bits they contain.

Figure 1 shows the latency of a representative sample of SLC, MLC and TLC chips. For each chip we measure the time to write random data to each page in 16 blocks. We divide these measurements into fast, slow and (for TLC) medium page latencies. Slow pages from the average MLC chip are 4.8× slower than fast pages, with D_MLC_50nm exhibiting the largest gap (6×) and F_MLC_41nm the smallest at 3.5×. Our data show that fast page program latency is comparable to SLC program latency in devices from similar technology generations [13].

Our previous work reveals two common organizations for fast and slow pages within an MLC block. We now extend those observations to TLC parts as well. With the exception of one manufacturer, the chips exhibit the organization in Figure 2A. In MLC devices, the first four pages are fast, the last four are slow and every pair of pages mid-block alternate between fast and slow. TLC devices cycle through the three latencies with pairs of

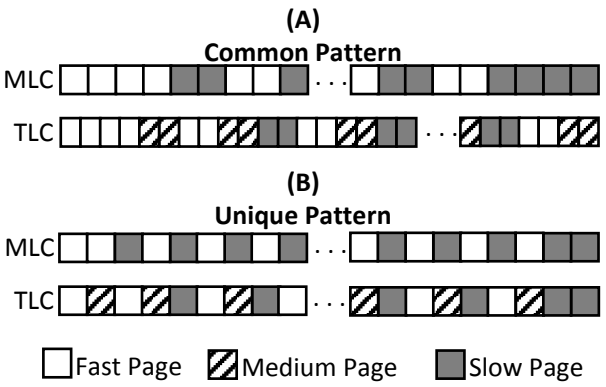


Figure 2: **Latency Pattern** Pages' read and write latencies follow the same pattern within each block of a given chip.

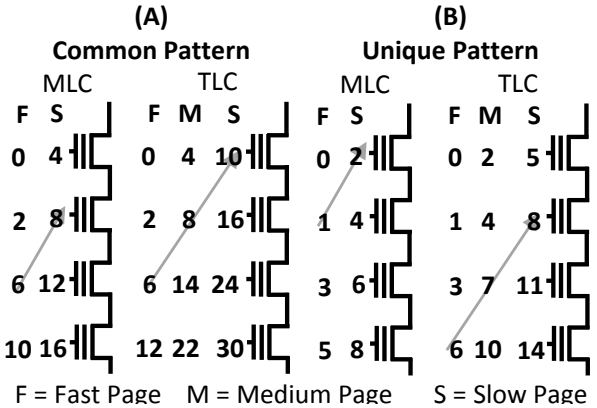


Figure 3: **Memory Cell Anatomy** Fast pages consist of each memory element's first-written bit. In-order programming causes the final bit of a memory cell to be written after most programs to the surrounding cells.

pages as well. The unique manufacturer follows the single-page alternating patterns in Figure 2B.

Figure 3 shows how a single bit from each page maps to the chain of flash memory cells. The numbers correspond to the page's location within the block and are in columns corresponding to the time required to program the bit. Figure 3A shows the even-numbered NAND chains from MLC and TLC parts made by most manufacturers (the corresponding odd chain is similar), and Figure 3B shows the pattern used by the manufacturer with a unique pattern.

Because of the in-order programming constraint, the final program of a cell occurs after most of the program operations to adjacent cells are complete. This reduces the program disturb that is a major hindrance to enabling multi-bit technology [21]. The blocks of most manufacturers alternate between page speeds in pairs because they separate pages into even and odd chains, while the unique manufacturer uses only one chain. Also, we observe most of the variation in the latency of slow pages (indicated by the wide error bars in Figure 1) comes from the even chain being slower than the odd chain, though we are unfamiliar with the cause.

The techniques we develop in the following sections depend on the FTL knowing the layout of fast and slow pages within a block. Since the layout is consistent for a given part number and does not vary over time, it is sufficient for the manufacturer to detect this pattern using a single block and configure the FTL accordingly. An FTL could perform the measurement at initialization time by monitoring the programming time of pages in a block, reducing the cost of moving to a new type of flash chip in an existing SSD design. There is also a non-destructive technique for determining page type. Page read latencies exhibit the same variation pattern. Furthermore, differentiating between the small number of possible patterns (either mentioned in the datasheets or derived empirically) requires only a few page reads.

Overall, as shown in Figure 1, the dramatic differences in page program latency provide a better opportunity to exploit diversity to improve SSD performance. In Section 4, we describe our extensions to the baseline FTL (from Section 3) which leverage these variations in program latency.

3 Baseline FTL

SSDs contain both an array of flash and a controller to manage wear leveling and access requirements while presenting a block interface. The following sections describe the basic algorithms needed in all FTLs, how we structure the algorithms to isolate important policy decisions, and our simulation infrastructure and array parameters.

3.1 FTL Basics

SSDs maintain a mapping between the logical block addresses (LBA) that the host system uses and the physical block addresses (PBA) that identify particular pages within the flash array. The FTL maintains this map with the goal of minimizing wear and maximizing performance. FTLs fall into three broad categories based on the granularity of this map – block-based, page-based and hybrids of the two. Improving the FTL is the object of intense work both in industry and academia (see Section 7).

In this work, we study variability-aware enhancements to a page-based FTL, but the concepts extend to other designs as well. We begin with the parallelized FTL architecture described in [7]. It uses log-structured write operations, filling up one block before moving on to another. To improve bandwidth, the FTL maintains one log for each chip in the array. We refer to the head of each

log as a *write point*.

As the FTL writes new data at a write point, the old version of the data for that LBA becomes invalid but remains in the array. The effects of this *copy-on-write* procedure requires that we provide functionality to (1) recover the physical-to-logical address mapping after unexpected power failure and (2) convert pages containing stale data to erased flash through garbage collection (GC).

First, for the FTL to recover from unexpected power failure it must track each page's logical address (LBA) as well as which copy of data for a given LBA is most-recent. With a single-write point array, a block sequence number suffices. However, when the system contains more than one write point, the FTL must use a page sequence number to maintain strict ordering. (See [6] and [7] for more details.)

Second, the FTL must remove the stale copies and create room for new data by performing GC. GC algorithms copy valid data from partially-invalid blocks to write points on or off chip, and erase the now fully-invalid blocks to make them ready for new write operations.

GC must constantly maintain a pool of erased blocks on each chip. When a write point reaches the end of a block, the block is full and the FTL must locate a new, erased block for that write point to continue writing. When a chip starts to run short on erased blocks, GC begins to consolidate valid data to create additional erased blocks for that chip. In the best case, garbage collection makes use of idle periods to hide its impact on performance. However, GC latencies are a significant source of performance variability in SSDs.

Our FTL uses two thresholds as parameters for the GC routines. The FTL maintains these thresholds on a per-chip basis, so in the worst case, any single chip can free up resources by taking itself off-line for cleaning. The first threshold is the *background (BG)* threshold. When the FTL finds any chip in the array idle, it performs GC operations on that chip up to the BG threshold. If the number of erased blocks on any chip drops below the second, *emergency* threshold, GC becomes the FTL's top priority for that chip and it will divert all incoming traffic to other chips or block entirely while GC proceeds. In normal operation, the FTL should very rarely enter this "emergency mode."

3.2 Design for Flexible Policy Choices

Figure 4 shows the high-level structure of the FTL's operation scheduler. The FTL maintains three queues. The queues hold write, erase, read and *cleanup* operations waiting to execute. External accesses to the SSD enter the *external* queue, background GC operations reside in the *background* queue, and the *emergency* queue holds

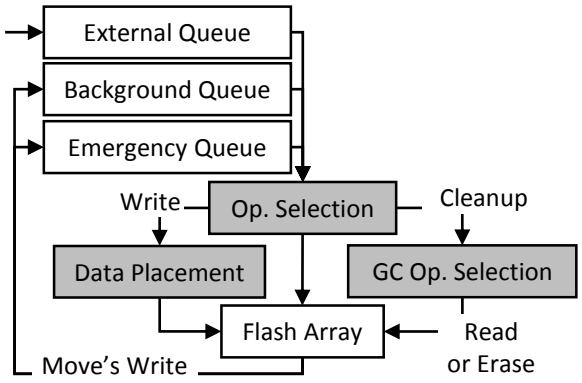


Figure 4: **Operation Flow** Operations move through the FTL's queues and a series of policy decisions (the gray boxes) before executing on a flash chip.

emergency GC operations. Emergency mode is a rare occurrence.

Operations pass from the queues to the flash array via three distinct policies, marked by the gray boxes in Figure 4:

Operation Selection Policy First, the FTL chooses which operation to execute next. Operations in the emergency queue have the highest priority. If the emergency queue is empty, or contains operations that cannot yet execute (for example, they must access a busy chip or wait for data being read), then an operation is taken from the external queue. Finally, operations are taken from the background queue when the system is idle.

Data Placement Policy The second policy in the FTL determines where to schedule writes. Because the physical address of an LBA changes with each write, the FTL has the freedom to choose, for example, the fastest page available. In our baseline design, the FTL follows a round robin approach which avoids busy chips and seeks to maintain a uniform number of valid LBAs on each chip.

GC Operation Selection Policy

The third policy is critical to efficient and flexible operation of GC. Rather than enqueue a list of move operations followed by one erase, we enqueue *cleanup* operations that represent one step in cleaning a block. The "Cleanup Operation Selection" policy in Figure 4 determines whether to start a read, write or erase operation. Delaying the choice of which page to move allows GC to adapt as pages become invalid due to external writes.

With GC policy reduced to the decision of executing one flash operation at a time, the particular algorithm is simple. Erasing fully invalidated blocks is the best option. When no such blocks are available, we move a page from a block with the least number of valid pages. A move begins with a read operation which, once complete

Parameter	Value
Channels	4 or 8
Dies per channel	2 or 16
Blocks per chip	2048
Pages per block	64 or 128
Bytes per page	4096
Fast Page Read Latency	27 μs
Slow Page Read Latency	40 μs
Fast Page Write Latency	253 μs
Slow Page Write Latency	1359 μs
Erase Latency	2871 μs

Table 1: **SSD Configuration** Architectural dimensions of the flash array and operation latencies to the flash chips.

pushes the paired write operation to the front of the queue from where the cleanup operation originated.

We will use this platform to demonstrate how to more effectively harness the variable performance available in high density flash. Many of these concepts and algorithms will transfer to the more memory-efficient hybrid FTL designs.

3.3 Simulation Setup

To evaluate these alternative organizations, we have developed a detailed trace-driven flash storage system simulator. It supports parallel operations between flash devices, models the flash buses and implements our FTL.

Table 1 details the array's dimensions. We model two moderately-sized SSDs – one to quickly simulate results for our microbenchmarks and a larger configuration to run the workloads. We also simulate an *All Fast* configuration, which models a half-capacity SLC-speed array by (1) reducing block size from 128 to 64 pages and (2) using only the fast read and write latencies.

Our SSD manages the array of flash chips and presents a block-based interface. The controller in the SSD coordinates 4 or 8 channels that each connect 2 chips to the controller via a 400 MB/s bus. Larger SSD configurations are possible, but the configurations we choose provide similar performance trends with much shorter simulation times.

To ensure steady state behavior, we arrange all of the LBAs randomly throughout the chips in the SSD before starting the simulations. We add enough invalidated pages to fill all blocks to the background threshold. The write points begin on a random page in the write point's assigned block.

4 Leveraging Variability

In this section, we describe our mechanisms for scheduling flash operations based on flash page performance variation. We demonstrate how careful, variation-aware scheduling can improve performance under both bursty and sustained workloads. With both mechanisms, we show how increasing the number of write points on each chip increases the FTL's ability to leverage the variability in its flash array.

4.1 Many Write Points for More Flexibility

Making good scheduling decisions requires the scheduler to have multiple options available, and without multiple options, no scheduling policy can have much impact on performance. Since each write point is associated with a single block, and the FTL must write to pages in the block in order, a single write point offers limited options: The FTL can either write to the next page (which may not be the type of page it would prefer) or it can skip the page, writing to the page of its choice, but wasting space.

Our baseline FTL maintains one write point per chip, which can only provide multiple options under light load (and some chips are idle). Under heavy load the FTL's only choice is to schedule an access to the most recently idled chip. Even under light load, a large burst of write traffic will use up the fast pages available on each write point. Both of these scenarios force the FTL to choose between the two undesirable options described above.

To provide flexibility, we extend the baseline FTL with multiple write points per chip, ensuring that the FTL will have choices and can make wise scheduling decisions. In the following subsections, we demonstrate how increasing the number of write points in the system and on each chip increases the policies' ability to access its desired page type.

While additional write points provide the flexibility to access fast and slow pages on demand, their number and use constitute a trade-off with over-provisioned capacity and data placement policies the FTL designer wishes to incorporate. Because each write point requires an open block, when the FTL maintains too many write points the over-provisioned space becomes too fractured across open blocks. In particular, the number of blocks between the background and emergency thresholds (for the GC routines described in section 3) provide a hard limit for the possible number of write points in our design. The FTL designer will also have to carefully weigh the value of placing data to potentially improve the efficiency of future GC with the effects of using a high or low latency page.

4.2 Handling Bursty Workloads

In this section, we present a policy called *Return to Fast* (RTF) that allows the FTL to service bursts of performance-critical operations exclusively with fast pages. The algorithm seamlessly provides nearly the speed of SLC while using all of the MLC pages.

We can apply the RTF policy in a number of situations. With an interface that passes information about the criticality of writes to the device, the system could schedule critical operations to fast pages. Such an interface could, for example, enable fast distributed locking protocols that require persistent writes for ordering via a log.

Even without changes to the interface, we can significantly enhance the performance of bursty workloads by treating user accesses as performance critical and GC operations as non-critical. In this case, we use fast pages exclusively until we run out, and then return to our baseline policy. We focus on this application in this paper.

RTF aims to service as many external writes as possible with fast pages. One approach is to skip over slow pages in order to move write points to the fast pages, but that would waste those skipped pages – reducing SSD capacity, invoking GC sooner, and increasing wear and potentially decreasing performance.

RTF avoids skipping pages by returning all write points to fast pages during the idle periods through GC writes. The FTL saves up a reserve of fast pages which it can spend on performance-critical operations. The number of write points in the system controls the size of reserve of fast pages.

The most common pattern of fast and slow pages provides up to two fast pages per write point. The FTL can fully exploit both pages in *Strongly RTF*, which ensures the write points reach the first of the pair of fast pages. The FTL can store an average of 1.5 writes per write point in *Weakly RTF*, which returns the write points to any fast page. Strongly RTF will give us the largest number of fast pages available after a large enough idle period.

We can further enhance the FTL with *preemptive GC*. During idle periods, the FTL continues to GC until each write point points to a fast page. This runs the risk of increased wear, when external writes or trims invalidate the pre-emptively moved data. However, simulation results show this is not a problem.

Increasing the number of write points in a system increases the performance of the bursts, even when the workload is a complex mix of reads, writes and potentially short idle times. In order for the FTL to direct an external write to a fast page, (1) there must be a write point already pointing at a fast page and (2) this write point must point to a chip which is not busy with another operation. Under a complex workload, the number of write points in the system is directly related to the likelihood of both of these conditions. The more write points there are, the more write points there will be pointing to fast pages. So, even with very little idle time we have increased the number of fast pages for the next burst.

A similar argument holds when you consider the contention over access to chips in the system. Imagine all but one of the chips in the array are blocked with operations. The single available chip is more likely to have a fast page available if there are more write points (and more possible pages available).

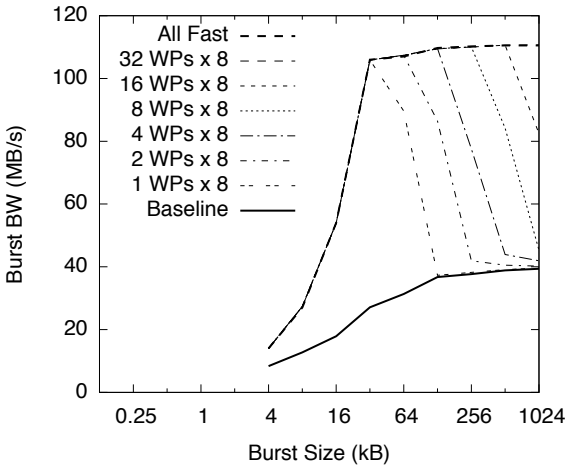


Figure 5: **Performance of Weakly RTF** The weakly RTF policy maintains performance comparable to using only fast pages for burst sizes up to the number of write points before dropping to the performance of using all page speeds.

4.2.1 Evaluating RTF

We explore the potential of the RTF policy by studying its behavior under a synthetic workload of page-sized accesses to uniformly distributed LBAs, grouped into bursts. The gap between bursts is sufficient to complete all necessary GC and return all the write points to fast pages, when applicable. Each trace uses a different burst size from 4 kB to 4 MB (1 to 1024 pages) and writes a total of 16 MB of data.

Figure 5 shows the performance of the Weakly RTF policy for 1-32 write points per chip on an 8 chip array (x8). For burst sizes less than 32 kB, the array is underused, but as the burst size reaches between one and two pages per chip the performance increases significantly for RTF and the All-Fast configuration. The baseline remains low with a maximum performance of 39.4 MB/s because it uses both fast and slow pages.

At burst sizes greater than 32kB, we observe the positive effect of additional write points in enabling RTF. With one write point, the FTL can manage only short bursts at high speed. Increasing the number of write points per chip provides a larger reserve of fast pages from which to draw and lets the scheduler make better decisions. For weakly (strongly) RTF, the maximum burst size serviced at high speed is equal to ($2\times$) the number of write points in the system times the page size.

4.3 Sustained Write-Intensive Workloads

RTF provides an effective tool for selective performance enhancement. However, under sustained write traffic, external operations must compete for resources with GC, which eclipses the performance benefits of RTF.

In this section, we develop a rate matching technique that allocates fast and slow SSD resources among GC and external operations for the best performance during long periods of sustained load. We begin with a variability-informed analytical model of an FTL, its page scheduling policy, and its GC. The model shows that in most cases the intuitive choice for page variability will lead to performance losses while the counter-intuitive choice improves performance. Finally, we study the potential of the FTL operating with these parameters.

4.3.1 Analyzing FTL Behavior Under Load

In order to maintain the erased block pool during periods of sustained, heavy load, the FTL must match the rate at which it erases pages with its external write rate. The per-chip bandwidths for these two operations remains constant, so the FTL matches these rates by establishing the correct number of chips performing each of the two sets of operations. Equations 1 and 2 describe the two per-chip bandwidths. For Equation 2, we assume 20% over-provisioning and include a parameter (*pgsMvd*) for the number of page moves GC must perform on the average block (which is determined by the workload's locality).

$$ExternalWriteBW = \frac{pageSize}{wLat} \quad (1)$$

$$GC\_BW = \frac{0.2 * blockSize}{pgsMvd * (mvLat) + eLat} \quad (2)$$

With respect to write latency variability, we consider two choices. The FTL could use slow pages to service GC writes and fast pages to service user writes (SGC), or vice versa (FGC).

Figure 6 plots the SSD's bandwidth for these policies and a baseline, latency agnostic configuration over a range of workload localities. Our model assumes the FTL always has access to the preferred page speed without skipping pages. For the FGC configuration, for example, we determine the per-chip user write BW and the cleaning BW using slow page write latency for Equation 1 and fast page write latency for Equation 2, respectively. The ratio of the two yields the correct ratio of chips to use for each operation. The chip counts are averaged over time, so they do not need to be integers. Ultimately these values yield the user-visible write bandwidth.

Without the analytical model, our initial choice was to accelerate external operations, corresponding to the SGC configuration. However, as Figure 6 shows, the highest performance configuration allocates fast pages to online GC instead (FGC).

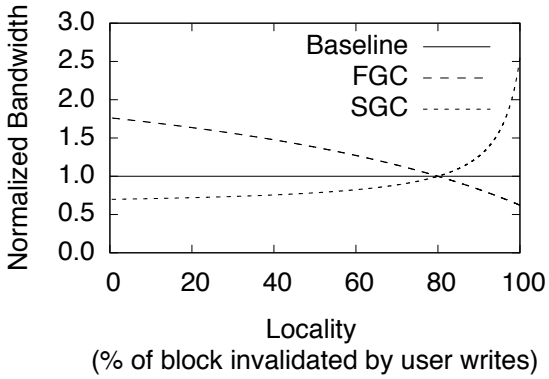


Figure 6: **Design Space for Rate Matching** Which configuration to use under heavy load depends on the workload's locality. If locality is low (less than 80% on this graph), GC must move lots of data and prioritize those writes to fast pages to improve overall performance.

Scenarios with average to low page locality will do best under FGC, because GC reclaims relatively few erased pages for many moves. SGC experiences a disadvantage because fast user writes and slow GC writes exacerbate the inherent slowness of GC. FGC, on the other hand, uses the speed of fast pages to help GC to keep pace with the user accesses. Because block erase is necessary, and such a heavy weight process, the FTL does best by completing it quickly.

The crossover point falls exactly at 80% locality because of the particular amount of over-provisioning in our array (20%). The analytical model frees 20% of the pages in a block for the average whole-block GC sequence. With 80% locality, the number of pages erased per block GC equals the number of pages moved, and so external write BW is the same as GC write bandwidth for all configurations. As locality decreases from this crossover point, GC requires more moves and higher-performing writes (FGC).

In order to study FGC and SGC, we make two changes to the baseline FTL. The first does not include knowledge of page variability and is simply to maintain the pool under sustained write traffic. To do this, we modify the operation selection policy. We calculate the ratio of per-chip GC bandwidth to per-chip external write bandwidth, called the *target ratio*. The FTL maintains a *chip use ratio* by monitoring the ratio of time spent on GC and external write operations for the recent history. The FTL then chooses the next operation by attempting to match the chip use ratio to the target ratio.

The second policy change accounts for page variability in the data placement policy by directing pages to match

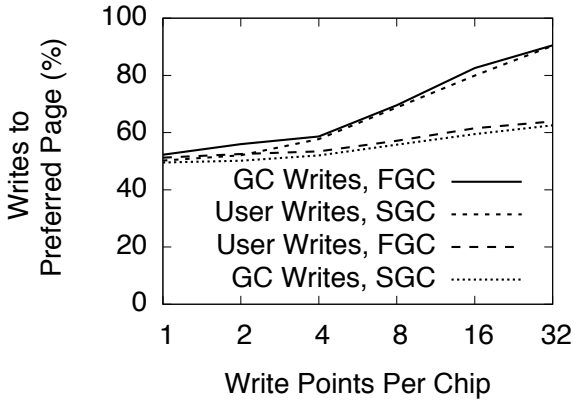


Figure 7: **Page Preference Improvement** Increasing the number of write points per chip increases the availability of the preferred page type when the SSD is under heavy load.

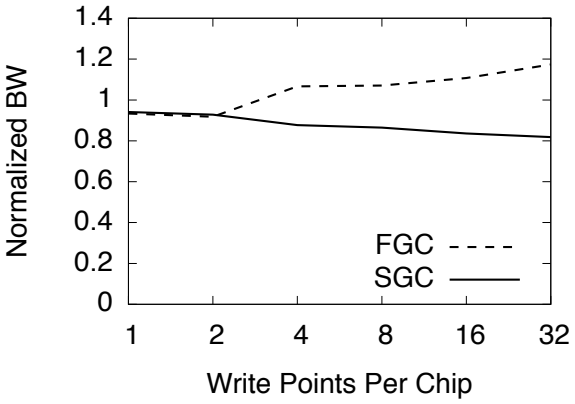


Figure 8: **Sustained Performance** Adding write points allows fast online GC to improve SSD performance by 20%.

either the SGC or FGC configurations. We implement a page preference policy whereby given the choice between several locations to write, the FTL prefers to direct the previously chosen operation according to the SGC or FGC configuration.

The baseline for studying the FTL under sustained load includes the changes to the operation choice policy, but retains the original round robin baseline for the write point choice policy.

4.3.2 Evaluating FGC and SGC

To study rate matching with page preference under the complex constraints imposed by a real FTL, we apply a write-intensive synthetic load to our simulator. The workload consists of 5 s pulses of infinite load followed by 4 s of idleness. This cycles repeats 80 times, and the load consists purely of writes with evenly distributed LBAs.

Under such a load, all operations reach the Data Placement policy with only one idle chip in the flash array. Because each chip only has one write point, the page preference has no effect, and all operations have an equal probability of being written to fast or slow pages. Skipping pages is not a good option because its negative effect on performance overwhelms any advantage gained from using fast pages, due to the added GC.

Write points again provide the flexibility needed for the FTL to leverage the fast pages in the FTL. With multiple write points on each chip, when the operation arrives with only one idle chip from which to select, it still has multiple options for where it can write.

Figure 7 shows how, as the number of write points increases, the FTL can run operations on the desired pages type more frequently. With one write point, both SGC and FGC direct their operations to the two page speeds with equal probability. As the number of write points increases, a larger percentage of operations are scheduled to their preferred page speed. This is especially true when that preferred page speed is fast.

Figure 8 shows the performance resulting from the FTL accessing its preferred pages more often, normalized to the baseline of no page preference. As more write points allow the FTL to select its preferences, the performance of FGC improves while the performance of SGC declines.

These results verify that the optimal choice for page preference under heavy write load is to save fast pages for servicing online garbage-collecting moves (FGC), and that increasing the number of write points on each chip better enables the FTL to tap into that supply of fast pages.

5 Results

In this section we evaluate the effectiveness of our variability aware FTL policies – RTF, FGC and SGC – on a set of five benchmarks.

5.1 Workloads

Table 2 describes the five trace files we use to explore our proposed FTL enhancements. Their burst sizes span a range as do the idle times between each burst.

The conventional method of replaying traces does not accurately retain fixed computation time (seen by the SSD as idle time). This runs the risk of mixing the idle and active parts of the workload which could both (1) eat into the idle time needed for RTF and (2) lessen the load FGC and SGC are intended to accommodate.

We pre-process our trace files to alleviate these problems. Instead of each trace line indicating what time it arrives at the SSD, it indicates how much later than the previous trace line it arrives. Then, if the delta is below a

Trace Name	Min. Δ (Thresh.)	Avg. Burst Size (pgs)	Avg. Idle Time (s)	Description
Build	0.087 s	3.56	1.74	Compilation of the Linux 2.6 kernel.
Financial	18 ms	0.140	0.0620	Live OLTP trace for financial transactions.
WebIndex	48 μs	212	0.000564	Indexing of webpages using Hadoop.
Swap	150 ms	0.0645	0.0218	Virtual memory trace for desktop applications.
DeskDev	0.7 s	4.48	3.82	24 hour trace of a software development work station.

Table 2: **Workload Statistics** Characteristics of the burstiness of our tracefiles and the idle times between the bursts.

particular per-trace threshold, we group that access in the same burst with the previous access by setting the delta to zero. In this way, we ensure the SSD experiences the full brunt of the burst without added idle time.

We assume that a large enough idle period (i.e. that greater than the threshold) indicates the program is executing calculations using the previous burst's data. We also assume that the amount of time before issuing its next burst will remain constant for a given processor architecture. We then enforce the delta time between each burst by issuing the first access of a given burst *delta* seconds after the previous burst completes (i.e. after the completion of the last access).

We set the delta threshold to be the average time between each trace line for a given file. Table 2 details the delta threshold for each trace file as well as the average size of the bursts and average amount of idle time between them.

Measuring the performance of an SSD running a trace file that includes idle time requires some care. To factor out the effect of idle time in the trace file, we divide the amount of data written in a given burst by the time it takes to complete that burst (this is the burst's write bandwidth). We then report the average of these bandwidths for each policy normalized to the baseline.

5.2 Return To Fast

Figure 9 shows the performance of the delta traces running under the Strongly RTF (sRTF) and weakly RTF (wRTF) policies with 1, 8 and 32 write points per chip. The All-Fast configuration shows a potential for 19% to 62% increase in write performance (34% on average) over the baseline and all traces realize at least a portion of these gains. On average, traces realize a 9% performance increase going from 1 to 32 write points per chip and no increase in performance for using strongly RTF rather than weakly RTF.

Financial (*Fin.* in the figures) works well with RTF – it contains a significant amount of idle time between bursts for recovery, and has very few reads which could block and stall the burst. Financial also has very few writes in each burst, so the SSD is able to realize the full potential of the fast pages with very few write points. For other workloads, added performance comes with more write points because a larger pool of fast pages increases the options for where to write, getting around the effect of blocking reads. All workloads on both strongly and weakly RTF achieve more than 24% of the All Fast configuration's gains and most see more than 64%.

While RTF consistently improves the write performance, it has negligible effect on the read performance. On average the RTF configurations gain less than 0.1% in read bandwidth.

Figure 10 shows the wear out experienced by our SSD under the different workloads and RTF configurations. Trying to achieve high performance by using only the fast pages significantly increases the wear – up to 2.0×, and 1.7× on average. However, if we instead fill the slow areas with garbage collected data we were planning on moving anyway, our wear increases by 5% relative to the baseline on average, and never more than 34%.

5.3 Rate Matching with FGC and SGC

Figure 11 shows the performance of the traces running on the FGC and SGC rate matching policies using 1, 8 and 32 write points per chip. The All-Fast configuration is able to realize much larger gains over the baseline, because the FTL makes use of all of the pages during external activity. Even so, the FGC configuration on most workloads achieves a significant portion of these gains while the more intuitive SGC configuration remains at baseline levels. DeskDev reaches the highest performance at 95% above baseline, and the average of all the traces except for WebIndex reaches 65% over baseline.

The spacial locality in the WebIndex's writes set this workload apart – in this case the intuitive choice of directing external operations to fast pages (SGC) provides better performance. WebIndex exhibits an average of 31% fewer moves per erase, placing it in the right-most region of Figure 6. The advantage of saving fast pages for online operations in FGC is a result of completing GC as fast as possible to match the rate of external writes. However, when the access stream exhibits good spacial locality, the act of writing external operations invalidates pages on a small set of blocks, accelerating GC.

Increasing the number of write points on each chip al-

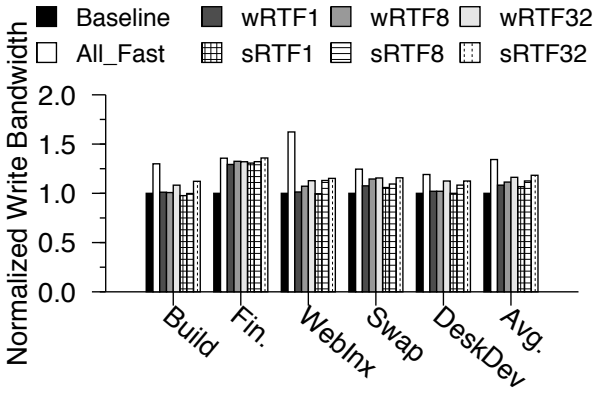


Figure 9: **Performance of RTF** More write points in the flash array increases the reserve of fast pages the FTL can build during idle periods, allowing the FTL to absorb larger burst with only fast pages.

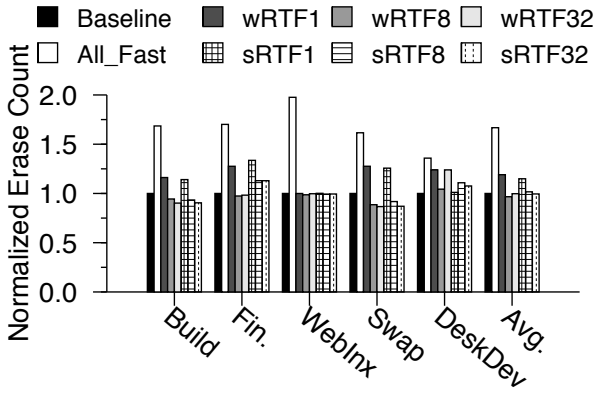


Figure 10: **Wear of RTF** While RTF improves performance, on average its wear is nearly that of the baseline.

lows each configuration to approach the predicted behavior. SGC almost always performs on par or worse than the baseline, often declining from baseline as the number of write points decreases. The opposite trend holds for FGC, frequently beginning with a performance better than baseline and increasing as the number of write points increases. This makes sense because increasing the number of write points increases the impact of each policy. Since SGC hurts performance, adding write points makes performance worse.

While FGC and SGC produce performance gains and losses, respectively, in most cases they both perform a number of erases on-par with the baseline (Figure 12). Excluding WebIndex, the erase count declines by as much as 32% for the SGC-32 configuration on DeskDev, and increases by no more than 2% (Excluding Financial). On average, FGC and SGC experience a 3% decline in wear while the All-Fast configuration is 56% more wear compared to the baseline.

6 Application

Although we propose distinct mechanisms for bursts and heavy load, we discuss their coordination with other policies in the system to address a variety of workloads with mixed access patterns. This section describes how this could be done either through coordination with the operating system or by further enhancing the FTL.

OS Support Coordination with the operating system constitutes one avenue of leveraging the Harey Tortoise techniques. The OS could provide hints with the accesses made to the SSD. For example, the FTL could use RTF to service latency-critical accesses (marked as high priority), providing the functionality of the variability aware FTL in [13] without the added wear. Alternately, the OS could signal a course-grained switch between workload style when, for example, a server transitions between workloads or activities that change between peak and off-peak periods. An enhanced interface, such as NVMe [2], would facilitate these implementations.

Dynamic FTL Without hints from the OS, the FTL could combine the Harey Tortoise's policies to accommodate mixed workloads. It would adjust as a burst of accesses of unknown length progresses – employing RTF early in the burst before transitioning to RM techniques as the "burst" lengthens to a sustained load. This technique would result in RTF accommodating small bursts while the FTL treats long bursts with RM techniques.

For long bursts and sustained load, the FTL would step through several phases combining our techniques proposed in this work. For such a policy, GC during idle period should employ RTF to return as many write points as possible to fast pages. Then, when accesses arrive, the FTL would achieve maximum possible performance from using only fast pages under RTF, before gradually transitioning to RM policies.

During the transition period the FTL would (1) adjust the preference for fast or slow pages of the external and GC writes and (2) tailor the use and cleaning rates to use up the over-provisioned space and create a graceful degradation of performance. The latter could be achieved by relating the target and chip time ratios by some factor which dynamically adjusts to one.

Finally, when the pool of erased blocks reaches a sustainable minimum, the FTL would work exclusively with the RM policies until an idle period allows for additional GC. In this way, the FTL would provide high performance to small bursts and gradually ramp down to a maximum, sustainable performance.

The inversion of preference (for RM) with good write locality suggests another dimension for exploring how to detect and adapt to the correct choice of page preference.

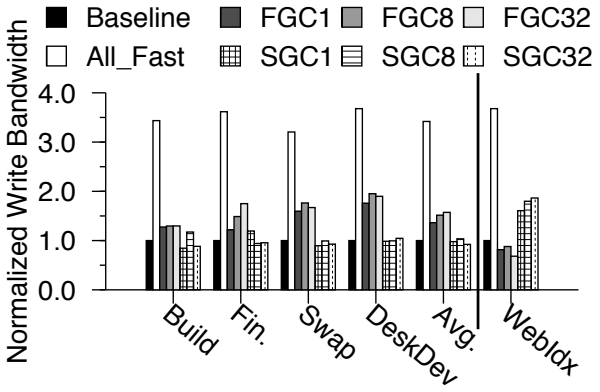


Figure 11: **Performance of FGC and SGC** The counter intuitive choice of servicing online operations with fast pages (FGC) improves the performance, when spacial locality is low.

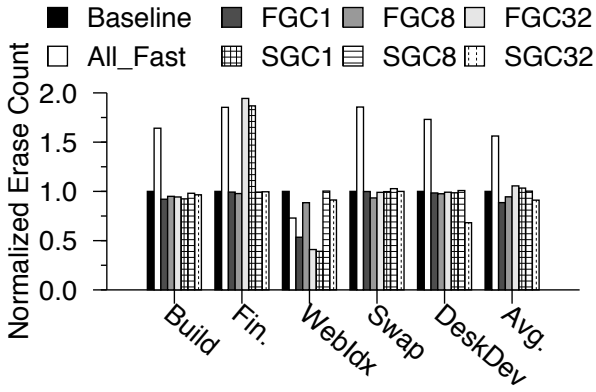


Figure 12: **Wear of FGC and SGC** Leveraging page variability during heavy load does not effect device wear out in most cases.

7 Related Work

There is a large body of flash-based storage research spurred on by the promise of high performance, low energy, and the limitations imposed by its idiosyncrasies. The research most closely related to our work falls in four categories: Mode-switching Flash, FTL algorithms, SSD interleaving, and write buffers. All of these topics try to improve the performance, endurance and/or reliability of the SSD, but do not leverage or address the variability inherent in MLC flash. The final section of related work discusses the emerging research that embraces flash page variability.

Mode-Switching Flash: Changing the cell bit density has been proposed in research [18] and implemented by SSD vendors [24, 20] to improve reliability, endurance, and performance. Switching between MLC mode and SLC mode does have the drawback of sacrificing half of the system capacity. In our work, by leveraging write latency asymmetry across the pages, we are able to approximate the performance of SLC without sacrificing device capacity, the best of both worlds. Furthermore, because we use all the pages in the block by not throwing away the slow pages, we reduce the number of erase cycles, improving overall system endurance and reliability.

FTL Algorithms: There is a large body of work focused on FTL optimizations to improve SSD performance, endurance and reduce memory overhead based on access pattern or application behavior. By using an adaptive page- and block-level addressing mapping scheme, KAST [17], ROSE [10] and WAFTL [27] are able to improve performance, reduce garbage collection overhead and reduce FTL address mapping table size. DFTL [15] goes one step further by caching a portion of the page-level address mapping table for reduced size and fast translation. MNFTL [23] reduces the number of valid page copies for garbage collection, explicitly targeting MLC flash. Finally, CAFTL [9], removes unnecessary duplicate writes and increases the lifespan of the SSD. While some of these FTLs address workload variability, none address the variability in the underlying MLC flash.

SSD Interleaving: Intra-SSD parallelism has been explored by many groups [3, 7, 22, 28, 8, 25, 4]. By not only issuing operations in parallel at the package-, die-, and plane-level, others have also shown that rescheduling operations can improve performance [28]. Our work dives deeper into parallel data placement by providing multiple write points for fast pages within the plane that can adsorb burst and sustain high write performance, on par with SLC devices.

Write Buffers: Historically, buffers have been used in HDD to improve read and write performance. Likewise, write buffers have been shown to improve random write performance in SSDs [19]. These write buffers are also sufficient for handling small burst sizes. More recently, research has shown that per package queues and operation reordering provide more opportunities for parallel operations and further improve performance over LRU-based write buffer mechanisms [25]. Write points can be used in conjunction with write buffers, providing the FTL with more flexibility in data placement, in light of the performance asymmetries that exist in MLC flash.

Variability: The quest for higher density flash has provided opportunities to exploit the variability in flash page latency. Previous work [13] has exposed these asymmetries and predicted their impact on future SSDs [14]. Other work has exploited the differences in the flash to improve error correction [12] or guarantee other properties, like secure erasure [26]. We demonstrate that the FTL can take advantage of flash variability to improve performance while not sacrificing endurance or capacity.

8 Conclusion

In this paper, we developed an FTL that leverages systematic variability in flash memory to provide the speed of the hare (SLC) with the capacity of the tortoise (MLC). We propose increasing the number of write points on each chip to increase the flexibility of the FTL to schedule accesses to pages with a variety of latencies, and we demonstrate how to use this flexibility to achieve up to 100% of the performance an SLC array (or an average of 89%) by using MLC flash without additional wear. Further, we show that the counterintuitive approach of scheduling garbage collection operations on fast pages improves performance by an average of 65% and as much as 95% in workloads with little spacial locality.

Acknowledgements

We would like to thank the reviewers and shepherd of this paper for their valuable input. This work was supported by the NSF Variability Expedition under award number 1029783.

References

[1] Densbits technologies. memory modem: Technology overview. April 2012.

[2] Nvm express. http://www.nvmexpress.org/. 2013.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, 2008.

[4] S. Bai and X.-L. Liao. A parallel flash translation layer based on page group-block hybrid-mapping method. *Consumer Electronics, IEEE Transactions on*, may 2012.

[5] A. Berman and Y. Birk. Constrained Flash memory programming. In *IEEE International Symposium on Information Theory*, pages 2128–2132, 2011.

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. Technical Report MSR-TR-2005-176, Microsoft Research, December 2005.

[7] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Architectural Support for Programming Languages and Operating Systems*, pages 217–228, 2009.

[8] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[9] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *USENIX Conference on File and Storage Technologies*, pages 77–90, 2011.

[10] M.-L. Chiao and D.-W. Chang. ROSE: A Novel Flash Translation Layer for NAND Flash Memory Based on Hybrid Address Translation. *IEEE Transactions on Computers*, 60:753–766, 2011.

[11] H.-T. L. et. al. Radically extending the cycling endurance of flash memory (to ¿100m cycles) by using built-in thermal annealing to self-heal the stress-induced damage. 2012.

[12] R. Gabrys, E. Yaakobi, L. M. Grupp, S. Swanson, and L. Dolecek. Tackling intracell variability in tlc flash through tensor product codes. In *International Symposium on Information Theory*, ISIT, 2012.

[13] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *International Symposium on Microarchitecture*, pages 24–33, 2009.

[14] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *USENIX Conference on File and Storage Technologies*, 2012.

[15] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2009.

[16] K.-C. Ho, P.-C. Fang, H.-P. Li, C.-Y. Wang, and H.-C. Chang. A 45nm 6b/cell Charge-Trapping Flash Memory Using LDPC-Based ECC and Drift-Immune Soft-Sensing Engine. In *Solid-State Circuits IEEE International Conference*, 2013.

[17] H. jin Cho, D. Shin, and Y. I. Eom. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Design, Automation, and Test in Europe*, pages 507–512, 2009.

[18] T. Kgil, D. Roberts, and T. Mudge. Improving nand flash based disk caches. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, june 2008.

[19] H. Kim and S. Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, 2008.

[20] G. e. a. Marotta. A 3bit/cell 32gb nand flash memory at 34nm with 6mb/s program throughput and with dynamic 2b/cell blocks configuration mode for a program throughput increase up to 13mb/s. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 444 –445, feb. 2010.

[21] K.-T. Park, M. Kang, D. Kim, S.-W. Hwang, B. Y. Choi, Y.-T. Lee, C. Kim, and K. Kim. A Zeroing Cell-to-Cell Interference Page Architecture With Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories. *IEEE Journal of Solid-state Circuits*, 43:919–928, 2008.

[22] S.-H. Park, S.-H. Ha, K. Bang, and E.-Y. Chung. Design and analysis of flash translation layers for multi-channel nand flash-based storage devices. *Consumer Electronics, IEEE Transactions on*, august 2009.

[23] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan. Mnftl: An efficient flash translation layer for mlc nand flash memory storage systems. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, 2011.

[24] D. Raffo. Fusionio builds ssd bridge between slc,mlc, july 2009.

[25] X. Ruan, Z. Zong, M. I. Alghamdi, Y. Tian, X. Jiang, and X. Qin. Improving write performance by enhancing internal parallelism of solid state drives. In *Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International*, dec. 2012.

[26] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, 2011.

[27] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada. WAFTL: A workload adaptive flash translation layer with data partition. In *Symposium on Mass Storage Systems*, pages 1–12, 2011.

[28] S. yeong Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee. Exploiting internal parallelism of flash-based ssds. *Computer Architecture Letters*, jan. 2010.

Janus: Optimal Flash Provisioning for Cloud Storage Workloads

Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji
François Labelle, Nate Coehlo, Xudong Shi, C. Eric Schrock
{calbrecht,aamerchant,mstokely}@google.com,mhwaliji@gmail.com
{flab,natec,xdshi,eschrock}@google.com
Google, Inc.

Abstract

Janus is a system for partitioning the flash storage tier between workloads in a cloud-scale distributed file system with two tiers, flash storage and disk. The file system stores newly created files in the flash tier and moves them to the disk tier using either a First-In-First-Out (FIFO) policy or a Least-Recently-Used (LRU) policy, subject to per-workload allocations. Janus constructs compact metrics of the *cacheability* of the different workloads, using sampled distributed traces because of the large scale of the system. From these metrics, we formulate and solve an optimization problem to determine the flash allocation to workloads that maximizes the total reads sent to the flash tier, subject to operator-set priorities and bounds on flash write rates. Using measurements from production workloads in multiple data centers using these recommendations, as well as traces of other production workloads, we show that the resulting allocation improves the flash hit rate by 47–76% compared to a unified tier shared by all workloads. Based on these results and an analysis of several thousand production workloads, we conclude that flash storage is a cost-effective complement to disks in data centers.

1 Introduction

Disks are slow, and not getting much faster, even as their capacities grow: the random I/O operations possible per gigabyte stored on disk continues to decline. We can compensate for this by adding flash storage, which supports a much higher I/O rate per byte of storage capacity. Since flash is expensive per byte compared to disk, it is best to provision a relatively small amount of flash to store the most frequently accessed data.

Storage needs in a large cloud environment are often highly varied between different users and workloads [15, 23]. Hence, distributing the available flash capacity uniformly between the workloads is not ideal from either a performance or a cost perspective. Instead, we seek to leverage the differences between the competing users and workloads to optimize the provisioning of flash.

Our system, Janus, provides flash provisioning and allocation recommendations for both individual users and system administrators of large cloud data centers, where many users share the resources. Janus uses sparse traces, such as Dapper traces [22], to build a compact characterization of how effective flash storage is for different workloads. Where flash provisioning decisions are made by individual users, this characterization can be used to determine how much flash storage is cost-effective to purchase. For the case where resources are provisioned and allocated centrally by a system operator, we set up an optimization problem to partition the available flash between workloads so as to maximize the overall reads from flash and show how to solve it efficiently.

Janus recommendations are used by several production workloads in our distributed file system, Colossus [17]. We provide evaluations of the effectiveness of the recommendations from measurements on some of these workloads, and additional evaluations using traces of other production workloads. Our workload characterizations show that most I/O accesses are to recently created files. Based on this observation, files are placed in the flash tier upon creation and moved to the disk tier using FIFO or LRU eviction policies. Our results show that the recommendations allow 28% of read operations to be served from flash by placing 1% of our data on flash.

The three main contributions of this paper are:

- A characterization of storage usage patterns in a large private cloud focusing on the age of data stored and I/O rates to recently written data (§ 4).
- An optimization problem formulation for flash allocation to groups of files to maximize read rates offloaded to flash weighted by priorities and bounded by maximum flash write rates (§ 6).
- Experimental results from an implementation for the Colossus file system (§ 8).

2 Related Work

Several types of multi-tier storage systems [16] have been developed for memory, solid state drives, disk, and tape. These include Hierarchical Storage Management (HSM) [7, 12], multi-tier stores [24], multi-tier file systems [2], hybrid disk/flash storage [19], and extent-based enterprise volume management [24, 13]. Most include automated methods for migrating data between tiers based on I/O activity levels, performance requirements set by administrators, or explicit rules defined by users or administrators. However, none of these have focused on a distributed, cloud-scale deployment, which adds issues of provisioning policies and workload monitoring compatible with distributed management.

Several storage design tools, such as Minerva [3] and DAD [4], advocate principled, automated approaches to choose appropriate storage parameters for disk arrays based on workloads and desired availability characteristics. However, these tools typically provide only coarse-grained recommendations about RAID levels for storage volumes, unlike the data placement decisions for different files in a multi-tiered cloud storage environment described in this paper.

Studies on the distributed file system in Sprite [5] and the local file system in 4.2 BSD [20] showed the utility of characterizing user activity, access patterns, and file lifetimes when evaluating caching strategies. Blaze [6] analyzed access patterns affecting caching in a distributed file system using traces of I/O activity obtained by monitoring storage related remote procedure calls (RPCs). We similarly monitor storage RPCs in our distributed file system, but we also needed to use sampling and other statistical techniques due to the system scale.

TIP [21] used explicit hints of future I/O accesses provided by the application programmer to determine which data to prefetch and when. Janus does not rely on the explicit programmer action of adding hints to the API usage of the system. Instead, we predict the cacheability of different user workloads automatically from online measurements of past usage. Kroeger [14] predicts file access patterns in the context of prefetching at the Linux kernel level by using the sequence of past accesses; however, it is not clear how it could be extended to the distributed case.

Our approach is most closely related to the work of Narayanan et al. [18], which analyzed several enterprise workload traces to evaluate the economic feasibility of replacing disks with flash storage. We focus on a larger cloud storage environment, develop an algorithm for making good allocation choices between different workloads, and reach significantly different conclusions about the effectiveness and economics of using flash in this manner.

3 System Description

Janus provides flash storage allocation recommendations for workloads in a distributed file system, such as Colossus, in a large private cloud data center. The underlying storage is a mix of disk and flash storage on distinct *chunkservers*, structured as separate tiers. Upon creation, files may be placed in the flash tier, and later moved to disk using a FIFO or LRU policy. We use this *insertion on write* mechanism rather than the *insertion on read* used in most caches because it is more suitable for our system. The distributed nature of file systems like GFS and Colossus makes insertion on read policies more expensive than insertion on write for some metrics we intend to optimize, in particular the volume of read activity. Because data access occurs directly between chunkserver nodes and clients, and not every chunkserver node contains flash capacity, an insertion on read policy that does not rely on the client for write back must perform an additional read in order to populate the data into flash storage. Additionally, the write back into flash storage can not be assumed to be instantaneous as the operation requires reading data from disk, transferring across a local network link, and finally a write into the flash media.

Many users and applications may use this system, either directly, or through higher level storage components such as Bigtable [8]. The flash tier can be partitioned between workloads. The main scenario we consider is where the system operator has a fixed amount of total flash available in the system, and wants to maximize the fraction of reads offloaded to flash storage; however, in some cases, it may be preferable to offload high-priority workloads.

Workloads can correspond to users, applications, or specified groups of files. For example, an application may have separate logging, data, and metadata components, and these could be different groups. In structured data, a table or a set of columns may be a group. Exactly how the workloads are formed is outside the scope of this paper; we just assume that these groupings exist, perhaps manually created. We may associate a *priority weight* with each workload; the higher the priority weight, the more important it is to accommodate its reads from flash storage. The precise mechanism for choosing workload weights is again outside the scope of this paper, but for example, the administrator could assign different weights to different workload types. Our goal is to determine automatically how to divide the available flash between the workloads to optimize the reads from flash.

The allocation recommendations are made by an offline optimization solver that runs periodically to adjust to changes in the workload behaviors and the available flash storage. A key input to the solver is a compact representation of both the age of the data stored in each

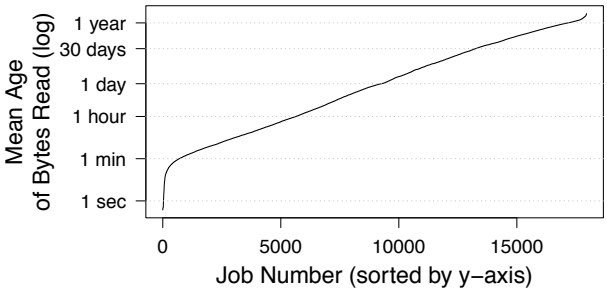


Figure 1: Mean age of bytes read differs significantly by user.

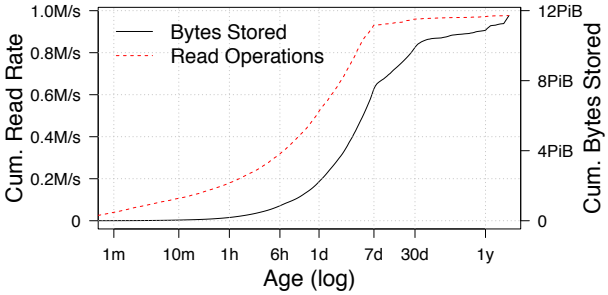


Figure 2: Cumulative distribution function of the bytes stored, and read operations sorted by the (FIFO) age of the data for a particular workload. These CDFs are a graphical representation of the histograms collected as inputs to the cacheability curves, which are different for each user and used in the optimization formulation. 50% of the data stored by this particular user is less than 1 week old, but that corresponds to over 90% of the read activity.

workload group, and the read rate of the data by age. These are obtained by scanning the file system metadata and sampled traces of I/O activity.

The operation of Janus can be broken into three steps:

- Collection of input data about the age of bytes stored and age of data accessed for different workloads to generate a characterization of how cacheable each workload is (§ 4).
- Solving an optimization problem to allocate flash amongst the workloads (§ 6).
- Coordination with the distributed file system to place data from different workloads on flash using the computed write probabilities and flash sizes from the solver (§ 8).

4 Workload Characterization

Storage in our data centers is shared between thousands of users and applications. Applications include content indexing, advertisement serving, Gmail, video processing, as well as smaller applications, such as MapReduce jobs owned by individual users. A large application may have many component jobs. The workload characteristics and demands of jobs in data centers are typically highly varied between users and jobs. Figure 1 shows the variation of mean read age over different jobs in our data centers. All read ages are well represented: there are jobs accessing very young (1 minute old) to very old (1 year old) data. However, different jobs also have very different read hotness, as shown in Figure 4, so we cannot conclude that the aggregate reads are evenly distributed over data of different ages. Instead, we need to define a metric that lets us compare how many read operations would be served by flash storage for a given flash allocation to that workload.

4.1 Cacheability Functions

The cacheability function (which we define more formally below) tells us the rate of read hits we are likely to get for a workload if we allocate it a given amount of flash. To compute this for FIFO eviction, we need two inputs for each workload: how much data there is of a given age, and how many reads there are to files of a given age. For LRU eviction, the corresponding two inputs are the amount of data with a given temporal locality and the rate of reads to files with that temporal locality.

We define two age metrics: FIFO age and LRU age, which are used with the corresponding eviction policies (although we will just say "age" where the disambiguation is not needed). The FIFO age of a file (and of all the data in the file) is the time since the file was created. The LRU age of a file, which is a measure of the temporal locality of its reads, is approximately the maximum time gap between reads to the file since it was created (see Section 8.7 for a precise definition).

Obtaining the distribution of FIFO age is straightforward: we scan the file system metadata, which includes the create time of each file, to build a histogram of the FIFO age of bytes stored for each group. To build a histogram of the read rates of data by FIFO age, we need to look at the read accesses, which we obtain from traces. Since the read rate in the data centers is enormous, it is not practical to consider every read to the data in each workload. Instead, we sample the reads from every job using Dapper [22, 9], an always-on system for distributed tracing and performance analysis. Dapper samples a fraction of all RPC traffic and, by looking at the age of the requested data at the time of each RPC, we can populate a second histogram of the number of read operations binned by age of the data read. Crucially, each of these two histograms has the same bucket boundaries for data age, which later lets us join the histograms.

Computing the corresponding histograms for LRU age is similar, except that computing the LRU age requires the time-gaps between read operations to a file. Dapper traces do not suffice in this case, since not every I/O to

Figure 3: The number of read operations for a given amount of the youngest data (by FIFO age) for a particular user.

a file is captured. We built instead a distributed tracing mechanism that samples based on the file identifier and captures every I/O for the files so selected.

The two input histograms of data age and read age for a specific workload can then be combined to construct a cacheability function.

Definition (Cacheability Function): For a workload the cacheability function ϕ maps the amount of flash allocated to the workload to the rate of reads absorbed by the flash storage. In particular, $\phi(x)$ gives the number of read operations that go to the youngest x bytes of data.

Figure 3 shows an example of a cacheability function computed by joining the histograms of read rate and data size by age in Figure 2. Joining the histograms is simple because the age bins are the same. From the histograms for a specified workload we derive the cumulative function f giving the amount of data younger than a certain age, and the cumulative function g giving the read operations to files that are younger. Essentially, for each flash allocation x, we can look up the age $f^{-1}(x)$ of the files that can be stored (assuming the youngest are stored) and then look up the rate of read hits for files of that age or younger:

$$\phi(x) = (g \circ f^{-1})(x) = g(f^{-1}(x))$$

We compute the cacheability function by linear interpolation between the bins, and hence the function is piecewise linear, a fact we later use in the optimization. Assuming that these distributions are stationary, the composition gives us the read hit rate for the flash allocation. Also, because of the way we separately defined file age for FIFO and LRU eviction, this method works in both cases.

5 Economics and Provisioning

Narayanan et al. [18] argued that replacing disk with flash storage was not cost-effective. Prices for flash have fallen considerably since then, but has this conclusion

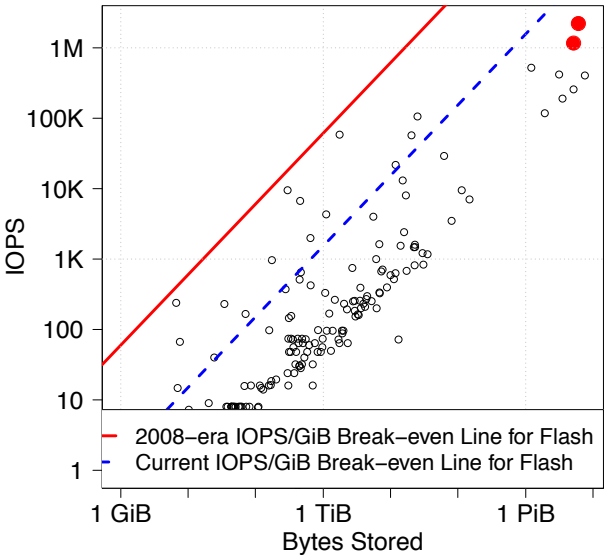


Figure 4: Peak IOPS and capacity requirements for user workloads in a shared data center. IOPS is the 95th percentile over 10 minute intervals. Workloads above the break-even line would be cost-effective to store entirely on flash. The two filled red dots are for the workloads in Table 1, and come from other data centers. The lower dot is for the workload in Figure 3.

changed? To analyze this, following Narayanan, we find the *break-even point*, which is the IOPS/GiB threshold determining whether a workload would be cheaper on flash storage or on disk. This threshold can be derived from the IOPS/$ of disk, I_d, and the GiB/$ of flash, G_f, since a workload with I IOPS and G GiB will cost G/G_f on flash and at least I/I_d on disk (more for cold data). Therefore, workloads with IOPS/GiB greater than I_d/G_f are better served from flash, and by using a disk with high I_d for this cutoff, we are being conservative in recommending workloads to go entirely on flash.

For our example IOPS/$ efficient retail drive, we use the Seagate Savvio 10K.3, which costs around $100. The disk specifications [1] indicate an IOPS capacity around 150 ((seek time+avg latency)$^{-1}$), or 1.5 IOPS per dollar for disk. On the other hand, recent news reports [10] indicate that we can get about 1 GiB of flash per dollar; together these give a break even point of 1.5 IOPS per GiB, which is much smaller than the 2008 value ≈ 60. As displayed in Figure 4, we find that, at least for some workloads, it is cost-effective to place all data in flash. Even for other workloads close to the break-even point, using flash may be justified by the resulting improvement in latency.

In addition, many workloads could benefit from putting their youngest data on flash using Janus. We now consider how much flash is cost-effective for an individual workload. For a workload with read operations rate rate$_r$, write operations rate rate$_w$, capacity size c, and

Workload	1	2
Data size (PiB)	5.2	6.1
Access rate (k ops/sec)	1172	2214
Janus Savings (%)	29	12
Janus Flash (%)	0.42	2.1

Table 1: Storage demands and savings from a price optimization using Janus, which correspond to solid red dots in Figure 4. The savings is over the best all-disk or all-flash solution. The flash % is the percentage of the user data in flash.

cacheability function $\phi()$, a disk (IOPS, GiB_{disk}) demand of (rate_r + rate_w, d), could be replaced with a disk + flash (IOPS_{disk}, GiB_{disk}, GiB_{flash}) demand of

$$(\text{rate}_r + \text{rate}_w - \phi(x),\ d - x,\ x)$$

To determine the amount of flash, x, that a user should purchase, and their benefit from using the system, we impose a pricing structure, then have each user purchase flash to minimize costs. We avoid pricing complications arising from the balance of cold and hot data in a shared storage system, and put ourselves in the IOPS constrained framework where we sell disk based entirely on IOPS, so that cost is determined by

$$\text{cost}(x) = \frac{(\text{rate}_r + \text{rate}_w - \phi(x))}{I_d} + \frac{x}{G_f} \qquad (1)$$

and we note that the optimization of cost is simplified by the fact that ϕ is piecewise linear between histogram buckets.

In Table 1 we consider this optimization for some workloads and display the price savings, along with the percentage of data that goes on flash, in the optimal configuration. We note that while workload 2 is hotter on average, workload 1 gets a greater benefit from a smaller amount of flash because of its steep cacheability curve (Figure 3).

6 Optimizing the Flash Allocation for Workloads

We now describe how we determine the best flash allocation for each workload, given the cacheability functions derived in Section 4. Specifically, we seek to maximize the aggregate rate of read operations served from flash subject to a bound on the total available flash. The workloads may have different priority weights, in which case we maximize the aggregate weighted rate of reads from flash.

We assume that the cacheability functions are piecewise linear and concave. As mentioned previously, the piecewise linear assumption always holds since we compute the function by linearly interpolating between a finite number of points (corresponding to the bins of the histogram from which we derive it). The concavity assumption is equivalent to assuming that the read rates for each workload's data decrease monotonically with increasing data age. This assumption holds usually, but not always. We will show in the next section how to relax the assumption where it matters.

Weighted Max Reads Flash Allocation Problem
Instance:

- A set of workloads; for each workload i is given the total data d_i, the cacheability function as a piecewise linear function $\phi_i : [0, d_i] \to \mathbb{R}$, and a priority weight ρ_i.
- A bound on the total flash capacity F.

Task:

Find for each workload i the allocated flash capacity x_i, $0 \le x_i \le d_i$, maximizing the total priority weighted flash read rate $\sum_i \rho_i \phi_i(x_i)$, and subject to the constraint of the total flash capacity $\sum_i x_i \le F$.

Let the segments of the piecewise linear function $\rho_i \phi_i$ be $a_{i,j} + b_{i,j} x$ for $j = 1, \ldots, n_i$. Since ϕ_i is concave, $\rho_i \phi_i$ can be expressed as the minimum of the functions corresponding to its linear segments:

$$\rho_i \phi_i(x) = \min_{1 \le j \le n_i} \{a_{i,j} + b_{i,j} x\}$$

By replacing $\rho_i \phi_i(x)$ with the variable y_i, we transform the task into a linear program:

$$\begin{aligned} \max \quad & \sum_i y_i \\ \text{s.t.} \quad & y_i \le a_{i,j} + b_{i,j} x_i \quad \text{for each workload } i \\ & \qquad\qquad \text{and each segment } j \\ & \sum_i x_i \ \le\ F \\ & 0 \ \le\ x_i \ \le\ d_i \quad \text{for each workload } i \end{aligned} \qquad (2)$$

This optimization problem can be solved with an LP solver. We solve it directly as explained at the end of the next section.

7 Optimization with Bounded Write Rates

Limiting flash write rate is important to avoid flash wear out and also reduces the impact of flash writes (which are slow) on read latencies. We now describe how to allocate flash so as to maximize the reads from flash while limiting the write rate to flash. We also show how to approximately relax the concavity assumption on the cacheability function. The cacheability function for a workload may be non-concave if the read rate increases some time after it is created, for example, if there is a workload that begins processing logs with some delay after they are created.

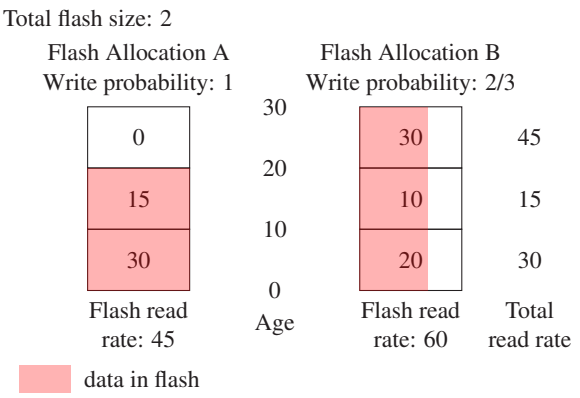


Figure 5: Example for non-concave cacheability and fractional write probability: Data blocks and read rates of a workload for different age ranges are shown at steady state. The workload has one block of data with age between 0 and 10 and a read rate of 30, a second block of data with age between 10 and 20 and a read rate of 15, and a third block of data with age between 20 and 30 and a read rate of 45. Storing all data younger than age 20 in flash (highlighted) gives a hit rate of 45 (left). With a write probability of $2/3$, we place less new data in the same flash but keep it longer, until age 30 (right). This captures the higher read rate for data between ages 20 and 30, for a total flash hit rate of 90 * 2/3 = 60. The write rate to flash also decreases by $1/3$.

We only consider insertion into flash upon write, so that the write rates per workload are independent of the flash allocation. For simplicity, we also ignore priority weights here, but this extension is straightforward.

The flash write rate can be controlled either by limiting the workloads that have data in flash or by writing only a fraction of each workload's new data into flash. We implement the latter by setting a write probability, and for each new file, deciding randomly with that probability whether to insert it into flash. Figure 5 shows an example with one workload where decreasing the write probability decreases the flash write rate *and* increases the flash read hits. This is only possible if the cacheability function is non-concave.

In general, if the workload i has a flash capacity x_i and a write probability p_i the data can stay in flash for as long as if the workload has a flash capacity of $\frac{x_i}{p_i}$ but all new data is written to flash. Hence, the flash read rate for the workload i with cacheability function ϕ_i is $p_i\,\phi_i(\frac{x_i}{p_i})$.

Bounded Writes Flash Allocation Problem

Instance:

- A set of workloads; for each workload i is given the total data d_i, a continuous piecewise linear cacheability function $\phi_i : [0, d_i] \to \mathbb{R}$, and a write rate w_i.
- A bound on the total flash write rate W.
- A bound on the total flash capacity F.

Task:

Find, for each workload i, the allocated flash capacity x_i, $0 \le x_i \le d_i$ and the flash write probability p_i, $0 \le p_i \le 1$, maximizing the total flash read rate $\sum_i p_i\,\phi_i(\frac{x_i}{p_i})$ and subject to the constraint of the total flash write rate and total flash capacity. Formally:

$$\begin{aligned} \max \quad & \sum_i p_i\,\phi_i\left(\frac{x_i}{p_i}\right) \\ \text{s.t.} \quad & \sum_i p_i\,w_i \le W \\ & \sum_i x_i \le F \\ & 0 \le x_i \le d_i \quad \text{for each workload } i \\ & 0 \le p_i \le 1 \quad \text{for each workload } i \end{aligned} \tag{3}$$

While the problem has linear constraints, the objective is not linear. Our approach is to (a) relax the constraint on the write rate via Lagrangian relaxation; (b) remove the dependence on the write probability p_i in the objective function; (c) linearize the objective; and (d) solve the resulting linear program with a greedy algorithm.

We relax (remove) the write rate bound $\sum_i p_i w_i \le W$ and change the objective function by subtracting the write rate with a write penalty factor $\lambda \ge 0$:

$$\sum_i p_i\,\phi_i\left(\frac{x_i}{p_i}\right) - \lambda\,p_i\,w_i \tag{4}$$

An optimal solution for the relaxed problem with a total write rate equal to the bound (i.e., $\sum_i p_i w_i = W$) is an optimal solution of the original problem (3). Proof: If there is a better solution for the original problem (3), its read rate is higher but its write rate cannot be larger. Hence, this solution is also a better solution for the relaxed problem, which contradicts the optimality.

Since the total write rate found by the relaxed optimization decreases monotonically with increasing λ, we can find the best λ, where the total write rate closely matches the bound, using binary search.

Since the constraints on the write probabilities p_i are independent of the other variables, we can remove the dependence on the write probabilities as follows. Let $h_i^\lambda(x)$ represent the contribution of workload i with allocated flash size x to the objective (4) when maximized. Then:

$$\begin{aligned} h_i^\lambda(x) &= \max_{0\le p\le 1} p\,\phi_i\left(\frac{x}{p}\right) - \lambda\,p\,w_i \\ &= \max_{z\ge x} \frac{x}{z}\,(\phi_i(z) - \lambda\,w_i) \\ &= x \max_{z\ge x} \frac{\phi_i(z) - \lambda\,w_i}{z} \end{aligned}$$

Since the function ϕ_i is continuous and piecewise linear, $(\phi_i(z) - \lambda\,w_i)/z$ is monotonic with z in each segment. Hence the above maximum can be found by evaluating it only at the breakpoints of ϕ_i. By processing the breakpoints of ϕ_i in decreasing x-coordinate the function $h_i^\lambda(x)$ can be computed in linear time.

Next, we linearize the resulting objective $\sum_i h_i^\lambda(x_i)$. h_i^λ is concave if ϕ_i is, which is usually the case because read rates decline with age. If not, we replace it with its concave upper bound by removing some breakpoints of the piecewise linear function. We argue later that this has only a small impact on the optimality of the result. As in the previous section, we rewrite $h_i^\lambda(x)$ as the minimum of the linear functions corresponding to its segments and get a linear program that has the same form as (2).

Finally, we solve this linear program with a greedy algorithm. We start with the solution $x_i = 0$, $y_i = 0$ for each workload i and then successively increase the allocated flash x_i of the workload that has the highest ratio of increase in the objective function to flash allocation, as long as flash can be allocated. Except for the last incremental allocation, the flash allocation to each workload corresponds to a breakpoint of its cacheability function. The algorithm has a runtime complexity of $O(n\,k\,\log k)$ where n is the maximum number of pieces of the piecewise linear functions ϕ_i and k is the number of workloads.

The result is optimal if h_i^λ is concave. If not, we can show that the error in the objective value due to the concave approximation is bounded by the objective increment of the last step. Hence, we are close to optimality if the last incremental flash allocation is small. This is certainly the case if the workload is partitioned into many small workloads, which is, in any case, preferable for optimal allocation.

8 Evaluation

In this section, we evaluate the effectiveness of the algorithms described in the previous sections on production storage workloads in Google data centers. Section 8.1 describes the production environment, and Section 8.2 introduces the datasets and terminology used for the evaluation. The remainder of the section evaluates the recommendations produced by Janus both on production workload deployments that used the recommendations and on traces of other production workloads.

8.1 File Placement in Colossus

Colossus (the successor of GFS [11]) is a distributed storage system with multiple master nodes and many chunkservers that store the file data. File system clients create new files by a request sent to a master node, which allocates space for it on chunkservers it selects. We evaluated Janus in a Colossus system extended as follows.

When a file is created, a Colossus master node decides, based on the amount of flash space available for the corresponding workload and the write probability assigned to it, whether it should be placed on disk or on flash, and accordingly allocates space. Eviction from flash is designed to take advantage of the already existing file maintenance scanner process. The file is tagged with an eviction time (TTL), which is computed from the flash allocated to that workload and the workload's write rate. The scanner process periodically checks whether the file has exceeded its eviction time, and if so, moves it to disk. The eviction time (TTL) in its current implementation is not updated after it is set, effectively producing an *approximate FIFO* eviction policy. An arriving file creation request sometimes finds the flash storage full; in this case, the master will write it to disk, regardless of whether it would otherwise have chosen to write it to flash.

8.2 Datasets and definitions

In the remainder of Section 8, we evaluate Janus based on several datasets.

A Colossus *cell* is a separate instance of the Colossus system. Separate cells are typically located in different facilities. Each cell has its own masters, chunkservers, and files, and each cell independently manages user quota.

The first three datasets come from multi-user cells, with workloads corresponding to different users of the cell.

Dapper A 37-day Dapper sample of read-write activity over 10 cells. The first 30 days are used for training (computing the cacheability functions), and the last 7 days are used for evaluation.

Janus Deployment Data from limited deployments of production workloads using Janus recommendations in 4 cells. In these deployments, flash was assigned only to a single workload. The training period consisted of a 30 day of Dapper sample prior to the deployment.

Multi-User Cell A 1-week trace of all read-write activity to a 1% sample of files in a single cell. The 6th day is used for training, and the 7th day is used for evaluation. The first 5 days are used in Section 8.7 to determine whether a file is cached by LRU. This cell had 407 workloads.

The last dataset comes from a cell where all activity comes through Bigtable. Files are separated into workloads based on tokens that Bigtable encodes in the file name for different tables and columns.

Single-User Cell A full trace of read-write activity in a single-user Colossus cell for 30 minutes. The first 15 minutes are used for training, and the second 15 minutes are used for evaluation. The cell had 541

workloads, and contained over 10,000 machines. A configuration change was needed to collect the data, leading to the short duration of the trace, but it contains adequate data because of the size of the cell and the fact that the trace is not sampled.

The *read rate* for each workload is the number of read operations per second, excluding in-memory cache hits. The *flash read rate* is the number of read operations per second that is served from flash. The *flash hit rate* is the flash read rate as a percent of the read rate. In some cases, we report the *normalized flash hit rate*, which is the flash read rate for a workload as a percent of the total read rate among all workloads in the cell. In particular, the cell-wide flash hit rate is the sum over all workloads of the normalized flash hit rate.

The *size* of a workload is the logical number of bytes stored, excluding overhead from replication or erasure coding. Analogous to the terminology for read rates, we also have *flash size*, *flash size percentage* and *normalized flash size percentage*.

The *write rate* of a workload is the number of bytes per second of new data written. Again, this excludes overhead from replication or erasure coding. Again, we also have *flash write rate* and *flash write percentage*.

Given a cell-wide flash size, or equivalently a flash size percentage, we form an *allocation* of flash to different workloads using the optimization. The allocation consists of a Flash Size and a Write Probability for each workload in the cell. This allocation is used in the evaluation period to compute various metrics of interest, such as flash hit rates.

8.3 Does the Past Predict the Future?

Our optimization is based on sampled historical data. Here, we investigate the stability of estimated per-workload flash hit rates between training and evaluation periods in the Dapper dataset. In Section 8.4, we will consider how well estimated flash hit rates correspond with values from an actual deployment.

We chose 10 cell-wide flash size percentages between 0.1% and 10%. For each flash size percentage and each cell, we optimized the allocation of flash to workloads. Figure 6 uses these allocations to plot per-workload flash hit rates in the evaluation period against those in the training period. The figure shows that flash hit rate during evaluation is typically within 7% of the flash hit rate during training. This range of variability is small enough for the resulting system to be effective.

8.4 Janus Deployment

Due to the Colossus's use of approximate FIFO (described in Section 8.1), we must compute eviction TTLs

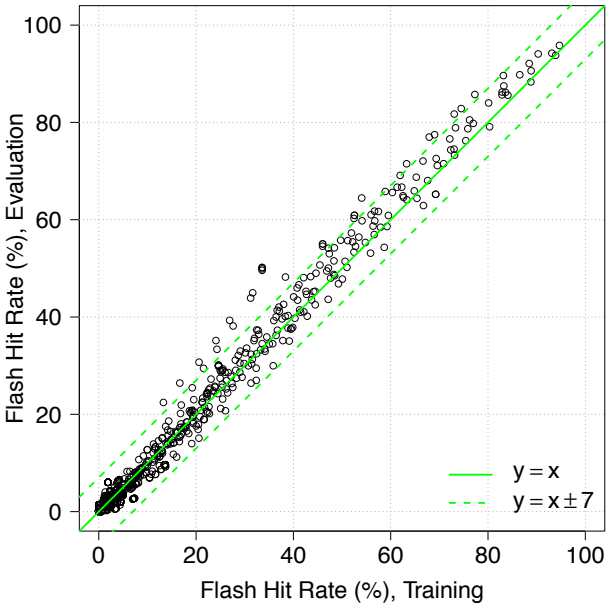


Figure 6: Flash hit rate during training and evaluation periods, estimated from the Dapper dataset. Each point represents a single workload in a single cell with a given cell-wide flash size percentage.

from each workload's allocation. Janus computes the TTLs using file age histograms from the training period. However, the file age distribution may change between training and deployment. For example, a workload may start writing new data at a high rate, or it may exhibit peak-to-trough variability not captured in histograms averaged over the 30-day training period. In these cases, using fixed TTLs may cause the workload to exceed its allocated flash size, and Colossus will write new files to disk until flash usage decreases. Hence, a workload's actual flash usage can fluctuate over time.

Figure 7 shows flash usage for a single workload over two days in one cell. The workload's allocated flash size was 100 TiB. Each day, actual flash usage fluctuates from 45 TiB to 100 TiB due to peak-trough variations. We accommodated this variation by decreasing the allocated flash size so as not to exceed the actual allocation.

Figure 7 also shows the workload's flash read rate during the period. On average, we get around 30k flash read ops/sec, with a peak of more than 40k flash read ops/sec. From the 30 day training period, we predicted a flash read rate of 33k flash read ops/sec.

Table 2 shows results for this workload over deployments in four different cells. The average of estimated and measured flash hit rates over those cells were 22.8% and 23.5% respectively, a 3% difference. For cell A, the measured flash hit rate (27%) was significantly higher than the estimated value (17%), partly because we manu-

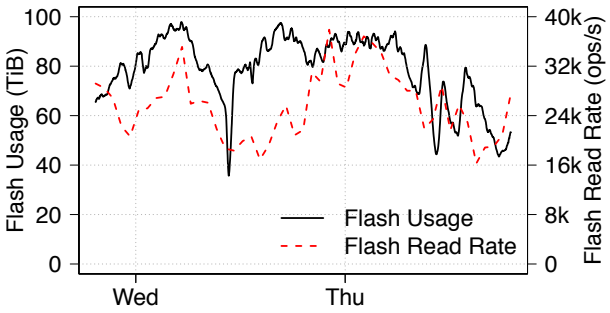


Figure 7: Flash usage and Flash read rate for one workload over two days after Janus deployment.

Cell	workload size	allocated flash size	average flash usage	estimated flash hit rate	measured flash hit rate
A	3.26 PiB	80 TiB	62 TiB	17%	27%
B	3.34 PiB	100 TiB	72 TiB	29%	31%
C	3.47 PiB	60 TiB	50 TiB	19%	16%
D	3.26 PiB	100 TiB	63 TiB	26%	20%
Avg	3.33 PiB	85 TiB	62 TiB	22.8%	23.5%

Table 2: Janus performance with one workload in four cells.

ally adjusted the parameters to maximize the space usage and allow the group to hit the quota.

8.5 Comparing Alternative Allocation Methods

In this section, we compare alternative methods for generating flash allocations. *Optimized FIFO* allocation uses the methods described above. We can also set per-workload flash size *proportional to read rate* or *proportional to size*. Both read rate and size are the average usages measured over the training period. Lastly, we can assign flash size such that the eviction TTL is the same for all workloads. This is effectively a *single FIFO* for all workloads. These and all subsequent comparisons are made using trace-based analysis rather than direct measurement, since Janus was only deployed with optimized FIFO eviction (denoted *Opt FIFO* in the tables).

Table 3 shows cell-wide flash hit rates for the single and multi-user cells. In the multi-user cell, the flash hit rate improves from 19% to 28% when changing from single FIFO to optimized FIFO, representing a 47% improvement. In the single-user cell, the relative improvement was even larger — from a 42% hit rate to 74%, a 76% relative improvement.

Especially in the single-user cell, optimized allocation outperforms the other methods. Table 4 shows that the poor performance of non-optimized methods in the single-user cell is due to allocating large amounts of flash to workload 117. This workload comprises 10% of the cell's read rate, but 43% of the cell's size. Optimized

Dataset	Multi-User	Single-User	Single-User
Flash Size (%)	1.0%	5.3%	5.3%
Additional Constraints			No flash for workload 117
Opt FIFO	28%	74%	74%
Prop. Read Rate	26%	64%	64%
Single FIFO	19%	42%	45%
Prop. Size	14%	15%	21%

Table 3: Flash hit rates achieved by 4 different allocation methods for the single and multi-user cells. The cell-wide flash size percentages were 5.3% for the single-user cell and 1.0% for the multi-user cell.

allocation assigns no flash to this workload, since other workloads provide a better read rate to size ratio.

The last column of Table 3 shows that the flash hit rate under Single FIFO and Proportional to Size improves if we constrain workload 117 to receive no flash. However, Proportional to Read Rate does not improve, as removing workload 117 exposes the next few workloads that have a high read rate to older data.

The improvement between single FIFO and optimized FIFO in the multi-user cell can also be attributed to a single workload. This is discussed further in Section 8.7.

Normalized Flash Hit Rate (%)

Workload	Opt FIFO	Prop Reads	FIFO	Prop Size
1	11.8	10.8	8.1	3.5
2	7.9	7.9	4.2	1.3
3	7.7	5.5	6.0	2.2
4	7.4	7.4	3.8	1.1
5	5.7	5.7	2.2	0.5
9	4.0	4.0	4.0	0.3
10	3.9	4.2	4.2	0.3
117	**0.0**	**0.6**	**0.9**	**1.0**
Others	25.4	17.6	8.5	4.6
Total	73.7	63.8	41.9	14.7

Normalized Flash Size Percentage (%)

Workload	Opt FIFO	Prop Reads	FIFO	Prop Size
1	0.82	0.59	0.22	0.04
2	0.08	0.41	0.02	0.00
3	1.19	0.62	0.71	0.17
4	0.09	0.38	0.02	0.00
5	0.04	0.31	0.01	0.00
9	0.01	0.20	0.01	0.00
10	0.00	0.21	0.01	0.00
117	**0.00**	**0.25**	**0.99**	**2.26**
Others	3.02	2.27	3.26	2.77
Total	5.25	5.25	5.25	5.25

Table 4: Flash hit rates and size for selected workloads in the single-user cell. Workloads are numbered in decreasing order of flash read rate under optimized FIFO allocation.

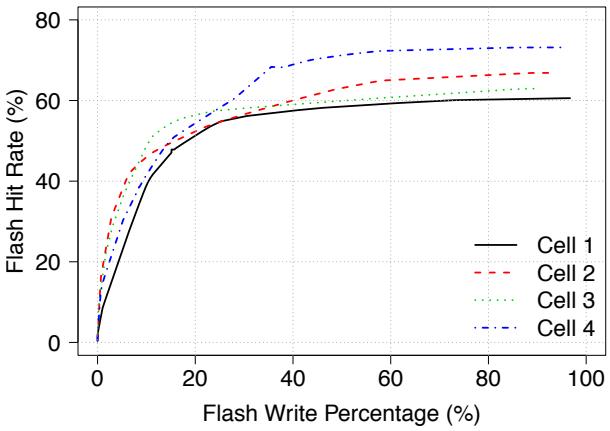


Figure 8: Flash hit rate for given bounds on the flash write percentage for four cells in the Dapper dataset.

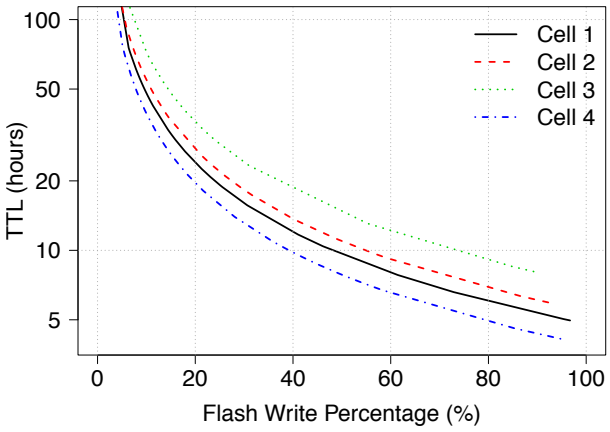


Figure 9: Average TTL of data written to flash for given bounds on the flash write percentage for four cells in the Dapper dataset.

8.6 Impact of Bounded Flash Write Percentage

In Section 7 we showed how the flash write percentage can be bounded at the cost of a lower flash hit rate.

Figure 8 shows the optimized flash hit rate for various bounds on the flash write percentage. The rightmost point on each curve corresponds to unbounded flash write percentage. In each cell, the optimized unbounded value is above 90%. As we decrease the bound, the flash hit rate decreases slowly at first, and it decreases quickly once the bound falls below 60%.

Figure 9 shows the average TTL of the new data written to flash for the same cells and the same flash allocation solutions. As the bound on the write rate is tightened, less data is written to flash but it stays there longer.

8.7 Evaluation of LRU Eviction

We have so far seen the performance of Janus with FIFO eviction. We now turn to an evaluation with LRU eviction.

LRU Cacheability Functions and Censoring

In Section 4, we briefly described cacheability functions for LRU eviction. We make this description more formal here.

A file will be in the cache if the maximum gap between reads is lower than the TTL. We re-define the notion of *age* to reflect this heuristic. The LRU age of a file at time t is

$$\text{Age}(t) = \max\left(t_1 - t_0, ..., t_n - t_{n-1}, t - t_n\right)$$

where t_0 is file creation time, and $t_1, ..., t_n$ are the times of the n reads in interval $[t_0, t)$. The smaller the LRU age of a file is, the more temporal locality its reads have. The cacheability function, $\phi(x)$, gives the flash read rate if the x bytes with the lowest LRU age are placed on flash.

To compute the age of a file at t, we require a full trace of read operations during $[t_0, t)$. In many cases, the full trace is not available. Suppose the trace is available only during $[t_S, t)$, with $t_S > t_0$. The resulting read age measurement is censored.

We deal with censoring by considering the two extremes. *Upper bound age* assumes that there were no reads between t_0 and t_S. *Lower bound age* assumes that there was continuous read activity between t_0 and t_S, so that $\text{Age}(t_S) = 0$. The upper bound of the cacheability function is obtained by using upper bound age for size and lower bound age for read rates, and vice versa for the lower bound of the cacheability function.

Evaluation of LRU using Multi-User Cell Dataset

Figure 10 shows the cacheability function for a single FIFO / LRU. With 1% flash size percentage, the flash hit rates are 19% for a single FIFO and 36%–40% for a single LRU. The marked points allocate flash to workloads using optimized FIFO / LRU. The flash hit rates are 28% for optimized FIFO and 44%–48% for optimized LRU.

Table 5 shows normalized flash hit rates for various workloads. Most of the improvement between FIFO and LRU can be attributed to the cacheability of workload 1, which is a Bigtable service shared by many users. LRU assigns 3.4x–3.5x as much flash to workload 1, and obtains 4.5x–5.1x as high a flash read rate as FIFO. This accounts for a 14.4–16.7 percentage-point increase in the cell-wide flash hit rate. Even optimized FIFO does not achieve this high a flash read rate for workload 1, because the workload's cacheability function is much steeper for LRU than for FIFO.

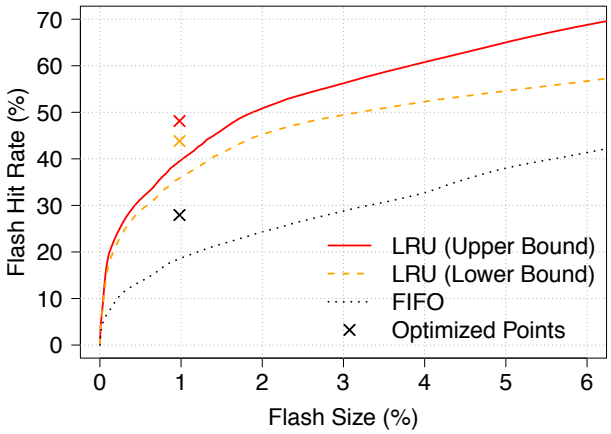


Figure 10: Cacheability curve for a single FIFO and single LRU. The marked points represent optimized flash hit rates for FIFO and LRU.

Normalized Flash Hit Rate (%)

Workload	FIFO	Opt FIFO	LRU	Opt LRU
1	4.1	5.3	18.5 – 20.8	15.8 – 16.9
2	0.0	7.3	0.0 – 0.0	7.3 – 7.3
3	0.1	0.9	1.3 – 2.1	1.6 – 6.1
4	3.0	1.9	5.6 – 6.1	4.9 – 5.3
5	6.4	5.7	4.6 – 4.7	6.8 – 5.2
Others	4.9	6.7	5.8 – 5.9	7.2 – 7.2
Total	18.5	27.8	35.9 – 39.5	43.6 – 47.9

Normalized Flash Size Percentage (%)

Workload	FIFO	Opt FIFO	LRU	Opt LRU
1	0.13	0.22	0.46 – 0.44	0.20 – 0.15
2	0.00	0.08	0.00 – 0.00	0.08 – 0.08
3	0.00	0.08	0.03 – 0.03	0.04 – 0.17
4	0.18	0.03	0.11 – 0.11	0.07 – 0.08
5	0.42	0.35	0.20 – 0.21	0.38 – 0.28
Others	0.25	0.22	0.18 – 0.19	0.22 – 0.21
Total	0.98	0.98	0.98 – 0.98	0.98 – 0.98

Table 5: Flash hit rate and size per workload assuming single and optimized FIFO/LRU. Workloads are ordered in decreasing order of flash read rate under Optimized LRU. The two numbers for LRU respectively use the lower and upper bounds of the cacheability function. Assumes cell-wide flash size percentage of 1% during the training period, which became 0.98% during evaluation since the amount of data increased slightly.

Table 5 also shows that workload 2 accounts for most of the difference between optimized and single versions of FIFO/LRU. In fact, both optimized FIFO and LRU put the entire contents of workload 2 on flash, increasing the cell-wide flash read hit by 7.3%. Workload 2 is a Bigtable used to serve static webpage content.

These results are robust to adjusting the period used for training. Of our 7-day dataset, we used the 7th day for evaluation and the 6th day for training; the remaining days were used only to compute the file ages. If the 5th day is used instead for training, then the cell-wide flash hit rate is 28.5% under optimized FIFO, and 44.4–49.1% under optimized LRU. Using the 4th day, we get 27.0% under optimized FIFO and 42.7–48.1% under optimized LRU. These numbers are similar to those in Table 5.

While LRU eviction performs better than FIFO for many workloads, there is a substantial associated overhead. The LRU age of a file depends on accesses to all its component chunks, and hence the eviction scanner must gather information from multiple chunkservers before determining whether a file should be evicted. By comparison, computing the FIFO age is simple because it depends only on the static creation time of the file.

9 Conclusions

The falling price of flash storage has made it cost-effective for some workloads to fit entirely in flash. As the I/O rate per byte supported by disks continues to decline, flash storage also becomes a critical component of the storage mix for many more workloads in modern storage systems. However, because flash is still expensive, it is best to use it only for workloads that can make good use of it. With Janus, we show how to use long-term workload characterization to determine how much flash storage should be allocated to each workload in a cloud-scale distributed file system.

Janus builds a compact representation of the cacheability of different user I/O workloads based on sampled RPC traces of I/O activity. These *cacheability curves* for different users are used to construct a linear optimization problem to determine the flash allocations that maximize the read hits from flash, subject to operator-set priorities and write-rate bounds.

This system has been in use at Google for 6 months. It allows users to make informed flash provisioning decisions by providing them a customized dashboard showing how many reads would be served from flash for a given flash allocation. Another view helps system administrators make allocation decisions based on a fixed amount of flash available in order to maximize the reads offloaded from disk.

Based on evaluations from workloads using these recommendations and I/O traces of other workloads, we conclude that the recommendation system is quite effective. In our trace-based estimates, flash hit rates using the optimized recommendations are 47-76% higher than the option of using the flash as an unpartitioned tier. We find that application owners appreciate learning how much flash is cost-effective for their workload.

Acknowledgments

Janus would not have been possible without the help of many individuals and teams. We are especially grateful to Jun Luo, Adam Gee, Denis Serenyi, and the entire Colossus team for their early collaboration on this project. Andy Chu, Herb Derby, Lawrence Greenfield, Sean Quinlan, Paul Cychosz, Salim Virji, and Gang Ren also contributed ideas or helped with the data collection and analysis on which Janus is built. We are grateful to John Wilkes, Florentina Popovici, our shepherd Kai Shen, and our anonymous referees for their feedback on improving the presentation.

References

[1] Retrieved 2013/01/09: `http://www.google.com/shopping/product/7417866799343902880/specs`.

[2] AGUILERA, M. K., ET AL. Improving recoverability in multi-tier storage systems. In *DSN* (2007), IEEE, pp. 677–686.

[3] ALVAREZ, G. A., ET AL. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (2001), 483–518.

[4] ANDERSON, E., ET AL. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst. 23*, 4 (2005), 337–374.

[5] BAKER, M., ET AL. Measurements of a distributed file system. In *SOSP* (1991), ACM, pp. 198–212.

[6] BLAZE, M. A. *Caching in large-scale distributed file systems*. PhD thesis, Princeton University, 1993.

[7] CANAN, D., ET AL. *Using ADSM Hierarchical Storage Management*. IBM Redbooks. 1996.

[8] CHANG, F., ET AL. Bigtable: a distributed storage system for structured data. In *OSDI* (2006), USENIX, pp. 205–218.

[9] COEHLO, N., MERCHANT, A., AND STOKELY, M. Uncertainty in aggregate estimates from sampled distributed traces. In *Workshop on Managing Systems Automatically and Dynamically (MAD 2012)*, USENIX.

[10] GASIOR, G. SSD prices down 38% in 2012, but up in Q4, 2013. Retrieved 2013/01/29: http://techreport.com/review/24216/ssd-prices-down-38-in-2012-but-up-in-q4.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP* (2003), ACM, pp. 29–43.

[12] GRAY, J., AND PUTZOLU, F. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD* (1987), ACM, pp. 395–398.

[13] GUERRA, J., ET AL. Cost effective storage using extent based dynamic tiering. In *FAST* (2011), USENIX, pp. 273–286.

[14] KROEGER, T., AND LONG, D. Design and implementation of a predictive file prefetching algorithm. In *ATC* (2001), USENIX, pp. 105–118.

[15] LOBOZ, C. Z. Cloud resource usage: extreme distributions invalidating traditional capacity planning models. In *Workshop on Scientific Cloud Computing (ScienceCloud 2011)*, ACM, pp. 7–14.

[16] MASSIGLIA, P. Exploiting multi-tier file storage effectively. Retrieved 2013/01/29: `https://snia.org/sites/default/education/tutorials/2009/spring/file/PaulMassiglia_Exploiting_Multi-Tier_File_StorageV05.pdf`, 2009.

[17] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on fast-forward. *Communications of the ACM 53*, 3 (2010), 42–49.

[18] NARAYANAN, D., ET AL. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys* (2009), ACM, pp. 145–158.

[19] OH, Y., ET AL. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage filesystems. In *FAST* (2012), USENIX.

[20] OUSTERHOUT, J. K., ET AL. A trace-driven analysis of the UNIX 4.2 BSD file system. In *SOSP* (1985), ACM, pp. 15–24.

[21] PATTERSON, R. H., ET AL. Informed prefetching and caching. In *SOSP* (1995), ACM, pp. 79–95.

[22] SIGELMAN, B. H., ET AL. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[23] STOKELY, M., ET AL. Projecting disk usage based on historical trends in a cloud environment. In *Workshop on Scientific Cloud Computing (ScienceCloud 2012)*, ACM, pp. 63–70.

[24] WILKES, J., GOLDING, R. A., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (1996), 108–136.

Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store

Christopher Mitchell Yifeng Geng* Jinyang Li
New York University **Tsinghua University*
{cmitchell, ygeng, jinyang}@cs.nyu.edu

Abstract

Recent technological trends indicate that future datacenter networks will incorporate High Performance Computing network features, such as ultra-low latency and CPU bypassing. How can these features be exploited in datacenter-scale systems infrastructure? In this paper, we explore the design of a distributed in-memory key-value store called Pilaf that takes advantage of Remote Direct Memory Access to achieve high performance with low CPU overhead.

In Pilaf, clients directly read from the server's memory via RDMA to perform `gets`, which commonly dominate key-value store workloads. By contrast, `put` operations are serviced by the server to simplify the task of synchronizing memory accesses. To detect inconsistent RDMA reads with concurrent CPU memory modifications, we introduce the notion of *self-verifying* data structures that can detect read-write races without client-server coordination. Our experiments show that Pilaf achieves low latency and high throughput while consuming few CPU resources. Specifically, Pilaf can surpass 1.3 million ops/sec (90% `gets`) using a single CPU core compared with 55K for Memcached and 59K for Redis.

1 Introduction

The network implementations found in High Performance Computing (HPC) clusters have historically differed from those in datacenters in a few key aspects: low latency, low CPU overhead, and high cost. Recent trends in the networking world indicate that these distinctions are beginning to disappear as HPC network prices drop and datacenter network equipment begins to adopt features previously found only in HPC clusters. Products are already being offered that implement kernel or CPU bypassing (two common HPC network features) over 10Gbps Ethernet [29, 23], while the prices for the popular Infiniband HPC interconnect have dropped dramatically and are now competitive with 10Gbps Ethernet hardware. For example, a Mellanox 40Gbps Infiniband adapter costs ~$500, while 10Gbps Ethernet cards range in price from ~$300 to $800. Surprisingly, low-latency Infiniband switches are now less expensive than their 10Gbps Ethernet counterparts. Given these changes, it is important that we understand how to leverage the features of these high-performance networks to build general-purpose applications. In this paper, we focus on how to effectively use Remote Direct Memory Access (RDMA), a common component of high performance networking fabrics.

RDMA operations allow a machine to read (or write) from a pre-registered memory region of another machine without involving the CPU on the remote side. Compared to traditional message passing, RDMA achieves the smallest round-trip latency (~3μs), highest throughput, and lowest (zero) CPU overhead. These advantages are offset by the difficulty of incorporating RDMA into distributed system designs. In a traditional design, the server processes all service requests from clients and thus acts as a single point of coordination for memory accesses. With RDMA, clients can directly access the server's memory to implement a service request without any involvement by the server. However, without the server's coordination, races in memory accesses by different machines become a serious concern.

In this paper, we present Pilaf, a distributed in-memory key-value store that leverages RDMA to achieve high throughput with low CPU overhead. We argue that the sweet spot in the design space is to restrict the use of RDMA to read-only service requests, namely `gets`, while letting the server handle all other requests via traditional messaging. As practical key-value workloads tend to be dominated by read operations [1], this approach can capture most of RDMA's performance benefits while facilitating a much simpler design than using RDMA for all types of requests. In particular, this approach restricts the class of memory access races that can occur: clients might read inconsistent data while the server is concurrently modifying the same memory addresses.

We use *self-verifying* data structures to address read-write races between the server and clients. A self-

verifying data structure consists of checksummed root data objects as well as pointers whose values include a checksum covering the referenced memory area. Starting from a set of root objects with known memory locations, clients are guaranteed to traverse a server's self-verifying data structure correctly, because the checksums can detect any inconsistencies that arise due to concurrent memory writes done by the server. When a race is detected, clients simply retry the operation.

Other projects have also used RDMA to enhance the performance of Memcached-like key-value stores [27, 13, 12]. In these designs, RDMA is treated simply as a means for accelerating standard message-passing. For example, each client sends a `get` request to the server which retrieves the corresponding key-value pair and directly stores it in the client's memory using RDMA. In contrast, in Pilaf, clients can process `get` requests without involving the server process at all, resulting in optimal (zero) CPU overhead. To the best of our knowledge, Pilaf is the first system design where clients can completely bypass the server's CPU for processing read requests.

We have implemented Pilaf on top of Infiniband, a popular HPC network interconnect. Our experiments on a cluster of machines equipped with 20Gbps Infiniband cards show that Pilaf achieves high performance with very low CPU overhead. In a workload consisting of 90% `gets` and 10% `puts`, Pilaf achieves 1.3 million ops/sec while utilizing only a single CPU core, compared to 55K for Memcached and 59K for Redis.

2 Opportunities and Challenges

This section gives an overview of RDMA and other HPC networking features and discusses how they might impact the design of distributed systems. Our discussion of the performance implications is based on Infiniband, a popular HPC interconnect.

Manufactured by Intel and Mellanox, Infiniband hardware provides 10, 20, or 40 Gbps of bandwidth in each direction. Applications running on top of Infiniband have several communication options:

IP over Infiniband (IPoIB) emulates Ethernet over Infiniband. As with normal Ethernet, the kernel processes packets and copies data to application memory. IPoIB allows existing socket-based applications to run on Infiniband with no modification.

Send/Recv Verbs provide user-level message exchange: these Verbs messages pass directly between user space applications and the network adapter, bypassing the kernel. Send/Recv Verbs are commonly referred to as two-sided operations since each Send operation requires a matching Recv operation at the remote process. Unlike IPoIB, applications must be rewritten to use the Verbs API.

RDMA allows full remote CPU bypass by letting one machine directly read or write the memory of another machine without involving the remote CPU. Unlike Send/Recv Verbs, RDMA operations are one-sided, since an RDMA operation can complete without any knowledge of the remote process. RDMA is technically a type of Verbs message. In this paper, we use the term RDMA specifically to refer to RDMA Verbs and the phrase verb messages to refer to Send/Recv Verbs, both of which we use in reliable mode.

We note that Infiniband is not the only network to support RDMA and user-level networking. Similar features have recently been made available in 10 Gbps Ethernet environments. For example, both Myricom and Solarflare offer 10GE adapters that support kernel bypass, and Intel offers 10GE iWARP adapters capable of RDMA over Ethernet. Although it remains unclear which specific hardware proposal will dominate the datacenter market, one can realistically expect future datacenter networks to support some form of CPU bypassing.

2.1 Performance Benefits of RDMA

How fast and efficient is RDMA? How does its performance compare to alternatives such as verb messages or traditional kernel-based TCP/IP transport? We answer these questions by benchmarking the various Infiniband communication options.

Our experiments were run on a small cluster of machines equipped with Mellanox ConnectX-2 20Gbps Infiniband cards. For RDMA experiments, each client node performs RDMA reads on the server. For verb message experiments, each client node issues a request (as a verb message in reliable mode) to which the server responds immediately with a reply. The IPoIB and Ethernet experiments are similar except that we use TCP/IP for exchanging requests and replies. We vary the size of the RDMA read or the request message while fixing the reply size at 10 bytes.

Figure 1 shows the roundtrip latencies of different communication methods. For small operations (< 1024 bytes), a verb message exchange takes less than 10μs, while the RTT of IPoIB or Ethernet is over 60 μs. Our Infiniband switch imposes a lower delay than our Ethernet switch, but the IPoIB latency is similar to that of Ethernet, suggesting that packet processing through the kernel adds significant latency. RDMA achieves the lowest RTT ($\sim 3\mu$s), half that of verb messages. This is because the request/reply pattern of traditional messaging involves two underlying Verbs exchanges. By contrast, an RDMA operation involves only one underlying

Verbs exchange, thereby reducing the latency by up to half.

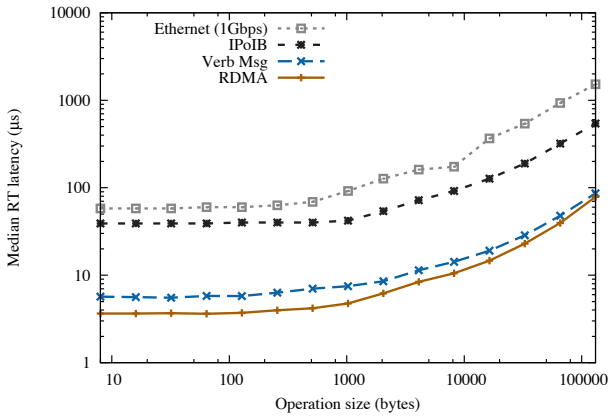


Figure 1: Median round-trip latency. The error bars depict 1% and 99% latency.

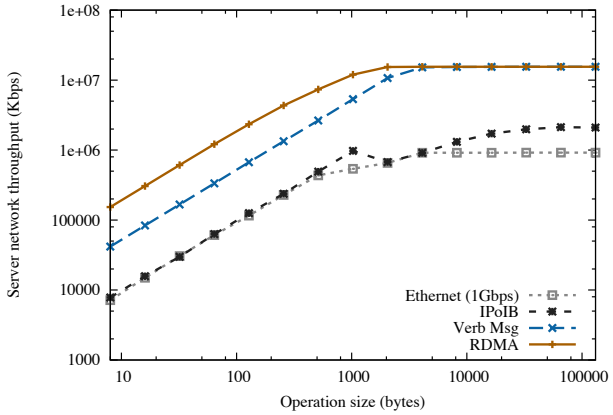


Figure 2: Server's network throughput under different communication methods.

Transport	Throughput (M ops/sec) 16-byte	1024-byte	4096-byte
RDMA	2.449	1.496	0.472
Verbs Message	0.668	0.668	0.464
IPoIB	0.126	0.122	0.028
Ethernet (1Gbps)	0.120	0.068	0.029

Table 1: Throughput (in million operations/sec) for 16 byte, 1Kbyte and 4Kbyte operations.

Figure 2 shows the throughput (in Kbps) achieved by the server. Since different communication methods incur varying CPU overhead, we limit the server's CPU consumption to a single core (AMD Opteron 6272) in all experiments. In Figure 2, large operations (>1024 bytes) over all communication methods except IPoIB can saturate their respective network's peak throughput. For smaller operations, both RDMA and Verbs messages are able to saturate the Infiniband network card's capacity when running the server on a single CPU core. By contrast, kernel-based transports require more than one core to saturate the network card, hence the much lower throughputs achieved in IPoIB and Ethernet experiments.

RDMA not only incurs zero CPU overhead on the server, it also saturates the network card at the highest throughput. As shown in Table 1, a server can sustain 2.45 million operations/second with 16-byte RDMA reads. By contrast, the server can only achieve 0.668 million operations/sec when exchanging Verbs request/reply messages. There are two reason for this performance gap. First, each request/reply exchange uses two underlying Verbs messages compared to one in RDMA. Second, because there is less bookkeeping for RDMA, our network card can perform RDMA at a higher throughput (~2.45 million reads) than sending (~700K) or receiving (~1.5 million) Verbs messages per second for short messages.

2.2 Opportunities for System Builders

As we have seen, bypassing the kernel and CPU allows for reduced latency and CPU overhead. Of these two, CPU bypass via RDMA is particularly powerful in that it achieves the highest throughput while incurring zero CPU overhead. As future datacenter networks embrace RDMA, what will the implications be for the designs of distributed systems infrastructure such as distributed storage systems or computation frameworks? To better understand the ensuing opportunities and challenges, we have chosen to design a distributed key-value store to exploit RDMA. We decided to use the key-value store as a case study system because it is a popular infrastructural service with demanding performance requirements [20]. Key-value stores are also used as a building block for other more sophisticated storage systems (e.g. BigTable [2], Spanner [4], Cassandra [16]) or distributed computation frameworks (e.g. Piccolo [22]).

Our experience in exploring the design space for a key-value store leads to two observations, both of which are applicable to other distributed systems besides key-value stores.

High performance is feasible with fewer CPU resources. With traditional Ethernet-based distributed systems, the performance bottleneck is often the CPU despite the availability of multiple cores [19]. With kernel and CPU bypass, servers can saturate the network using many fewer cores. The improvement in CPU efficiency is particularly notable with RDMA, which potentially allows clients to process service requests without involving the server at all. Efficient CPU usage is crucial in datacenters, which often operate a shared environ-

ment by running multiple applications on a single machine [6]. With less CPU overhead, one can pack more applications onto each machine, use fewer machines, rely on wimpier cores [27] and yet achieve the same or better performance.

Multi-round operations are practical. Because the roundtrip latency on Ethernet is substantial, traditional systems designs aim to minimize the rounds of communications required to complete an operation. For example, existing key-value stores process each `get` or `put` operations in one roundtrip. With RDMA's ultra-low latency, it becomes feasible to use multi-round protocols without adversely affecting end-to-end operation latency. In particular, each `get` operation in Pilaf requires at least two roundtrips.

Challenges. It is technically challenging to fully exploit RDMA's performance advantage in a system design. The common existing practice is to use RDMA to optimize verb message exchange [18, 13, 11]. Specifically, in order to send (or receive) a large message, a client first transmits some control information to the server using a verb message. The server then performs an RDMA read (or write) to the client to fetch (or store) the actual payload. This design maintains the traditional request/reply communication pattern, but does not fully exploit the benefits of RDMA since the overall latency and throughput is still bottlenecked by sending/receiving verb messages.

A more efficient system design is one in which all or a large fraction of the existing request/reply traffic is *replaced* (instead of supplemented) by RDMA operations. However, letting clients directly perform RDMA on the server's memory introduces serious synchronization problems: multiple concurrent RDMA accesses to the same server can cause races, and the server may also simultaneously perform local memory access that conflict with remote accesses. Unfortunately, there are limited hardware mechanisms for synchronizing multiple RDMA accesses, and no efficient capability at all for coordinating local and remote memory accesses.

3 Pilaf Design

This section traces the evolution of Pilaf's design up to its current form. We first motivate Pilaf's overall architecture, which processes write operations at the server and uses RDMA for read-only operations (Section 3.1). We then explain how clients perform `get`s using RDMA reads and discuss how Pilaf synchronizes clients' RDMA accesses with the server's local memory writes. Last, we describe the Cuckoo hashing optimization that reduces the number of required roundtrips in the worst case.

3.1 Overview

The most straightforward design would be to take a traditional key-value store and re-implement its messaging layer using verb messages instead of TCP sockets. However, this design fails to reap the benefits of RDMA, which has much lower latency and CPU overhead than verb messages. Therefore, our goal is to find a system design that can exploit one-sided RDMA operations without adding too much complexity.

A key-value store has two basic operations: $V \leftarrow get(K)$ and $put(K, V)$, where both the key K and value V are strings of arbitrary length. In our initial design iterations, we tried to use one-sided RDMA operations for both `get`s and `put`s. In other words, each client performs RDMA reads to implement `get`s and RDMA writes to implement `put`s.

We quickly discovered that using RDMA for all operations leads to complex and fragile designs. First, clients must synchronize their RDMA writes so as not to corrupt the server's memory. The Infiniband card supports atomic operations (such as compare-and-swap) on top of which one could build an explicit locking mechanism. However, locking over the network not only incurs a performance hit, but also introduces the complication of clients failing while holding a lock. Second, a `put` operation requires memory allocation to store key-value strings of arbitrary length; such memory management becomes unwieldy in the presence of remote writes. Having clients implement memory management remotely is expensive, with excessive locking and round-trips required. On the other hand, letting the server perform memory management introduces write-write races between the server and clients. Unfortunately, unlike synchronization among concurrent clients, there exists no efficient hardware mechanism to synchronize memory accesses initiated by the CPU and the network card. Last but not least, by making all operations transparent to the server, debugging becomes a painstaking, as race conditions involving remote accesses are much more difficult to find and reproduce than those involving local accesses.

Our first major design decision is to have the server handle all the write operations (i.e. `put` and `remove`) and have the clients implement read-only operations (i.e. `get` and `contains`) using one-sided RDMA reads. Since real-world workloads are skewed towards reads (e.g., Facebook reported read-to-write ratios ranging from 68%-99% for its active key-value stores [1]), this design captures most of the performance benefits of RDMA while drastically simplifying the problem of synchronization. In fact, the beauty of this design is that it incurs no write-write races, but only read-write races between RDMA reads and the server's local memory writes, Write-write races are the main source of design

complexities since they must be avoided at all costs to prevent memory corruption. In contrast, read-write races can be made harmless by detecting the presence of such races and re-trying the affected operation. Thus, no fragile and expensive locking protocol is needed.

Figure 3 shows Pilaf's overall architecture. Using verb messages, clients send all `put` requests to the server, which inserts them in its in-memory hashmap before sending the corresponding replies. By contrast, `gets` are transparent to the server in that the clients perform RDMA reads over multiple roundtrips to directly fetch data from the server's memory.

As in other key-value store designs [19, 24], the server asynchronously logs updates to its local disk.

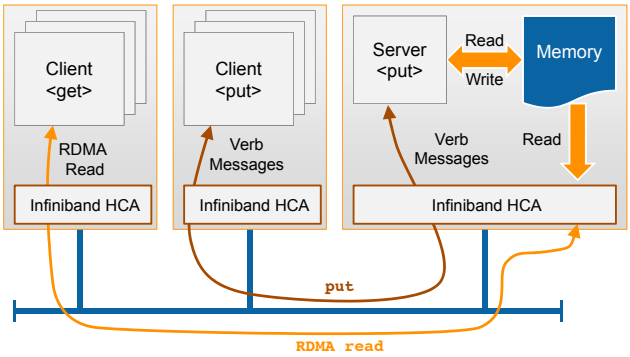


Figure 3: Pilaf restricts the clients' use of RDMAs to read-only `get` operations and handles all `puts` at the server using verb messages.

3.2 Basic `get` Operation Using RDMA

We first explain how Pilaf performs `gets` without involving the server's CPU. We defer the challenge of coping with concurrent `puts` and `gets` to Section 3.3.

To allow RDMA reads, the server must expose its data structure for storing the hash table, as shown in Figure 4. There are two logical memory regions: an array of fixed size hash table entries and an extent area for storing the actual keys and values, which are strings of arbitrary length. The server registers both memory regions with the network card, and clients obtain the corresponding registration keys of these two memory regions (as well as the size of the hash table array) when they first establish a connection to the server. Subsequently, clients can issue RDMA requests to any memory address in these two regions by specifying the memory's registration key and an offset.

In the basic design, a client looks up a key in the hash table array using linear probing [25]. Each probe involves two RDMA reads. The first read fetches the hash table entry corresponding to the key. If the entry is currently filled (indicated by an `in_use` bit), the client initiates a second RDMA read to fetch the actual key and value strings from the extent region according to the address information stored in the corresponding hash table entry. The client checks whether the fetched key string matches the requested key. If so, the `get` operation finishes. Otherwise, the client continues with the next probe.

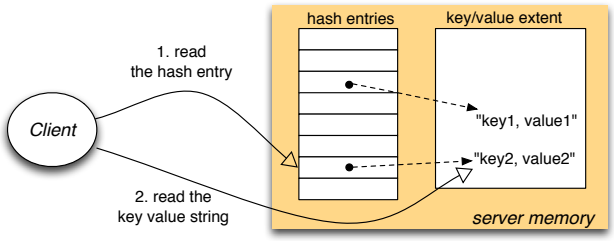


Figure 4: The memory layout of the Pilaf server's hash table. Two memory regions are used, one contains an array of fixed-size hash table entries, the other is an extent storing variable sized key-value extents. Clients perform `get` operations in two RDMA reads, first fetching a hash table entry, then using the address information in that entry to fetch the associated key-value string.

3.3 Coping with Read-Write Races

The Pilaf server handles all `put` operations. Thus, local memory writes performed by the server's CPU unavoidably create potential read-write races with concurrent RDMA reads done by clients. This is a challenge as there exists no efficient hardware mechanism to coordinate the CPU and the network card. To inhibit RDMA reads during a write, the server could resort to resetting all existing connections, or temporarily de-register memory regions with the network card. However, both mechanisms are far too expensive to be used for every `put` operations.

To implement a read operation, clients need to traverse the server's data structure. The traversal starts from a set of "root" objects with known memory locations and recursively follows pointers read previously. In the context of Pilaf, we can view each hash table entry as a "root" object which points to additional key-value information. Read-write races introduce the possibility that clients can traverse the server's data structure incorrectly.

Two scenarios can result in incorrect traversal. First, a root object can be corrupt. In Pilaf, this happens when the server modifies a hash table entry while a client is reading that entry. Consequently, the client will read a partially-modified or corrupt hash table entry, potentially causing it to read the key-value string from an invalid memory location. Second, a client's pointer reference can become invalid. For example, in Pilaf, the server may delete or modify an existing key/value pair while a client is holding a pointer reference to the old string from its first RDMA read of the hash table entry.

Thus, during its second RDMA access, the client might read garbage or an incorrect key-value string.

To permit correct traversal in the face of read-write races, we introduce the notion of a *self-verifying* data structure by making both root objects and pointers self-verifying. For a root object, we append a checksum that covers the object's entire content. Thus, any ongoing modification on the root object result in a checksum failure. To make a pointer self-verifying, we store it as a tuple combining a memory location, the size of the memory chunk being referenced, and a checksum covering the content of the referenced memory. Therefore, a client can detect the inconsistency between a pointer's intended memory reference and the actual memory content. For example, if the server de-allocates the memory chunk being referenced and re-uses parts of it later while a client is still holding a pointer to it, the client will fail to verify the checksum when it retrieves the memory content using the pointer. Figure 5 shows Pilaf's self-verifying hash table. As a root object, each hash table entry contains a checksum covering the whole entry. The pointer stored in each hash table entry contains a checksum verifying the key-value string being referenced.

Self-verifying data structures ensure correct traversal starting from a set of known root object locations. On rare occasions, the server may need to change the root object locations. This can be accomplished correctly by having the server reset all its existing RDMA connections to clients to inhibit clients from reading stale root object locations. In Pilaf, whenever the server needs to resize its hash table array, it resets connections so that clients are prevented from performing RDMA reads until the resize is complete. They are allowed to reconnect once the resize operation is complete to obtain up-to-date information about the location and size of the hash table array. Since hash table resizing is infrequent, there is a minimal performance penalty from resetting connections.

A self-verifying data structure allows clients to perform consistent reads in the face of concurrent writes. In addition, the Pilaf server uses a memory barrier to force any updates from the CPU cache to the main memory before replying to a `put` request. Doing so ensures that a subsequent `get` always reads the effect of any completed `puts`. As a result, Pilaf provides the strongest consistency semantics, i.e. linearizability [10].

3.4 Improving a Hash Table's Memory Efficiency

In the basic design, a client performs linear probing to look up a key in the server's hash table array. This simple hash scheme does not achieve a good tradeoff between memory efficiency and operation latency. For example, when the hash table is 60% full, the maximum number of probes required can be as high as 70. To achieve good

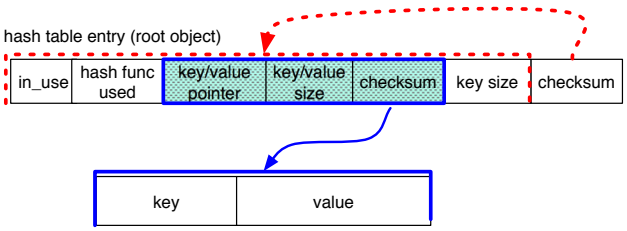


Figure 5: *Self-verifying* hash table structure. Each hash table entry is protected by a checksum. Each entry stores a self-verifying pointer (shown in shaded fields) which contains a checksum covering the memory area being referenced.

memory efficiency with fewer probes, Pilaf uses n-way Cuckoo hashing [21, 15]. This hashing scheme uses n orthogonal hash functions, and every key is either at one of n possible locations or absent. If all n possible locations for a new key are filled, the key is inserted anyway, kicking the resident key-value pair to one of that key's alternate locations. That operation may in turn kick out another pair, ad infinitum. The table is resized when a limit is reached on the number of kicks performed or when a cycle is detected.

The main challenge in using Cuckoo hashing for Pilaf lies in the process of moving an existing entry to a different hash table location. Ordinarily, bulk key movements such as resizing the hash table requires that the server reset all existing RDMA connections. This is not desirable, as the need to move a key occurs much more frequently than table resizing with Cuckoo hashing. Without resetting connections, there is the danger that a key-value pair might appear to be "lost" to the clients while the server is moving it to a new location. To address this issue, during a `put` operation the server first calculates the new locations of every affected key without actually moving the keys. Then, starting from the last affected key, the server shifts each key to its new location, thereby ensuring that a key is always stored in at one or two (instead of zero or one) hash table entries during movement.

We explored different parameter values for n and determined that 3-way Cuckoo hashing achieves the best memory efficiency with few hash entry traversals per read. As Figure 6 shows, at a fill ratio of 75%, the average and maximum number of probes in 3-way Cuckoo hashing is 1.6 and 3, compared to 2.5 and 213 respectively for linear probing.

4 Implementation

We implemented Pilaf in C++. Pilaf uses the `libibverbs` library from the OpenFabrics Alliance, which allows user-space processes to use verb messages and RDMA directly. The Pilaf server continuously polls the network card for new events, including the reception of verb messages or the completion of recently-sent

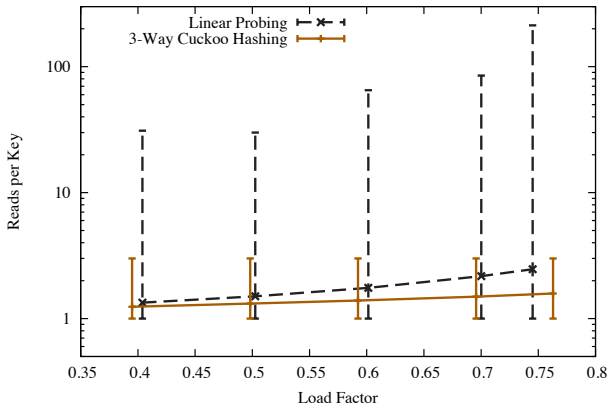


Figure 6: The average number of probes required during a key lookup in 3-way Cuckoo hashing and Linear probing. The error bars depict the median and maximum values.

RDMA operations or verb messages. Since Pilaf is able to saturate the network card's performance using a single thread, our implementation uses the same polling thread to process `puts` as well.

RDMA-Friendly Extents: The server must register a region of memory and gives clients the registration key for that memory before clients can perform RDMA on the region. This process is relatively expensive and should be made infrequent. Therefore, Pilaf allocates and registers a large contiguous address space for the key-value extents. We ported the `mem5` memory management unit from SQLite to C++ to "malloc" and "free" strings in the key-value extents. Whenever the extents region becomes full, the server resets all existing connections, expands the extents, and then allows clients to re-connect and obtain new registration keys. As with hash table resizing, we expect and observe extents resizing to be an infrequent event.

Self-Verifying Data Structures: Our implementation uses CRC64 as the checksum scheme for our self-verifying data structures. CRCs are not effective for cryptographic verification. Instead, they were originally intended to detect random errors, making them ideal for our application. The ideal n-bit CRC will fail to detect 1 in 2^n message collisions. Although 32-bit CRC is popular (e.g. for Ethernet and SATA checksums), we believe that CRC32 is insufficient for Pilaf. Every `put` incurs two CRC updates, one on the hash table entry and one on the key-value string. As will be shown in Section 5.2, Pilaf can process 663K `puts` per second. Therefore, up to 1.326 million CRCs may be calculated per second. Since each CRC32 incurs a collision with probability 1 in 2^{32}, we expect a collision once every 3239 seconds (54 minutes). We find this rate to be unacceptably high. Using CRC64, we can expect a collision once every $1.35 * 10^{13}$ seconds, or once per 428 millenia.

CRC64 is fast. Our implementation consumes about a dozen CPU cycles for each checksummed byte, and incurs the same overhead as CRC32 when running on 64-bit CPUs.

Logging: By default, Pilaf server asynchronously logs all `put` and `delete` operations to the local disk, similar to the logging facility in other key-value stores including Redis [24], Masstree [19] and LevelDB [8]. Using a single solid state disk, Pilaf is able to log 663K (our peak `put` throughput) writes per second if the average key-value size is smaller than 500 bytes. Should one desire a high logging capacity, multiple SSDs must be used.

5 Evaluation

We evaluate the performance of Pilaf on our Infiniband cluster. The highlights of our results are the following:

- Pilaf achieves high performance: its peak throughput reaches 1.3 million ops/sec. The end-to-end operation latency is very low with a 90-percentile latency of $\sim 30\mu$s.
- Pilaf is CPU-efficient. Even when running on a *single* CPU core, Pilaf is able to saturate the network hardware's capacity to achieve 1.3 million ops/sec. By comparison, Memcached and Redis achieve less than 60K ops/sec per CPU core, so they require at least 20$\times$ the CPU resource to match Pilaf's performance.
- Self-verifying data structures are effective at detecting read-write races between the clients' RDMA operations and the server's local memory accesses.

5.1 Experimental setup

Hardware and configuration. Our experiments are run on a cluster of ten machines, each with two AMD or Intel processors and 32GB of memory. Each machine is equipped with a a Mellanox ConnectX-2 20 Gbps Infiniband HCA as well as an Intel gigabit Ethernet adapter. The machines run Ubuntu 12.10 with the OFED 3.2 Infiniband driver.

For each experiment, we run a server process on one physical machine, while the clients are distributed among the remaining machines to saturate the server. By default, we restrict the server process to run on one CPU core. For Ethernet experiments, we configure the kernel's network interrupt processing to trigger on the same core used by the server process.

We disable Pilaf's asynchronous logging in the experiments. With logging turned on, Pilaf incurs no measurable reduction in achieved throughput for key-value

sizes less than 500 bytes. With larger operations, the I/O bandwidth of the server's single local SSD becomes the bottleneck.

Workload. We use the YCSB [3] benchmark to generate our workloads. YCSB constructs key-value pairs with variable key and value lengths, modelled on the statistical properties of real-world workloads. Furthermore, with YCSB, the keys being accessed follow a long-tailed zipf distribution. The original YCSB software is written in Java. We ported it to C so that fewer client machines are required to saturate the server.

In all experiments, we vary the size of the value string from 16 to 4096 bytes while keeping the average key size at 23 bytes, the default value in YCSB. We use two mixed workloads, one consisting of 10% `puts` and 90% `gets`, the other 50% `puts` and 50% `gets`. Since Facebook has reported that most of their Memcached deployments are read-heavy [1], our mixed workloads give reasonable representations of real workloads.

Points of comparison. We compare Pilaf against Memcached [7] and Redis [24] (with logging disabled). Additionally, we also compare Pilaf to an alternative implementation of itself, which we refer to as Pilaf-VO (short for Pilaf using Verb messages Only). In Pilaf-VO, clients send all operations (including `gets`) to the server for processing via verb messages. The performance gap between Pilaf and Pilaf-VO demonstrates the importance of bypassing the CPU using RDMA.

5.2 Microbenchmarks

The microbenchmarks measure the throughput and latency of individual `get` and `put` operations.

Throughput: Figure 7 shows Pilaf's peak operation throughput, achieved with 40 concurrent clients. Pilaf can perform 1.28 million `get` and 663K `put` operations per second for small key-values. Of note is that Pilaf's high throughput is achieved using a single CPU core which saturate the Infiniband card's performance for in most cases.

`Get` operations via RDMA impose zero CPU overhead on the server. Furthermore, `get` operations also have the highest throughput. As shown in Table 1 (Section 2), the network card's achieved RDMA throughput is much higher than that of verb message, especially for small messages. In particular, the card can satisfy 2.45 million RDMA reads per second for small reads. Since each `get` requires at least two RDMA reads, the overall throughput is approximately half of the raw RDMA throughput at 1.28 million `gets`/sec. By contrast, the peak verb throughput is 667K request/reply pairs/sec for small messages, resulting in 667K ops/sec for `puts`.

For larger key-value pairs, the throughputs of `get` and `put` converge as they both approach the network bandwidth. For example, for 4096-byte key-values, Pilaf consumes 11.7Gbps of the 16Gbps data bandwidth supported by the network card. Interestingly, we find that when processing `puts` with large values, the Pilaf server becomes CPU-bound when using a single core. Specifically, for 1024-byte value size, Pilaf achieves 75.4% of its network-bound `put` throughput (500K ops/sec) with one core and 100% (663K ops/sec) with two cores.

We also measure the throughput of Pilaf-VO's `get` operation, which is processed by the server using verb messages instead of by the client using RDMA. As Figure 7 shows, the throughput of performing `gets` using verb messages is similar to that of `puts` and is much smaller than the throughput of `gets` done via RDMA for small key-value pairs.

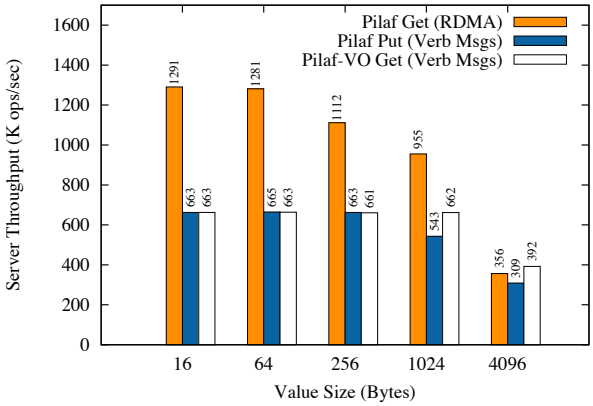


Figure 7: Server throughput for `put` and `get` operations as the average value length is increased. All tests are performed with 40 connected clients.

Latency: Figure 8 shows the latency of `get` and `put` operations with 10 concurrent clients. With 10 concurrent clients performing operations as fast as possible, queuing effects are minimized. With 40 or more clients, the latency is mostly determined by queuing effects and thus is much higher. With a single client (not shown in the figure), the latency of `get` is slightly more than 2 RDMA roundtrips and is twice the latency of `put`. With more clients and thus more load, we found that the RDMA latency scales better than that of verb messages. For small `gets`, the average latency is 12μs, while small `puts` take around 15μs. For large key-values, the latencies of `get` and `put` are similar and both bounded by the packet transmission time.

5.3 Performance of self-verifying data structure

Pilaf uses a self-verifying hash table structure to detect read-write races during concurrent `gets` and `puts`. We expect such races to be rare in a normal workload. To artificially vary the conflict rate, we inject the maximum achievable `get` and `put` loads, simultaneously reading

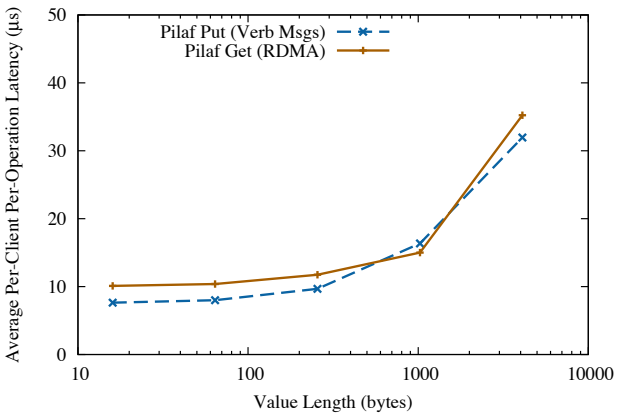


Figure 8: Average operation latency for `put` and `get` operations as the average value size increases. All tests are performed with 10 connected clients; though not pictured, we observe a linear relationship between the number of connected clients and latency due to queuing effects.

and writing a varying number of unique key-value pairs. Therefore, the probability of races increases as the `gets` and `puts` are restricted to fewer and fewer unique keys.

Figure 9 shows the probability of detecting a read-write race as measured by the fraction of `gets` that need to be re-tried. The two lines in Figure 9 illustrate the probabilities of a retry due to a race when reading the hash table entry or when reading the key-value extent. As we can see, there is non-negligible race rate only when the hash table is extremely small. When the table contains more than 20,000 keys, the probability of racing is less than 0.01% even under peak `put` and `get` loads.

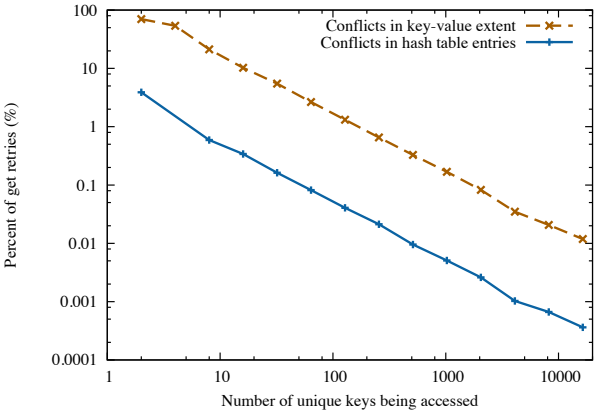


Figure 9: Percentage of re-reads of extents and hash table entries due to detected read-write races. We control the rate of conflicts by varying the number of unique keys being read or updated. The Pilaf server is operating under peak operation throughput.

5.4 Pilaf versus Memcached and Redis

We compare Pilaf to two existing popular key-value systems, Memcached [7] and Redis [24]. Both systems are widely deployed in the industry, including Facebook [1], YouTube [5], and Instagram [14]. Memcached is commonly used as a database query cache or a web cache to speed up the server's generation of a result web page and improve throughput. Low operation latency is vital in such a usage scenario: the faster the key-value cache can fulfill each request, the faster a page involving many cache lookups can be returned to the client. High throughput and low CPU overhead are also crucial, since these properties allow more clients can be served with fewer server resources.

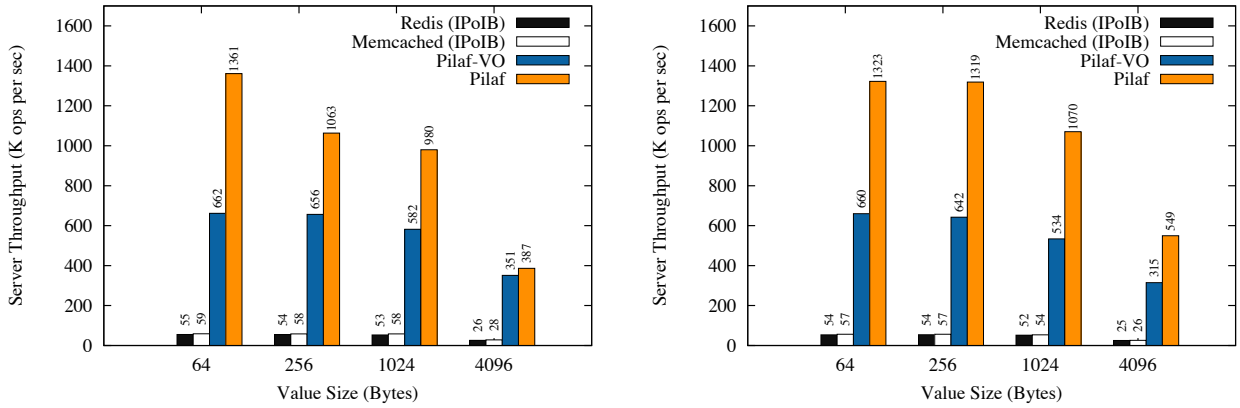
Because Memcached and Redis are written to use TCP sockets, we run them on our Infiniband network using IPoIB. It's important to note that we do not batch requests for any of the systems, unlike in [19].

In our experiments, the peak throughput of each system is achieved when running 40 concurrent client processes. We use two mixed workloads, one containing 90% `gets` and 10% `puts` and the other containing 50% `gets` and 50% `puts`.

Throughput Comparison: Figure 10 shows the achieved operation throughput using a single CPU core for various value sizes in a workload with 90% `gets`. We can see that the performance of Pilaf far exceeds that of Redis and Memcached running on top of IPoIB. For small operations (64-byte values) Pilaf achieves 1.3 million ops/sec compared to less than 60 Kops/sec for Memcached and Redis. Both Memcached and Redis are bottlenecked by the single CPU core and are unable to saturate the Infiniband card's performance. Because of the CPU bottleneck, their single core performance is the same when running on 1 Gbps Ethernet. We elided those numbers from Figure 10 for clarity.

The throughputs of Memcached and Redis can be scaled by devoting more CPU cores to each system. For example, both systems can saturate the 1Gbps Ethernet card when running on four CPU cores. We were not able to scale Memcached and Redis' performance on IPoIB using more CPU cores because the IPoIB driver is unable to spread network interrupts across multiple cores. Nevertheless, even if we optimistically assume perfect scaling, Memcached and Redis require 17× CPU cores to match the performance of Pilaf running on a single core for small key-values. In reality, these systems do not exhibit perfect scaling. For example, [19] reported a 11× throughput improvements for non-batched Memcached puts when scaling from 1 core to 16 cores.

When comparing against Pilaf-VO, we see that Pilaf also achieves substantially better throughput across all operation sizes. In particular, the throughput of Pilaf is



(a) Peak throughput (90% `gets`, 10% `puts`)

(b) Peak throughput (50% `gets`, 50% `puts`)

Figure 10: Throughput achieved on a single CPU core for Pilaf, Pilaf-VO, Redis, and Memcached.

2.1× that of Pilaf-VO for 64-byte values and this performance advantage decreases to 1.1× for 4096-byte values. The shrinking performance gap between Pilaf and Pilaf-VO for larger values reflects the increasingly dominant network transmission overhead for large messages.

Figure 10(b) shows the peak throughput of different systems in a second workload with 50% `gets` and 50% `puts`. Not surprisingly, the performance of Memcached and Redis are similar under both workloads.

We were surprised to see that Pilaf achieves identical and sometimes better throughput in the second workload compared to the first. Since RDMA-based `get` operation has much higher performance than verb message-based `put` (Figure 7), we initially expected the second workload to achieve worse throughput since it contains a larger fraction of `puts`. On further investigation, we found that our Infiniband cards appear to be able to process verb messages and RDMA operations somewhat independently. Quantitatively, the card can reach ~80% of its peak RDMA throughput while *simultaneously* sending and receiving verb messages at ~95% of the peak verb throughput. This explains why the second workload has better throughput. For example, with 256-byte values, the first workload achieves 0.9 million `gets`/sec (80% of peak RDMA performance) and 0.1 million `puts`/sec (far less than the card's verb message sending capacity). By contrast, the second workload produces 0.65 million ops/sec for both `get` and `put` which represents 60% of the card's peak RDMA performance and 94% of the card's verb message performance. Thus, the second workload has a total throughput of 1.3 million ops/sec, better than that achieved by the first workload.

Latency: Figure 11 shows the cumulative probability distribution of operation latencies under different systems in the workload with 90% `gets`. The underlying experiments involved 10 concurrent clients issuing operations with 1024-byte values as fast as possible.

In Figure 11, Pilaf's median latency 15μs which is determined by the `get` operation latency. From the earlier experiments in Section 2 (Figure 1), we know the average RDMA roundtrip latency is 4μs for 1024-byte reads with a single client. With an average of 1.45 probes (each involving two RDMA reads) to find a particular key-value in a 65%-filled 3-way Cuckoo hash table, the ideal `get` latency would be 11.2 μs. The extra 4μs reflects the overhead in calculating CRCs on the clients' side plus the queuing effects incurred by having ten connected clients. The latency tail in Figure 11 is very short.

As expected, IPoIB also maintains lower latency than Ethernet for both Memcached and Redis. Median Ethernet latency is 209μs for Redis and 230μs for Memcached. Pilaf beats Redis' and Memcached's median Ethernet latency by 14×-15×, and their median IPoIB latency by 9×-11×. The experiments for Figure 11 involve ten clients connected to a single server. In these experiments, Pilaf-VO reaches 95% of its peak throughput, Memcached is at 75% of its maximum throughput, and Redis and Pilaf at half their peak throughput. Therefore, queuing effects are uneven for these systems in Figure 11. When tested under very light loads (e.g. a single client), Pilaf-VO and Pilaf have similar latency while Memcached and Redis running on IPoIB also have similar latency.

6 Related Work

There has been much work in the HPC community to exploit performance critical features like kernel and CPU bypassing. Many MPI implementations, e.g. MPICH/MVAPICH [17, 18] and OpenMPI [26], supports an Infiniband network layer, leveraging both verb messages and RDMA to reduce latency and increase bandwidth.

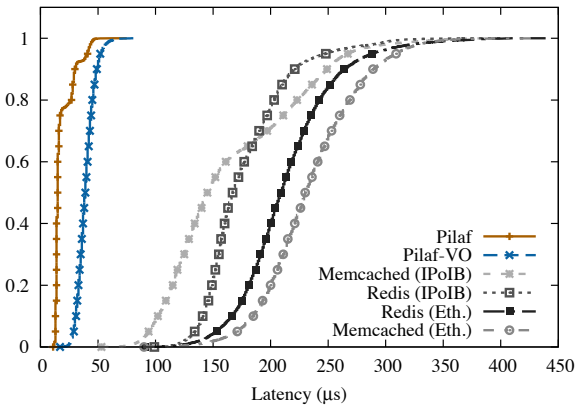


Figure 11: CDF of Pilaf latency compared with Memcached, Redis and Pilaf-VO in a workload consisting of 90% `gets` and 10% `puts`. The average value size is 1024 bytes. The experiments involved 10 clients.

RDMA as a powerful HPC networking feature has been recognized in the system community in several works. Due to the perceived cost of specialized HPC hardware, some have advocated software RDMA over traditional Ethernet. Soft-iWARP is a version of the iWARP protocol implemented entirely in software [28]; it reduces TCP latency by 5%-15% by minimizing data copying and limiting the number of context switches required. Another project later used soft-iWARP to realize a 20% reduction in per-`get` CPU load for Memcached without Infiniband hardware [27].

Many have leveraged RDMAs to improve the throughput and reduce the CPU overhead of existing networked systems such as PVFS distributed filesystem [30], NFS [9], Memcached [13, 12, 27], HBase [11]. All of these works port existing system designs to a modified networking backend which utilizes RDMA within traditional request/reply message exchange pattern. In other words, RDMAs are used as a supplement mechanism to optimize data transfers while verb or other messaging mechanisms are required before each RDMA to signal control information before the transfer. As an example, a client sends a verb message to instruct the server to perform an RDMA read (or write) to the client. When the server completes the RDMA operation, it replies with another verb message informing the client that the transfer is complete. This strategy uses RDMA effectively only for large messages since the throughput and CPU overhead of processing small messages are still bounded by the verb message performance. By contrast, our work aims to replace a large fraction of the request/reply message exchanges with RDMA reads by the clients, thereby significantly reducing the server's CPU overhead.

The three projects that implement Memcached over RDMA on Infiniband [13, 12] or soft-iWARP [27] also adopt the usual combination of control messages plus RDMAs write to process `gets` and `puts` at the server. In [13], the client uses a verb message to send the server a local buffer address, which the server then copies data into using an RDMA write. `Put` operations also involve two verb messages and one RDMA read, wherein the client gives the server an address, from which the server pulls a key-value pair via RDMA read. Both `put` and `get` include short-operation optimizations that combine the data normally read or written via RDMA into one of the verb messages exchanged. Compared to Pilaf, this design achieves much lower throughput. Their reported performance in a Infiniband cluster similar to ours is 300 Kops/sec for small operations, significantly lower than that achieved by Pilaf (1.3 million ops/sec). The other Memcached over Infiniband project [12] combines Infiniband's Reliable Connection (RC, with guarantees similar to TCP) and Unreliable Datagram (UD, resembling UDP) modes. The resulting performance is also lower than achieved by Pilaf, despite running on a QDR Infiniband cluster which is twice as fast as ours (DDR).

7 Conclusion

As future datacenter networks move towards incorporating HPC network features, it is time to rethink networked system designs that can fully exploit powerful features like RDMA. We have demonstrated such a design by building a high-performance key-value store with very low CPU overhead. Pilaf replaces the usual request/rely messaging pattern for read-only operations by having the clients directly read from the server's memory using RDMA. It uses *self-verifying* data structures to detect read-write races in the face of concurrent RDMA reads done by the clients and the local memory accesses done by the server. Pilaf is able to achieve a peak throughput of over 1.35M read and 663K write operations per second from a single CPU core, outperforming existing systems running over Ethernet or IPoIB by more than an order of magnitude.

Acknowledgements

Members of the NeWS group – Yang Zhang, Russell Power, and Aditya Dhananjay – gave valuable feedbacks that helped improve this work. Our special thanks go to Yang Zhang, who first suggested using CRCs to check for data inconsistency. Frank Dabek suggested useful experiments to evaluate Pilaf's self-verifying data structure. This work was partially supported by NSF Award CSR-1065114 and a Google Research Award. Yifeng Geng was supported by a Tsinghua visitor scholarship.

References

[1] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale

key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[2] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W., WALLACH, D., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[3] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[4] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, pp. 251–264.

[5] CUONG, C. Youtube scalability. In *Google Seattle Conference on Scalability* (2007).

[6] DEAN, J. Software engineering advice from building large-scale distributed systems. Slides.

[7] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[8] GHEMAWAT, S., AND DEAN, J. Leveldb, 2011.

[9] GIBSON, G., AND TANTISIRIROJ, W. Network file system (nfs) in high performance networks. Tech. rep., Carnegie Mellon University, 2008.

[10] HERLIHY, M., AND WING, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[11] HUANG, J., OUYANG, X., JOSE, J., UR RAHMAN, M. W., WANG, H., LUO, M., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High-performance design of HBase with rdma over infiniband.

[12] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012).

[13] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011).

[14] KRIEGER, M. What powers instagram: Hundreds of instances, dozens of technologies.

[15] KUTZELNIGG, R., AND DRMOTA, M. *Random bipartite graphs and their application to Cuckoo Hashing*. PhD thesis, PhD thesis, Vienna University of Technology, 2008.

[16] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[17] LIU, J., JIANG, W., WYCKOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (april 2004), p. 16.

[18] LIU, J., WU, J., KINI, S., BUNTINAS, D., YU, W., CHANDRASEKARAN, B., NORONHA, R., WYCKOFF, P., AND PANDA, D. Mpi over infiniband: Early experiences. In *Ohio State University Technical Report* (2003).

[19] MAO, Y., KOHLER, E., AND MORRIS, R. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), pp. 183–196.

[20] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcached at facebook. In *Proceedings of USENIX NSDI 2013* (2013).

[21] PAGH, R., AND RODLER, F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[22] POWER, R., AND LI, J. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2010).

[23] RASHTI, M., AND AFSAHI, A. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), pp. 1–8.

[24] SANFILIPPO, S., AND NOORDHUIS, P. Redis.

[25] SEDGEWICK, R., AND WAYNE, K. *Algorithms*. Addison-Wesley, 2011.

[26] SHIPMAN, G., WOODALL, T., GRAHAM, R., MACCABE, A., AND BRIDGES, P. Infiniband scalability in open mpi. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), pp. 10–pp.

[27] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of USENIX Annual Technical Conference* (2012).

[28] TRIVEDI, A., METZLER, B., AND STUEDI, P. A case for rdma in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), p. 17.

[29] VIENNE, J., CHEN, J., WASI-UR-RAHMAN, M., ISLAM, N., SUBRAMONI, H., AND PANDA, D. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on* (aug. 2012), pp. 48 –55.

[30] WU, J., WYCKOFF, P., AND PANDA, D. Pvfs over infiniband: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on* (2003), pp. 125–132.

Lightweight Memory Tracing

Mathias Payer
ETH Zurich

Enrico Kravina
ETH Zurich

Thomas R. Gross
ETH Zurich

Abstract

Memory tracing (executing additional code for every memory access of a program) is a powerful technique with many applications, e.g., debugging, taint checking, or tracking dataflow. Current approaches are limited: software-only memory tracing incurs high performance overhead (e.g., for Libdft up to 10x) because every single memory access of the application is checked by additional code that is not part of the original application and hardware is limited to a small set of watched locations.

This paper introduces *memTrace*, a lightweight memory tracing technique that builds on dynamic on-the-fly cross-ISA binary translation of 32-bit code to 64-bit code. Our software-only approach enables memory tracing for unmodified, binary-only x86 applications using the x64 extension that is available in current CPUs; no OS extensions or special hardware is required. The additional registers in x64 and the wider memory addressing enable a low-overhead tracing infrastructure that is protected from the application code (i.e., uses disjunct registers and memory regions). MemTrace handles multi-threaded applications. Two case studies discuss a framework for unlimited read and write watchpoints and an allocation-based memory checker similar in functionality to memgrind.

The performance evaluation of memTrace shows that the time overhead is between 1.3x and 3.1x for the SPEC CPU2006 benchmarks, with a geometric mean of 1.97x.

1 Introduction

Analyzing memory accesses in large applications is a hard problem due to limitations of the current tracing infrastructure and hardware. Dynamic program instrumentation that naively instruments every memory access results in high execution overhead (20x for Valgrind's memcheck [18], up to 10x for libdft [14], up to 21.1x for compression for PTT [9], and up to 40x for taintcheck [19]), and the execution overhead makes it often impossible to execute large instrumented applications up to the point where a specific bug is triggered. Hardware watchpoints are limited to a small set of memory locations but allow tracing at native performance.

Memory tracing allows the execution of *memlets* for every memory access of the instrumented application. Memlets are code sequences that are woven into the executed application code. These memlets can execute additional code for each memory access depending on: (i) the data value that is read or written, (ii) the address that is read from or written to, or (iii) the state associated with the address that is read or written (the tracing infrastructure may provide additional state – a *shadow value* – for every memory location that is used in an application). Memory tracing is lightweight if the overall performance overhead added through the memlets is low. Memlets can use the state and the value of each memory location to implement high-level functionality like (unlimited) watchpoints, dataflow tracking, or taint checking.

This paper presents *memTrace*, a framework for lightweight memory tracing for single-threaded and multi-threaded 32-bit applications. MemTrace combines an API to set and check shadow values for every byte used in the application with an interface to implement different user-defined memlets. We present two example memlets that (i) support an unlimited number of memory watchpoints and (ii) enforce explicit safety regions around every memory allocation for C/C++ applications to find memory corruption bugs like buffer overwrites and buffer underwrites, and these memlets handle arbitrary unmodified 32-bit binary applications.

Current memory tracing systems use software binary translation to instrument all memory accesses of an application with a pre-determined set of instructions (i.e., current systems do not support user-configurable memlets). Some systems reuse "unused" registers (e.g., minemu [4] uses the SSE registers and therefore only supports applications that do not use SSE instructions, LIFT [23] uses x64 registers) while other systems (e.g., PIN [15], or Valgrind [18]) reallocate registers during the binary translation process. Unused registers speed up memory tracing because the memlets and the memory checks use these registers, and no spill code is needed.

All recent Intel and AMD x86 CPUs are x64 capable, on the other hand most applications are based on the 32-bit x86 ISA (e.g., the recommended Ubuntu end-user image uses only 32-bit applications, and all Windows and MacOS operating system images exist in a 32-bit and a 64-bit version). A drawback of x64 is the increased memory usage due to the 64-bit pointer width and the larger page tables. Most applications fit well into a 32-bit memory space. MemTrace enables lightweight memory tracing for these common x86 applications and uses the available features of the already dominant x64 hardware. The combination of free registers to implement the lookup checks and a data structure that supports fast and efficient lookup for individual memory locations is key to low execution overhead.

MemTrace uses cross-ISA translation for 32-bit applications to a 64-bit ISA to offer both a wider address space and additional registers to user-defined memlets. The memTrace

prototype implementation leverages the x64[1] ISA to implement efficient memory tracing for unmodified x86 applications. The x86 code is dynamically translated to x64 code. The x64 ISA is the 64-bit extension of x86. Most instructions are available in both ISAs and can be translated easily. The cross-ISA translation provides 8 additional registers that can be used for the memlets. A shadow memory area above the 4GB limit of the 32-bit x86 application (i.e., application code uses only 32-bit pointers and is therefore unable to interfere with the shadow memory area) is used to store the data used by the memlets. Our prototype implementation of memTrace supports all x86 instructions, including all FPU and SSE extensions.

The flexible implementation of memlets combined with shadow data enables additional fine-grained operations that build on top of memory tracing like dataflow tracking or taint checking. The memlets update the data or taint information for each memory location and check the integrity of the data upon every memory access.

A key observation of Greathouse et al. [12] is that a fast memory tracing framework needs some form of additional hardware extensions to achieve low overhead. This paper shows that low overhead memory tracing can be achieved in software by using additional hardware resources (more registers and a wider address space) that are available through dynamic cross-ISA translation. The memTrace memory tracing technique offers new opportunities for debugging, dataflow tracking, or other user-defined memlets that evaluate fine-grained memory access.

The memTrace prototype implementation supports arbitrary applications like the OpenOffice office suite or the Apache webserver. A performance evaluation of the memTrace prototype implementation for x64 Linux kernels with the SPEC CPU2006 benchmarks shows low overhead with a geometric mean of 1.97x. The contributions of this paper are as follows:

1. The architecture of *memTrace*, a lightweight memory tracing technique for binary-only 32-bit applications that supports user-defined memlets and leverages cross-ISA translation.
2. A case study that shows two memlets: one that supports unlimited watchpoints and a second one that checks an application for memory allocation errors (allocation over-writes and under-writes).
3. An evaluation and discussion of a prototype implementation of the memory tracing technique for x86/x64 and the corresponding memlets.

The rest of the paper is organized as follows: Section 2 lists requirements for lightweight memory tracing; Section 3 describes cross ISA binary translation; Section 4 shows two case studies that use memory tracing; Section 5 presents the memTrace implementation; Section 7 discusses related work; and Section 8 concludes.

[1]Multiple different names are used for the 64-bit extension of x86: x64, EM64T, AMD64, IA-32e, and x86-64. This paper uses x64.

2 Requirements for lightweight memory tracing

MemTrace is a technique for lightweight memory tracing that builds on dynamic cross-ISA binary translation. Dynamic binary translation keeps the overhead low and cross-ISA translation from 32-bit to 64-bit enables the memlets to access a broader memory space than the original ISA permits. This paper discusses 32-bit programs running on a 64-bit ISA. Other combinations work analogously, e.g., 16-bit code running on a 32-bit ISA, as long as the address space of the target ISA is a super-set of the source ISA. A lightweight memory tracing technique must fulfill the following requirements:

Unchanged application address space: the 32-bit application has access to the full 4GB memory space. The larger target address space allows memTrace to hide the binary translation framework and all the data structures needed by the memlets from the application. Neither the binary translator nor the memlets store any internal state in the application memory space. The memlets may change application memory values as part of their functionality. This requirement ensures that the binary translator does not interfere with the original memory layout of the application and, e.g., the placement of shared libraries.

Unmodified execution: the translated application follows the same control flow pattern as the original application. The application uses the original return addresses on the stack, the original function pointers, and the original targets for indirect jumps. The translated code executes additional lookups in a mapping table to transparently map from translated to untranslated code targets. This requirement ensures that the application can use original addresses, e.g., as control flow targets.

Full isolation: the application has no access to data of the binary translator or to data of the memlets. The translated application cannot access any data above the original application segment (due to the restriction of 32-bit pointers). This requirement ensures that the application cannot modify any internal data.

Flexible memlets: the memory tracing technique enables the implementation of flexible memlets that use shadow memory or registers as state. MemTrace allows the implementation of any memlet that needs one or more bytes of state for each byte that the application uses. The memlets can use additional available free registers in the target ISA.

Low overhead: the overall overhead of the memory tracing technique must be low, and the other requirements must not preclude a fast implementation.

The technique for lightweight memory tracing presented in this paper fulfills the criteria above.

3 Cross-ISA binary translation

Cross-ISA binary translation takes a program written in a source ISA and executes the program on a different target ISA. Multiple reasons for cross-ISA translation exist, e.g., program portability, or additional resources in the target ISA. Depending on the differences between the two ISAs the translation is non-trivial.

This paper discusses two different architectures that are both extended from 32-bit to 64-bit, namely the Intel x86 platform and the ARM platform. The first example covers the x86 ISA. The x86 ISA evolved over more than 20 years and was extended multiple times. 32-bit x86 and the 64-bit extension x64 are closely related. x64 widens the registers and the address size to 64-bit, adds 8 general purpose registers, and introduces new instructions. Some instructions are removed as well: 16 one-byte x86 instructions are replaced and reused as prefixes for the new x64 instructions. The original 16 instructions are no longer available on x64 and must be emulated using longer instructions. Additional changes include (i) the limitation of segmentation which makes binary translation for x64 harder [27] and (ii) the way system calls are executed.

A second example is binary translation for the ARM platform. The ARMv8-A architecture supports two ISAs: the AArch64 ISA is a 64-bit extension of the 32-bit AArch32 ISA. Similar to the x64 extension of x86 AArch64 supports a wider address space and a wider register file. The prototype implementation focuses on x86/x64 but the design of the memTrace technique is applicable to AArch64/AArch32 as well because our technique relies on a wider address space and similar instructions between the two ISAs. A notable difference between x86 and ARM is that the instruction pointer (EIP) cannot be accessed directly on x86 while it is a regular register on ARM. The binary translator modifies the EIP to execute translated code from the code cache but emulates all instructions that indirectly use the EIP to keep up the illusion of an unmodified application (e.g., `call foo` is translated into `push orig_eip; jmp transl_foo`. In contrast to x86, an ARM implementation must emulate all instructions that use the program instruction counter directly as well.

There are two problems that must be solved for binary translation for 32-bit x86 programs: register pressure and location of internal data structures of the binary translator. Register allocation on x86 is a hard problem [1, 2, 29] and register reallocation in a binary translator without type information and control-flow information is even more complex. Translating 32-bit x86 applications to x64 code solves the register pressure problem. The 8 additional registers are used by both the dynamic binary translator to implement the translation process and the memlets to implement the memory tracing. The translated application uses the unchanged original registers except for the program counter. Same-ISA binary translators modify the original memory address space of an application by placing internal data structures somewhere into the existing memory space. Cross-ISA translation from x86 to x64 enables a wider address space. Consequently, the translated application uses the low 4GB of memory and the binary translator and the memlets place their data in the upper memory areas. The translated application keeps using 32-bit pointers and cannot access the memory of the binary translator.

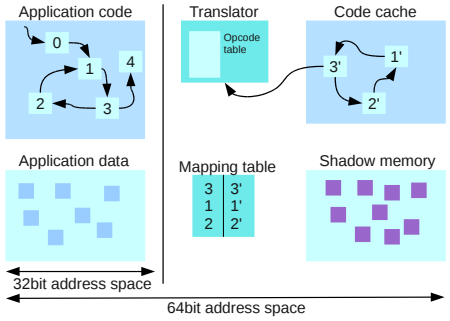


Figure 1: Binary translator runtime layout. Basic blocks are translated and placed in the code cache using opcode tables. The mapping table maps addresses in the program to translated addresses. Trampolines invoke the translator for untranslated basic blocks.

To summarize, the advantages of cross-ISA binary translation are: (i) additional registers available for the instrumentation, (ii) memory separation as translated x86 code cannot access the code of the translator, and (iii) full encapsulation of the translated application.

A possible disadvantage is that some hardware features like segmentation are limited. Fortunately segmentation is not used in user-space applications except for thread local data. Segmentation for thread local data is still supported on x64.

3.1 Dynamic binary translation

Dynamic binary translation instruments a user-space application on the fly. Figure 1 shows the design of the dynamic binary translator and the memory layout. The translator compiles individual basic blocks of the original x86 application on demand and places the translated code in a code cache. Translated control flow transfers use the mapping table to translate targets in the original application to targets in the code cache. Untranslated target fall back to the translator. All executed code is either a part of the translator or of the generated code.

Instructions are translated using a table-based translation scheme as described in libdetox [21]. Most instructions are copied verbatim. For cross-ISA translation some instructions must be adapted due to different memory encodings or addressing schemes, other instructions are emulated by the translation layer. In addition, all instructions that alter control flow (e.g., jump instructions, call instructions, return instructions, system calls, or interrupts) are adapted so that the binary translator keeps control of the translated application.

3.2 Memory layout

The x64 ISA uses 64-bit wide pointers (whereas the physical memory bus is up to 48bit wide). The binary translator maps the original application to the low 4GB. The binary translator library, the code cache, the mapping table, and all shadow memory are placed above the 4GB limit of the translated application.

The translated application still uses 32-bit pointers and 32-bit registers and has therefore no access to any data of the binary translator. This enables *hardware enforced protection* of the internal data from the translated application as the application is not able to generate a memory access to that region due to the 32-bit wide pointers used in the source ISA. In addition, the application has exclusive access to the original 32-bit address space; the binary translator keeps all data in higher memory areas.

3.3 MemTrace design summary

All state for the memlets and the internal data for the binary translator are stored in the area of the 64-bit address space above the first 4GB. The wider address space of a 64-bit ISA like x64 in comparison to a 32-bit address space allows the binary translator to place the shadow memory data structure and all binary translator data structures into an address area that is not accessible from the original application. The translated application uses the low 4 GB of the 64-bit address space that overlaps with the complete 4 GB address space of a 32-bit application. The binary translator is completely concealed; translated code is put in a code cache and every control flow transfer uses a mapping table to map the original target in the application memory space to the translated target in the binary translator space. The application is fully isolated from the binary translator: all pointers in the application domain are 32-bit; the application has no access to any data of the binary translator. The evaluation of the prototype implementation shows low performance overhead with a geometric mean of 1.97x for the SPEC CPU2006 benchmarks.

4 Memory tracing case studies

This section presents two case studies that use the lightweight memory tracing technique. The first case study designs a memlet for unlimited watchpoints. The memlet for unlimited watchpoints supports both read and write watchpoints and can be used to overcome the hardware limitation of 4 write watchpoints on current x86 platforms.

The second case study implements a memory allocation checker. Upon every allocation in a C or C++ program the memory checker adds additional safe zones around the allocated memory region. Any out-of-bounds reads and writes are detected and stop the program.

4.1 Case study: a memlet for unlimited watchpoints

Watchpoints are used to debug applications and enable the inspection of specific memory addresses. Read watchpoints are triggered whenever the location is read and write watchpoints are triggered whenever the location is written. For example, if a certain address is read or written by a bug in the application then a watchpoint can be used to find the code location and context where that read or write access is executed. x86 supports up to 4 (up to 8 byte wide) hardware watchpoints that can be set using debug registers. For many use cases 4 watchpoints are not enough as a wider memory region must be protected to find a specific bug.

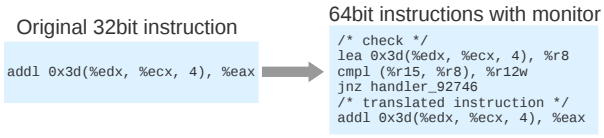


Figure 2: Translation of a memory accessing instruction.

The lightweight memory tracing technique facilitates the design of a simple watchpoint memlet that implements unlimited read and write watchpoints with constant overhead. The overhead is constant for every memory access and does not increase with the number of watchpoints.

The watchpoint memlet uses a shadow memory segment of the same size as the original application. The shadow memory is mapped with a 4GB offset (i.e., the address `0xdeadbeef` is shadowed at `0x1deadbeef`). Every byte in shadow memory is either 0 (if no watchpoint should be triggered) or non-0 if either a read or write watchpoint is set. Figure 2 shows the translation of a sample instruction that reads a memory address. The instruction is translated to 64-bit by expanding all pointers in the instruction. MemTrace adds the memlet before the memory-accessing instruction and checks if the shadow data is 0. The register `%r15` holds the constant offset `0x100000000`, `%r12` keeps the value 0, and `%r13` is used to store the watchpoint information.

The memlet is optimized for fast execution: the instruction cache (i-cache) pressure is reduced by using shorter instruction encodings for memlets and moving the watchpoint handler (the cold path) out into a trampoline. The memlet uses two registers (`%r12` and `%r15`) to store constants. Each replacement of a constant with a register saves 8 bytes in the instruction length. In addition, the translator generates a cold path trampoline for each instruction that accesses memory. The trampoline stores the context (i.e., original IP of the instruction that triggered the watchpoint) and transfers control to the general watchpoint handler.

An interesting feature of the shadow memory segments is that unaligned multi-byte memory accesses are supported. If an instruction accesses a multi-byte value then the shadow bytes of all bytes are combined. The memlet checks for non-0 and detects with a single check if a watchpoint is set for at least a single byte of the multi-byte access.

The watchpoints can either be used by a debugging script/program or can be used as regular watchpoints in GDB. GDB allows remote stubs as backends with the standard GDB

frontend using a remote serial protocol [10]. The backend implements a simple protocol to, e.g., read registers, set breakpoints, and to set watchpoints. The remote stub starts the application under the control of the lightweight memory tracing prototype implementation. Watchpoints are forwarded from the GDB frontend and activated using the watchpoint memlets in translated code.

4.2 Case study: safe heap memory allocator

Ptmalloc2 [11], the standard allocator for C and C++ is an in-place memory allocator that stores information about each allocated memory block before and after the block. This information may be overwritten by buffer overflows or random memory writes. Such bugs are hard to find because memory corruption bugs might only cause a segmentation fault when the block is reused the next time.

This case study uses the watchpoint memlet to set watchpoints before and after every allocated memory block. Calls to the memory allocator are intercepted by the binary translation framework and new watchpoints are added dynamically. If a block is collected (freed) then the watchpoints are removed.

If a bug in the application writes to a watchpoint or reads a watchpoint (i.e. the application accesses an illegal memory region) then the application is either terminated with an information message or a debugger is attached dynamically so that a programmer can analyze the problem.

5 Implementation

The prototype implementation of memTrace extends the libdetox [21] binary translation platform. The libdetox platform is a table-based x86 to x86 binary translator. Our prototype implementation extends the translator with a cross-ISA translation module that transforms x86 instructions to equivalent x64 instructions. The complete prototype implementation is released as open source.

The prototype implementation maps the 32-bit version of the standard loader `ld.so` into the 32-bit address space and prepares the application stack with the correct parameters that `ld.so` expects for the initialization of a 32-bit application. Next the binary translator starts translation and execution of the loader code which loads and initializes all needed shared libraries and starts the execution of the application.

The following sections discuss the translation of individual instructions, present how the memory layout of a translated application looks, and focus on specific translation details.

5.1 Instruction translation

Due to the similarity of the two ISAs the encoding of most instructions is similar as well, the translation is straight-forward and follows the concept of other table-based translators. For most instructions the available encodings on x64 are a super-set of the available encodings of x84. The binary translator uses linked instruction tables to decode the current instruction. If the instruction accesses memory then the pointers are zero expanded to 64-bit memory addresses and the memlet is emitted before the translated instruction. The binary translator uses one of the following translation schemes for each instruction:

Emulation: instructions that are not available on x64 (e.g., `pusha`, or `popa`) are replaced by a sequence of instructions that emulates the removed instruction transparently.

Exception: instructions that are no longer used (e.g., `aaa`, or `aad`) raise an exception. The binary translator fails gracefully and prints an error message. An emulation of these instructions can be added if needed.

Encoding: some instructions are encoded differently on x64 (e.g., `inc`, or `dec`). These instructions are replaced during the translation process.

Addressing mode: x64 uses an instruction pointer relative addressing mode instead of an absolute addressing mode. Absolute references are translated dynamically to absolute addresses during code generation.

Different semantics: some instructions change their semantics (e.g., `push`, or `pop`) and operate on quadwords on x64. These instructions are translated to operate on doublewords during the translation.

Rep prefix: the handling of the rep prefix changes for x64. String operations (e.g., `rep stosb`) that use the rep prefix are translated to loops during the translation process.

On x64 segment-based addressing is restricted compared to x86. Current user-space x86 applications use segmenting only for thread local storage in current applications. The x64 ISA supports segmentation for thread local storage and the prototype implementation support 32-bit thread local storage in a 64-bit environment.

5.2 Shadow memory

The 64-bit address space enables the implementation to use address regions that cannot be encoded using 32-bit memory pointers. The application uses the low 4GB of the 64-bit address space and no data in that region is changed through the binary translator (the data may be changed as a function of the memlets). Figure 3 shows the memory layout of a running application under the control of memTrace.

The next 4GB are used as shadow memory of the application memory at offset `0x100000000`. The memlets store information about the corresponding memory addresses of the application in the shadow memory. The shadow memory regions are mapped at the same time when the application memory is mapped. Virtual memory allocates physical pages only if the page is accessed by a memlet (e.g., code regions are not accessed by our memlets and the physical pages are therefore not allocated). An upper bound for the memory consumption for the shadow memory is 1x the

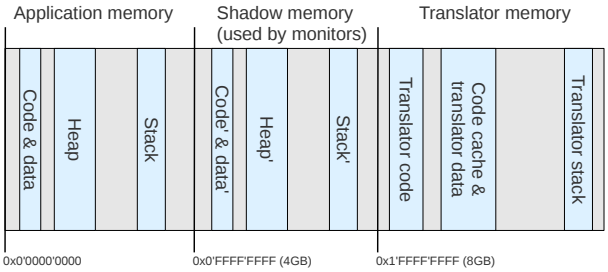


Figure 3: Memory layout of the translated application.

memory that the application uses. Memlets may use multiple shadow memory regions to store additional information (i.e., at offsets `0x200000000`, `0x300000000`, and so on).

5.3 Register allocation

The x64 ISA offers 8 additional general registers (`%r8` to `%r15`) that can be used by the lightweight memory tracing technique. The binary translator component is compiled for the x64 ISA and uses all available registers during the translation of x86 code. The transition between translator and translated application code saves and restores all general purpose registers. The memlets are native x64 code and can use the 8 additional x64 registers during the execution of the translated code.

Both memlets discussed here use registers `%r8` and `%r9` as temporary scratch registers. The memory watchpoint memlet uses three registers `%r10`, `%r11`, and `%r13` to track usage of the `eflags` register. Saving and restoring the `eflags` register before and after the execution of a memlet adds overhead, therefore reducing the number of save and restore operations is important.

The register `%r12` holds the constant `0x0` and the register `%r15` holds the constant offset to the shadow memory (`0x100000000`). Using registers to hold constants instead of encoding constants in the instruction itself saves 8 bytes per used constant in the emitted code. If needed, these registers may be used for other purposes.

5.4 String instructions

String instructions use the `rep` prefix to repeat a single instruction n times. String instructions access multiple memory locations in a sequence of incrementing or decrementing addresses.

MemTrace replaces string instructions with a short loop that first checks the source address and the destination address, executes a single instruction with the current parameters, and increments or decrements the source and target registers.

5.5 System calls, signals, and threads

An x86 application requests system calls either using interrupts (`int $0x80`) or using the `sysenter` instruction. On x64 the application uses the `syscall` instruction. The mapping between system call and system call number is different between x86 and x64. The parameters of individual system calls can change as well (i.e., 64-bit wide addresses instead of 32-bit wide addresses).

MemTrace uses a mapping table to map between x86 and x64 system calls. Most system calls can be mapped easily. For system calls that access memory, pointers are dynamically extended to 64-bit and returned pointers from the kernel are truncated to 32-bit. All memory management system calls (`mmap`, `munmap`, `mremap`, and `brk`) are redirected to a special handler function that checks and adapts the specified parameters and manages the shadow memory as well.

Signals are handled in the binary translator as well. The binary translator intercepts all system calls that install signals and replaces the signal handlers with its own internal signal handler. This signal handler then handles the switch to the application stack if the signal was caught while in the translator and executes the corresponding translated application signal handler.

The cross-ISA binary translator supports Linux pthreads by translating thread-related system calls of the application into the necessary 64-bit system calls. Thread support is a difficult problem for memory tracing due to possible synchronization issues. Two threads may concurrently modify the same memory address and the corresponding memlets may therefore access the same shadow value. As long as the application synchronizes access to the memory location the access to the shadow value is implicitly synchronized as well. If the original program has a data-race then the memlets must synchronize concurrent writes, e.g., by using regional locks. Simply adding a `lock` prefix to the original memory access is not enough as the memlet will access a second memory location. Adding explicit locks for each memory access adds high overhead and is currently not implemented.

The accesses of the memlets to the shadow table follow the same pattern as the memory accesses in the original application. If the application locks the memory region for a specific thread then the corresponding shadow memory region is implicitly locked as well. No other application thread can access the original memory region, therefore no memlet of another thread will access the implicitly locked shadow memory region. This implicit locking approach only works if the application has no data races between threads. User-defined memlets that analyze inter-thread behavior, e.g., to check for data races, must lock the shadow memory themselves.

5.6 Flag tracking optimizations

This section presents two optimizations for the binary translator that help lowering the overhead for memory tracing. The first optimization tracks the usage of the `eflags` register and allows the memlets to change the `eflags` register if the `eflags` register is not used between instructions that affect the flags. The second optimization stores the operands of the relevant arithmetic instruction in two free registers and reexecutes the instruction with a bogus target.

A big advantage of the cross-ISA translation is that

```
addl %ebx, %eax # sets flags, unused
subl %ecx, %eax # sets flags, used
movl (0xdeadbeef), %ebx # mem. read
movl %eax, (0xdeadbeef) # mem. write
jnz next_block  # uses flag
```

Listing 1: Example of a basic block in an application

memTrace can use additional free registers without the need for register reallocation. Unfortunately one register is shared between the binary translator, the memlets, and the translated application code: the `eflags` register. The situation is worsened by the fact that access to this register is very slow when using `pushf` and `popf` instructions. Instead of using `pushf` and `popf` instructions we use a short sequence of instructions (`lahf` and `seto` to save and `addb` and `sahf` to restore) to handle the `eflags` register.

On x86 all arithmetic instructions, the "compare" instruction, and the "test" instructions set the `eflags` register. But the `eflags` register is only read after a subset of the instructions. Table-based binary translators do not build an intermediate representation (IR) which makes `eflags`-usage tracking more complicated. MemTrace uses a triple-pass approach to track usage of the `eflags` register in each basic block. The first two passes decode all instructions and analyze which instructions use the `eflags` register. The third pass emits translated instructions (including the memlets which change the application-set `eflags` register) but inserts code that saves the `eflags` register only when necessary.

The second optimization saves the operands of the `eflags`-relevant arithmetic instruction in two x64 registers (`%r10`, and `%r11`). In front of the instruction that reads the `eflags` register the arithmetic instruction is executed again (with the saved operands) to reproduce the state of the `eflags` register. This optimization reduces the overhead of saving and restoring the `eflags` register in tight loops.

Listing 1 shows an example of a basic block. The first two instructions set the `eflags` registers, but only the result of the `subl` instruction is used. The two `movl` instructions execute memory accesses and the memlets in the instrumented code overwrite the status of the `subl` instruction. MemTrace restores the status of the `eflags` register of the last arithmetic instruction before the `jnz` instruction.

6 Evaluation

The prototype implementation is stable and runs applications like, e.g., the parsec benchmarks, OpenOffice, gedit, and the complete set of SPEC CPU2006 benchmarks. The evaluation uses the SPEC CPU2006 benchmarks to evaluate the performance of the memTrace prototype implementation, including two different user-defined memlets.

This evaluation uses all SPEC CPU2006 benchmarks except 481.wrf which no longer compiles on modern systems. This is not a limitation of our prototype implementation but a limitation of the SPEC CPU2006 benchmarks.

All benchmarks are executed on a 64-bit version of Ubuntu 12.04. The machine uses an Intel Core i7-2640M CPU with 2 cores at 2.80 GHz with 4 GB of memory. The benchmarks are compiled using gcc version 4.6.3 and use the glibc version 2.15. The benchmarks are compiled for 32-bit.

6.1 SPEC CPU2006

This section evaluates the performance of memTrace, our prototype implementation, using the SPEC CPU2006 version 1.0.1 benchmarks using the flags `-O3 -m32`. We evaluate different configurations of memTrace to show the overall overhead and relative performance changes for individual optimizations. The evaluations use the `runspec` script to produce reproducible runs with 3 iterations.

We perform the measurements on both the *reference* dataset and on the *test* dataset. The test dataset is used to evaluate the overhead for short running programs while the reference dataset shows the overhead for long running benchmarks. The following configurations are used:

NAT: A native configuration that runs without binary translation or memory tracing.

ID: The benchmarks execute with binary translation.

EFL: This configuration measures the overhead for storing and restoring the `eflags` register for memory tracing. Code that saves and restores the `eflags` register is added before as if memory tracing is executed but no memlets are added. All optimizations discussed in Section 5.6 are enabled.

MT: This configuration shows the performance of the baseline memory tracing framework. MT extends the EFL extension and measures the impact of reading the shadow memory address for each memory access.

WP: This configuration executes full memory tracing using the watchpoint memlets (without any active watchpoints).

Table 1 shows the overhead of the four different memTrace configurations compared to native execution of the 32-bit binaries. Most benchmarks exhibit moderate overhead for the different memTrace configurations. The overhead is always below 3.11x and for 16 of 28 applications the overhead is below 2x.

The ID configuration measures the overhead for cross-ISA translation. The overhead for cross-ISA binary translation is low, 15% on average with a geometric mean of 17%. The usual culprits 400.perlbench, 403.gcc, 445.gobmk, 458.sjeng, and 453.povray result in an overhead of more than 40% for binary translation due to the high number of indirect control flow transfers. The ID configuration shows that the binary translator is a reasonable baseline to implement memory tracing.

Benchmark	NAT [s]	ID	EFL	MT	WP
400.perlbench	324.00	1.74	2.10	2.60	2.82
401.bzip2	498.00	1.08	1.28	1.91	1.97
403.gcc	305.00	1.40	1.65	2.08	2.20
429.mcf	278.00	1.10	1.10	2.00	2.21
445.gobmp	434.00	1.49	1.85	2.26	2.74
456.hmmer	433.00	1.01	1.43	2.63	2.63
458.sjeng	485.00	1.56	1.99	2.35	2.72
462.libquantum	543.00	1.01	1.03	1.23	1.24
464.h264ref	609.00	1.22	1.42	2.71	2.86
471.omnetpp	308.00	1.37	1.44	1.89	1.94
473.astar	412.00	1.09	1.25	1.60	1.62
483.xalancbmk	252.00	1.90	2.23	2.68	2.99
410.bwaves	405.00	1.01	1.20	1.96	1.97
416.gamess	760.00	1.09	1.57	2.30	2.43
433.milc	441.00	1.00	1.06	1.32	1.34
434.zeusmp	540.00	1.02	1.22	1.69	1.72
435.gromacs	658.00	1.01	1.15	1.43	1.49
436.cactusADM	1120.00	0.99	1.24	2.21	3.11
437.leslie3d	447.00	1.02	1.11	1.44	1.44
444.namd	412.00	1.02	1.23	1.61	1.63
447.dealII	318.00	1.35	1.56	2.08	2.13
450.soplex	298.00	1.07	1.23	1.46	1.51
453.povray	181.00	1.49	2.05	2.62	2.78
454.calculix	696.00	1.03	1.21	1.55	1.59
459.GemsFDTD	503.00	1.04	1.14	1.61	1.66
465.tonto	559.00	1.15	1.35	1.71	1.81
470.lbm	440.00	1.00	1.02	1.13	1.14
482.sphinx3	521.00	1.05	1.23	1.59	1.61
Average	470.71	1.15	1.36	1.90	2.06
Geo. mean	439.54	1.17	1.37	1.86	1.97

Table 1: Performance evaluation using the SPEC CPU 2006 benchmarks (*reference* dataset). NAT shows native execution in seconds, the remaining columns show memTrace configurations relative to NAT.

Benchmark	NAT [s]	ID	EFL	MT	WP	VAL	VMEM
400.perlbench	3.57	1.67	1.75	1.88	1.95	6.53	err
401.bzip2	5.79	1.10	1.30	2.09	2.16	4.40	16.29
403.gcc	1.00	2.01	2.27	2.74	3.07	11.42	err
429.mcf	1.69	1.15	1.19	2.17	2.27	3.44	10.00
445.gobmk	15.00	1.49	1.85	2.26	2.71	8.07	31.20
456.hmmer	2.51	1.10	1.49	2.37	2.39	5.66	28.53
458.sjeng	3.39	1.53	1.87	2.25	2.58	7.73	31.27
462.libquantum	0.05	1.40	1.56	2.01	2.01	8.30	17.35
464.h264ref	12.30	1.24	1.59	2.60	2.72	4.73	34.47
471.omnetpp	0.31	3.66	4.33	4.90	5.92	18.15	73.25
473.astar	7.93	1.08	1.23	1.54	1.59	2.86	11.92
483.xalancbmk	0.08	4.36	4.52	5.39	5.79	22.93	65.79
410.bwaves	5.13	1.02	1.21	2.03	2.05	5.85	61.79
416.gamess	0.33	1.47	1.79	2.40	2.58	11.23	43.69
433.milc	5.06	1.12	1.34	1.77	1.77	6.52	31.03
434.zeusmp	13.80	1.01	1.24	1.64	1.64	err	err
435.gromacs	1.37	1.11	1.25	1.52	1.60	5.64	20.44
436.cactusADM	2.47	1.02	1.44	3.00	4.66	6.15	err
437.leslie3d	11.50	1.02	1.13	1.48	1.48	4.83	12.00
444.namd	11.10	1.05	1.26	1.63	1.65	7.09	23.78
447.dealII	13.20	1.42	1.58	2.20	2.24	err	err
450.soplex	0.03	2.57	2.79	3.03	3.19	err	err
453.povray	0.50	1.62	2.11	2.68	2.88	11.15	52.52
454.calculix	0.05	2.38	2.75	3.24	3.22	16.57	44.01
459.GemsFDTD	2.10	1.32	1.45	2.00	2.07	5.62	17.14
465.tonto	0.80	1.43	1.71	2.10	2.25	8.39	31.54
470.lbm	3.27	1.00	1.02	1.10	1.11	3.24	11.77
482.sphinx3	1.45	1.30	1.49	1.84	2.00	8.90	34.69
Average	4.49	1.22	1.45	1.98	2.12	5.24	24.30
Geo. mean	1.68	1.43	1.67	2.21	2.36	7.13	26.39

Table 2: Performance evaluation using the SPEC CPU2006 benchmarks (*test* dataset). NAT shows native execution in seconds, the next four columns compare different memTrace configurations to NAT. The last two columns compare Valgrind nullgrind (VAL) and memcheck (VMEM) to NAT.

The EFL configuration measures the performance overhead induced by `eflags` tracking, saving, and restoring needed if additional code is executed for every memory-accessing instruction. No memlet code is executed for this configuration. The average performance overhead for this configuration is 36% and the geometric mean is 37%. Different benchmarks show different increase in the performance overhead. These differences hint at the number of memory-accessing instructions that are executed for each benchmark. If the overhead increases over-proportional, then the benchmark executes more memory accessing instructions than the average benchmark.

The MT configuration extends the EFL configuration by reading the shadow value for each accessed memory location. No additional computation is executed. The performance difference between EFL shows the impact of one additional `mov` instruction per memory-accessing instruction combined with additional cache pressure for accessing twice as many memory locations in hot code regions. Several benchmarks exhibit a performance impact of 2.0x to 2.7x for this configuration. The average overhead is 1.90x with a geometric mean of 1.86x. This configuration shows the overhead for memory tracing without executing any memlets.

The WP configuration extends the MT configuration with the memlet for unlimited watchpoints. No watchpoints are set in this configuration, but the difference in execution time between configurations with set watchpoints and configurations without set watchpoints is negligible if no watchpoints are taken. If watchpoints are taken then the execution time of the watchpoint handlers must be added to the overhead as well. We measure the highest performance impact for the cactusADM benchmark with 3.11x performance impact compared to native execution due to the high frequency of memory accesses. The average overhead is 2.06x and the geometric mean is 1.97x. These two values show that the additional overhead

for the user-defined memlet is low compared to the execution overhead of the baseline memory tracing framework.

The performance analysis shows that the overall overhead for the prototype of the lightweight memory tracing framework using cross-ISA translation (the MT configuration) is below 2.0x and that the additional performance impact for the execution of user-definable memlets is low (the WP configuration). Lightweight memory tracing is a technique that can be used in practice to trace every single memory access of an application using user-definable memlets with tolerable execution overhead.

6.2 Comparison to other systems

This section evaluates the prototype implementation of the memTrace technique with other, similar software products that are capable of memory tracing. We tried running the minemu 0.8 open-source version on a 64-bit Ubuntu 12.04 system. Unfortunately the current version of minemu crashes during the initialization of thread local storage of the SPEC CPU2006 benchmarks when running 32-bit x86 binaries on an x64 system.

6.2.1 Valgrind

We evaluate valgrind [18] version 3.7.0-0ubuntu3 as the second system in two configurations: nullgrind (VAL) to evaluate the Valgrind overhead and (VMEM) to evaluate Valgrind's memcheck overhead. Table 2 shows the timings of the SPEC CPU2006 benchmarks and compare different configurations against the native execution of the *test* dataset. The test dataset uses shorter input files and simpler problems. This comparison uses only the *test* dataset due to the higher translation overhead of Valgrind. The 434.zeusmp, 447.dealII, and 450.soplex benchmarks did not complete under Valgrind's nullgrind configuration and the 400.perlbench, 403.gcc, 434.zeusmp, 436.cactusADM, 447.dealII, and 450.soplex benchmarks did not complete under Valgrind's memcheck configuration. MemTrace uses the same configurations as in Section 6.1.

The evaluation for memTrace shows a similar picture like the performance analysis of the *ref* dataset. In general the overhead increases due to the fact that translated code in the code cache is reused less often. The geometric mean for the MT configuration is 2.21x and the average overhead is 1.98x (compared to a geometric mean of 1.86x and an average of 1.90 for the *ref* dataset).

Valgrind on the other hand exhibits an average overhead of 5.24x and a geometric mean of 7.13x for the *test* dataset in the nullgrind configuration. The nullgrind configuration is comparable to the ID configuration of memTrace and does not execute any memlets or other user-defined code. The memcheck configuration of Valgrind results in an average overhead of 24.3x and a geometric mean of 26.4x. The memcheck configuration is comparable to memTrace's WP configuration.

	1 WP [s]	10 WP [s]	100 WP [s]
GDB SW WP	180	330	1670
memTrace	0.01	0.01	0.01

Table 3: Evaluation of the microbenchmark in with the first watchpoint at the 1,000 element.

6.2.2 GDB

We use a CPU-bound microbenchmark to evaluate the performance of the watchpoint memlet compared to GDB. The microbenchmark sets W consecutive watchpoints in a large array and processes the array in multiple passes, where the n^{th} pass accesses the first n elements of the array. In each pass, the elements are accessed using several patterns: a forward linear sweep, a convolution, and a sparse backward sweep. The microbenchmark measures the time until the first watchpoint is hit and handled by the debugger.

To compare memTrace performance with hardware watchpoint performance we configure the microbenchmark with the first watchpoint at the $500,000$ array element (i.e., memTrace needs to execute a large amount of memlets that do not trigger a watchpoint). With one active watchpoint the hardware watchpoint configuration executes in 52.8 seconds while the memTrace implementation uses 80.5 seconds, resulting in 52% overhead compared to the hardware implementation. While hardware watchpoints support only up to 4 simultaneous watchpoints memTrace supports unlimited watchpoints at a constant overhead (10^4 watchpoints in 80.5 seconds and 10^8 watchpoints in 81.5 seconds). Even at 10^8 watchpoints the performance of memTrace remains stable.

Table 3 compares memTrace performance with the performance of GDB software watchpoints with the first watchpoint at the $1,000$ array element. Even for 1 GDB software watchpoint memTrace is 18,000x faster than software watchpoints. For 100 GDB software watchpoints memTrace is 167,000x faster. The prototype implementation of the memTrace watchpoint memlet fully supports the remote serial protocol of GDB and works as a fast drop-in replacement for the GDB software watchpoints.

6.3 Memory overhead

Table 4 presents an analysis of the memory overhead for the SPEC CPU2006 benchmarks when run natively and under the control of the memTrace prototype implementation. The table shows the peak amount of mapped memory of the benchmark. This benchmark measures the number of mapped memory pages, not the number of allocated memory pages. The allocated memory pages are a subset of the mapped memory pages.

The memory overhead for binary translation only is low with an average of 9.2 MB and a maximum of 12.7 MB. Binary translation only needs few data structures (8 MB for the mapping table plus data structures for the code cache, signal handlers, and trampolines). These numbers show that

Benchmark	NAT [MB]	ID [MB]	Ovhd.	WP [MB]	Ovhd.
400.perlbench	565.24	574.92	1.7%	1142.16	102.1%
401.bzip2	628.27	636.76	1.4%	1265.34	101.4%
403.gcc	84.05	96.74	15.1%	186.92	122.4%
429.mcf	856.22	864.64	1.0%	1721.12	101.0%
445.gobmk	38.66	48.66	25.9%	89.76	132.2%
456.hmmer	21.07	29.61	40.5%	51.11	142.6%
458.sjeng	192.09	200.64	4.4%	393.17	104.7%
462.libquantum	114.02	122.44	7.4%	236.78	107.7%
464.h264ref	81.00	89.97	11.1%	172.15	112.5%
471.omnetpp	119.57	129.02	7.9%	250.28	109.3%
473.astar	140.52	149.00	6.0%	289.96	106.3%
483.xalancbmk	329.77	340.42	3.2%	673.82	104.3%
410.bwaves	891.57	900.11	1.0%	1792.19	101.0%
416.gamess	655.31	665.31	1.5%	1323.43	102.0%
433.milc	687.87	696.41	1.2%	1384.78	101.3%
434.zeusmp	1136.98	1146.00	0.8%	2284.24	100.9%
435.gromacs	34.60	43.39	25.4%	78.87	127.9%
436.cactusADM	1017.70	1026.80	0.9%	2045.81	101.0%
437.leslie3d	141.02	149.68	6.1%	291.39	106.6%
444.namd	63.85	72.52	13.6%	137.12	114.7%
447.dealII	501.38	510.89	1.9%	1014.33	102.3%
450.soplex	509.26	518.04	1.7%	1028.18	101.9%
453.povray	21.84	31.10	42.4%	54.44	149.3%
454.calculix	179.23	188.73	5.3%	369.89	106.4%
459.GemsFDTD	845.69	855.01	1.1%	1702.32	101.3%
465.tonto	53.91	64.46	19.6%	122.37	127.0%
470.lbm	426.93	435.35	2.0%	862.53	102.0%
482.sphinx3	57.81	66.59	15.2%	125.16	116.5%
Average	371.27	380.47	2.5%	753.20	102.9%
Geo. mean	201.55	219.56	8.9%	413.58	105.2%

Table 4: Memory consumption in megabytes of the SPEC CPU2006 benchmarks (NAT) and additional memory consumption of different configurations of the memTrace prototype implementation. ID represents a binary translation only configuration.

the memory overhead for cross-ISA binary translation is low.

The last column of Table 4 shows the memory overhead of the WP configuration. The amount of mapped memory is roughly doubled due to the shadow memory region.

7 Related work

There are several areas of related work that are relevant for lightweight memory tracing. Binary translation is needed to dynamically weave the memlets into the executed application code. The following sections discuss different systems for binary translation and different systems that implement some forms of memory tracing.

7.1 Binary translation

Binary translation enables late code modification to, e.g., instrument a binary application, to offer late code optimization, or to execute an application on a different ISA than it was originally compiled for.

Full-ISA emulation is too slow for real-world scenarios and mostly used for evaluation of new hardware features. Efficient binary translation is implemented using either table-based approaches or IR-based approaches.

Same-ISA binary translation translates an application to the same ISA (x86 to x86). A drawback of same-ISA translation is the register pressure on x86. Only 8 general purpose registers are available for x86 applications and only 6 or 7 registers are available for general computation (depending on the calling conventions). Memlets used for memory tracing need to execute additional computation for each memory access, starving the register allocator even further.

IR-based binary translators translate the application by using a traditional compiler approach. The binary translator transforms code into an IR, adds the desired instrumentation, and generates machine code for the desired platform. Translation is either dynamic like in a just-in-time compiler or static ahead-of-time. DynamoRIO [5], PIN [15], QEMU [3], and Valgrind [18] are dynamic IR-based binary translators. The IR-based approach enables compiler optimization to produce high-quality code at some translation cost.

Dynamic table-based binary translators (e.g., HDTrans [25], fastBT/libdetox [20, 21], or StarDBT [28]) use translation tables to decode original instructions and to generate translated instructions. The advantage is the low-overhead translation speed combined with reasonable code quality.

StarDBT [28] and QEMU [3] are two binary translation systems that support cross-ISA translation. StarDBT translates x86 code to x64 code and QEMU translates (almost) any ISA to (almost) any other ISA.

MemTrace is a cross-ISA table-based dynamic binary translator that translates user-space applications from x86 to x64. The binary translator component offers near-native performance. The StarDBT binary translator is similar to our binary translator but uses two compilation stages (baseline and optimized) while memTrace uses only one fast table-based translation scheme. In addition, memTrace allows the definition of user-defined memlets that may use fixed registers to speed up memlet execution.

7.2 Memory tracing and watchpoints

Memory tracing allows the execution of memlets for each memory access. A baseline memory tracing infrastructure is needed to implement higher-level memlets like watchpoints, or taint checking.

Greathouse et al. [12] present a case for unlimited watchpoints and light-weight, hardware-assisted memory tracing. They reason that additional hardware is needed to achieve low overhead for unlimited watchpoints. MemTrace shows that cross-ISA translation realizes low-overhead memory tracing (and watchpoints) for x86 applications when executed on modern processors that support x64 extensions.

Metric [16] is a memory tracing framework that collects and stores selected memory access traces. Memcheck [17],

System	Arch.	Underlying BT	Shadow memory	Overhead
memTrace	x86 to x64	libdetox	1 byte per byte	2.0x for SPEC CPU2006
Libdft [14]	x86 to x86	PIN	flexible	1.14x to 10x slowdown SPEC CPU2000
Minemu [4]	x86 to x86	dynamic BT, no SSE[a]	1 byte per byte	2.4x for SPEC INT2006
PTT [9]	x86 to x86	QEMU	32-bit vector per byte	21.1x for compression benchmark
Saxena et al. [24]	x86 to x86	static BT	1 bit per byte	1.9x (stack only) to 2.8x (SPEC INT95 subset)
Panorama [30]	x86 to x86	QEMU	4 byte pointer per byte	20x on selected benchmarks
Dytan [7]	x86 to x86	static BT	1 bit vector per byte	30x to 50x for gzip
LIFT [23]	x86 to x64	StarDBT	1 bit per byte	1.7-7.9x, 3.6x for SPEC INT2000
Argos [22]	x86 to x86	QEMU	1 bit per byte (phys. mem)	"at least 16x overhead"
Xentaint [13]	x86 to x86	Xen and QEMU	1 bit per byte	61.5x to 88.4x for micro-benchmarks
Vigilante [8]	x86 to x86	static BT on start-up	1 bit per 4k page	no numbers on performance overhead reported
Taintcheck [19]	x86 to x86	Valgrind	4 byte pointer per byte	1.5x to 40x
Suh et al. [26]	Alpha	HW extension	1 bit per page/quad word/byte	1.44% for SPEC CPU2000

[a]Minemu internally uses the SSE registers and cannot support any SSE instructions in applications. Modern compilers use SSE instructions to speed up memory transfers, for vectorization, and for floating point computation.

Table 5: Comparison of different taint checking and dataflow analysis systems.

Umbra [31], EDDI [32], and Dr. Memory [6] are four frameworks for memory tracing that use same-ISA binary translation to add hard-coded memlets for watchpoints. Memcheck builds on Valgrind and reports an overhead of 22.2x for the SPEC CPU2000 benchmarks. Umbra, EDDI, and Dr. Memory build on DynamoRIO. Umbra reports an overhead of 2.33x for SPEC CPU2006 for memory tracing of an x64 application; an example tool that extends Umbra with a memlet that monitors thread's memory accesses imposes a 6.49x overhead for a set of benchmarks. EDDI reports an overhead of 2.59x for 0 watchpoints and 3.68x for watching the complete data region on the SPEC CPU2000 benchmarks in the FI configuration. The PI configuration of EDDI only reports on a subset of the SPEC CPU2000 benchmarks. Dr. Memory reports a slowdown of 10.2x for the SPEC CPU2006 benchmarks. Umbra implements memory tracing without additional memlets; memcheck, EDDI, and Dr. Memory add hard-coded instructions into the executed application code to check memory accesses for validity.

MemTrace improves on related work by offering user-definable memlets that implement high-level memory checkers and offers better performance than previous solutions: memTrace reports an average overhead of 2.06x and a geometric mean of 1.97x for tracing all memory accesses of all SPEC CPU2006 benchmarks.

7.3 Taint checking and dataflow analysis

Taint checking and data flow analysis extend memory tracing and analyse the flow of data inside an application. Every memory cell and every register has an associated tag. Taint checking uses a single taint bit per address while dataflow analysis supports multiple different tags. Compared to single-threaded approaches of other related work memTrace fully supports memlets for concurrent threads.

Some of the systems in the following list use taint checking or dataflow analysis as a technique in their system. Table 5 focuses on the taint checking or dataflow analysis component of the presented systems.

MemTrace does not change the address space layout of the original application, all data of the memlets is stored at a higher location in the 64-bit memory space. This design decision solves the problem of accesses to the shadow memory by the application. For the shadow memory itself memTrace uses 1 byte per byte, enabling threads to update the (shared) shadow memory data structure concurrently without locking. Only if memlets rely on bit-granularity then the programmer must add a locking scheme to ensure correctness.

8 Conclusion

This paper presents memTrace, a technique for dynamic lightweight memory tracing for unmodified binary applications. This technique adds shadow memory and state for each memory address of an application and allows the execution of user-defined memlets to inspect memory accesses.

The practical value of memTrace is demonstrated by the implementation of two memlets: a memory checking memlet that allows the debugging of memory errors and a memlet that allows an unlimited number of watchpoints in a running application. We evaluated the prototype implementation and show that the overhead for SPEC CPU2006 is low with a geometric mean of 1.97x and an average of 2.05x.

The open source release of the memTrace prototype is available at `http://nebelwelt.net/projects/memTrace` and can be used to implement other memlets, e.g., for taint checking, dataflow analysis, or control flow integrity checks.

Acknowledgments

We thank the anonymous reviewers for their comments, Albert Noll for his comments on an early draft of this paper, and Jonas Pfefferle and Tobias Hartmann for working on a same-ISA version of a simple memory tracing infrastructure.

References

[1] ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., AND STICHNOTH, J. M. Fast, effective code generation in a just-in-time java compiler. In *PLDI'98* (1998), pp. 280–290.

[2] ALPERN, B., BUTRICO, M. A., COCCHI, A., DOLBY, J., FINK, S. J., GROVE, D., AND NGO, T. Experiences porting the jikes rvm to linux/ia32. In *Java Virtual Machine Research and Technology Symposium* (2002), pp. 51–64.

[3] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC* (2005), pp. 41–41.

[4] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: the world's fastest taint tracker. In *RAID'11: Proc. 14th conf. on Recent Advances in Intrusion Detection* (2011), pp. 1–20.

[5] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (2003), pp. 265–275.

[6] BRUENING, D., AND ZHAO, Q. Practical memory checking with dr. memory. In *CGO'11* (2011), pp. 213–223.

[7] CLAUSE, J. A., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Intl. Symp. on Software Testing and Analysis* (2007), pp. 196–206.

[8] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A. I. T., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *SOSP'05* (2005), vol. 39, pp. 133–147.

[9] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L. L., AND MCCAULEY, M. Towards practical taint tracking. Tech. Rep. UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010.

[10] GDB. GDB remote serial protocol. `http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html`, 2010.

[11] GLOGER, W. Dynamic memory allocator implementations in linux system libraries. `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`, May 1997.

[12] GREATHOUSE, J. L., XIN, H., LUO, Y., AND AUSTIN, T. A case for unlimited watchpoints. In *ASPLOS'12* (2012), pp. 159–172.

[13] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys'06* (2006), pp. 29–41.

[14] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. libdft: practical dynamic data flow tracking for commodity systems. In *VEE'12* (2012), pp. 121–132.

[15] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05* (2005), pp. 190–200.

[16] MARATHE, J., MUELLER, F., MOHAN, T., MCKEE, S. A., DE SUPINSKI, B. R., AND YOO, A. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Trans. Program. Lang. Syst. 29*, 2 (Apr. 2007).

[17] NETHERCOTE, N., AND SEWARD, J. How to shadow every byte of memory used by a program. In *VEE'07* (2007), pp. 65–74.

[18] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07* (2007), pp. 89–100.

[19] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS'05* (2005).

[20] PAYER, M., AND GROSS, T. R. Generating low-overhead dynamic binary translators. In *Proc. 3rd Annual Haifa Experimental Systems Conf.* (2010), SYSTOR '10, ACM, pp. 22:1–22:14.

[21] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments* (2011), pp. 157–168.

[22] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys'06* (2006).

[23] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO'06* (2006), pp. 135–148.

[24] SAXENA, P., SEKAR, R., AND PURANIK, V. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *CGO'08* (2008), pp. 74–83.

[25] SRIDHAR, S., SHAPIRO, J. S., AND BUNGALE, P. P. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News 35*, 1 (2007), 135–140.

[26] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *ASPLOS'04* (2004), pp. 85–96.

[27] VMWARE. Software and hardware techniques for x86 virtualization. `http://www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf`, 2009.

[28] WANG, C., HU, S., KIM, H.-S., NAIR, S., BRETERNITZ, M., YING, Z., AND WU, Y. Stardbt: An efficient multi-platform dynamic binary translation system. In *Advances in Computer Systems Architecture*, vol. 4697. 2007, pp. 4–15.

[29] WIMMER, C., AND FRANZ, M. Linear scan register allocation on ssa form. In *CGO'10* (2010), pp. 170–179.

[30] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS'07* (2007), pp. 116–127.

[31] ZHAO, Q., BRUENING, D., AND AMARASINGHE, S. Umbra: efficient and scalable memory shadowing. In *CGO'10* (2010), pp. 22–31.

[32] ZHAO, Q., RABBAH, R., AMARASINGHE, S., RUDOLPH, L., AND WONG, W. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *CC'08*.

Flash Caching on the Storage Client

David A. Holland, Elaine Angelino, Gideon Wald, Margo I. Seltzer
Harvard University

Abstract

Flash memory has recently become popular as a caching medium. Most uses to date are on the storage server side. We investigate a different structure: flash as a cache on the client side of a networked storage environment. We use trace-driven simulation to explore the design space. We consider a wide range of configurations and policies to determine the potential client-side caches might offer and how best to arrange them.

Our results show that the flash cache writeback policy does not significantly affect performance. Write-through is sufficient; this greatly simplifies cache consistency handling. We also find that the chief benefit of the flash cache is its size, not its persistence. Cache persistence offers additional performance benefits at system restart at essentially no runtime cost. Finally, for some workloads a large flash cache allows using miniscule amounts of RAM for file caching (e.g., 256 KB) leaving more memory available for application use.

1 Introduction

Recently flash memory has become popular not only as a storage medium but also as a caching layer in high-end storage systems. The typical scenario has been to combine flash with disks, either locally or on a file server. We look at the opposite case: flash combined with the operating system's buffer cache, on the client side of a networked storage system.

We consider compute servers running storage-intensive workloads that are themselves clients in a networked storage environment. There are many examples of such servers: application servers in three-tier web applications, compute servers in data centers, render farms used in animation, and compute nodes in scientific computation clusters all fit this model. Our analysis explores a range of design issues arising from this configuration:

- Must the flash cache be managed together with the file system RAM cache or can it act as an independent layer below it?
- Should the RAM cache be a proper subset of the flash cache or should the two caches be treated as a single unified cache to avoid duplication?
- How large must the flash cache be relative to RAM?
- What writeback policies should be used from RAM to flash and from flash to the file server?
- Should a flash cache be persistent and recoverable?
- How critical is consistency across multiple caches?

This design space is already enormous, so we put aside other relevant but secondary considerations, such as cache replacement policy (we use LRU) and wear leveling algorithms. We assume our flash device comes equipped with a flash translation layer that handles wear leveling, erase cycles, and other considerations that arise if one uses raw flash chips directly.

We explore this design space via trace-driven simulation, which allows us to examine the behavior of an extensive range of configurations and cache sizes. We validated our simulator and traces against actual workloads, but use stochastically generated workloads for our analysis, because we could not find real-world traces with workloads large enough to stress the flash.

Our results show that all simple writeback policies, short of synchronously writing from RAM all the way through to the file server, produce comparable results. This means that flash caches can be write-through, which simplifies cache consistency handling. We also find the primary benefit of flash caching comes from its density. A volatile cache medium available for a reasonable price in similar sizes would also be attractive.

In the next section, we briefly discuss the various ways flash is being used to boost storage performance. We then outline the flash cache design space in Section 3. We describe our traces in Section 4 and our simulator in Section 5. We discuss how we validated our tools and models in Section 6 and then present the results of our simulation study in Section 7. Our conclusions are in Section 8.

2 Related Work

Flash is widely used in high end storage servers [2, 3] and more recently in hybrid drives that package flash and spinning media inside a single device [18, 20]. The NetApp FlashCache[17] is a device that transparently sits in front of a storage server, using the persistent cache to reduce latency. FlashTier [19] is a disk controller with an on-board persistent flash cache. It explores the possibilities of using a custom flash translation layer optimized for caching rather than storage. All of these solutions place flash on the storage side of a network (or local SATA), combining flash with disk drives. Our work examines flash on the client side, combining flash with the operating system buffer cache.

NetApp's Project Mercury [6] is a client-side flash cache that avoids explicit integration with the operating system. It is a block-level cache that can be deployed in various ways: a hypervisor filter driver, an OS filter driver, an application cache, or a proxy cache for network storage protocols. Mercury is one point in the design space this study explores. In Mercury, RAM stores a proper subset of the data stored in the flash cache, the writeback policy from RAM is the operating system's, and the writeback policy from flash is write-through.

Microsoft's ReadyBoost [15] is a software solution in recent Windows releases that uses a standard flash device as an extension to memory for random read caching. Windows gradually fills the flash cache with data and then services random reads from that cache, when doing so improves performance.

Recently, Koller et al. [11] experimented with a range of more sophisticated writeback policies for a flash cache. They found (as we did) that synchronous write-through all the way to disk is slow. Their work is otherwise complementary to ours as it explores write-back policies more sophisticated than those we considered. (They found, for example, that their policies can increase write throughput by improving the batching of back-end write requests; our simulator does not model this effect.) One key difference is that they were working in an environment where applications wait until writes propagate all the way to disk. We concentrate on a more conventional environment where writes return to the application once the data is written into the operating system's buffer cache. As we will see, this hides the write latency of the underlying storage tiers except under heavy write traffic. We also assume a high-performance filer with sophisticated read-ahead, nonvolatile cache, and large server memory at the back end, rather than a simple disk array.

3 Flash Cache Design Space

We model an application server environment consisting of one or more compute servers ("hosts") and a file server ("filer") connected by private network segments. Each host runs one or more applications, involving one or more threads of execution. Each host has cache space that is partially RAM and partially flash. As previously mentioned this environment reflects a number of real-life situations. We consider storage-intensive workloads.

We now address the design issues from Section 1.

3.1 Flash-RAM Integration

We begin by asking whether flash cache support should be integrated into the operating system's buffer manager or if it performs acceptably as an independent entity, as in Mercury. The former case requires substantial kernel modifications. The latter case allows deploying the flash cache in (or as) a self-contained device driver.

The need for integration depends on the level of coordination required between the RAM and flash caches. If accessing the flash via ordinary block reads and writes performs adequately, the flash cache can be independent. On the other hand, if special policies are required, or extra metadata must be provided to the flash cache, then kernel support is required.

3.2 Placement

Our second design question is whether the RAM cache can be a subset of the flash cache. This is effectively a choice of block placement policy. The straightforward approach is to structure the flash cache as an additional independent tier of cache below the RAM cache. The flash cache services the RAM cache and the file server services the flash. Newly referenced blocks are first placed in flash, then into RAM; the RAM cache is always a subset of the flash cache. This policy wastes some of the capacity of the flash, but is relatively simple.

Alternatively, one could use two separate layers of cache, but choose some more elaborate policy; for example, one might place blocks initially into RAM and

then migrate less recently (or less frequently) used blocks down to flash. Another option is to treat the two stores as a single unified cache and come up with some policy for initial placement and perhaps also internal migration.

The basic question is whether the simple approach is good enough. We would also like to estimate how much better (if at all) an alternate placement scheme performs.

3.3 Cache Architecture

We handle integration and placement as a single choice of cache architecture. Because the number of possible fill and migration policies is near infinite, we chose three simple alternatives to implement and test. Other options are certainly possible and may be a worthwhile subject of future research. These are the three architectures:

- **Naive.** The flash cache is treated as an independent cache layer beneath the RAM cache; the RAM cache is always a subset of the flash cache, requiring no integrated management.
- **Lookaside.** Based on Mercury [6], writes go directly from RAM to the file server instead of being routed through the flash. The flash is updated *after* the file server and never contains dirty data. Applications see persistence guarantees identical to a system without flash. The RAM cache is a subset of the flash cache, requiring no integrated management.
- **Unified.** RAM and flash are managed together using a single LRU chain. Data blocks are placed into the least recently used buffer, whether RAM or flash, and are never migrated. No attempt is made to prefer RAM to flash. Here the RAM cache is not a subset of the flash, so integrated management is needed.

3.4 Relative Size

What size does the flash cache need to be relative to the RAM cache to be effective? We use 8 GB as the baseline RAM size and examine flash sizes ranging from 8 GB to 128 GB (1x to 16x RAM). We use 64 GB as the baseline flash size based on the old rule of thumb that each successive layer of cache should be roughly an order of magnitude larger. (Note that the RAM size actually reflects the amount of RAM available for file system caching. For many real-life workloads this is substantially smaller than the total amount of RAM in the machine.)

3.5 Flash Writeback Policy

We next consider the question of when dirty blocks move from flash to the file server. We chose four policies:

- *write-through* - data is immediately written to the server, blocking the requester until completion.
- *asynchronous write-through* - data is immediately written to the server without blocking the requester.
- *periodic* - dirty data remains in the cache until a syncer thread flushes the data back to the server.
- *none* - dirty data remains in the cache until evicted for capacity reasons.

We run the periodic case with syncer periods of 1, 5, 15, and 30 seconds, resulting in seven different policies.

3.6 RAM Writeback Policy

We now consider RAM writeback policies. Since (at least for the *naive* architecture) these writebacks go to the flash cache, it does not necessarily follow that the standard behavior of file system RAM caches is correct.

We tested the same seven writeback policies that we used for flash writeback, yielding 49 different policy configurations for each of the three architectures.

We did not try other more elaborate policies (such as trickle-flushing, writing back asynchronously after a delay, etc.) for either flash or RAM, because we found that nearly all the policy combinations perform identically.

3.7 Cache Persistence

Volatile RAM caches are emptied by system restart and are typically left to refill naturally. However, a cache kept in persistent memory can potentially be recovered after a crash, to avoid the performance degradation that occurs when refilling the cache [12]. The Rio File Cache research prototype demonstrated the potential of such approaches as early as 1996 [7]. Today, the NetApp Mercury cache exploits persistence to avoid performance degradation after reboot [6], and high end file servers typically use battery-backed memory similarly to accomplish such warm restarts [1, 2]. With flash caches, cached data can survive a restart, but the system must take precautions to ensure that the data is valid.

Our results show that the price/performance of flash makes it attractive simply as a larger cache. However, taking advantage of its persistence can provide additional benefit. There are three chief obstacles: First, cache consistency needs to be maintained; this is discussed in the next section. Second, the cache indexing structures must themselves be kept in the flash and kept up to date and consistent with the data blocks in the flash. This creates additional flash traffic and additional overhead. A naive implementation adds an additional flash write latency every time the flash cache is updated; a clever implementa-

tion can batch those writes. Third, if the crash was caused by corruption in the flash itself, a simple reboot may not be sufficient to restore the system to a running state.

In the *lookaside* architecture blocks in the flash are never dirty, so the system cannot crash with dirty blocks that *must* be recovered and written back to the file server.

3.8 Cache Consistency

Normally one writes updated blocks in the RAM cache back to the file server quickly, because RAM is volatile. This motivation disappears with a persistent cache. If the flash cache is recoverable, as discussed in the previous section, cache writebacks can be delayed. Some writes will then die in the cache, reducing network contention.

However, for shared data, it also complicates cache consistency handling. Data not written back to the file server right away must still be reported back to the server so other hosts do not read stale versions. And, of course, unmodified data retained in the cache must also be tracked in case some other host updates it.

Cache consistency is not a new problem [9, 16, 21] and does not need a new solution; however, two new issues arise. The size of flash caches may affect the scalability of consistency protocols; detailed modeling of this effect is beyond the scope of our work. Furthermore, a recoverable cache is unavailable during a reboot; it cannot flush dirty data or participate in cache consistency protocols until afterwards. As reboots typically take at least minutes, this may induce unacceptable delays.

We concentrate primarily on non-shared data, e.g., disk images provided to clients over a SAN. We touch briefly on cache consistency only to quantify the magnitude of the problem. The simulator invalidates stale copies of blocks instantly (using global knowledge) when a new version is first written into a cache. This exposes the overhead caused when these blocks must be fetched again later. However, we only count invalidations; we do not model the overhead of cache consistency traffic, nor do we adopt any particular real-world cache consistency model. This information gives designers a basic overview of the circumstances that arise with the much larger caches that flash allows.

4 Traces

For our trace-driven simulation, we use block-level traces containing read and write operations. Each operation identifies a file and a range of blocks within that file. Each operation also carries a thread ID and host ID.

During development and validation, we used traces from the SNIA repository and the Mercury traces, but for our analysis we use synthetic traces. Adequately large real traces are, by and large, not available; when working with a 128GB flash device, we need a trace that churns through enough data to fill it and then work with it for long enough to access plenty of data that both is and is not in the original fill. The largest trace for which we present results moves roughly 2.5 TB of data, all told; we were unable to locate any real traces this large.

We wrote a trace generator to produce large traces with characteristics similar to real traces. The trace generator starts from a list of files and file sizes from the Impressions file system generator [4]. It samples this file server model to produce working sets, then samples these to produce I/O requests. A portion of the I/O requests are sampled instead from the whole file server. The distribution of I/Os among hosts and threads is uniform; the distribution of I/Os among files (and selection of files for working sets) is weighted by popularity, where small integer popularities are generated from a Zipfian distribution. The distribution of I/O sizes (and selection of file subregions for working sets) is Poisson, modified by clamping to the filesize. The distribution of I/O starting points (and file subregion starting points) is uniform.

All traces used in the results presented are based on the same 1.4 TB file server model we generated with Impressions. (This is larger than any of the cache sizes we use.) They use 4K blocks and have 80% of the I/Os coming from the working set. They also use eight threads per host. They grind through a total volume of data that is, in all cases, four times the working set size, half of it being devoted to a warmup period for which statistics are not collected. This ensures the cache fills thoroughly. (We checked the results of changing the working set percentage and the number of threads; these did not affect the conclusions about our key questions.)

The two traces we use as a baseline use one host, one working set, working set sizes of 60 and 80 GB (for use with a 64 GB flash), and 30% writes. For many of the experiments we vary one or more of these parameters.

5 Simulator

As discussed earlier, we model an environment where some number of computation servers ("hosts") share a single networked file server. We wrote a trace-driven simulator for this environment.

The simulator issues I/O requests from the trace as quickly as possible given that each application thread can have only one I/O in progress. I/O requests may stall at

various points in the system; all executions are fully interleaved. We do not try to produce realistic application-level I/O schedules; not only is scheduling I/O traces a known hard problem [10, 14, 22], but flash substantially changes the timing. Timestamps taken from environments without flash would have dubious value.

We model the caches in detail; each is a single LRU chain of blocks. We treat the flash itself as a block device; that is, we write blocks to it and read them back. We assume a flash translation layer but do not model it directly. We use average per-block access times derived from testing real flash devices. (See Sections 6.1 and 6.2.)

The network is modeled less exactly: each segment can carry one packet at a time, and each I/O request uses one packet in each direction. Each packet is assumed to incur a fixed latency (for headers, block information, and so forth) plus a small amount of additional time per bit of block data transferred.

We do not attempt to model the caches or prefetching behavior of the filer directly. Many man-years of effort have gone into providing high-end file servers with clever and aggressive caching logic, and modeling this is irrelevant to the main goals of this work. Instead we use a simple model: a "fast" latency for cache hits, a "slow" latency for misses, and a prefetch success rate that determines what fraction of reads are fast. (*Which* reads are fast is random. Writes are buffered and always fast.)

We do not model application overhead, user-kernel transitions, hypercall delays, processing latency in the nework stack, etc. Most of these are invariant under caching or can be incorporated elsewhere.

6 Validation

We validate two parts of our system that could produce fallacious results if not done properly. First, we validate our simulator against data using NetApp's Mercury flash cache. Second, we validate that average read/write latencies for our device reasonably approximate actual flash latencies.

6.1 Simulator Validation

We validated our simulator against NetApp's Mercury [6], a hardware implementation of a client-side flash cache. Working with the Mercury group, we took four days of traces from a NetApp Windows laptop and played them back both on their hardware and on our simulator. These traces were collected below the file system, i.e., under the buffer cache, so we played them back directly through a 32GB flash cache. (In our simulator, that means we set the RAM cache size to zero.)

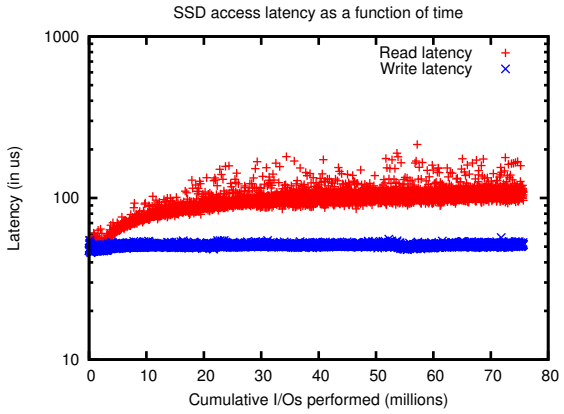


Figure 1: Flash device read (top) and write (bottom) latency; 60GB working set workload on a 58GB device. Each point is the average of 10,000 block I/Os.

We debugged the simulator and adjusted our timing models as necessary until the I/O throughput and latencies seen above and below the flash cache, as well as accessing the flash device, plus the cache hit rates, all or nearly all matched within 10%. Many of the statistics matched more closely. A perfect alignment is not possible, because (besides the inherent limitations of simulators) Mercury is not structured identically to the simulator. The simulator also does not account for an additional application-level or other systemic overhead of roughly 10% seen in the end-to-end run times.

These measurements gave us confidence that the simulator accurately models the system behavior and that its results are meaningful.

6.2 Flash Modeling Validation

We worried that average write latencies might not adequately model the behavior of a real device in the presence of flash erase cycles. We bought two low-end consumer grade SSDs and evaluated their latency behavior.

We modified the simulator to log I/Os to the flash as it ran and captured the results for a variety of workloads. Then we replayed these I/Os to the SSDs and recorded the actual read and write latencies. We also tried fully random reads and writes with a read/write mix similar to that found in the simulator logs.

We found three things of possible interest. First, while both devices exhibited high variance in their access latency, this variance is short-term; across a group of 10,000 to 100,000 block accesses (much less than the length of our traces) the variance is high, but from group

to group the average behavior is quite reasonable. Second, and perhaps of more interest, both devices maintained a single average write latency from beginning to end across essentially all the workloads. This included workloads with up to 90% (application) writes. Only the read latency fluctuated significantly over time as the device filled. We observed a weak relationship between higher write volumes and worse read performance; whether this is due to erase cycles or caching or some other internal phenomenon is anyone's guess.

Third, the read performance replaying the simulator logs is much better than the read performance doing purely random I/Os. Caching workloads are not random.

Figure 1 shows a scatter plot of the read and write latencies against time for a typical workload run. Each point is the average of 10,000 block I/Os.

Our conclusion was that a single average access latency is fine for modeling writes, and viable, though not ideal, for reads. However, our experience with flash devices is that each model is different, exhibiting its own average latencies and behavioral quirks. Fortunately the system performance does not appear to be highly sensitive to flash performance; see Section 7.7.

7 Results

We chose a per-block RAM access time of 400 ns, corresponding to roughly 10 GB/sec memory bandwidth. An internal limitation of the simulator restricts it to integer multiples of 100 ns, so this speed roughly reflects the 10-12 GB/s expected (and observed on an Intel Core i7 [13]) bandwidth of DDR3 RAM.

We used the performance data from validating against Mercury to choose timing models for the flash and the combined network and file server accesses. We then picked latencies loosely corresponding to a gigabit network for the network and attributed the rest of the combined network and file server times to the file server. Table 1 summarizes the timing parameters.

In evaluating possible configurations, we use the latency experienced by the application as the governing metric. Although the simulator captures a variety of other metrics (including throughput and latencies at every level of the stack), we use those only to explain behavior rather than to evaluate policies.

Parameter	Value
RAM read	400 ns / 4K block
RAM write	400 ns / 4K block
Flash read	88 μs / 4K block
Flash write	21 μs / 4K block
Network base latency	8.2 μs / packet
Network data latency	1 ns / bit
File server fast read	92 μs / 4K block
File server slow read	7952 μs / 4K block
File server write	92 μs / 4K block
File server fast read rate	90%

Table 1: Timing Model Parameters

7.1 Architecture and Writeback Policy

We begin our analysis by evaluating our *naive*, *lookaside*, and *unified* architectures and how they are affected by the 49 combinations (seven each for RAM and flash) of writeback policies. Identifying the promising configurations from among the 147 possibilities allows for a more focused comparison in the rest of the evaluation.

We used the two baseline traces described in Section 4. We ran these traces on the corresponding baseline simulator configuration: 8 GB of RAM and 64 GB of flash.

Figure 2 shows the average read and write latency seen by the application across all 49 policies for the three different architectures. We show the 80 GB workload; the 60 GB graphs are nearly identical.

Cursory inspection of the figures reveals the first important result: excepting policies that result in synchronous writes to the filer (synchronous or none) the writeback policy does not matter. The "none" policy leads to synchronous evictions once the cache fills. When the RAM policy allows this effect in the flash cache to show through to the application, as seen in the front left and right corners of the write latency graph, multiple threads doing evictions contend for the network, convoy, and slow down to (less than) the speed of the file server.

While this result initially surprised us, it is entirely reasonable: flash caches are so large that any reasonable writeback policy maintains an ample supply of clean blocks to evict and replace; the latency exposed above the flash cache is never greater than the flash write latency.

For the application to observe greater latency, it would have to sustain a write bandwidth greater than the writeback bandwidth to the file server for sufficiently long to fill many gigabytes of flash with dirty blocks. While workloads exhibiting this behavior probably exist, we expect them to be rare. Furthermore, upon filling the flash, write latency will largely revert to that of the file server. This produces the same effect as having no flash cache.

Based on this exploration, we use one policy combination for most of the remaining analysis: a one-second periodic RAM writeback policy (as this most closely matches real system behavior) and asynchronous write-

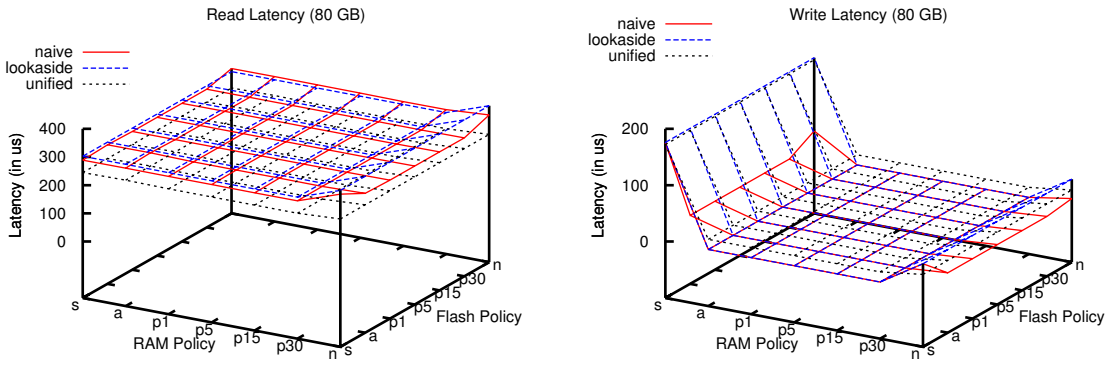


Figure 2: Application read and write latency on the 80 GB working set as a function of RAM and flash writeback policies.

through for the flash cache. Asynchronous write-through seems like the best overall choice for the flash, as it is equivalent to synchronous write-through for consistency and integrity purposes. Meanwhile it avoids exposing synchronous file server writes if the RAM cache becomes synchronous through dysfunction, e.g., thrashing.

Figure 2 also shows the *unified* architecture produces the lowest read latencies while the *naive* and *lookaside* architectures produce the lowest write latencies. The read latency results are unsurprising, because the effective capacity of the *unified* architecture is greater: it is the sum of the RAM and flash sizes (72 GB) instead of just the flash size (64 GB). When the working set fits in the flash (60 GB), the difference is tiny, only 3.5%. However, when the working set falls out of the flash (80 GB), we see that the larger effective cache size produces a significant benefit, improving read latency by as much as 20%. Figure 3 illustrates in more detail how the effective total cache size affects performance. For two of the cases in this graph we pretended that the flash has the same access latency as RAM. This allows distinguishing the structural effects from the latency properties of the cache materials. Although it is difficult to see in the graph, the performance of the RAM-only *unified* architecture with 8 and 56 GB caches is identical to that of the RAM-only *naive* architecture with 8 and 64 GB caches. The difference between that line and the one above it reflects the effect the slower flash has on read latency.

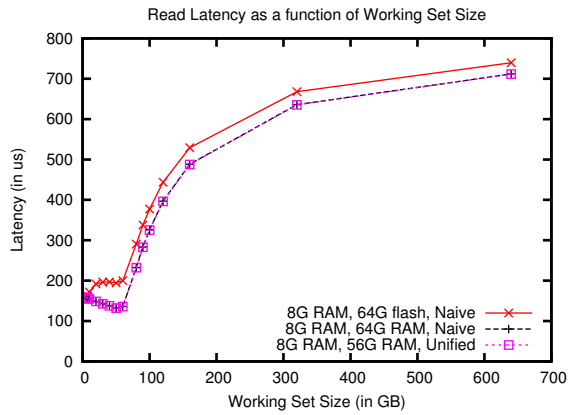


Figure 3: Application read latencies comparing effective cache sizes. See discussion in text.

Returning to the policy comparison in Figure 2, on the write side, the *naive* and *lookaside* architectures perform at RAM speed, because all writes go directly to RAM (except for very high write rates). The *unified* architecture also exposes flash latency by nature; since only $1/9$ of the data is placed in RAM and the rest in flash, on average we see $8/9$ of the 21 μs flash latency.

Stepping back, these results suggest that for read performance, bigger is better and that for write performance, the key is to avoid exposing applications to the flash timing. If we assume a given cost budget, an attractive strategy is to use only enough RAM to act as an effective write buffer and then buy as much flash as the budget allows. We explore this option in Section 7.5. Unless otherwise specified, we use the *naive* architecture in the remaining analyses, as it hides the flash write latency and offers the simplest implementation alternative.

7.2 Flash vs. No Flash

Having settled on policies, we now investigate the advantage the flash cache offers. To this end we ran a range of working set sizes, ranging from 5 GB to 640 GB, on three sizes of flash cache (32 GB, 64 GB, and 128 GB)

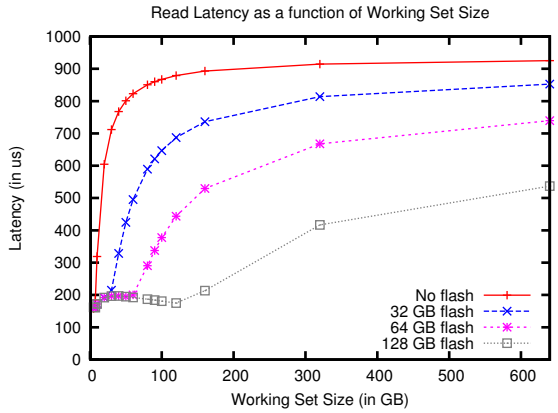


Figure 4: Read latencies as a function of working set size across a variety of flash sizes. As expected, when the working set fits in the flash, read latency improves dramatically over a RAM-only system.

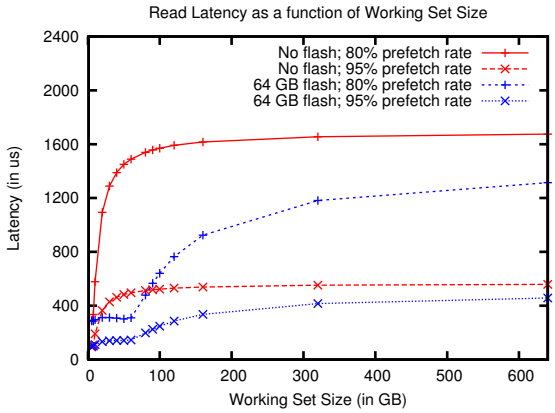


Figure 5: Application-level read latency for different workload sizes and two filer prefetch rates. Comparing the lines of similar shape demonstrates the dramatic effect that filer prefetching has on the resulting latency.

as well as with no flash cache. The RAM cache size is 8 GB. The working set sizes range from smaller than RAM to substantially larger than the largest flash cache.

Figure 4 shows that even when the working set far exceeds the flash size, the flash improves performance significantly, because the difference between flash performance and filer performance is substantial. In all configurations, the RAM hit rate is only 3.4%, but the flash hit rate varies from 0 (with no flash) to 47% in the 128 GB configuration. Although the filer fast read time (92 μs) is quite close to that of flash (88 μs), the two orders of magnitude difference between fast and slow filer read times is significant, even with the 90% fast filer read rate. As we shall see in the next section, the filer's ability to read ahead is critical in any configuration. The write latency figures from this experiment are not interesting: all writes see the RAM write latency of 0.4 μs.

7.3 Filer Read-Ahead

An effect observed in Mercury [6] suggests that a large cache reduces the file server's ability to prefetch data. We cannot yet quantify this effect, but we can bound it. In Figure 5 we show the spread between an 80% prefetch rate, which we believe to be a reasonable lower bound, and a 95% prefetch rate, which serves as a plausible upper bound. The graph shows the spread for the 64 GB flash, as well as for no flash, using the same range of working set sizes used in the previous section.

The application read latency is dominated by the cost of file server misses, which cost milliseconds. In an ideal world, installing the flash cache would not affect the file server's prefetch ability. Then the flash cache is beneficial for almost all workload sizes, as can be seen in the figure. In a pessimal world, the prefetch rate might drop substantially; in this case the cache is beneficial for a much narrower range of workloads: those that fit in flash but not in RAM. This can be seen in Figure 5 as the pocket between the lower (better) no-flash curve and the upper (worse) with-flash curve.

Avoiding the pessimal world is an engineering challenge and a critical issue for the adoption of flash caching. In the presence of a flash cache, the filer cache transitions from a second level cache to a third level cache; its prefetching and replacement policies must therefore adapt accordingly [5, 8, 23].

However, in environments where the back end is not a filer but a plain disk array [11], the prefetch rate will be negligible and a flash cache is a huge win.

7.4 Flash Cache Size

We next examined the converse case: given a fixed workload, what happens as we increase the flash cache size. As expected, the read latency decreases as a greater portion of the working set falls in the cache until the flash cache is large enough to capture the entire working set, at which point the read latency is that of flash. As there is nothing unexpected in these results, we have omitted the corresponding graphs.

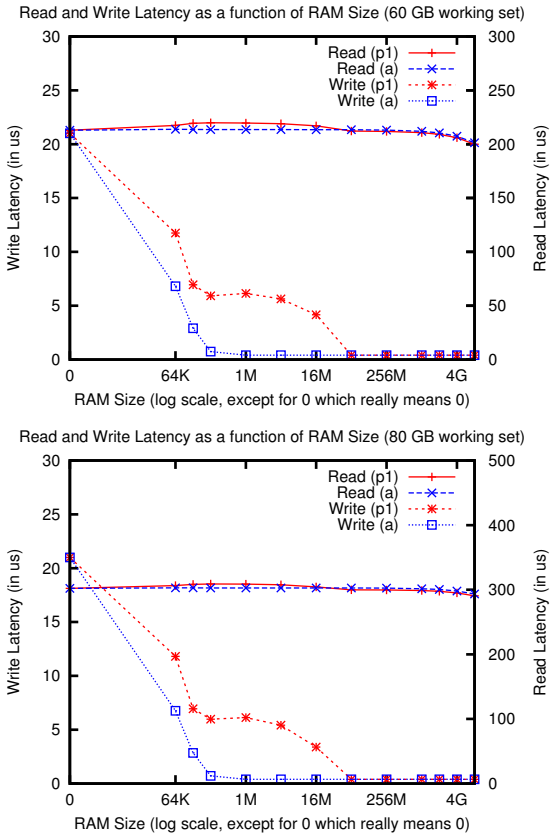


Figure 6: Application read and write latencies with small RAM cache sizes. The (a) and (p1) notations in both graphs refer to the RAM write-back policy: asynchronous write-through and 1-second periodic respectively. Surprisingly, a small (256 KB) cache achieves performance comparable to much larger ones.

7.5 No RAM Cache

One intriguing possibility suggested by the previous results is to dispense with the RAM cache entirely. We run the baseline workloads with a fixed 64 GB flash cache and RAM cache sizes ranging from zero to the baseline 8 GB. We run these with both the asynchronous write-through RAM policy (a) as well as the default 1-second periodic writeback (p1) we chose above.

Figure 6 shows the application read and write latencies for the 60 GB and 80 GB working sets, respectively. The X axis is the base 2 log of the RAM size or zero for none.

The no-RAM configuration does not work well, but it is surprising how well a relatively small (e.g., 64 MB) RAM cache performs. If we use the asynchronous write-through policy, a tiny 256 KB is sufficient as a write buffer. For the smallest caches the periodic syncer does not run often enough, so the RAM cache fills with dirty

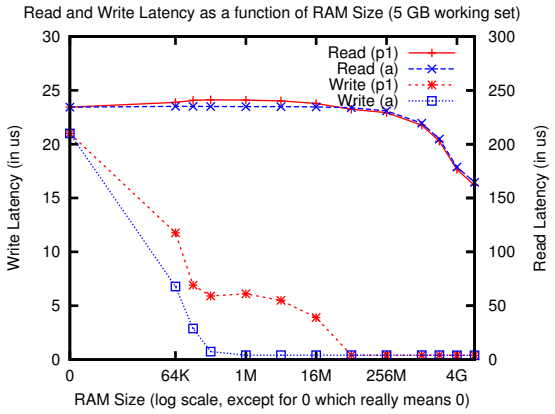


Figure 7: Application read and write latencies with small RAM cache sizes and a small workload.

blocks and performance drops.

The somewhat startling conclusion is that with a large, cheap flash cache, and a workload much larger than RAM, we can allocate minimal RAM (large enough to act as a speed-matching buffer) to file system caching, leaving the rest of memory available for application or operating system use!

This was tantalizing, so we tried the small RAM configuration on RAM-sized workloads. Figure 7 shows the latencies for a workload with a 5GB working set. As seen at the right, this configuration carries a 25-30% penalty, which is noticeable but far less than the factor of five or so seen without the flash cache. It may be an acceptable tradeoff in some circumstances.

7.6 Read-mostly vs. Write-mostly

The previous results all assumed a 30% write percentage. We next investigate the sensitivity of our results to the write percentage. We use our baseline working set sizes (60 GB and 80 GB) and cache sizes (8 GB RAM cache and 64 GB flash cache), while varying the percentage of writes in the trace from 0% to 100%. Figure 8 shows the application-level read and write latencies. As expected, read latency remains stable. The write latency is also unaffected except at very high write rates, where we start seeing synchronous writebacks from the RAM cache that expose the flash's write latency. As the proportion of writes increases, the trace runs faster, because writes are faster than reads. At very high write rates the 1-second RAM-to-flash syncer starts to fall behind. Several other effects come into play as well, such as network saturation, resulting in complex behavior that may be imperfectly modeled. The portion of the graphs above 90%

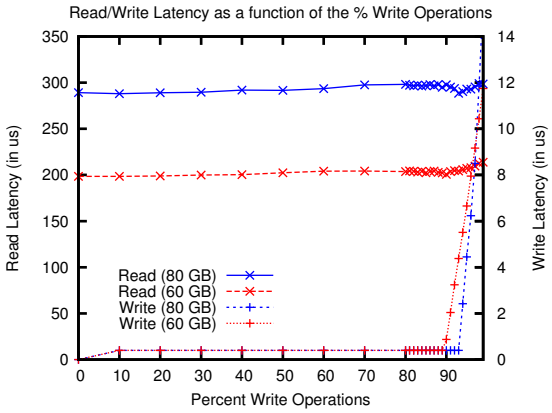


Figure 8: Application read and write latencies (in seconds) as a function of write percentage. As long as the write percentage remains below 90%, avoiding synchronous RAM evictions, performance is independent of the write rate.

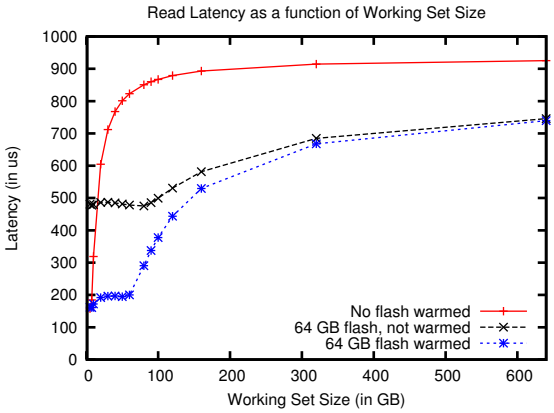


Figure 10: Effect of persistence. The not-warmed case is equivalent to having a non-persistent cache and crashing at the beginning of the simulator run. The no flash case is provided for comparison.

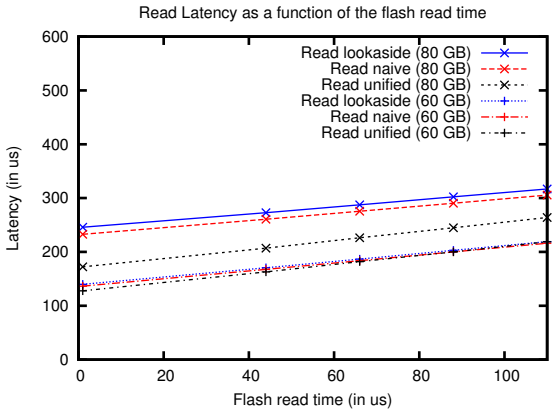


Figure 9: Application read latencies (in μs) for a range of flash read latencies (shown) and write latencies (proportional), in μs.

writes should be taken with a grain of salt.

The benefit of flash caching increases with write ratio because writes never incur a file server latency by missing in the cache: they always go straight to cache and are written back in the background.

7.7 Flash Timings

As flash devices vary a good deal in performance, we wanted to test a variety of flash timing configurations. Once again, the results were as expected: where the flash latencies appear directly, they scale with the flash speed; where they are hidden, changing the flash speed has no effect; and where they participate in the total latency, the overall latency scales linearly.

Figure 9 shows the application-level read latency for a range of flash timings for both standard traces and all three cache architectures. The leftmost point represents the potential performance of phase-change memory.

When the working set fits in flash, the architecture makes little difference, but when it falls out, we see the benefit of the larger effective sizes of the *unified* architecture. In all cases, however, application latency scales linearly with the flash latency, so improvements in flash timings are readily visible to the application.

7.8 Persistence

We approximated the cost making the flash persistent by doubling the flash write latency to model performing two flash writes per block, one of the data and one for the meta-data describing the block. (We did not attempt to simulate the recovery phase.) We investigated the benefit by skipping the warming phase of our traces; this is equivalent to having a non-persistent flash cache and crashing at the start of the simulator run.

The result is that the increased flash write latency associated with persistence is invisible to the application. This is consistent with our other results where the flash write latency is also invisible. However, the benefit of persistence, or rather the potential cost of not providing persistence, is substantial, as shown in Figure 10.

7.9 Cache Consistency

As discussed in Section 3.8, flash caches introduce two problems related to consistency: their larger size, and,

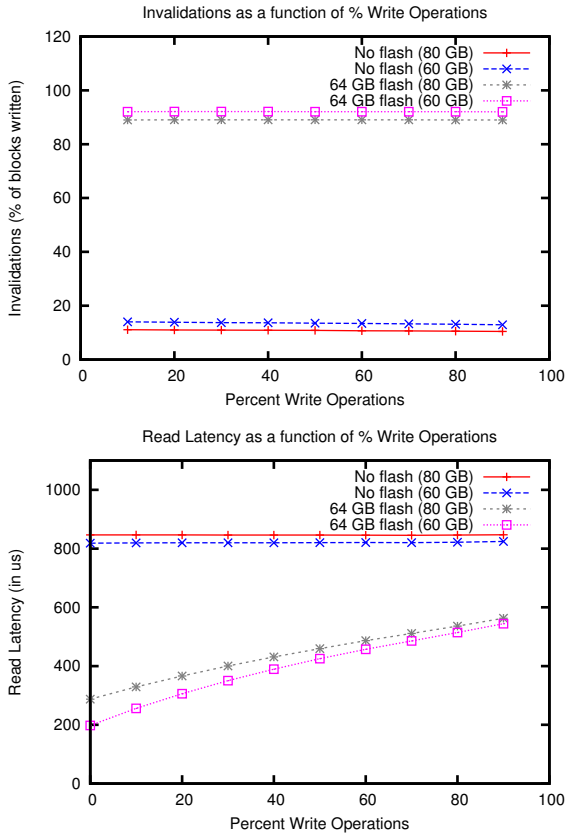


Figure 11: Invalidations required, and read latency, as a function of write percentage.

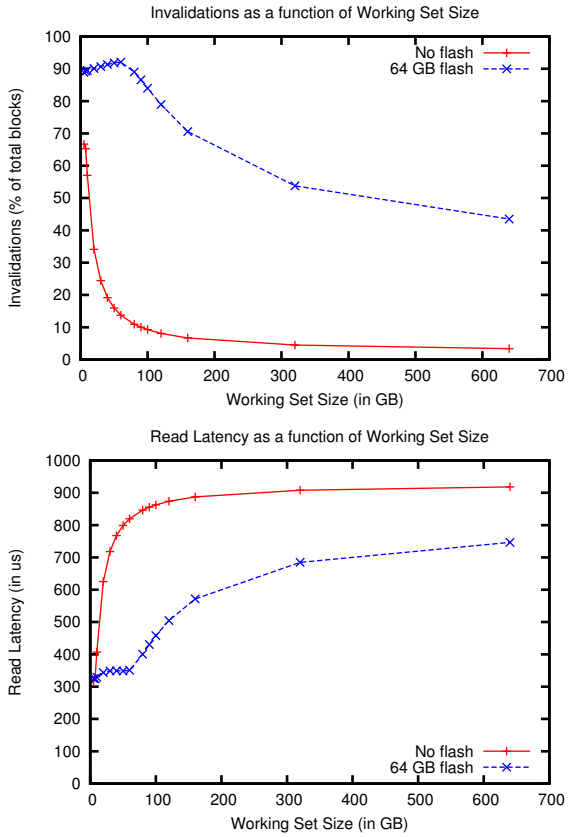


Figure 12: Invalidations required, and read latency, as a function of working set size.

for recoverable caches, that the cache is offline during reboots. These may affect cache consistency protocols.

We generated two additional families of traces, using two hosts, to investigate the effect of size on consistency control. As a worst-case scenario we make the two hosts share one working set. In the first family, we examine varying write percentages; in the second, we examine a range of working set sizes. Writing a new version of a block into a cache must invalidate all copies in other caches. We measure the fraction of (application-level) block writes that require invalidations.

Figure 11 shows the percentage of blocks written requiring invalidation and application read latency, as a function of the write percentage. The write latencies (for the 64 GB flash) are comparable to those in Figure 8.

Figure 12 shows, for the baseline setting of 30% writes, the percentage of invalidations and the application read latency as a function of the working set size. The write latency results are uniform and are not shown.

The primary finding is that for workloads that fit in flash, the percentage of writes requiring invalidation is high, even relative to workloads that fit in RAM with no flash. The invalidation rate drops off for out-of-cache workloads, but neither as quickly nor as significantly as with the smaller RAM cache. This has implications for read performance as well. Comparing the application read latency graphs (Figure 11 to Figure 8 and Figure 12 to Figure 4), we see that while the flash provides an advantage, read latency increases with the fraction of invalidations, because invalidated blocks must be reread from the filer. Although this is a worst case analysis (both servers share the entire working set), these results highlight critical areas in cache management design.

8 Conclusions

The results of our simulations show that even in its simplest implementation, a client-side flash cache provides significant benefits to applications. We now review our findings regarding the design questions from Section 1.

The flash cache does not need to be integrated with the

file system. While doing so increases the effective size of the cache, given the relative sizes (and prices) of RAM and flash this effect is fairly small and may not justify the implementation complexity.

The flash cache can be as large relative to RAM as desired. In fact, except for workloads that fit entirely into RAM, it makes sense to limit the RAM cache to the space needed to buffer writes, keeping the cache only in flash.

Any writeback policy that avoids synchronous writes and does not allow the cache to become full of dirty data produces good performance. Prompt writeback from flash exposes cache consistency events at no cost, and these cache consistency events are potentially important.

It is not necessary to make the cache persistent (that is, recoverable) to benefit from it. However, doing so offers significant additional benefit.

Cache consistency is a serious issue when multiple hosts actively modify a shared working set. Even with a write-through flash cache, such workloads cause substantially higher invalidation traffic than we see with traditional RAM-based caches. Also, traditional cache consistency protocols may not be able to cope with a recoverable cache being offline while recovering.

There is much follow-on work to be done. The most important area of further research is adapting file servers to these larger caches, ensuring that we can retain excellent read-ahead behavior when we do miss in the flash. In the presence of data shared among multiple hosts, each with its own flash cache, it is necessary to explore the details of maintaining cache consistency among the multiple caches. Finally, flash caching is a good candidate for a custom flash translation layer [19] – exploring approaches and algorithms as well as establishing satisfactory lifetime for this application remains as future work.

9 Acknowledgements

This work was supported by NetApp. In addition, James Lentini, Keith Smith, and Chris Small, all of NetApp, were tremendously helpful in providing us with the means and expertise to validate our simulator.

References

[1] Smart Array technology: Advantages of battery-backed cache. http://h10032.www1.hp.com/ctg/Manual/c00257513.pdf, 2002.

[2] Oracle, Sun launch high-end OLTP server. PCWorld, Sep 2009.

[3] EMC outlines strategy to accelerate flash adoption. In *EMCWorld 2011* (May 2011), http://www.emc.com/about/news/press/2011/20110509-05.htm.

[4] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. *Trans. Storage 5* (December 2009), 16:1–16:30.

[5] BUTT, A. R., GNIADY, C., AND HU, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. SIGMETRICS 2005* (Banff, Alberta, Canada, 2005), ACM, pp. 157–168.

[6] BYAN, S., ET AL. Mercury: Host-side flash caching for the data center. In *28th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2012)* (April 2012), pp. 1 –12.

[7] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *Proc. ASPLOS* (October 1996).

[8] FORNEY, B. C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Storage-aware caching: revisiting caching for heterogeneous storage systems. In *Proc. FAST* (Monterey, CA, 2002), USENIX Association, pp. 5–5.

[9] HOWARD, J. H., ET AL. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6* (February 1988), 51–81.

[10] JOUKOV, N., WONG, T., AND ZADOK, E. Accurate and efficient replaying of file system traces. In *Proc. FAST* (San Francisco, CA, 2005), USENIX Association, pp. 25–25.

[11] KOLLER, R., ET AL. Write policies for host-side flash caches. In *Proc. FAST* (San Jose, CA, 2013), USENIX Assoc., pp. 45–58.

[12] KOURAI, K. CacheMind: Fast performance recovery using a virtual machine monitor. In *Dependable Systems and Networks Workshops (DSN-W)* (July 2010), pp. 86 –92.

[13] MCCALPIN, J. D. Stream: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2011. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[14] MESNIER, M. P., ET AL. Trace: parallel trace replay with approximate causal events. In *Proc. FAST* (San Jose, CA, 2007), USENIX Association, p. 24.

[15] MICROSOFT. ReadyBoost. http://windows.microsoft.com/en-US/windows7/products/features/readyboost, 2009.

[16] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6* (February 1988), 134–154.

[17] NETAPP. Flash Cache. http://www.netapp.com/us/products/storage-systems/flash-cache/.

[18] RAIDON. HyBrid RunneR iH2420-2S-S2 data sheet. http://www.raidon.com.tw/content.php?sno=0000462&p_id=113, 2010.

[19] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: a lightweight, consistent and durable storage cache. In *Proc. EuroSys* (Bern, Switzerland, 2012), ACM, pp. 267–280.

[20] SEAGATE. Momentus XT product data sheet. http://www.seagate.com/docs/pdf/datasheet/disc/ds_momentus_xt_retail.pdf, 2009.

[21] SHEPLER, S., ET AL. NFS version 4 protocol. http://www.ietf.org/rfc/rfc3530.txt, April 2003.

[22] VIJAYAKUMAR, K., MUELLER, F., MA, X., AND ROTH, P. C. Scalable I/O tracing and analysis. In *Proc. Workshop on Petascale Data Storage* (Portland, Oregon, 2009), ACM, pp. 26–31.

[23] YADGAR, G., FACTOR, M., AND SCHUSTER, A. Karma: know-it-all replacement for a multilevel cache. In *Proc. FAST* (San Jose, CA, 2007), USENIX Association, pp. 25–25.

Practical and effective sandboxing for non-root users

Taesoo Kim and Nickolai Zeldovich
MIT CSAIL

Abstract

MBOX is a lightweight sandboxing mechanism for non-root users in commodity OSes. MBOX's sandbox usage model executes a program in the sandbox and prevents the program from modifying the host filesystem by layering the sandbox filesystem on top of the host filesystem. At the end of program execution, the user can examine changes in the sandbox filesystem and selectively commit them back to the host filesystem. MBOX implements this by interposing on system calls and provides a variety of useful applications: installing system packages as a non-root user, running unknown binaries safely without network accesses, checkpointing the host filesystem instantly, and setting up a virtual development environment without special tools. Our performance evaluation shows that MBOX imposes CPU overheads of 0.1–45.2% for various workloads. In this paper, we present MBOX's design, efficient techniques for interposing on system calls, our experience avoiding common system call interposition pitfalls, and MBOX's performance evaluation.

1 Introduction

In this paper, we present MBOX, a lightweight sandboxing mechanism for non-root users in commodity OSes. MBOX provides two attractive benefits as a sandbox; first, protection of the host filesystem from modifications by sandboxed programs; and second, flexibility in controlling the execution of the sandboxed program.

To protect the host system, MBOX overlays the host filesystem with a sandbox filesystem and confines all modifications made by the sandboxed program to the sandbox filesystem. As MBOX stores the sandbox filesystem as a regular directory in the host filesystem, users can use standard Unix tools to examine the modifications, commit them back to the host filesystem, or even archive them for later use as a layered sandbox filesystem for other programs.

MBOX implements the layered sandbox filesystem with system call interposition. By interposing on system calls, MBOX can provide additional features missing from commodity OSes, which are useful to non-root users in a variety of real-world scenarios: enabling non-root users to install system packages with standard package managers, checkpointing the whole filesystem instantly, running unknown binaries safely without network access, and setting up virtual development environments without special tools. More importantly, all use cases neither require root privilege nor require modification to the OS kernel and applications.

Overview MBOX aims to make running a program in a sandbox as easy as running the program itself. For example, one can sandbox a program (say `wget`) by running as below:

```
$ mbox -- wget google.com
...
Network Summary:
 > [11279] -> 173.194.43.51:80
 > [11279] Create socket(PF_INET,...)
 > [11279] -> a00::2607:f8b0:4006:803:0
 ...
Sandbox Root:
 > /tmp/sandbox-11275
 >  N:/tmp/index.html
[c]ommit, [i]gnore, [d]iff, [l]ist, [s]hell, [q]uit ?>
```

`wget` is a utility to download files from the web. In the above example, MBOX prevents `wget` from writing the downloaded `index.html` to the host filesystem, and instead redirects it to the sandbox filesystem (stored at `/tmp/sandbox-11275`). Since the sandbox filesystem is just a regular directory in the host filesystem, the user can use standard Unix tools to perform operations on the files modified by the program. For example, the user can commit the `index.html` file back to the place where `wget` would have downloaded the file if it was not sandboxed.

The advantages of using MBOX come from the fact that we can restrict the sandboxed program or change its behavior while protecting the host filesystem. For example, we can enable interesting use cases like monitoring where `wget` connects to and what it downloads, or restricting its remote network accesses (see §2).

Contributions In this paper, we

- describe the MBOX abstraction, usage model, and a wide range of use cases.
- present `seccomp/BPF` as an efficient system call interposition technique, and our experience with avoiding common system call interposition mistakes [4].
- implement and evaluate these ideas in MBOX, a Linux-based open source tool that requires no changes to the OS kernel or applications.

Outline §2 provides practical use cases of MBOX. §3 describes its design. §4 explains MBOX's interposition

technique. §5 discusses its implementation, §6 evaluates, §7 compares MBOX with related work, and §8 concludes.

2 Use cases

We motivate the usefulness of MBOX by describing five real-world use cases that are difficult to achieve in commodity OSes as a non-root user.

2.1 Installing packages without root access

```
$ mbox -R -- apt-get install git
  (-R: emulate a fakeroot environment)
```

Installing packages requires root privilege in Linux because normal users do not have write access to system directories such as `/bin` and `/lib`; so, to install a package, non-root users need to perform tedious jobs like resolving dependencies manually and compiling source code, even though package managers already perform these jobs. With MBOX, users can instead install packages with standard package managers by running them in a sandbox with a writable sandbox filesystem. As package managers often check for root privilege, MBOX optionally emulates a root-like environment (fakeroot) so users can execute them without any modification. After installing a package with MBOX, the sandbox filesystem contains not only newly installed files, but also the corresponding package databases, separate from the host filesystem. Users, therefore, can even install or remove packages by reusing the same sandbox filesystem (see §2.4). We tested that MBOX supports Ubuntu's `apt-get`, Debian's `dpkg`, and Python's `pip` package managers.

2.2 Running unknown binary safely

```
$ mbox -n -- wget google.com
  (-n: disable remote network accesses)
```

When running unknown binaries, users can protect the host filesystem from modification by running them with MBOX. However, if these binaries misbehave or are compromised, they still can access a user's private data and disclose it to attackers. To prevent this, MBOX provides a way to restrict or monitor remote network accesses of sandboxed processes. If users want to restrict network accesses, MBOX blocks all socket-related system calls; for example, the above command kills `wget` at the first `socket()` system call. However, by default, MBOX interprets socket-related system calls and summarizes network activity, as in the `wget` example in §1.

2.3 Checkpointing filesystem

```
$ mbox -i -- sh
  (-i: enable interactive commit-mode)
```

Using MBOX, one can instantly branch out a new filesystem from the current host filesystem by running a new shell. The shell and all subsequent processes created from the shell run in the same sandbox, and share the same layered filesystem view. For example, editing `emacs` configuration files often requires killing and rerunning `emacs` to check if it works with the new configuration. When it fails with an error, we might need to run vanilla `emacs` to continue fixing the error. With MBOX, one can checkpoint the host filesystem and edit configuration files with `emacs` running in the sandbox; `emacs` instances on the host system still function correctly, even if the edited file has an error. When done with editing, users can commit the modified configuration files to the host filesystem, revert them by discarding changes, or stash them for later use. These workflows are what make users feel comfortable when using SCM tools like Git; with MBOX, users get similar safety and convenience for filesystem data.

2.4 Build/development environment

```
$ mbox -r outdir -- make
  (-r dir: specify a sandbox directory)
```

When building a project's source tree, we often see the directory entangled with both original source files and generated object files. By running a build script with MBOX, we can redirect all generated object files to the sandbox filesystem; also, cleaning up the project directory (say `make clean`) becomes a simple `rm -rf outdir`. Combined with package installations (§2.1), any user can conveniently setup a development environment that is safely separated from the system libraries. For example, without using `virtualenv` for Python and `cabal-dev` for Haskell, we can create virtual environments with the `pip` and `cabal` tools that major distributions come with.

2.5 Profile-based sandbox

```
$ mbox -p build.prof -- ./configure
  (-p prof: enable profile-based policy)
```

MBOX supports another important use case poorly supported by commodity OSes. In Unix-like OSes, a process created by a user runs with that user's privilege, and can access the user's private files. In some cases, the process needs access to the user's files to do useful work; however, often there are cases where the user does not want to expose sensitive data to the process. For example, when a user executes a `./configure` script, she does not want the script to read her private ssh key stored in the `$HOME/.ssh` directory. With MBOX, users can easily hide private directories, and allow access to only the necessary parts of a filesystem by describing them as below.

```
# build.prof
[fs]
    allow: .
    hide: ~
```

If a user runs the `./configure` script with the above profile, MBOX hides the user's home directory yet allows access to the current working directory. Therefore, the script cannot steal the user's private files, but can still

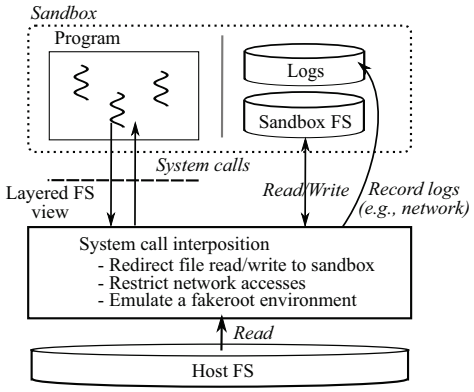


Figure 1: Overview of MBOX's design. MBOX interposes on a sandboxed program's system calls to provide a sandbox filesystem overlaid on the host filesystem; to restrict network accesses; and to emulate a fakeroot environment.

configure properly by accessing the system libraries and header files. In addition, for scripts that are never expected to access the network, users can additionally specify an option to restrict network accesses (§2.2).

3 MBOX abstraction

When a user runs a program with MBOX, MBOX creates a layered sandbox filesystem, where all modifications by the program take place, on top of the host filesystem. The host filesystem remains intact and is never modified by the sandboxed program. When the sandboxed program terminates, users can examine modified files in the sandbox filesystem, and commit them back to the host filesystem if they want. Since the sandbox filesystem is stored in persistent storage, users have complete control over files and directories afterward, and can even reuse them later as a sandbox layer of other programs. We call this usage model the MBOX abstraction. Figure 1 provides an overview of the MBOX design.

3.1 Layered filesystems

Unlike traditional filesystems in which every process has the same namespace, MBOX needs to provide a private filesystem to each process running in different sandboxes. MBOX stacks a private filesystem layer on top of the host filesystem, and provides a logically unified view of both filesystems to a sandboxed program. We call the private filesystem layer, where all modification happens by the program, the *sandbox filesystem*, and call both the sandbox and host filesystems together the *layered filesystem*. To provide a layered filesystem, MBOX interposes on system calls of a sandboxed program. On every system call entry, MBOX decides which system call arguments should be rewritten so that changes by the system call redirect to the sandbox filesystem, rather than affecting the host filesystem.

Copy-on-write The sandbox filesystem is created with no content when a user executes a program with MBOX. Since the sandbox filesystem is empty, all reads by the program will be forwarded to the host filesystem. Once the sandboxed program writes to a file, the sandbox filesystem will contain the modified file and subsequent reads will be redirected to the sandbox filesystem. Thus, the application running inside the sandbox is able to access the modified file and works as it would without the sandbox. The layered filesystem in effect implements copy-on-write: MBOX duplicates the file into the sandbox filesystem and protects the original file from modifications.

Persistent storage The sandbox filesystem is not a filesystem, but is a regular directory in the host filesystem, so it can persist even after the sandboxed program terminates. The persistent sandbox gives users more freedom to examine, archive, and even duplicate the sandbox filesystem, as normal files and directories, with familiar utilities. Also, users can reuse the previous sandbox filesystem as a sandbox layer of any other program, so that users can consider the layered filesystem persistently branched out of the host filesystem, yet easy to discard.

3.2 Committing changes

When a sandboxed program terminates, users can commit modified files back to the host filesystem with tools that MBOX provides. To help users decide what files to commit, MBOX allows the user to check the differences of files in host and sandbox filesystems before committing.

When committing a modified file back to the host filesystem, the original file that the sandbox branched out from might have been changed by programs running on the host filesystem. Faced with such concurrent modifications to the same file in both the host and the sandbox filesystem, MBOX flags a conflict, and requires the user to decide how to merge the changes, much like any version control system.

To detect conflicts, MBOX records a hash of the original file contents when creating a copy of the file in the sandbox filesystem, and checks if the contents of the file in the host filesystem still match the hash before committing any changes from the sandbox. For conflicts in text files, standard Unix tools like `diff` and `patch` can often resolve the conflict, but in other cases like custom or binary files, users should manually merge them with application-specific tools.

4 Interposing system calls

In this section, we describe the recently introduced `seccomp/BPF` [1] as a means for interposing system calls; common pitfalls of using `ptrace` and `seccomp/BPF` for sandboxing; and how to use them to restrict network accesses and construct a fakeroot environment.

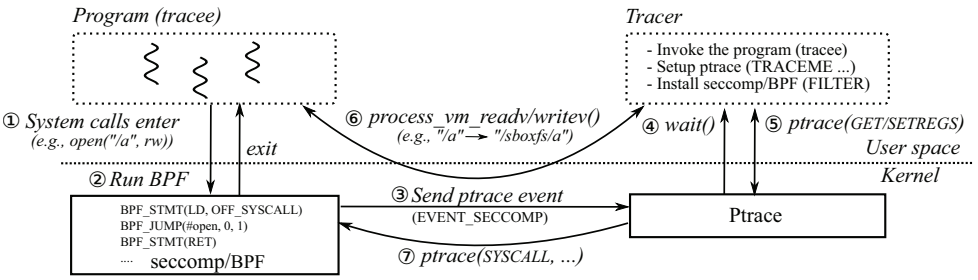


Figure 2: Interposing a system call with `seccomp/BPF` and `ptrace`. At startup, the tracer invokes the tracee, sets up `ptrace` and installs a `BPF` program. When the tracee generates a system call ①, the `BPF` program runs and decides whether to intercept or not ②. If the system call needs to be handled by MBOX, `BPF` will send a `seccomp` event ③ to the tracee waiting for a `ptrace` event ④. Then, the tracer queries the states of the tracee via the `ptrace` interface ⑤, or overwrites the tracee's memory with the `process_vm_writev()` system call ⑥. To continue the tracee, but stop at the exit of the current system call, the tracer needs to invoke `ptrace` with `SYSCALL`.

4.1 Using `seccomp/BPF`

`seccomp` [2] is a mechanism for isolating a process by allowing only a certain set of system calls. Linux 3.5 further introduced support for using Berkeley Packet Filter (`BPF`) bytecode to examine system calls when using `seccomp` [1]; for example, the `BPF` bytecode can decide whether the process can invoke the `socket()` system call. In `seccomp/BPF`, the input to the `BPF` program is the system call number, its arguments, and the instruction pointer, and it is invoked on every entry and exit of a system call. The `BPF` program decides whether to allow the system call to proceed or not; an additional option is to generate a `ptrace` event to the tracer, if the current process has one. Using `seccomp/BPF`, the tracer can download a `BPF` program and wait for a `ptrace` event, as described in Figure 2, instead of stopping on every tracee system call. This allows MBOX to interpose on just the necessary system calls, improving overall performance, as we show in §6.

4.2 Avoiding common pitfalls

It is easy to make mistakes when implementing a sandbox mechanism, making the resulting implementation vulnerable to adversaries due to minor mistakes. In particular, `ptrace` and `seccomp/BPF` are difficult to use correctly for interposing on system calls. We will now describe our experience in trying to avoid some of the pitfalls in using `ptrace` and `seccomp/BPF` for system call interposition.

4.2.1 Time-of-check-to-time-of-use (TOCTTOU)

Using `ptrace` to intercept system call entry allows us to examine, sanitize, and rewrite the system call's arguments. If an argument points to process memory, we can read remote memory and interpret it as the system call handler does. However, the read value can be different from what the system call handler will see in the kernel. For example, an adversary's thread can overwrite the memory that the current argument points to, right after the tracer checks the argument. Even verifying that sanitized arguments still point to the right value at system call exit does not help, because an adversary can restore it by that time.

To avoid TOCTTOU problems in rewriting memory arguments, MBOX takes advantage of two properties of `ptrace`. First, system call arguments examined using `PTRACE_GETREGS` are the actual values that the handler will see, because `x86-64` uses registers to pass system call arguments, and copies them to kernel space when entering the system call handler. Second, `ptrace` allows the tracer to write to read-only memory in the tracee with `PTRACE_POKEDATA`.

MBOX avoids TOCTTOU problems by mapping a page of read-only memory in the tracee process. When MBOX needs to examine, sanitize, or rewrite an in-memory data structure, such as a path name, used as a system call argument, MBOX copies the data structure to the read-only memory (using `PTRACE_POKEDATA` or the more efficient `process_vm_writev()`), and changes the system call argument pointer to point to this copy. For example, at the entry of an `open(path, O_WRONLY)` system call, the tracer first gets the system call's arguments, rewrites the path argument to point to the read-only memory, and updates the read-only memory with a new path pointing to the sandbox filesystem. Since no other threads can overwrite the read-only memory without invoking a system call (e.g., `mprotect()`), MBOX avoids TOCTTOU problem when rewriting path arguments. To ensure that the sandboxed process cannot change this read-only virtual memory mapping (e.g., using `mprotect()`, `mmap()`, or `mremap()`), MBOX intercepts these system call and kills the process if it detects an attempt to modify MBOX's special read-only page.

4.2.2 Replicating OS state

Another common mistake is to improve performance by replicating some state of the tracee process in the tracer. For example, in handling an `openat(fd, ...)` system call, one might think that keeping track of a path for `fd` whenever opening a path can improve performance, instead of reading the actual path for `fd`. However, it is

impractical to correctly emulate in userspace all subtle system calls that can change the state of a file descriptor. In MBOX, we design a set of stateless rules for deciding whether to rewrite a path argument of the current system call, by paying the cost of examining states at its entry. By controlling how a process can obtain a file descriptor in the first place, MBOX does not need to interpose on system calls that take only file descriptor arguments.

Rules for rewriting path arguments MBOX rewrites a path argument as follows:

- If `path` exists in the sandbox filesystem, then it was already modified by previous write operations. MBOX rewrites `path` to point at the sandbox filesystem, so that subsequent read/write should see the one in the sandbox filesystem.
- If `path` was deleted before, then to pretend that `path` in the host filesystem is deleted, `path` is rewritten to the non-existent path in the sandbox filesystem.
- If the current system call will modify the host filesystem, then, since `path` does not exist in the sandbox filesystem, MBOX copies the file from the host filesystem to the sandbox filesystem. The subsequent read/write will see the duplicated copy in the sandbox filesystem, by the first rule.

As one example, at the entry of an `open(path, O_RDWR)` system call, if `path` does not exists in the sandbox filesystem and was not deleted before, MBOX will copy the file from the host filesystem to the sandbox filesystem by the last rule, and rewrite the path to point to the sandbox filesystem, where any later `write()`s will be reflected. Any subsequent `open(path, O_RDONLY)` on the same path will also be rewritten to access the sandbox filesystem, by the first rule.

5 Implementation

We implemented a prototype of MBOX for Linux by extending `strace 4.7`, which is a system utility to trace system calls. To improve performance, we modified `strace` to use `seccomp/BPF`. For OSes that do not support `seccomp/BPF` yet, MBOX falls back to using `ptrace` as the main system call interposition mechanism (`seccomp/BPF` is supported on Linux 3.5 and above). MBOX has been tested on the x86-64 Arch distribution with the 3.8.10 Linux kernel, and the Ubuntu 12.04.1-LTS distribution with the 3.2.0-36 Linux kernel.

6 Evaluation

To analyze the performance characteristics of MBOX, we ran benchmarks used in Apiary [10] in three environments: without a sandbox, with MBOX using `ptrace`, and with MBOX using `seccomp/BPF` to intercept system calls. We carried out all experiments on a system with an Intel Core i7-2640M CPU, using one core with hyperthreads disabled, and 16GB RAM, running Arch Linux with kernel 3.8.10, if not stated specifically. Table 1 summarizes the results.

6.1 End-to-end performance overhead

In the computation-heavy Octave benchmark, Octave [6] in Table 1, MBOX exhibits negligible performance overheads, 0.1%, because it spends 98% of its execution time in userspace, with few system calls. However, when compressing files (Zip), decompressing files (Untar) or building the Linux kernel (Build Linux), MBOX incurs more significant overheads, 12.0%–20.9%, because these benchmarks invoke a lot of file-related system calls.

6.2 Interposing system calls

In the Zip and Untar benchmarks in Table 1, using `seccomp/BPF` was a lot more efficient than using `ptrace`. With `seccomp/BPF`, MBOX can intercept just the system calls that it needs to examine, and skip system calls such as `read()` and `write()` that take a file descriptor as an argument. Untar generates a total of 543k system calls, out of which 330k (60.8%) are `read()` and `write()`. Using `seccomp/BPF`, MBOX interposes on just 90k system calls (16.5%). These results show that `seccomp/BPF` helps MBOX reduce interposition overhead.

6.3 Concurrency

With `seccomp/BPF`, we can improve concurrency by avoiding unnecessary serialization of system calls, which enables each process to invoke system calls without being interleaved by the tracer. For example, `ptrace` imposed 110.1% overhead when building the Linux kernel in parallel, but using `seccomp/BPF` incurred 45.2% overhead, because the tracer interposed only on the necessary system calls, thereby allowing multiple system calls to execute simultaneously.

7 Related work

Layered filesystems UnionFS [8, 11] strongly influenced the design of MBOX; we follow its namespace unification rules and strategies for copy-on-write. However, MBOX enables them for non-root users by using `seccomp/BPF` in Linux, and also provides a variety of applications without requiring any modification of existing software. Cowdancer [12] and FL-COW [7] similarly provide a way to redirect modifications by a process, but since they use `LD_PRELOAD`, they cannot isolate a malicious process, unlike MBOX. Apiary [10] confines applications using UnionFS, but its main purpose of using the layered filesystem is to save storage by sharing package dependencies of confined applications.

Task	Normal	Sandbox				Description
		Ptrace		Seccomp/BPF		
Zip	15.6s	21.2s	36.5%	17.4s	12.0%	Compressing all files of linux-3.8
Octave	2.1s	2.1s	0.1%	2.1s	0.1%	Octave Benchmark [6] calculating matrix
Untar	13.6s	19.0s	40.3%	16.4s	20.9%	Decompressing linux-3.8 source files
Build Linux (-j1)	43.6s	53.2s	21.9%	49.7s	13.9%	Compiling linux-3.8 kernel
Build Linux (-j4)	21.7s	45.6s	110.1%	31.5s	45.2%	Compiling linux-3.8 kernel with 4 parallel jobs

Table 1: Performance benchmark results. Following the benchmark from Apiary [10], we measure the total execution time of each benchmark in normal execution, and in the sandbox using either `ptrace` and `seccomp/BPF`; for sandbox execution times, we also report the percent overhead on top of normal execution. We used two cores with hyperthreads enabled for the last benchmark, building the Linux kernel with 4 parallel jobs.

System call interposition Garfinkel used the system call interposition technique for enforcing security policies in Ostia [5], and studied common mistakes and pitfalls when using it for implementing a security tool [4]. In this paper, we summarized our experiences of avoiding those mistakes, especially the TOCTTOU attack, when using `seccomp/BPF` as a means for rewriting system calls.

Namespace The effectiveness of MBOX comes from the fact that every process can have a private namespace, detached from the host filesystem. Plan9 [9] originally proposed this idea; MBOX implements private namespaces by using `ptrace`, which commodity OSes provide to all users for debugging. MBOX, therefore, can use private namespaces for sandboxing without changing the kernel or applications. Docker [3] provides a container for applications by using namespaces, newly introduced in Linux 3.8, as a means to migrate processes transparently between OSes. We expect that the `mnt`, `net` and `ipc` namespaces, combined with Aufs [8], can be used for implementing an efficient layered filesystem, but without enabling all applications that MBOX provides with system call interposition.

8 Summary

We presented MBOX, a lightweight sandboxing mechanism for non-root users in commodity OSes. MBOX protects the host filesystem by layering the sandbox filesystem on top of it using efficient system call interposition based on `seccomp/BPF`. We showed that MBOX is effective in a variety of applications, and incurs reasonable CPU overhead. MBOX is available for download at `http://pdos.csail.mit.edu/mbox/`.

Acknowledgments

We thank Silas Boyd-Wickizer, Ramesh Chandra, Cody Cutler, Kavya Joshi, Meelap Shah, Keith Winstein, the anonymous reviewers, and our shepherd, David Presotto, for their feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] Dynamic seccomp policies (using BPF filters). `http://lwn.net/Articles/475019`, January 2012.

[2] A. Arcangeli. Seccomp: secure computing mode. `http://en.wikipedia.org/wiki/Seccomp`. January 2013.

[3] dotCloud. Docker: The Linux container engine. `http://www.docker.io`, 2013.

[4] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2003.

[5] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[6] P. Grosjean. Octave benchmark 2: speed comparison of various number crunching packages (version 2). `http://sciviews.org/benchmark`. January 2013.

[7] D. Libenzi. FL-COW 0.10. `http://xmailserver.org/flcow.html`. January 2013.

[8] J. R. Okajima. Aufs3: Advanced multi layered unification filesystem version 3.x. `http://aufs.sf.net`. January 2013.

[9] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[10] S. Potter and J. Nieh. Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 103–116, June 2010.

[11] D. Quigley, J. Sipek, C. Wright, and E. Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[12] J. Uekawa. Cowdancer: copy-on-write data access completely in userland. `http://www.netfort.gr.jp/~dancer/software/cowdancer.html.en`. January 2013.

TABLEFS: Enhancing Metadata Efficiency in the Local File System

Kai Ren, Garth Gibson
Carnegie Mellon University
{*kair, garth*}*@cs.cmu.edu*

Abstract

File systems that manage magnetic disks have long recognized the importance of sequential allocation and large transfer sizes for file data. Fast random access has dominated metadata lookup data structures with increasing use of B-trees on-disk. Yet our experiments with workloads dominated by metadata and small file access indicate that even sophisticated local disk file systems like Ext4, XFS and Btrfs leave a lot of opportunity for performance improvement in workloads dominated by metadata and small files.

In this paper we present a stacked file system, TABLEFS, which uses another local file system as an object store. TABLEFS organizes all metadata into a single sparse table backed on disk using a Log-Structured Merge (LSM) tree, LevelDB in our experiments. By stacking, TABLEFS asks only for efficient large file allocation and access from the underlying local file system. By using an LSM tree, TABLEFS ensures metadata is written to disk in large, non-overwrite, sorted and indexed logs. Even an inefficient FUSE based user level implementation of TABLEFS can perform comparably to Ext4, XFS and Btrfs on data-intensive benchmarks, and can outperform them by 50% to as much as 1000% for metadata-intensive workloads. Such promising performance results from TABLEFS suggest that local disk file systems can be significantly improved by more aggressive aggregation and batching of metadata updates.

1 Introduction

In the last decade parallel and Internet service file systems have demonstrated effective scaling for high bandwidth, large file transfers [48, 13, 17, 25, 38, 39]. The same, however, is not true for workloads that are dominated by metadata and tiny file access [34, 49]. Instead there has emerged a class of scalable small-data storage systems, commonly called key-value stores, that emphasize simple (NoSQL) interfaces and large in-memory caches [2, 24, 33].

Some of these key-value stores feature high rates of change and efficient out-of-memory Log-structured Merge (LSM) tree structures [8, 23, 32]. An LSM tree can provide fast random updates, inserts and deletes without scarificing lookup performance [5]. We believe that file systems should adopt LSM tree techniques used by modern key-value stores to represent metadata and tiny files, because LSM trees aggressively aggregate metadata. Moreover, today's key-value store implementations are "thin" enough to provide the performance levels required by file systems.

In this paper we present experiments in the most mature and restrictive of environments: a local file system managing one magnetic hard disk. We used a LevelDB key-value store [23] to implement TABLEFS, our POSIX-compliant stacked file system, which represents metadata and tiny files as key-value pairs. Our results show that for workloads dominated by metadata and tiny files, it is possible to improve the performance of the most modern local file systems in Linux by as much as an order of magnitude. Our demonstration is more compelling because it begins disadvantaged: we use an interposed file system layer [1] that represents metadata and tiny files in a LevelDB store whose LSM tree and log segments are stored in the same local file systems we compete with.

2 Background

Even in the era of big data, most things in many file systems are small [10, 28]. Inevitably, scalable systems should expect the numbers of small files to soon achieve and exceed billions, a known challenge for both the largest [34] and most local file systems [49]. In this section we review implementation details of the systems employed in our experiments: Ext4, XFS, Btrfs and LevelDB.

2.1 Local File System Structures

Ext4[26] is the fourth generation of Linux ext file systems, and, of the three we study, the most like traditional UNIX file systems. Ext4 divides the disk into block groups, similar to cylinder groups in traditional UNIX, and stores in each block group a copy of the superblock, a block group descriptor, a bitmap describing free data blocks, a table of inodes and a bitmap describing free inodes, in addition to the actual data blocks. Inodes contain a file's attributes (such as the file's inode number, ownership, access mode, file size and timestamps) and four extent pointers for data extents or a tree of data extents. The inode of a directory contains links to a HTree (similar to B-Tree) that can be one or two levels deep, based on a 32 bit hash of the directory entry's name. By default only changes to metadata are journaled for durability, and Ext4 asynchronously commits its journal to disk every five seconds. When committing pending data and metadata, data blocks are written to disk before the associated metadata is written to disk.

XFS[47], originally developed by SGI, aggressively and pervasively uses B+ trees to manage all file structures: free space maps, file extent maps, directory entry indices and dynamically allocated inodes. Because all file sizes, disk addresses and inode numbers are 64 bits in XFS, index structures can be large. To reduce the size of these structures XFS partitions the disk into allocation groups, clusters allocation in an allocation group and uses allocation group relative pointers. Free extents are represented in two B+ trees: one indexed by the starting address of the extent and the other indexed by the length of the extent, to enable efficient search for an appropriately sized extent. Inodes contain either a direct extent map, or a B+ tree of extent maps. Each allocation group has a B+ tree indexed by inode number. Inodes for directories have a B+ tree for directory entries, indexed by a 32 bit hash of the entry's file name. XFS also journals metadata for durability, committing the journal asynchronously when a log buffer (256 KB by default) fills or synchronously on request.

Btrfs[22, 36] is the newest and most sophisticated local file system in our comparison set. Inspired by Rodeh's copy-on-write B-tree[35], as well as features of XFS, NetApp's WAFL and Sun's ZFS[3, 18], Btrfs copies any B-tree node to an unallocated location when it is modified. Provided the modified nodes can be allocated contiguously, B-tree update writing can be highly sequential; however more data must be written than is minimally needed (write amplification). The other significant feature of Btrfs is its collocation of different metadata components in the same B-tree, called the FS tree. The FS tree is indexed by (inode number, type, offset) and it contains inodes, directory entries and file extent maps, distinguished by a type field: INODE_ITEM for inodes, DIR_ITEM and DIR_INDEX for directory entries, and EXTENT_DATA_REF for file extent maps. Directory entries are stored twice so that they can be ordered differently: in one the offset field of the FS tree index (for the directory's inode) is the hash of the entry's name, for fast single entry lookup, and in the other the offset field is the child file's inode number. The latter allows a range scan of the FS tree to list the inodes of child files and accelerate user operations such as $ls+stat$. Btrfs, by default, delays writes for 30 seconds to increase disk efficiency, and metadata and data are in the same delay queue.

2.2 LevelDB and its LSM Tree

Inspired by a simpler structure in BigTable[8], LevelDB [23] is an open-source key-value storage library that features an Log-Structured Merge (LSM) tree [32] for on-disk storage. It provides simple APIs such as GET, PUT, DELETE and SCAN (an iterator). Unlike BigTable, not even single row transactions are supported in LevelDB. Because TABLEFS uses LevelDB, we will review its design in greater detail in this section.

In a simple understanding of an LSM tree, an memory buffer cache delays writing new and changed entries until it has a significant amount of changes to record on disk. Delay writes are made more durable by redundantly recording new and changed entries in a write-ahead log, which is pushed to disk periodically and asynchronously by default.

In LevelDB, by default, a set of changes are spilled to disk when the total size of modified entries exceeds 4 MB. When a spill is triggered, called a minor compaction, the changed entries are sorted, indexed and written to disk in a format known as SSTable[8]. These entries may then be discarded by the memory buffer and can be reloaded by searching each SSTable on disk, possibly stopping when the first match occurs if the SSTables are searched from most recent to oldest. The number of SSTables that need to be searched can be reduced by maintaining a Bloom filter[7] on each, but with increasing numbers of records the disk access cost of finding a record not in memory increases. Scan operations in LevelDB are used to find neighbor entries, or to iterate through all key-value pairs within a range. When performing a scan operation, LevelDB first searches each SSTable to place a cursor; it then increments cursors in the multiple SSTables and merges key-value pairs in sorted order. Compaction is the process of combining multiple SSTables into a smaller number of SSTables by merge sort. Compaction is similar to *online defragmentation* in traditional file systems and *cleaning* process in

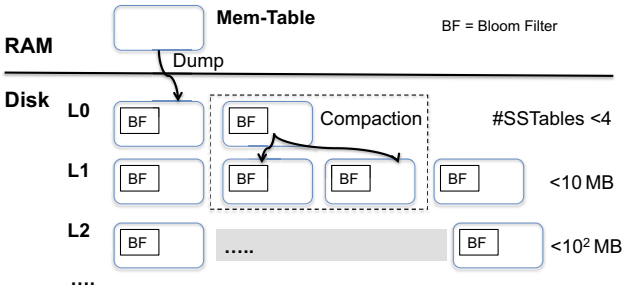


Figure 1: *LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs. SSTables are grouped into different levels with lower-numbered levels containing more recently inserted key-value pairs. Finding a specific pair on disk may search up to all SSTables in level 0 and at most one in each higher-numbered level. Compaction is the process of combining SSTables by merge sort into higher-numbered levels.*

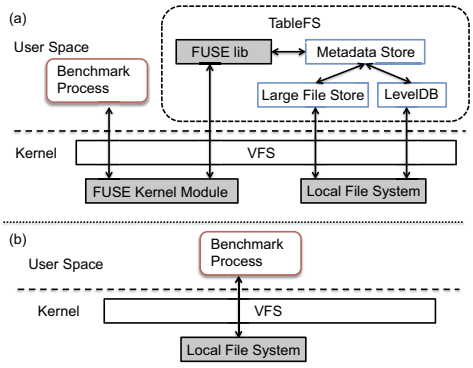


Figure 2: *(a) The architecture of* TABLEFS. *A FUSE kernel module redirects file system calls from a benchmark process to* TABLEFS*, and* TABLEFS *stores objects into either LevelDB or a large file store. (b) When we benchmark a local file system, there is no FUSE overhead to be paid.*

LFS [37].

As illustrated in Figure 1, LevelDB extends this simple approach to further reduce read costs by dividing SSTables into sets, or levels. Levels are numbered starting from 0, and levels with a smaller number are referenced as "lower" levels. The 0th level of SSTables follows a simple formulation: each SSTable in this level may contain entries with any key/value, based on what was in memory at the time of its spill. LevelDB's SSTables in level $L > 0$ are the results of compacting SSTables from level L or $L-1$. In these higher levels, LevelDB maintains the following invariant: the key range spanning each SSTable is disjoint from the key range of all other SSTables at that level and each SSTable is limited in size (2MB by default). Therefore querying for an entry in the higher levels only need to read at most one SSTable in each level. LevelDB also sizes each level differentially: all SSTables have the same maximum size and the sum of the sizes of all SSTables at level L will not exceed 10^L MB. This ensures that the number of levels, that is, the maximum number of SSTables that need to be searched in the higher levels, grows logarithmically with increasing numbers of entries.

When LevelDB decides to compact an SSTable at level L, it picks one, finds all other SSTables at the same level and level $L+1$ that have an overlapping key range, and then merge sorts all of these SSTables, producing a set of SSTables with disjoint ranges at the next higher level. If an SSTable at level 0 is selected, it is not unlikely that many or all other SSTables at level 0 will also be compacted, and many SSTables at level 1 may be included. But at higher levels most compactions will involve a smaller number of SSTables. To select when and what to compact there is a weight associated with compacting each SSTable, and the number of SSTables at level 0 is held in check (by default compaction will be triggered if there are more than four SSTables at level 0). There are also counts associated with SSTables that are searched when looking for an entry, and hotter SSTables will be compacted sooner. Finally, only one compaction runs at a time.

3 TABLEFS

As shown in Figure 2(a), TABLEFS exploits the FUSE user level file system infrastructure to interpose on top of the local file system. TABLEFS represents directories, inodes and small files in one all-encompassing table, and only writes large objects (such as write-ahead logs, SSTables, and large files) to the local disk.

3.1 Local File System as Object Store

There is no explicit space management in TABLEFS. Instead, it uses the local file system for allocation and storage of objects. Because TABLEFS packs directories, inodes and small files into a LevelDB table, and LevelDB stores sorted logs (SSTables) of about 2MB each, the local file system sees many fewer, larger objects. We use Ext4 as the object store for TABLEFS in all experiments.

Files larger than T bytes are stored directly in the object store named according to their inode number. The object store uses a two-level directory tree in the local file system, storing a file with inode number I as "/LargeFileStore/J/I" where $J = I \div 10000$. This is to circumvent any scalability limits on directory entries in the underlying local file systems. In TABLEFS today, T, the threshold for blobbing a file is 4KB, which is the median size of files in desktop workloads [28], although others have suggested T be at least 256KB and perhaps as large as 1MB [41].

3.2 Table Schema

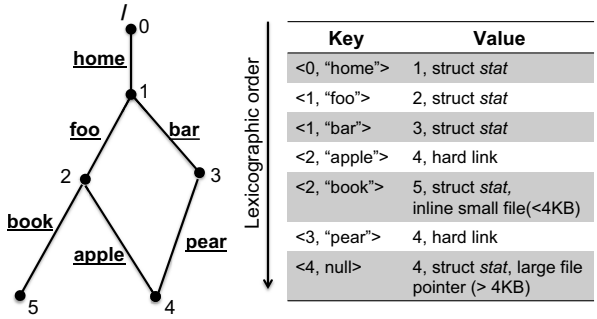
TABLEFS's metadata store aggregates directory entries, inode attributes and small files into one LevelDB table with a row for each file. To link together the hierarchical structure of the user's namespace, the rows of the table are ordered by a variable-length key consisting of the 64-bit inode number of a file's parent directory and its filename string (final component of its pathname). The value of a row contains inode attributes, such as inode number, ownership, access mode, file size and timestamps (*struct stat* in Linux). For small files, the file's row also contains the file's data.

Figure 3 shows an example of storing a sample file system's metadata into one LevelDB table.

All entries in the same directory have rows that share the same first 64 bits of their table key. For *readdir* operations, once the inode number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's inode number as the first 64 bits of their table key. To resolve a single pathname, TABLEFS starts searching from the root inode, which has a well-known inode number (0). Traversing the user's directory tree involves constructing a search key by concatenating the inode number of current directory with the hash of next component name in the pathname. Unlike Btrfs, TABLEFS does not need the second version of each directory entry because the entire attributes are returned in the *readdir* scan.

3.3 Hard Links

Hard links, as usual, are a special case because two or more rows must have the same inode attributes and data. Whenever TABLEFS creates the second hard link to a file, it creates a separate row for the file itself, with a null name, and its own inode number as its parent's inode number in the row key. As illustrated in Figure 3, creating a hard link also modifies the directory entry such that each row naming the file has an attribute indicating the directory entry is a hard link to the file object's inode row.



Key	Value
<0, "home">	1, struct *stat*
<1, "foo">	2, struct *stat*
<1, "bar">	3, struct *stat*
<2, "apple">	4, hard link
<2, "book">	5, struct *stat*, inline small file(<4KB)
<3, "pear">	4, hard link
<4, null>	4, struct *stat*, large file pointer (> 4KB)

Figure 3: *An example illustrates table schema used by* TABLEFS*'s metadata store. The file with inode number 4 has two hard links, one called "apple" from directory foo and the other called "pear" from directory bar.*

3.4 Scan Operation Optimization

TABLEFS utilizes the scan operation provided by LevelDB to implement *readdir()* system call. The scan operation in LevelDB is designed to support iteration over arbitrary key ranges, which may require searching SSTables at each level. In such a case, Bloom filters cannot help to reduce the number of SSTables to search. However, in TABLEFS, *readdir()* only scans keys sharing the common prefix — the inode number of the searched directory. For each SSTable, an additional Bloom filter is maintained, to keep track of all inode numbers that appear as the first 64 bit of row keys in the SSTable. Before starting an iterator in an SSTable for *readdir()*, TABLEFS can first check its Bloom filter to find out whether it contains any of the desired directory entries. Therefore, unnecessary iterations over SSTables that do not contain any of the requested directory entries can be avoided.

3.5 Inode Number Allocation

TABLEFS uses a global counter for allocating inode numbers. The counter increments when creating a new file or a new directory. Since we use 64-bit inode numbers, it will not soon be necessary to recycle the inode number of deleted entries. Coping with operating systems that use 32 bit inode numbers may require frequent inode number recycling, a problem beyond the scope of this paper and addressed by many file systems.

3.6 Locking and Consistency

LevelDB provides atomic insertion of a batch of writes but does not support atomic row read-modify-write operations. The atomic batch write guarantees that a sequence of updates to the database are applied in order, and committed to the write-ahead log atomically. Thus the *rename* operation can be implemented as a batch of two operations: insert the new directory entry and delete the stale entry. But for operations like *chmod* and *utime*, since all of an inode's attributes are stored in a single key-value pair, TABLEFS must read-modify-write attributes atomically. We implemented a light-weight locking mechanism in the TABLEFS core layer to ensure correctness under concurrent access.

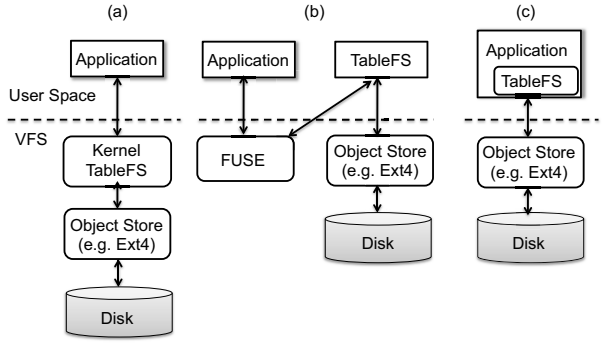


Figure 4: *Three different implementations of* TABLEFS*: (a) the kernel-native* TABLEFS*, (b) the FUSE version of* TABLEFS*, and (c) the library version of* TABLEFS*. In the following evaluation section, (b) and (c) are presented to bracket the performance of (a), which was not implemented.*

3.7 Journaling

TABLEFS relies on LevelDB and the local file system to achieve journaling. LevelDB has its own write-ahead log that journals all updates to the table. LevelDB can be set to commit the log to disk synchronously or asynchronously. To achieve a consistency guarantee similar to "ordered mode" in Ext4, TABLEFS forces LevelDB to commit the write-ahead log to disk periodically (by default it is committed every 5 seconds).

3.8 TABLEFS in the Kernel

A kernel-native TABLEFS file system is a stacked file system, similar to eCryptfs [14], treating a second local file system as an object store, as shown in Figure 4(a). An implementation of a Log-Structured Merge (LSM) tree [32] used for storing TABLEFS in the associated object store, such as LevelDB [23], is likely to have an asynchronous compaction thread that is more conveniently executed at user level in a TABLEFS daemon, as illustrated in Figure 4(b).

For the experiments in this paper, we bracket the performance of a kernel-native TABLEFS (Figure 4(a)), between a FUSE-based user-level TABLEFS (Figure 4(b)) with no TABLEFS function in the kernel and all of TABLEFS in the user level FUSE daemon) and an application-embedded TABLEFS library, illustrated in Figure 4(c).

TABLEFS entirely at user-level in a FUSE daemon is unfairly slow because of the excess kernel crossings and scheduling delays experienced by FUSE file systems [6, 45]. TABLEFS embedded entirely in the benchmark application as a library is not sharable, and unrealistically fast because of the infrequency of system calls. We approximate the performance of a kernel-native TABLEFS using the library version and preceding each reference to the TABLEFS library with a write("*/dev/null*", *N* bytes) to account for the system call and data transfer overhead. *N* is chosen to match the size of data passed through each system call. More details on these models will be discussed in Section 4.3.

4 Evaluation

4.1 Evaluation System

We evaluate our TABLEFS prototype on Linux desktop computers equipped as follows:

Linux	Ubuntu 12.10, Kernel 3.6.6 64-bit version
CPU	AMD Opteron Processor 242 Dual Core
DRAM	16GB DDR SDRAM
Hard Disk	Western Digital WD2001FASS-00U0B0
	SATA, 7200rpm, 2TB
	Random Seeks 100 seeks/sec peak
	Sequential Reads 137.6 MB/sec peak
	Sequential Writes 135.4 MB/sec peak

We compare TABLEFS with Linux's most sophisticated local file systems: Ext4, XFS, and Btrfs. Ext4 is mounted with "ordered" journaling to force all data to be flushed out to disk before its metadata is committed to disk. By default, Ext4's journal is asynchronously committed to disks every 5 seconds. XFS and Btrfs use similar policies to asynchronously update journals. Btrfs, by default, duplicates metadata and calculates checksums for data and metadata. We disable both features (unavailable in the other file systems) when benchmarking Btrfs to avoid penalizing it. Since the tested filesystems have different inode sizes (Ext4 and XFS use 256 bytes and Btrfs uses 136 bytes), we pessimistically penalize TABLEFS by padding its inode attributes to 256 bytes. This slows down TABLEFS doing metadata-intensive workloads significantly, but it still performs quite well. In some benchmarks, we also changed the Linux boot parameters to limit the machines' available memory below certain threshold, in order to test out-of-RAM performance.

4.2 Data-Intensive Macrobenchmark

We run two sets of macrobenchmarks on the FUSE version of TABLEFS, which provides a full featured, transparent application service. Instead of using a metadata-intensive workload, emphasized in the previous and later sections of this paper, we emphasize data-intensive work in this section. Our goal is to demonstrate that TABLEFS is capable of reasonable performance for the traditional workloads that are often used to test local file systems.

Kernel build is a macrobenchmark that uses a Linux kernel compilation and related operations to compare TABLEFS's performance to the other tested file systems. In the kernel build test, we use the Linux 3.0.1 source tree (whose compressed tar archive is about 73 MB in size). In this test, we run four operations in this order:

- **untar**: untar the source tarball;
- **grep**: grep "nonexistent pattern" over all of the source tree;
- **make**: run *make* inside the source tree;
- **gzip**: gzip the entire source tree.

After compilation, the source tree contains 45,567 files with a total size of 551MB. The machine's available memory is set to be 350MB, and therefore compilation data are forced to be written to the disk.

Figure 5 shows the average runtime of three runs of these four macro-benchmarks using Ext4, XFS, Btrfs and TABLEFS-FUSE. For each macro-benchmark, the runtime is normalized by dividing the minimum value. Summing the operations, TABLEFS-FUSE is about 20% slower, but it is also paying significant overhead caused by moving all data through the user-level FUSE daemon and the kernel twice, instead of only through the kernel once, as illustrated in Figure 4. Table 5 also shows that the degraded performance of Ext4, XFS, and Btrfs when they are accessed through FUSE is about the same as TABLEFS-FUSE.

Postmark was designed to measure the performance of a file system used for e-mail, and web based services [20]. It creates a large number of small randomly-sized files between 512B and 4KB, performs a specified number of transactions on them, and then deletes all of them.

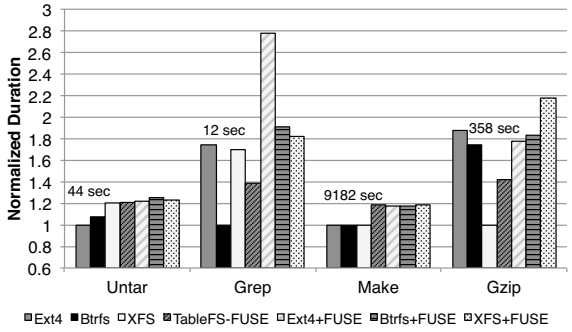


Figure 5: *The normalized elapsed time for unpacking, searching building and compressing the Linux 3.0.1 kernel package. All elapsed time in each operation is divided by the minimum value (1.0 bar). The legends above each bar show the actual minimum value in seconds.*

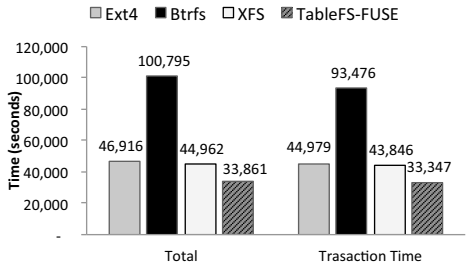


Figure 6: *The elapsed time for both the entire run of Postmark and the transactions phase of Postmark for the four tested file systems.*

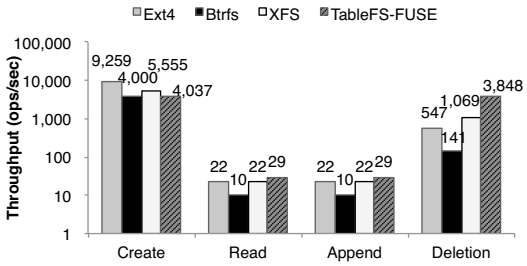


Figure 7: *Average throughput of each type of operation in Postmark benchmark.*

Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The configuration used for these experiments consists of two million transactions on one million files, and the biases for transaction types are equal. The experiments were run with the available memory set to be 1400 MB, too small to fit the entire datasets (about 3GB) in memory.

Figure 6 shows the Postmark results for the four tested file systems. TABLEFS outperforms other tested file systems by at least 23% during the transctions phase. Figure 7 gives the average throughput of each type of operations individually. TABLEFS runs faster than the other tested filesystems for *read*, *append* and *deletion*, but runs slower for the *creation*. In Postmark, *creation* phase is to create files in the alphabatical order of their filenames. Thus the *creation* phase is a sequential insertion workload for all file systems, and Ext4 and XFS perform very efficiently in this workload. TABLEFS-FUSE pays for the overhead from FUSE and writing file data at least twice: LevelDB first time writes it to the write-ahead log, and second time to an SSTable during compaction.

4.3 TABLEFS-FUSE Overhead Analysis

To understand the overhead of FUSE in TABLEFS-FUSE, and estimate the performance of an in-kernel TABLEFS, we ran a micro-benchmark against

TABLEFS-FUSE and TABLEFS-Library ((b) and (c) in Figure 4). This micro-benchmark creates one million zero-length files in one directory starting with an empty file system. The amount of memory available to the evaluation system is 1400 MB, almost enough to fit the benchmark in memory.

Figure 8 shows the total runtime of the experiment. TABLEFS-FUSE is about 3 times slower than TABLEFS-Libary.

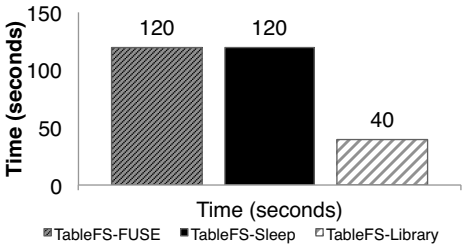


Figure 8: *The elapsed time for creating 1M zero-length files on three versions of* TABLEFS *(See Figure 4)*

Figure 9 shows the total disk traffic gathered from the Linux proc file system (*/proc/diskstats*) during the test. Relative to TABLEFS-Library, TABLEFS-FUSE writes almost as twice as many bytes to the disk, and reads almost 100 times as much. This additional disk traffic results from two sources: 1) under a slower insertion rate, LevelDB tends to compact more often; and 2) the FUSE framework populates the kernel's cache with its own version of inodes, competing with the local file system for cache memory.

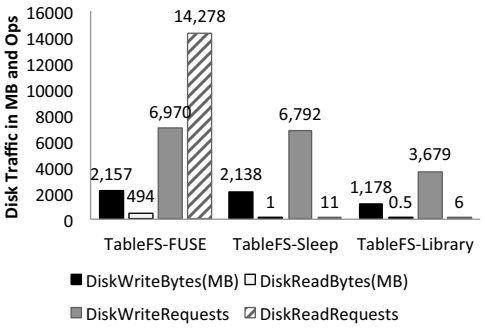
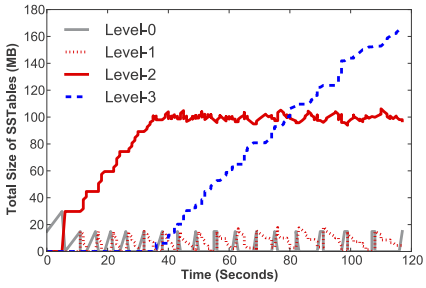
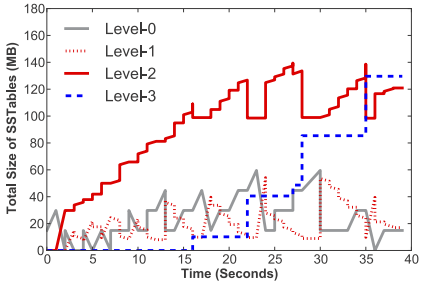


Figure 9: *Total disk traffic associated with Figure 8*

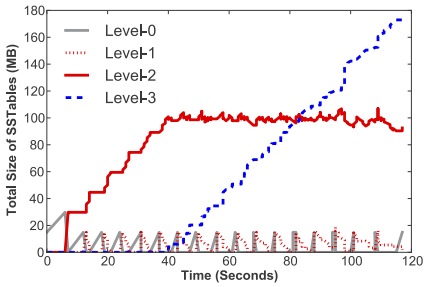
To illustrate the first point, we show LevelDB's compaction process during this test in Figure 10. Figure 10 shows the total size of SSTables in each Level over time. The compaction process will move SSTables from one level to the next level. For each compaction in Level 0, LevelDB will compact all SSTables with overlapping



(a) TABLEFS-FUSE



(b) TABLEFS-Library



(c) TABLEFS-Sleep

Figure 10: *Changes of total size of SSTables in each level over time during the creation of 1M zero-length files for three* TABLEFS *models.* TABLEFS-*Sleep illustrates similar compaction behavior as does* TABLEFS-*FUSE.*

ranges (which in this benchmark will be almost all SSTables in level 0 and 1). At the end of a compaction, the next compaction will repeat similar work, except the number of level 0 SSTables will be proportional to the data insertion rate. When the insertion rate is slower (Figure 10(a)), compaction in Level 0 finds fewer overlapping SSTables than TABLEFS-Library (Figure 10(b)) in each compaction. In Figure 10(b), the level 0 size (blue line) exceeds 20MB for much of the test, while in 10(a) it never exceeds 20MB after the first compaction. Therefore, LevelDB does more compactions to integrate the same arriving log of changes when insertion is slower.

To negate the different compaction work, we deliberately slow down TABLEFS-Library to run at the same

speed as TABLEFS-FUSE by adding *sleep 80ms* every 1000 operations (80ms was empirically derived to match the run time of TABLEFS-FUSE). This model of TABLEFS is called TABLEFS-Sleep and is shown in Figure 9 and 10 (c). TABLEFS-Sleep causes almost the same pattern of compactions as does TABLEFS-FUSE and induces about the same write traffic (Figure 9). But unlike TABLEFS-FUSE, TABLEFS-Sleep can use more of the kernel page cache to store SSTables than TABLEFS-FUSE. Thus, as shown in Figure 9, TABLEFS-Sleep writes the same amount of data as TABLEFS-FUSE but does much less disk reading.

To estimate TABLEFS performance without FUSE overhead, we use TABLEFS-Library to avoid double caching and perform a *write*("*/dev/null*", *N* bytes) on every TABLEFS invocation to model the kernel's system call and argument data transfer overhead. This model is called TABLEFS-Predict and is used in the following sections to predict metadata efficiency of a kernel TABLEFS.

4.4 Metadata-Intensive Microbenchmark

Metadata-only Benchmark

In this section, we run four micro-benchmarks of the efficiency of pure metadata operations. Each micro-benchmark consists of two phases: a) create and b) test. For all four tests, the create phase is the same:

- a) *create*: In "create", the benchmark application generates directories in depth first order, and then creates one million zero-length files in the appropriate parent directories in a random order, according to a realistic or synthesized namespace.

The test phase in the benchmark are:

- b1) *null*: In test 1, the test phase is null because create is what we are measuring.
- b2) *query*: This workload issues one million read or write queries to random (uniform) files or directories. A read query calls *stat* on the file, and a write query randomly does either a *chmod* or *utime* to update the mode or the timestamp attributes.
- b3) *rename*: This workload issues a half million rename operations to random (uniform) files, moving the file to another randomly chosen directory.
- b4) *delete*: This workload issues a half million delete operations to randomly chosen files.

The captured file system namespace used in the experiment was taken from one author's personal Ubuntu desktop. There were 172,252 directories, each with 11 files on average, and the average depth of the namespace is 8 directories. We also used the Impressions tool [4] to generate a "standard namespace". This synthetic namespace yields similar results, so its data is omitted from this paper. Between the create and test phase of each run, we umount and re-mount local filesystems to clear kernel caches. To test out-of-RAM performance, we limit the machine's available memory to 350MB which does not fit the entire test in memory. All tests were run for three times, and the coefficient of variation is less than 1%.

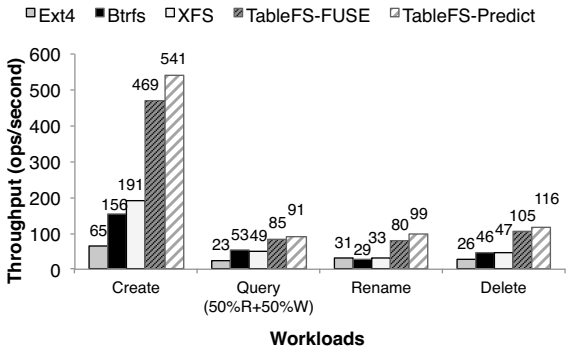


Figure 11: *Average throughput during four different workloads for five tested systems.*

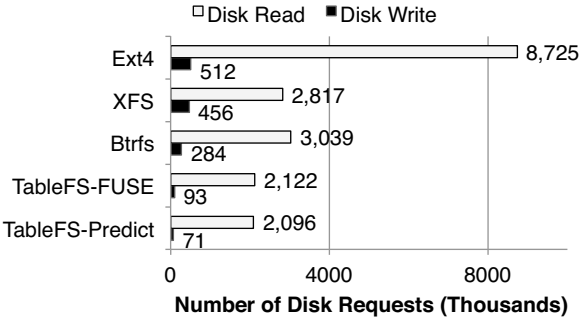


Figure 12: *Total number of disk read/write requests during 50%Read+50%Write query workload for five tested systems.*

Figure 11 shows the test results averaged over three runs. The create phase of all tests had the same performance so we show it only once. For the other tests, we show only the second, test phase. Both TABLEFS-Predict and TABLEFS-FUSE runs are almost 2 to 3 times faster than the other local file systems in all tests.

Figure 12 shows the total number of disk read and write requests during the query workload, the test in which TABLEFS has the least advantage. Both versions of TABLEFS issue many fewer disk writes, effectively aggregating changes into larger sequential writes. For read requests, because of bloom filtering and in-memory

indexing, TABLEFS issues fewer read requests. Therefore TABLEFS's total number of disk requests is smaller than the other tested file systems.

Scan Queries

In addition to point queries such as *stat, chmod* and *utime*, range queries such as *readdir* are important metadata operations. To test the performance of *readdir*, we modify the micro-benchmark to perform multiple *readdir* operations in the generated directory tree. To show the trade-offs involved in embedding small files, we create 1KB files (with random data) instead of zero byte files. For the test phase, we use the following three operations:

- b5) *readdir*: The benchmark application performs *readdir()* on 100,000 randomly picked directories.
- b6) *readdir+stat*: The benchmark application performs *readdir()* on 100,000 randomly picked directories, and for each returned directory entry, performs a *stat* operation. This simulates "ls -l".
- b7) *readdir+read*: Similar to *readdir+stat*, but for each returned directory entry, it reads the entire file (if returned entry is a file) instead of *stat*.

Figure 13 shows the total time needed to complete each *readdir* workload (the average of three runs). In the pure *readdir* workload, TABLEFS-Predict is slower than Ext4 because of read amplification, that is, for each *readdir* operation, TABLEFS fetches directory entries along with unnecessary inode attributes and file data. However, in the other two workloads when at least one of the attributes or file data is needed, TABLEFS is faster than Ext4, XFS, and Btrfs, since many random disk accesses are avoided by embedding inodes and small files.

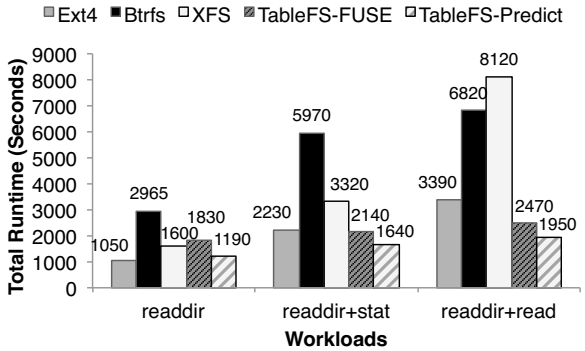


Figure 13: *Total run-time of three readdir workloads for five tested file systems.*

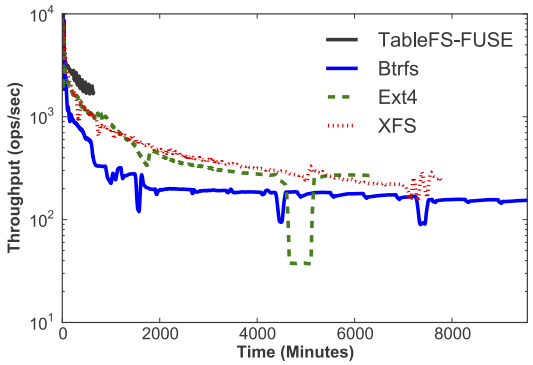


Figure 14: *Throughput of all four tested file systems while creating 100 million zero-length files.* TABLEFS*-FUSE is almost 10× faster than the other tested file systems in the later stage of this experiment. The data is sampled in every 10 seconds and smoothed over 100 seconds. The vertical axis is shown on a log scale.*

Benchmark with Larger Directories

Because the scalability of small files is of topical interest [49], we modified the zero-byte file create phase to create 100 million files (a number of files rarely seen in the local file system today). In this benchmark, we allowed the memory available to the evaluation system to be the full 16GB of physical memory.

Figure 14 shows a timeline of the creation rate for four file systems. In the beginning of this test, there is a throughput spike that is caused by everything fitting in the cache. Later in the test, the creation rate of all tested file systems slows down because the non-existence test in each create is applied to ever larger on-disk data structures. Btrfs suffers the most serious drop, slowing down to 100 operations per second at some points. TABLEFS-FUSE maintains more steady performance with an average speed of more than 2,200 operations per second and *is 10 times faster than all other tested file systems.*

All tested file systems have throughput fluctuations during the test. This kind of fluctuation might be caused by on disk data structure maintenance. In TABLEFS, this behavior is caused by compactions in LevelDB, in which SSTables are merged and sequentially written back to disk.

Solid State Drive Results

TABLEFS reduces disk seeks, so you might expect it to have less benefit on solid state drives, and you'd be right. We applied the "create-query" microbenchmark to a 120GB SATA II 2.5in Intel 520 Solid State Drive (SSD). Random read throughput is 15,000 IO/s at peak, and random write throughput peaks at 3,500 IO/s. Sequential read throughput peaks at 245MB/sec, and se-

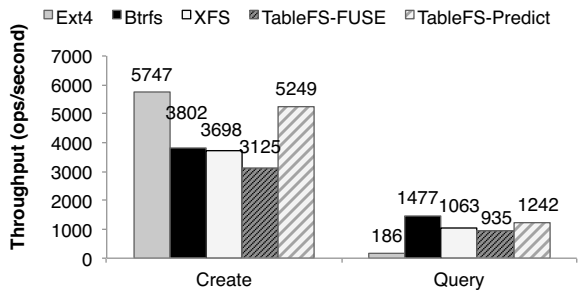
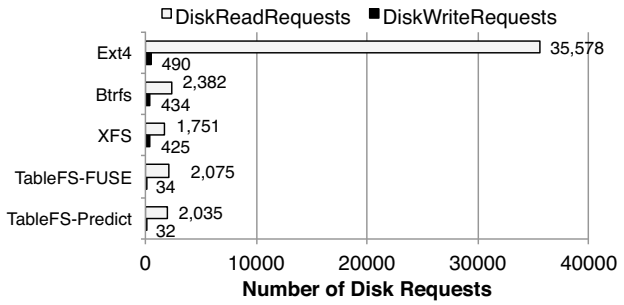
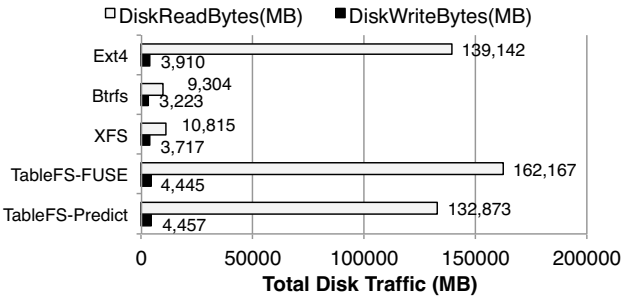


Figure 15: *Average throughput in the create and query workloads on an Intel 520 SSD for five tested file systems.*



(a) Disk Requests



(b) Disk Bytes

Figure 16: *Total number of disk requests and disk bytes moved in the query workload on an Intel 520 SSD for five tested file systems.*

quential write throughput peaks at 107MB/sec. Btrfs has a "ssd" optimization mount option which we enabled.

Figure 15 shows the throughput averaged over three runs of the create and query phases. In comparison to Figure 11, all results are about 10 times faster. Although TABLEFS is not the fastest, TABLEFS-Predict is comparable to the fastest. Figure 16 shows the total number of disk requests and disk bytes moved during the query phase. While TABLEFS achieves fewer disk writes, it reads much more data from SSD than XFS and Btfs. For use with solid state disks, LevelDB can be further optimized to reduce read amplification. For example, using SILT-like fine-grained in-memory indexing [24] can reduce the amount of data read from SSD, and using VT-Tree compaction stitching [45] can reduce compaction works for sequential workloads.

5 Related Work

File system metadata is structured data, a natural fit for relational database techniques. However, because of their large size, complexity and slow speed, file system developers have long been reluctant to incorporate traditional databases into the lower levels of file systems [31, 46]. Modern stacked file systems often expand on the limited structure in file systems, hiding structures inside directories meant to represent files [6, 14, 15, 21], even though this may increase the number of small files in the file system. In this paper, we return to the basic premise, file system metadata is a natural fit for table-based representation, and show that today's lightweight data stores may be up to the task. We are concerned with an efficient representation of huge numbers of small files, not strengthening transactional semantics [16, 19, 40, 45, 50].

Early file systems stored directory entries in a linear array in a file and inodes in simple on-disk tables, separate from the data of each file. Clustering within a file was pursued aggressively, but for different files clustering was at the granularity of the same cylinder group. It has long been recognized that small files can be packed into the block pointer space in inodes [29]. C-FFS [12] takes packing further and clusters small files, inodes and their parent directory's entries in the same disk readahead unit, the track. A variation on clustering for efficient prefetching is replication of inode fields in directory entries, as is done in NTFS[9]. TABLEFS pursues an aggressive clustering strategy; each row of a table is ordered in the table with its parent directory, embedding directory entries, inode attributes and the data of small files. This clustering manifests as adjacency for objects in the lower level object store if these entries were created/updated close together in time, or after compaction has merge sorted them back together.

Beginning with the Log-Structured File System (LFS)[37], file systems have exploited write allocation methods that are non-overwrite, log-based and deferred. Variations of log structuring have been implemented in NetApp's WAFL, Sun's ZFS and BSD UNIX [3, 18, 44]. In this paper we are primarily concerned with the disk access performance implications of non-overwrite and log-based writing, although the potential of strictly ordered logging to simplify failure recovery in LFS has been emphasized and compared to various write ordering schemes such as Soft Updates and Xsyncfs [27, 30, 43]. LevelDB's recovery provisions are con-

sistent with delayed periodic journalling, but could be made consistent with stronger ordering schemes. It is worth noting that the design goals of Btrfs call for non-overwrite (copy-on-write) updates to be clustered and written sequentially[36], largely the same goals of LevelDB in TABLEFS. Our measurements indicate, however, that the Btrfs implementation ends up doing far more small disk accesses in metadata dominant workloads.

Partitioning the contents of a file system into two groups, a set of large file objects and all of the metadata and small files, has been explored in hFS[51]. In their design large file objects do not float as they are modified, and hFS uses modified log-structured file system approach and an in-place B-Tree to manage metadata, directory entries and small files. TABLEFS has this split as well, with large file objects handled directly by the backing object store, and metadata updates approximately log structured in LevelDB's partitioned LSM tree. However, TABLEFS uses a layered approach and does not handle disk allocation, showing that metadata performance of widely available and trusted file systems can be greatly improved even in a less efficient stacked approach. Moreover, hFS's B-Tree layered on LFS approach is similar to Btrfs' copy-on-write B-Tree, and our experiments show that the LSM approach is faster than the Btrfs approach.

Log-Structured Merge trees [32] were inspired in part by LFS, but focus on representing a large B-tree of small entries in a set of on-disk B-trees constructed of recent changes and merging these on-disk B-trees as needed for lookup reads or in order to merge on-disk trees to reduce the number of future lookup reads. LevelDB [23] and TokuFS [11] are LSM trees. Both are partitioned in that the on-disk B-trees produced by compaction cover small fractions of the key space, to reduce unnecessary lookup reads. TokuFS names its implementation a Fractal Tree, also called streaming B-trees[5], and its compaction may lead to more efficient range queries than LevelDB's LSM tree because LevelDB uses Bloom filters[7] to limit lookup reads, a technique appropriate for point lookups only. If bounding the variance on insert response time is critical, compaction algorithms can be more carefully scheduled, as is done in bLSM[42]. TABLEFS may benefit from all of these improvements to LevelDB's compaction algorithms, which we observe to sometimes consume disk bandwidth injudiciously (See Section 4.3).

Recently, VT-trees [45] were developed as a modification to LSM trees to avoid always copying old SSTable content into new SSTables during compaction. These trees add another layer of pointers so new SSTables can point to regions of old SSTables, reducing data copying but requiring extra seeks and eventual defragmentation.

6 Conclusion

File systems for magnetic disks have long suffered low performance when accessing huge collections of small files because of slow random disk seeks. TABLEFS uses modern key-value store techniques to pack small things (directory entries, inode attributes, small file data) into large on-disk files with the goal of suffering fewer seeks when seeks are unavoidable. Our implementation, even hampered by FUSE overhead, LevelDB code overhead, LevelDB compaction overhead, and pessimistically padded inode attributes, performs as much as 10 times better than state-of-the-art local file systems in extensive metadata update workloads.

Acknowledgment

This research is supported in part by The Gordon and Betty Moore Foundation, National Science Foundation under awards, SCI-0430781, CCF-1019104, CNS-1042537 and CNS-1042543 (PRObE http://www.nmc-probe.org/), Qatar National Research Foundation 09-1116-1-172, DOE/Los Alamos National Laboratory, under contract number DE-AC52-06NA25396/161465-1, by Intel as part of ISTC-CC. We thank the member companies of the PDL Consortium for their feedback and support.

References

[1] FUSE. http://fuse.sourceforge.net/.

[2] Memcached. http://memcached.org/.

[3] ZFS. http://www.opensolaris.org/os/community/zfs.

[4] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. In *Proccedings of the 7th conference on file and storage technologies* (2009).

[5] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming B-trees. In *Proceedings of annual ACM symposium on parallel algorithms and architectures* (2007).

[6] BENT, J., GIBSON, G., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE conference on Supercomputing* (2009).

[7] BLOOM, B. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7* (1970).

[8] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (2006).

[9] CUSTER, H. Inside the windows NT file system. *Microsoft Press* (1994).

[10] DAYAL, S. Characterizing HEC storage systems at rest. In *Carnegie Mellon University, CMU-PDL-08-109* (2008).

[11] ESMET, J., BENDER, M., FARACH-COLTON, M., AND KUSZMAUL, B. The TokuFS streaming file system. *Proceedings of the USENIX conference on Hot Topics in Storage and File Systems* (2012).

[12] GANGER, G. R., AND KAASHOEK, M. F. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1997).

[13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th ACM symposium on Operating systems principles* (2003).

[14] HALCROW, M. A. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. *Proc. of the Linux Symposium, Ottawa, Canada* (2005).

[15] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011).

[16] HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in quicksilver. *In Proceedings of the Eleventh ACM Symposium on Operating System Principles* (1987).

[17] HDFS. Hadoop file system. `http://hadoop.apache.org/`.

[18] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference* (1994).

[19] KASHYAP, A. File system extensibility and reliability using an in-kernel database. *Master Thesis, Computer Science Department, Stony Brook University* (2004).

[20] KATCHER, J. Postmark: A new file system benchmark. In *NetApp Technical Report TR3022* (1997).

[21] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (2012).

[22] KRA, J. Ext4, BTRFS, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference* (2009).

[23] LEVELDB. A fast and lightweight key/value database library. `http://code.google.com/p/leveldb/`.

[24] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011).

[25] LUSTRE. Lustre file system. `http://www.lustre.org/`.

[26] MATHUR, A., CAO, M., AND BHATTACHARYA, S. The new Ext4 lesystem: current status and future plans. In *Ottawa Linux Symposium* (2007).

[27] MCKUSICK, M. K., AND GANGER, G. R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. *Proceedings of the annual conference on USENIX Annual Technical Conference, FREENIX Track* (1999).

[28] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and Storage Technologies* (2011).

[29] MULLENDER, S. J., AND TANENBAUM, A. S. Immediate files. *SoftwarePractice and Experience* (1984).

[30] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. *ACM Transactions on Computer Systems, Vol.26, No.3 Article 6* (2008).

[31] OLSON, M. A. The design and implementation of the Inversion file system. In *USENIX Winter Technical Conference* (1993).

[32] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica* (1996).

[33] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM symposium on Operating systems principles* (2011).

[34] PATIL, S., AND GIBSON, G. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of USENIX Conference on File and Storage Technologies* (2011).

[35] RODEH, O. B-trees, shadowing, and clones. *Transactions on Storage* (2008).

[36] RODEH, O., BACIK, J., AND MASON, C. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501 (ALM1207-004)* (2012).

[37] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles* (1991).

[38] ROSS, R., AND LATHAM, R. PVFS: a parallel file system. In *Proceedings of the ACM/IEEE conference on Supercomputing* (2006).

[39] SCHMUCK, F. B., AND HASKIN, R. L. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX conference on file and storage technologies* (2002).

[40] SEARS, R., AND BREWER, E. A. Stasis: Flexible transactional storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2006).

[41] SEARS, R., INGEN, C. V., AND GRAY, J. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? *Microsoft Technical Report* (2007).

[42] SEARS, R., AND RAMAKRISHNAN, R. bLSM: a general purpose log structured merge tree. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2012).

[43] SELTZER, M., GANGER, G., MCKUSICK, K., SMITH, K., SOULES, C., AND STEIN, C. Journaling versus soft updates: Asynchronous meta-data protection in file systems. *Proceedings of the annual conference on USENIX Annual Technical Conference* (2000).

[44] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. *USENIX Winter Technical Conference* (1993).

[45] SHETTY, P., SPILLANE, R., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-Trees. In *Proccedings of the 11th conference on file and storage technologies* (2013).

[46] STONEBRAKER, M. Operating System Support for Database Management. *Commun. ACM* (1981).

[47] SWEENEY, A. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference* (1996).

[48] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX conference on File and Storage Technologies* (2008).

[49] WHEELER, R. One billion files: pushing scalability limits of linux filesystem. In *Linux Foudation Events* (2010).

[50] WRIGHT, C. P., SPILLANE, R., SIVATHANU, G., AND ZADOK, E. Extending ACID Semantics to the File System. *ACM Transactions on Storage* (2007).

[51] ZHANG, Z., AND GHOSE, K. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (2007).

Characterization of Incremental Data Changes for Efficient Data Protection

Hyong Shim, Philip Shilane, and Windsor Hsu
Backup Recovery Systems Division
EMC Corporation

Abstract

Protecting data on primary storage often requires creating secondary copies by periodically replicating the data to external target systems. We analyze over 100,000 traces from 125 customer block-based primary storage systems to gain a high-level understanding of I/O characteristics and then perform an in-depth analysis of over 500 traces from 13 systems that span at least 24 hours. Our analysis has the twin goals of minimizing overheads on primary systems and improving data replication efficiency. We compare our results with a study a decade ago [20] and provide fresh insights into patterns of incremental changes on primary systems over time.

Primary storage systems often create snapshots as point-in-time copies in order to support host I/O while replicating changed data to target systems. However, creating standard snapshots on a primary storage system incurs overheads in terms of capacity and I/O, and we present a new snapshot technique called a **replication snapshot** that reduces these overheads. Replicated data also requires capacity and I/O on the target system, and we investigate techniques to significantly reduce these overheads. We also find that highly sequential or random I/O patterns have different incremental change characteristics. Where applicable, we present our findings as advice to storage engineers and administrators.

1 Introduction

Protecting data on primary storage systems often requires periodically creating secondary copies by transferring changed data to external target systems, which may be in the same facility or remotely located. However, as the size of data to be protected continues to grow exponentially, the traditional approach to data protection, e.g., copying all the data on the primary storage system to a target system (such as backup servers) at regular intervals, is fast becoming infeasible. A better approach is to only copy the data blocks that have been modified since the last transfer, unlike standard backup software that copies modified files or whole directories. So, understanding how data changes on primary storage over time is key to both improving existing data protection solutions and enabling new solutions.

Specifically, we analyzed the size, rate, and pattern of data changes over time under various host I/O access patterns on EMC Symmetrix VMAX systems [8], a tier-1 block-based primary storage system. We analyzed over 100,000 traces from 125 enterprise systems from some of the world's largest corporations to gain high-level insights into storage characteristics. We then selected over 500 traces that spanned at least 24 hours from 13 systems to analyze various incremental transfer intervals. We believe the number of traces and systems used for analysis is substantially larger than in previously published studies and our results are of value to any organization designing or configuring data protection architectures.

Replicating changed data from a primary system to a target system may take a substantial amount of time, depending on the change rate and transfer throughput. During the transfer period, the primary system must maintain the point-in-time version of storage until the transfer completes, even while hosts write to the primary system. Snapshots [2, 5, 10, 22] are a general purpose mechanism to capture the point-in-time view of data, and transferring snapshots to target storage is one technique for data protection [20]. As two examples, snapshots kept within primary storage allow a user to recover accidentally deleted files, and snapshots are increasingly used to maintain a consistent state of the system to be copied to target storage while a primary system continues operation. We have focused our analysis on snapshot overheads when used for replication.

We found that using standard snapshots for replication incurs significant overhead in terms of space usage and I/O. We observe that only the point-in-time state of the changed blocks (instead of all of the blocks) needs to be maintained, so we can relax the semantics of snapshots, which we call a **replication snapshot**. A replication snapshot protects the changed blocks that need to be replicated without necessarily maintaining the values of blocks that do not need to be copied to target storage. Typical snapshot implementations are designed to create semi-persistent versions, while replication snapshots are designed specifically to support periodic replication and are then released. Also, implementing replication snapshots along with a replication protocol allows separate primary storage and target storage vendors to jointly support efficient replication.

Storage overheads on primary storage can be avoided when host writes are protected with a synchronous remote mirroring mechanism [14], in which host writes are, in effect, sent to both primary and target storage.

1. 8% of capacity needs to be reserved for snapshot overheads to support incremental transfers every 12 hours. The reserve is as low as 2% of capacity with replication snapshots.
2. Primary I/O should be over-provisioned by 100% to support copy-on-write related write-amplification of host writes during replication. The over-provision can be as low as 20% with a replication snapshot.
3. Having a write buffer effectively decreases snapshot I/O overheads but has little impact on storage overheads.
4. The daily transfer size with small blocks is generally 40% of what hosts write.
5. Scheduling at least 6 hours between transfers allows blocks to achieve nearly peak dirtiness.
6. Scheduling at least 12 hours between transfers drastically reduces peak network bandwidth requirements. Volume capacity is not predictive of bandwidth requirements.
7. Target storage must support as much as 20% of the I/O per second capabilities of primary storage when the replication interval is at least one hour.

Table 1: Rules-of-thumb from our analysis

Such a mechanism, however, typically requires that target system have storage capacity and I/O performance similar to those of the primary system, which does not scale well to transferring data changes over a long distance to protect against site disasters. Our analysis focuses on data-protection cases where target systems have larger capacity but potentially lower I/O capabilities than primary systems. This is because data protection systems must be large enough to hold multiple versions of primary storage such as daily copies for a month or longer. As guidance to storage engineers and administrators, we summarize our findings in a set of rules-of-thumb, which are presented in Table 1. Our contributions include: a detailed analysis of data change characteristics for a large set of traces collected from deployed systems, a design for replication snapshots to reduce overheads on primary storage, and an evaluation of overheads on primary and target storage to guide design and configuration.

A related study by Patterson et al. [20] investigated how to efficiently create primary system snapshots at remote systems. The main differences between the present work and Patterson's include our investigation of using various units of data aggregation to transport changed data and their impact on the size of transferred data and I/O rate on the target system. We also investigate how incremental data changes are impacted by different host write I/O patterns used to produce the data change. Importantly, it has been a decade since the earlier study, and it is worth revisiting this analysis to understand how I/O properties have changed using a newer, larger set of traces.

2 Collected Traces

We collected I/O traces from over 100,000 logical volumes from 125 EMC Symmetrix VMAX [8] systems installed at enterprise customer sites. The number of logical volumes captured for each primary storage system ranged from 12 to over 14,000. These systems supported database, email, file system, and other business applications. Unfortunately, no other information is available regarding which applications wrote to and read from which logical volumes. While such information would have been useful, enterprise primary storage systems should be designed to support a wide range of applications.

Traced data includes sector-level read/write I/O requests received by primary storage systems as applications performed I/O operations on their hosts connected to the primary systems in, for example, storage area networks (SANs). Traced data was collected into a trace file per volume. The trace file (or simply trace) contains a number of records, each of which contains the following data fields: timestamp from the beginning of the trace, read/write command, port at which I/O is received, logical volume number, logical sector address (ranging from 0 to largest address), and number of sectors to read or write.

Table 2 and Table 3 summarize I/O activities, rate, and throughput in the traced systems. See the captions of the tables for the descriptions of analyzed I/O properties. Each row of the tables corresponds to a subset of logical volumes that share some common properties and are analyzed together. The trace sets are:

1hr_1Wrt: logical volumes traced for at least 1 hour and that received at least 1 write I/O

1hr_1GBWrt: a subset of **1hr_1Wrt**, which includes volumes traced for at least 1 hour and that received at least 1GB worth of writes

24hr_1GBWrt a subset of **1hr_1GBWrt**, which includes volumes traced for at least 24 hours and that received at least 1GB worth of writes

24hr_1GBWrt_Random: a subset of **24hr_1GBWrt**, which includes volumes that received largely random write I/O requests (See Section 2.1)

24hr_1GBWrt_Sequential: a subset of **24hr_1GBWrt**, which includes volumes that received largely sequential write I/O requests (See Section 2.1)

The **24hr_1GBWrt*** trace sets were selected for detailed analysis because they provide a consistent basis for a wide range of simulations across replication intervals.

As the large standard deviations in the tables indicate, the traced volumes widely vary in host I/O activities they supported. The tables do confirm the long-held view that hosts issue more read I/O requests than write I/O requests

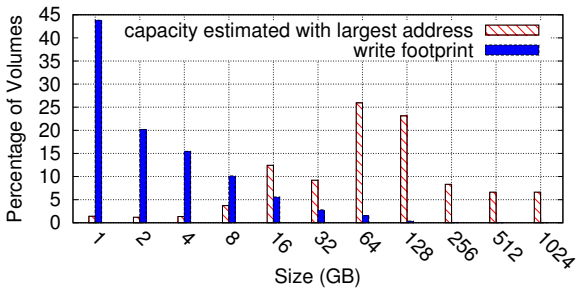


Figure 1: Storage capacity estimates and write footprint for **1hr_1GBWrt**.

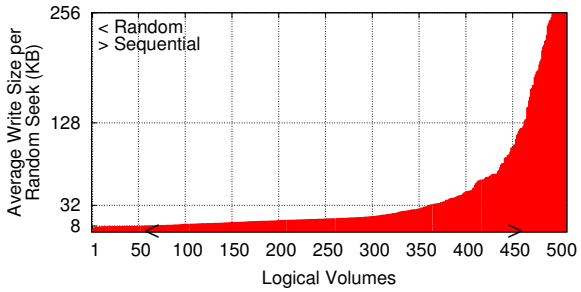


Figure 2: Average write size per random seek. We define random and sequential volumes as having <8.5KB and >102KB average writes per seek, respectively.

and (not surprisingly) read more data than they write. Note in the *W_rlen* column of Table 2 that the average run length of dirty data written after a random seek is much longer for **1hr_1Wrt** and **1hr_1GBWrt** trace sets than for the **24hr_1GBWrt** sets. This is because a small fraction of volumes included in these sets received long bursts of sequential writes that skewed the average values for the entire sets. This can be seen in the standard deviations, which are 10 times the corresponding averages. Such volumes are excluded from the **24hr_1GBWrt** sets as their trace periods are shorter than 24 hours.

We do not have access to the configured volume size, so the storage capacity of each volume is estimated with the largest logical address found in the corresponding trace. Figure 1 shows the estimated storage capacity distribution for the 16,100 volumes in the **1hr_1GBWrt** set. For comparison, we also show the write footprint distribution as percentage of volumes. The write footprint is the number of unique sectors written converted to bytes. Most volumes only had a few gigabytes of unique writes, though volumes were estimated as hundreds of gigabytes in capacity.

2.1 Sequential vs Random I/O

To determine if host I/O pattern has any significant impact on our major findings, we further distinguish traces in the **24hr_1GBWrt** set into **sequential** and **random**. Intuitively, a sequential trace is the result of a host writing data to consecutive locations. From surveying the literature, we have found multiple definitions of sequential I/O (e.g., [1, 4, 12, 17, 21, 23, 24]). For our metric, we measure how much data are written, on average, after seeking to a random sector. By random sector, we mean a sector that is not consecutive with the last sector written based on logical address.

Figure 2 shows the average sequential write size after a random seek for >500 logical volumes in **24hr_1GBWrt**. The volumes are arranged on the x-axis in increasing order of the average sequential write size. Towards the right end of the x-axis, hosts write >102KB of data in sequence after making a random seek in 11% of the volumes. Towards the left end of the x-axis, hosts write <8.5KB of data in sequence after making a random seek in 11% of the volumes. Unfortunately, there is not a clear division between sequential and random host I/O shown in the figure, so for the purpose of our analysis, we use the average sequential write sizes of >102KB and <8.5KB as threshold values in determining sequential volumes and random volumes respectively. The sequential and random volumes are denoted as **24hr_1GBWrt_Sequential** and **24hr_1GBWrt_Random** in Table 2. In the remainder of the paper, the 24hr_1GBWrt trace set is referred to as **All**, 24hr_1GBWrt_Sequential as **Seq** and 24hr_1GBWrt_Random as **Random**.

One drawback of our definition of sequential access is that it does not account for interleaving writes from different hosts because our tracing was lower in the storage system. Nevertheless, we have adopted this approach based on our observation that sequential write I/O requests often appear together in sequence in trace files. Another potential weakness of our trace analysis is that we have not specifically analyzed time-of-day effects. Partly, this is an artifact of our trace collection process that has retained relative, not absolute time stamps, so our replication intervals begin at the start of each trace. Since we have a relatively large number of traces, such effects are likely averaged out, but a future analysis could clarify the impact by comparing results after offsetting the start time.

3 Analysis Methodology

While analyzing logical volume traces, we have tracked incremental changes over time and measured various statistics. Our simulation entails three main components: **replication intervals**, **blocks**, and **transfer throughput**. The top half of Figure 3 illustrates host I/O as a sequence of writes and reads, and the bottom half shows affected sectors and blocks in a logical volume.

A **replication interval** simulates the fact that data protection mechanisms in primary systems often keep track of dirty data for a user-defined period of time and replicates dirty data to target storage at the end of each pe-

Trace Set	#Vol.	#Sys.	Dur. (hrs)	Est.Cap. (GB)	#W_reqs (1000s)	W_size (GB)	W_fp (GB)	W_rlen (KB)	#R_reqs (1000s)	R_size (GB)	R_fp (GB)	R_rlen (KB)
1hr_1Wrt	109263	125	30.4 [78.3]	71 [203]	72.2 [510.4]	1.7 [31.0]	0.7 [11.2]	947.0 [9230.3]	166.5 [1962.8]	5.2 [65.7]	2.2 [18.9]	667.0 [11301.9]
1hr_1GBWrt	16100	120	7.7 [6.7]	132 [262]	429.0 [1270.7]	10.7 [80.1]	4.6 [28.9]	948.5 [9270.8]	796.0 [4986.7]	24.9 [166.3]	9.8 [45.0]	491.2 [11065.7]
24hr_1GBWrt All	508	13	24.4 [1.2]	318 [439]	1802.8 [4838.7]	51.1 [337.6]	19.9 [103.7]	284.6 [256.1]	7824.3 [23875.4]	241.5 [763.2]	91.3 [172.1]	132.7 [3078.8]
24hr_1GBWrt Random	58	9	24.2 [0.8]	238 [328]	1365.5 [1819.7]	9.9 [13.8]	7.2 [12.1]	8.1 [0.4]	5677.1 [8587.6]	97.0 [111.2]	66.5 [84.2]	35.6 [27.3]
24hr_1GBWrt Sequential	54	9	24.9 [1.4]	343 [591]	2542.1 [7567.2]	280.2 [993.9]	102.6 [301.8]	461.4 [193.4]	2292.1 [7533.8]	247.8 [963.5]	64.1 [191.3]	687.9 [9118.1]

Table 2: Summary of I/O activities. The first four columns denote the number of logical volumes in a trace set, the number of primary systems, the average trace period, and the average estimated storage capacity. The rest of the columns show average I/O requests the host has issued. Footprint (*fp*) is the sum of unique sectors written or read at least once, while run length (*rlen*) indicates the average size of data accessed in sequence after a random seek. The values in square brackets are standard deviations for the corresponding averages.

	I/O Request Rate (1000s/sec)						I/O Request Throughput (MB/sec)					
Trace Set	Avg. W_rate	Peak W_rate 1 sec	Peak W_rate 10 ms	Avg R_rate	Peak R_rate 1 sec	Peak R_rate 10 ms	Avg. W_tput	Peak W_tput 1 sec	Peak W_tput 10 ms	Avg. R_tput	Peak R_tput 1 sec	Peak R_tput 10 ms
1hr_1Wrt	0.0007 [0.008]	0.2 [0.6]	1.8 [2.6]	0.002 [0.03]	0.3 [0.8]	1.7 [2.5]	0.02 [0.4]	6.5 [26.0]	64.0 [1669.7]	0.05 [0.8]	10.5 [85.5]	107.3 [8089.5]
1hr_1GBWrt	0.02 [0.04]	0.9 [1.3]	4.4 [4.4]	0.03 [0.1]	0.9 [1.4]	3.6 [4.1]	0.4 [1.8]	30.2 [60.6]	224.1 [4342.7]	0.9 [3.7]	32.8 [216.2]	359.7 [21K]
24hr_1GBWrt All	0.02 [0.06]	1.5 [1.8]	9.0 [8.2]	0.09 [0.3]	2.0 [2.5]	5.6 [7.0]	0.6 [3.9]	44.3 [76.7]	325.0 [460.7]	2.8 [8.8]	122.42 [1188.2]	5644.6 [119K]
24hr_1GBWrt Random	0.02 [0.02]	1.6 [1.4]	6.8 [5.6]	0.07 [0.1]	1.3 [1.0]	4.2 [3.9]	0.1 [0.2]	15.9 [13.5]	143.4 [326.6]	1.1 [1.3]	32.5 [50.0]	166.5 [316.9]
24hr_1GBWrt Sequential	0.03 [0.08]	1.2 [1.7]	5.3 [4.9]	0.03 [0.08]	1.5 [2.0]	4.3 [4.3]	3.2 [11.4]	98.1 [121.1]	584.7 [817.4]	2.8 [11.1]	70.4 [107.1]	517.6 [880.5]

Table 3: Summary of I/O rate and throughput. The peak values for each volume are selected by considering every 10ms and 1 second period. The peak values for a given set are the average of peak values of individual volumes.

riod. In our trace analysis, we model how host write I/O requests are collected for a given replication interval, and one or more dirty sectors are determined from those requests. Reads are ignored. We have used the following replication intervals for analysis in this paper: 24 hours, 12 hours, 6 hours, 3 hours, 1 hour, 30 minutes, and 15 minutes. We have performed some analysis down to 1 minute replication intervals, though to simplify figures, we generally do not show the intervals below 15 minutes. Organizations typically select a replication interval based on their recovery point objective, which defines the time period for which they can tolerate losing data changes due to a disaster. Organizations would like to shrink the replication interval to as short as possible while considering the cost and infrastructure requirements.

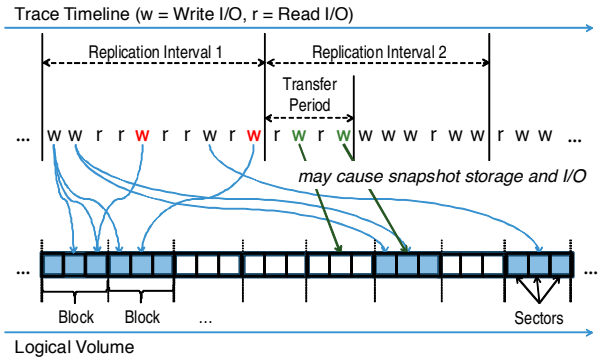


Figure 3: Example of processing a logical volume trace by removing read requests and recording affected sectors and blocks. The red 'w' indicates overwriting requests.

In addition, as shown in Figure 3, dirty sectors are mapped to a larger unit, called a **block** in our model. A block is a sequence of n consecutive sectors in logical volume space, where $n >= 1$. Blocks simulate the fact that many storage systems and data protection mechanisms aggregate dirty sectors into a larger unit and copy those units when replicating modified data to target storage. They do so to reduce memory and storage resources required to maintain, for example, a map of dirty sectors. Block sizes around 128KB are common in some storage systems [26]. For a 1TB volume, the memory requirements for bit vector tracking are: 512B blocks require 256MB of RAM, 128KB blocks require 1MB of RAM, and 1MB blocks require 128KB of RAM.

A block is called 'dirty' if it has one or more dirty sectors, and the figure shows dirty blocks in a darker shade

for Replication Interval 1. When determining the number of dirty sectors (and blocks) during a replication interval, over-writes to a given sector (and block) are counted once. For space and figure clarity reasons, we only show the results for extreme block-size values of 512B and 1MB, unless otherwise noted, because results changed in a gradual manner with block size.

We also consider the impact of **transfer throughput** within our model, which we define very broadly as the throughput from reading dirty blocks on the primary system, transferring across a network (LAN or WAN), and storing on a target system. Based on the transfer throughput and amount of data to transfer, we can determine the transfer period (see Figure 3) during which a primary system must maintain a consistent view of dirty blocks until transfer completes. I/O from the host during the transfer period may cause snapshot I/O (shown in the figure), and those modifications will be recorded and transferred at the end of Replication Interval 2. Note that all modified blocks will be transferred, but because of the point-in-time nature of transferring a snapshot, we have to carefully manage which version of a block exists when a snapshot is created. Managing multiple block versions causes snapshot I/O and storage overheads.

We have analyzed throughputs between 1.5Mb/s (T1) and 40Gb/s and typically discuss results for 1.5Mb/s (T1) and 1Gb/s representing WAN and LAN scenarios, respectively. Note that this throughput is per volume, and storage systems can have over 10,000 volumes. If all 10,000 volumes were replicated at T1 bandwidth individually, this would require 15Gb/s, which is impractical for many customers. Even with that consideration, our analysis provides general results for volumes selected for replication.

With the described trace analysis methodology and storage system model, we can determine how much data should be copied to a target system at the end of each replication interval. To determine the number of write I/O requests needed to copy the data, we assume the underlying data transfer protocol has an upper limit on transfer size, which is assumed to be 1MB, so a larger data run is split.

4 Findings for Primary Storage

At the end of a replication interval, the primary system begins transferring changed blocks to the target, which can take seconds to hours depending on the replication interval, the number of changed blocks, and transfer throughput. During that time, the primary system must maintain an accurate point-in-time representation of those changed blocks, while also supporting incoming host writes that may be directed at blocks that are in the process of being transferred as well as blocks not being transferred. In this section, we characterize storage and I/O overheads for primary storage while changed blocks are transferred to target storage under a variety of configurations.

For a logical volume, snapshots are a general purpose technique to preserve the values for all sectors, usually with a mapping from logical to physical sector addresses [2, 5, 10, 22]. Snapshots are often used to preserve copies on a primary system but are also increasingly being leveraged indirectly for data protection. While there are multiple ways snapshots could be implemented, copy-on-write and redirect-on-write are two prevalent implementations. Suppose snapshot s_t is created at time t. A host write to the volume at time $t+1$ causes the version of the block at time t to be copied into the snapshot (copy-on-write) or the write at $t+1$ is redirected to a snapshot (redirect-on-write). Snapshot s_t has meta data indicating whether the appropriate version of a block is in the main volume or exists in a snapshot region.

Depending on how sectors are modified, both snapshots techniques could be close to empty (no modified sectors) or as large as the active volume (all modified blocks). In terms of I/O, copy-on-write requires I/O to perform the read and write of the earlier block value. Redirect-on-write may require I/O for read-modify-write when a write is less than the block size, and redirect-on-write affects data locality. Creating a clone is an alternative to creating a snapshot, but a clone is less space efficient because it is a full point-in-time copy.

4.1 Replication Snapshot

While this paper focuses on transferring changed blocks, standard snapshot functionality is not designed for this purpose in that any incoming write I/O causes a copy-on-write or redirect-on-write. For replication snapshots, we finely track which blocks need to be transferred for a given replication interval (those that have changed since the last transfer). Only application writes to those blocks cause copy-on-write or redirect-on-write during the time it takes for a transfer to complete. Application writes to non-tracked blocks can happen normally, and all modifications will be transferred in the next replication interval. We present results from the baseline snapshot approach as well as from two versions of replication snapshots, which relax some of the requirements for generic snapshots such that only data that needs to be transferred are tracked.

When describing snapshot techniques, we refer to an example volume shown in Figure 4. Blocks shaded in blue are changed at the end of a replication interval and need to be transferred. Also, their values need to be preserved until replication completes while allowing host I/O to continue.

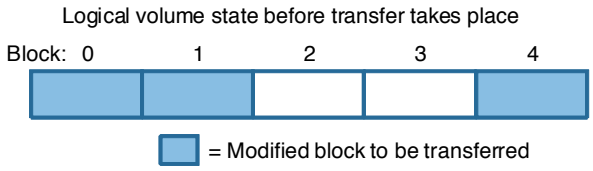


Figure 4: Changed blocks 0, 1, and 4 are transferred to target storage at the end of a replication interval, and a snapshot maintains their state while host I/O continues.

Baseline Snapshot: The standard approach performs snapshot I/O for all incoming host writes regardless of whether the affected block is being transferred or not. In Figure 4, host writes to any block $(0-4)$ result in snapshot I/O the first time. All blocks are released from snapshot protection once the three changed blocks are transferred.

Changed Block Replication Snapshot (CB): Only the changed blocks being transferred at the end of a replication interval are tracked, so host write I/O to these blocks causes snapshot I/O. Importantly, host write I/O to clean blocks is processed without snapshot I/O. In our example, host writes to blocks 0, 1, and 4 cause snapshot I/O, but writes to blocks 2 and 3 do not. All blocks are released from snapshot protection once the three changed blocks are transferred.

Changed Block with Early Release Replication Snapshot (CBER): Similar to the previous version, only changed blocks are tracked, but a block is released from replication snapshot tracking immediately once it is transferred, instead of waiting for the entire transfer to complete. In the figure, host writes to blocks 0, 1, and 4 will cause a snapshot I/O only if those blocks have not yet been transferred based on block-by-block tracking of transfer status.

Note that for all three snapshot versions, repeated host I/O to the same block only causes a single snapshot I/O. Also, the amount of data transferred is identical for all three snapshot techniques. The only difference is the overhead for snapshot I/O and storage. A property affecting snapshot performance is the transfer throughput, which affects how long a snapshot persists. In simulation, we have explored a range of throughputs described in Section 3 but only present a subset of results due to space limitations.

While CBER has lower overheads than CB in our experiments, there is extra tracking information required. There is also more communication with target storage to confirm when individual blocks have been transferred so that blocks can be released from replication snapshot tracking. We leave such analysis to future work. Depending on specific storage system implementations, one type of replication snapshot may be more appropriate than another.

4.2 Storage Overhead

We performed experiments to measure the amount of extra storage space required for blocks written due to snapshots, which is the same for copy-on-write and redirect-on-write. This storage overhead is required to maintain block values while changed blocks are transferred to a target system. Figure 5 shows results for a throughput of 1.5Mb/s for block sizes of 512B and 1MB and three snapshot alternatives for Random 5a, All 5b, and Sequential 5c hosts.

For all configurations, as the replication interval increases on the horizontal axis from 15 minutes to 12 hours, the average fraction of capacity required for snapshots increases. For Figure 5b, we see an average storage overhead of 8% for the Baseline approach with 1MB blocks at 12 hours, and we have even found a peak overhead of 100% in some traces. Unsurprisingly, we see a consistent pattern that the storage overhead is larger for 1MB blocks than 512B blocks. Replication snapshot techniques such as CB and CBER reduce storage overhead because of finer-grained tracking of block transfer state. Considering 1MB blocks at 12 hours, storage overheads decrease from 8% to 4% to 2% respectively, and we see the same trend for 512B blocks.

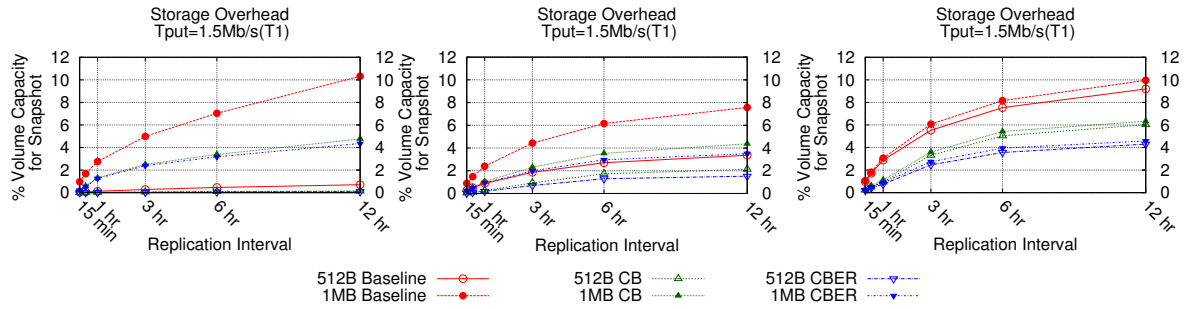
Our general conclusions hold for Random and Sequential traces, though there are several interesting differences. For Random traces, 512B blocks have very low capacity overheads because of the lower change rate for Random traces. For Sequential traces, the block size has little impact because blocks tend to be fully dirty.

Although not shown for space reasons, the trends are identical at a higher throughput of 1Gb/s. Larger blocks require more capacity overheads than smaller blocks, and finer-grained snapshots reduce overhead. Because of the higher throughput, transfer time is shorter (seconds versus minutes or hours), and storage overhead is a few percent on average for every configuration.

Rule-of-thumb 1: 8% of capacity needs to be reserved for snapshot overheads to support incremental transfers every 12 hours. The reserve is as low as 2% of capacity with replication snapshots.

4.3 I/O Overhead

We have further analyzed the I/O overhead for snapshots by measuring the fraction of host write I/O that causes a snapshot I/O during the transfer period. This can be thought of as I/O amplification because a host write can cause a read and second write for copy-on-write. For redirect-on-write, there may be a read-modify-write due to writes smaller than the block size as well as decreased data locality.



a. Random b. All c. Sequential

Figure 5: Snapshot storage overhead due to host write I/O for Random, All, and Sequentially written systems.

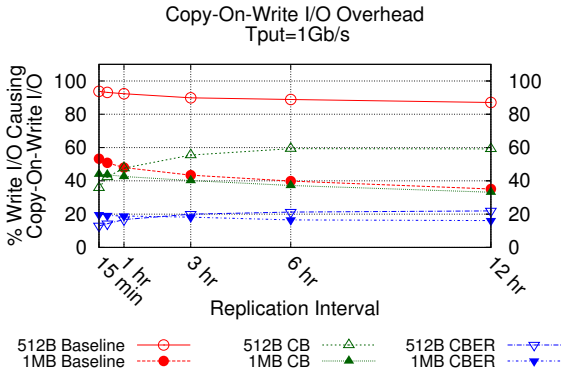


Figure 6: Fraction of host write I/O that causes copy-on-write I/O during the transfer period. The plotted lines are for **24hr_1GBWrt All**.

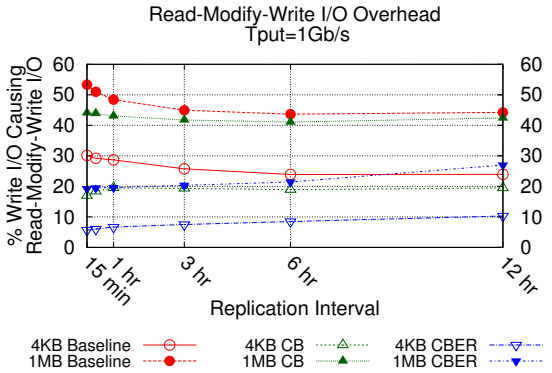


Figure 7: Fraction of host write I/O that causes read-modify-write I/O during the transfer period. The plotted lines are for **24hr_1GBWrt All**.

As shown in Figure 6, copy-on-write I/O can be almost 100% of the host I/O for 512B blocks and Baseline snapshots. In general, we find that smaller blocks cause a larger number of copy-on-write I/Os than larger blocks, though transferring larger blocks will include sectors that were not modified. This is because host write I/O tends to be at least somewhat sequential, and only the first I/O to a block causes a copy-on-write I/O. We also find a consistent pattern, in which improving the replication snapshot technique decreases the copy-on-write I/O overhead across block sizes and replication intervals.

In contrast, redirect-on-write has different patterns than copy-on-write, because redirect-on-write can cause read-modify-write operations as shown in Figure 7. We analyzed 4KB blocks instead of 512B blocks since there is never a read-modify-write for 512B blocks. We find that 1MB blocks have a higher fraction of read-modify-write I/O because host I/O sizes tend to be kilobytes.

These results presented for 1Gb/s throughput are qualitatively similar to results for lower transfer throughputs. One difference is that I/O overheads are larger for high throughput than low throughput, which may seem counter-intuitive. We present a representative transfer period with the Baseline snapshot technique in Figure 8 for one trace (System 1799). The horizontal axis shows

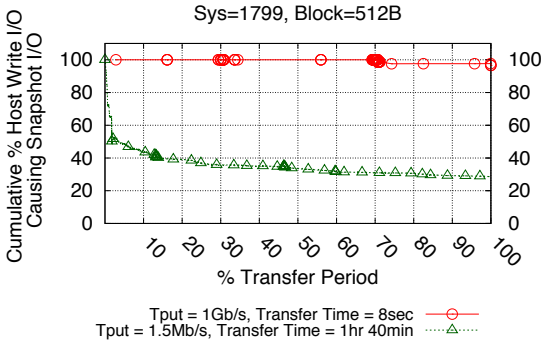


Figure 8: For high throughput, most host write I/Os cause a copy-on-write I/O, while at lower throughputs, there is less I/O overhead.

transfer time normalized to 100%, and the vertical axis shows the cumulative fraction of host I/O that causes a copy-on-write I/O for both 1.5Mb/s and 1Gb/s throughputs. For the 1Gb/s result, each mark represents a single I/O, while for 1.5Mb/s, each mark represents 1,000 I/Os.

Transfer periods can be quite short with 1Gb/s throughputs (8 seconds in this example) such that there are few I/Os during that time and those I/Os tend to be to unique blocks, which causes a copy-on-write I/O. At 1Gb/s throughput, 12-19% of systems did not experience

any host I/O during the transfer period for block sizes of 512B-1MB respectively. For slower throughputs, transfer time is longer (1 hour and 40 minutes for the T1 example), there are more I/Os, and many I/Os affect the same block. Over 99% of systems had at least some host write I/O during transfer at 1.5Mb/s.

Rule-of-thumb 2: Primary I/O should be over-provisioned by 100% to support copy-on-write related write-amplification of host writes during replication. The over-provision can be as low as 20% with a replication snapshot.

4.4 Analysis with Write Buffers

Our analysis thus far has not included the impact of buffering host write I/O on the primary storage server during incremental transfer. Write-buffering is common in practice [3], with flushes to disk either scheduled periodically or triggered through a storage API. To simulate the impact of buffering host write I/O, we have added a FIFO queue to our analysis throughout the replication interval. As host writes take place during transfer time, the corresponding blocks are added to the queue. When our queue fills, the oldest block is evicted from the queue and is written to storage, which causes a copy-on-write or redirect-on-write (for the first write to a block) with related snapshot I/O and storage overheads.

Snapshot I/O overhead for 1.5Mb/s throughput and a 12 hour replication interval is shown in Figure 9. Snapshot I/O overhead decreases rapidly as the write buffer increases from 0% to 1% of the volume's estimated capacity. Increasing the write buffer would further decrease overheads, but write buffers are typically much less than 1% of storage capacity due to differences in cost between memory and persistent storage. In contrast to snapshot I/O, we found that storage overhead for snapshots was nearly unaffected by buffer size because only the first write to a block requires snapshot storage space. We did find that both I/O and storage overheads decrease with improved replication snapshot techniques. A storage overhead figure is not shown due to space limitations.

Rule-of-thumb 3: Having a write buffer effectively decreases snapshot I/O overheads but has little impact on storage overheads.

5 Findings for Target Storage

Besides improving storage overheads for primary systems, we can also analyze how target data protection storage is impacted. How frequently can replication run? How much data will be stored? How much bandwidth is required? Answering these questions will guide the design of future data protection systems.

5.1 Transfer Size Analysis

We first investigate the amount of data to be transferred and stored for each replication interval. We investigate

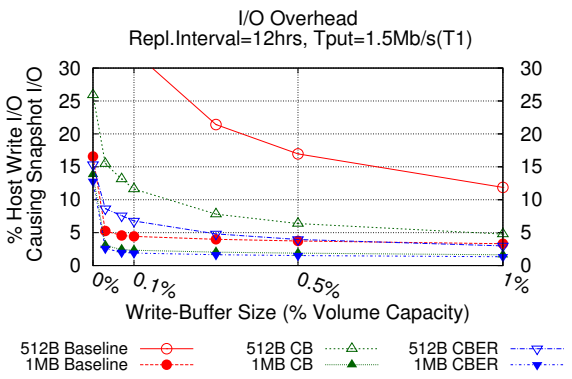


Figure 9: Snapshot I/O overhead decreases rapidly as write buffer size increases.

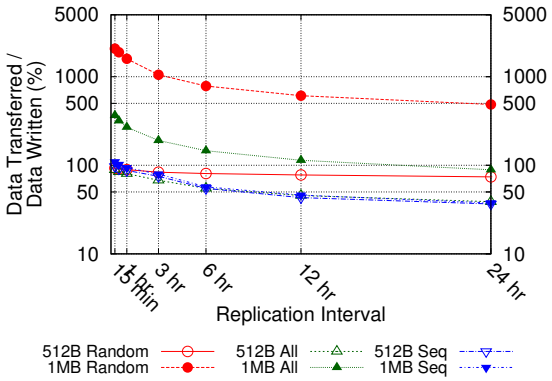


Figure 10: Data transferred as a fraction of data written gradually decreases as the replication interval increases.

the size of data transferred in Figure 10. Note that the vertical axis shows the normalized data transferred (log scale). For normalization, we divide the dirty blocks to be transferred by the amount of data written by the host to the primary system. Values will be less than 100% when a host writes to the same block multiple times, and the block only has to be transferred once because of write collapsing.

For a block size of 512 bytes across all volumes (the 512B All line overlaps with Seq), the data transferred starts at about 100% of the data written with the interval of 15 minutes and gradually decreases to about 40% with a 24-hour interval. For sequentially accessed logical volumes (Seq), results are consistent across block sizes: data transferred is >=100% of the data written when the interval is 15 minutes and gradually reaches about 40% of the data written at 24 hours. This is because sequential host write I/O tends to produce more completely dirty blocks than other I/O patterns.

Data transferred can be more than 100% because all of the sectors in a dirty block are transferred even if only a single sector in the block is actually dirty. As the interval increases, blocks are 'filled up' with more dirty data. Figure 11 shows that 512B blocks are always fully dirty

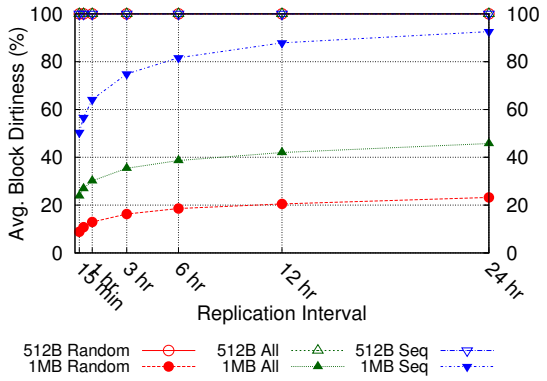


Figure 11: Except for the block size of 512B, dirty blocks are likely to contain various amounts of 'clean data,' with larger blocks more so than smaller blocks.

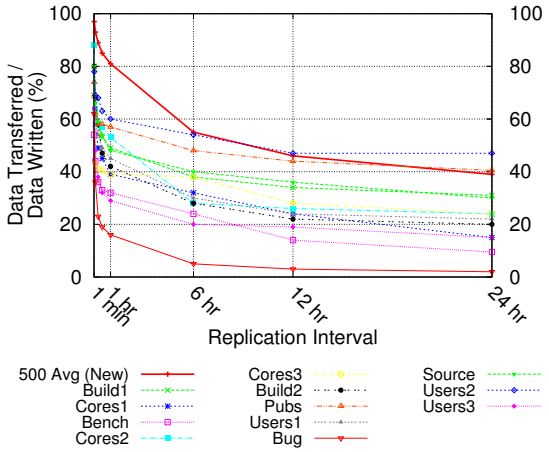


Figure 12: The **500 Avg (New)** line plots the data transferred normalized to data written by the host for each replication interval with our traces, while the other lines are reproduced from Patterson et al. [20]. All results are with 4KB blocks.

(line across the top). On the other hand, 1MB blocks become dirtier as the time between transfers increases, with most of the change in the first six hours. As expected, sequentially written volumes have much more fully dirty blocks than randomly written volumes, and block dirtiness is related to the reduction in normalized data transferred. Though not shown due to space limitations, we also found a distinct pattern that blocks were either fully dirty or dirty in multiples of 4KB or 8KB, likely due to file system and database allocation units.

These results suggest that using a large block size with a short interval can incur a significant overhead in transferring changed data to target storage. Even for small block sizes, >40% of data written daily is transferred to external systems.

Rule-of-thumb 4: The daily transfer size with small blocks is generally 40% of what hosts write.

Rule-of-thumb 5: Scheduling at least 6 hours between transfers allows blocks to achieve nearly peak dirtiness.

Comparison to previous study

A previous study in the SnapMirror system [20] of 12 file system servers examined reduction in data size to be transferred to a remote mirroring site over a range of replication intervals. In their study, the block size was fixed at 4KB. In Figure 12, our result for over 500 logical volumes (500 Avg (New)) with a 4KB block size is plotted along with a reproduction of their figure.

We find the reduction in data size to be much smaller than the SnapMirror results for intervals between 1 minute and 6 hours. Specifically, the SnapMirror study reports that all 12 systems achieve at least 30% reduction by 1 hour, while the average reduction for our traces is less than 20%. At longer intervals, our results are closer. For example, SnapMirror found a reduction in data size at 24-hour intervals to be between 53% and 98%, while we observe an average reduction of 60%.

In summary, our results are qualitatively similar, with transfer savings increasing with replication interval. The observed discrepancies are most likely due to different workloads used for analysis. The smaller number of systems studied for SnapMirror mostly supported software development and related applications, e.g., source code tree, bug tracking database, and engineer home directories, while the systems in our study support a mix of business and consumer applications and file systems.

5.2 Bandwidth Requirements

Transferring data requires sufficient bandwidth for the transfer to complete before the next replication interval or a cascade of failures occurs. Peak bandwidth was calculated for each trace, and the 90th percentile across traces is plotted in Figure 13. Results are per volume, so bandwidth for a storage system with many volumes would be higher. Logical volumes supporting sequential hosts require the most network bandwidth across all replication intervals. For replication interval > 6*hours*, the required bandwidth for the logical volumes in the Random set is similar to that for the volumes in the All set. For sequential hosts, the number of logical volumes that can simultaneously transfer changed data is largely bound by network bandwidth, while for the other volumes, the choice of block size has a significant impact. Based on the results from Figure 10, storage administrators can calculate how much bandwidth they will need to transfer changed data, which is a sizable fraction (approximately 40%) of what hosts write to primary storage.

There is clearly a relationship between the amount of data written by the host to primary storage and the

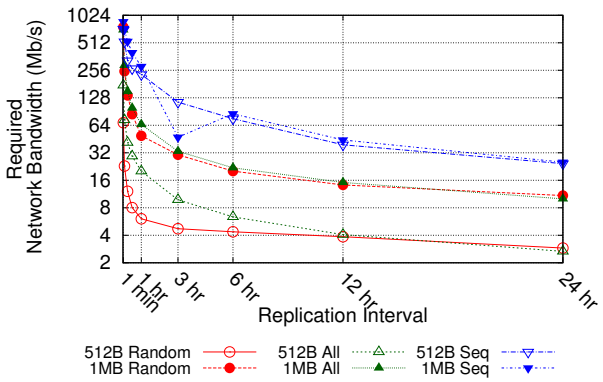


Figure 13: 90th-percentile peak network bandwidth needed to successfully transfer dirty blocks. Bandwidth is per logical volume, and the y-axis has a log scale of base 2.

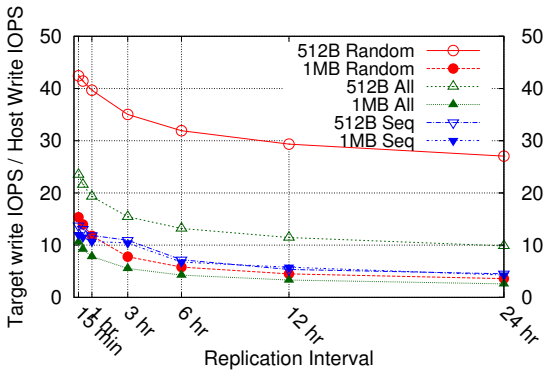


Figure 14: Ratio of target write IOPS to host write IOPS.

amount transferred to target storage, and analyzing 512B blocks shows a 99% correlation up to 30 minutes. The correlation is lower (62-85% depending on replication interval) for 1MB blocks, likely due to clean data also transferred in large blocks. We found a fairly low correlation (30-53%) between estimated volume capacity and transferred data, so capacity is less predictive of bandwidth requirements than other properties such as host write throughput.

Rule-of-thumb 6: Scheduling at least 12 hours between transfers drastically reduces peak network bandwidth requirements. Volume capacity is not predictive of bandwidth requirements.

5.3 I/O Analysis

A significant difference between primary storage and target storage designed for data protection is the I/O requirements of each system. Primary storage is designed to optimize for host I/O requirements related to email or web servers, shared file systems, or databases. While capacity matters, I/O per second is often a more critical feature. In comparison, target storage is designed for capacity and high throughput [26], so I/O per second may be of lower priority.

Figure 14 shows how the replication interval affects I/O per second requirements for target storage that is not log structured. The vertical axis is normalized relative to host I/O rates. Specifically, it shows the number of write I/O requests needed to transfer dirty blocks to the target as a percentage of the number of host write I/O requests for the same period in the original trace. See Section 3 for detailed information on how we compute write I/O to target storage.

For even a fifteen minute interval, the transfer I/O rate drops to between 10% and 40% of the host I/O rate, depending on the block size and write pattern. This sharp drop for a short interval is because we first order the dirty sectors accumulated over the interval by their logical addresses to compute write I/O needed for transfer to target storage. This ordering results in longer runs of sequential dirty sectors than created by original host write I/O requests (up to the assumed maximum 1MB transfer size). I/O savings continue up to 24 hours measured, though there is little change between 6 hours and 24, and the I/O rate for a larger block size is consistently lower than that of a smaller block size. Collecting host I/Os for a period of time is a well studied technique to reduce I/O requirements [3].

For sequentially accessed logical volumes, the transfer I/O rates for different block sizes are almost indistinguishable across all the intervals. This is because sequential host write I/O, along with our ordering of dirty sectors, produces runs of sequential dirty sectors that are $>>$ 1MB in size, so the 1MB network transfer size limitation becomes the dominating factor. For randomly accessed logical volumes, block size has a large impact on I/O requirements, requiring from 12% to 40% at 1 hour. These results indicate that it is worthwhile to configure block sizes and replication intervals for mixed and randomly accessed volumes.

While our work focuses on asynchronous replication to reduce I/O and storage requirements for target systems, an alternative is to consider synchronous replication. Synchronous replication requires a target system to have 100% of the I/O capabilities of the primary system, which would be a horizontal line added to Figure 14 at 100% on the vertical axis. Asynchronous replication can be more efficient than synchronous replication for two reasons: collapsing multiple writes to the same block before replication to reduce transferred data and reordering writes to reduce random I/O. We leave it as future work to explore the impact of each reason.

Rule-of-thumb 7: Target storage must support as much as 20% of the I/O per second capabilities of primary storage when the replication interval is at least one hour.

6 Related Work

Over the years, there have been many studies of storage workloads in various computing environments including aspects of file access and caching [3, 4, 11, 19, 23]. Leung et al. [17] analyzed I/O trace data collected from networked file servers deployed in a data center. Anderson [1] presented new techniques for collecting large, detailed traces. Analysis of high performance computing (i.e. supercomputing) workloads focused on bandwidth, I/O request inter-arrival times, idle time, and access rates [6, 7, 15, 16, 18]. Gulati et al. [9] studied characteristics and consolidation strategies for virtualized systems. Analysis for database workloads [12] has shown qualitatively similar properties to file systems.

Numerous studies have measured disk access properties including block lifetimes, access rates, response time, sequential patterns, and caching [23, 24]. Riska and Riedel [21] analyzed how I/O workloads on disk drives change depending on applications and computing environments, e.g., enterprise servers vs. desktop computers vs. consumer electronics.

Unlike these earlier works, we specifically focus on characterizing the overheads and I/O properties in transferring incremental changes on primary storage to target storage. Specifically, we analyze data changes at the physical (block) level, in part, because creating backups at the physical level is more efficient than doing so at the logical (file) level [13]. Roselli et al. [23] studied block lifetimes but not in the context of data protection. A study a decade ago by Patterson et al. [20] characterized changed data at the block level for a similar goal; see Section 5.1. Wallace et al. [26] described backup workload characteristics, though they intermixed full and incremental workloads.

Snapshots are a common technique to create a point-in-time version of data. WAFL [10], ZFS [5] and BTRFS [22] all natively support snapshots with copy-on-write as means of ensuring data consistency on disk and enabling fast restart after system crash. In these systems, snapshots are first-class objects that can be named and accessed by the end user. In the case of ZFS and BTRFS, snapshots are writable and can be updated independently from the original. In addition, snapshots are taken at the logical level, e.g., the entire file system, directories, and/or individual files. In contrast, a replication snapshot is mainly comprised of blocks written since the last transfer, is not writable, and does not persist; once the transfer is completed, the space allocated for copied-on-write blocks is reclaimed for use by primary storage or later snapshots.

There are several publications on snapshot overheads. Azagury et al. [2] and Shah [25] both report up to 7% degradation in I/O rate due to copy-on-write. We analyzed replication snapshots as a technique to reduce overheads of standard snapshots during replication. Our two versions of replication snapshots can be classified as **write-coalescing batches with atomic update** in a taxonomy for remote mirroring defined by Ji et. al. [14], with the batch size determined by replication intervals. Our asynchronous technique allows for write coalescing to reduce write size and I/O rate on target storage.

Synchronous remote mirroring [14] can also be used for protection of data changes, especially when the change rate is low and/or the geographical distance between primary and target systems is relatively short, e.g., [27, 28]. In this paper, we analyze an asynchronous approach to allow target systems whose I/O performance and storage capacity are characteristically different from primary storage, e.g., purpose-built backup appliances.

7 Discussion and Conclusion

In this paper, we have analyzed I/O traces from over 100,000 logical volumes in customer block-based primary storage systems to understand I/O characteristics and performed a detailed analysis of over 500 traces spanning at least 24 hours to gain a better understanding of incremental change patterns. New insights can help data protection expand from the realm of daily backups to more frequent updates.

Our analysis has uncovered several new findings for both primary and target storage. Overheads on primary storage due to snapshots can require both capacity and I/O to preserve point-in-time copies, though a write buffer decreases I/O requirements. For target storage, storage requirements depend on the write patterns of the host and can vary from 40% for most hosts to 100% for hosts that write sequentially. Replication requires bandwidth, which we have shown grows proportionally with the write-throughput of hosts. We have found that access patterns can change from highly sequential to highly random across different replication intervals, with a large change in data transfer characteristics. Given that the transfer interval is often statically configured by the target system administrator, our observations argue for dynamically changing block sizes and replication intervals at run time based on the host I/O access pattern.

Many findings about data patterns align with previous results: dirty blocks tend to be overwritten again within minutes or hours, the change rate grows less rapidly with longer replication intervals, and volumes tend to be modified in multiples of 4KB or 8KB. From the analysis of over 100,000 traces, we found that there is great diversity in storage requirements in terms of capacity, numbers of writes and reads, as well as average and peak throughput and I/O per second.

Acknowledgments

We would like to thank Fred Douglis, Kadir Ozdemir, Steve Smaldone, Grant Wallace, Ian Wigmore, and our reviewers for their feedback. We also thank Bill Glynn and the EMC VMAX team for providing the traces.

References

[1] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proc. of the 7th USENIX Conf. on File and Storage Tech.*, 2009.

[2] A. Azagury, M. E. Factor, J. Satran, and W. Micka. Point-in-time copy: Yesterday, today and tomorrow. In *Proc. IEEE/NASA Conf. Mass Storage Systems*, 2002.

[3] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. *ACM SIGPLAN Notices*, 27(9):10–22, 1992.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[5] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003.

[6] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Trans. on Storage*, 7(3), October 2011.

[7] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 Characterization of Petascale I/O Workloads. In *Proc. of the 1st Works. on Interfaces and Abstractions for Scientific Data Storage*, 2009.

[8] EMC. EMC Symmetrix VMAX. `http://www.emc.com/storage/symmetrix-vmax/symmetrix-vmax.htm`, 2013.

[9] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Proc. of the 2nd Inter. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, 2009.

[10] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, 1994.

[11] W. W. Hsu and A. Smith. The performance impact of I/O optimizations and disk improvements. *IBM Journal of Research and Development*, pages 255–289, March 2004.

[12] W. W. Hsu, A. J. Smith, and H. C. Young. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. *ACM Trans. on Database Systems*, 26:96–143, 2001.

[13] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[14] M. Ji, A. Veitch, J. Wilkes, et al. Seneca: remote mirroring done write. In *Proc. of the USENIX Annual Technical Conf.*, 2003.

[15] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemeyer. Workload characterization of a leadership class storage cluster. In *Proc. of the 5th Petascale Data Storage Workshop*, 2010.

[16] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proc. of Supercomputing*, November 2009.

[17] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proc. of the USENIX Annual Technical Conf.*, 2008.

[18] E. L. Miller, R. H. Katz, and Y. H. Katz. Analyzing the I/O behavior of supercomputer applications. In *Proc. of the 11th IEEE Symposium on Mass Storage Systems*, 1991.

[19] J. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. of the 10th Symposium on Operating System Principles*, 1985.

[20] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. In *Proc. of the 1st USENIX Conf. on File and Storage Tech.*, 2002.

[21] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proc. of the USENIX Annual Technical Conf.*, 2006.

[22] O. Rodeh, J. Bacik, and C. Mason. Brtfs: The linux b-tree filesystem. Technical report, IBM Research Report RJ10501 (ALM1207-004), 2012.

[23] D. Roselli, J. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the USENIX Annual Technical Conf.*, 2000.

[24] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proc. of the Winter USENIX Conf.*, 1993.

[25] B. Shah. *Disk performance of copy-on-write snapshot logical volumes*. PhD thesis, The University Of British Columbia, 2006.

[26] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proc. of the 10th USENIX Conf. on File and Storage Tech.*, 2012.

[27] H. Weatherspoon, L. Ganesh, T. Marian, M. Balakrishnan, and K. Birman. Smoke and mirrors: reflecting files at a geographically remote location without loss of performance. In *Proc. of the 7th USENIX Conf. on File and Storage Tech.*, 2009.

[28] M. Zhang, Y. Liu, and Q. Yang. Cost-effective remote mirroring using the iSCSI protocol. In *21st IEEE Conf. on Mass. Storage Systems and Tech.*, 2004.

On the Efficiency of Durable State Machine Replication

Alysson Bessani[1], *Marcel Santos*[1], *João Felix*[1], *Nuno Neves*[1], *Miguel Correia*[2]
{[1]FCUL/LaSIGE, [2]INESC-ID/IST}, University of Lisbon – Portugal

Abstract

State Machine Replication (SMR) is a fundamental technique for ensuring the dependability of critical services in modern internet-scale infrastructures. SMR alone does not protect from full crashes, and thus in practice it is employed together with secondary storage to ensure the durability of the data managed by these services. In this work we show that the classical durability enforcing mechanisms – logging, checkpointing, state transfer – can have a high impact on the performance of SMR-based services even if SSDs are used instead of disks. To alleviate this impact, we propose three techniques that can be used in a transparent manner, i.e., without modifying the SMR programming model or requiring extra resources: parallel logging, sequential checkpointing, and collaborative state transfer. We show the benefits of these techniques experimentally by implementing them in an open-source replication library, and evaluating them in the context of a consistent key-value store and a coordination service.

1 Introduction

Internet-scale infrastructures rely on services that are replicated in a group of servers to guarantee availability and integrity despite the occurrence of faults. One of the key techniques for implementing replication is the *Paxos* protocol [27], or more generically the *state machine replication* (SMR) approach [34]. Many systems in production use variations of this approach to tolerate *crash faults* (e.g., [4, 5, 8, 12, 19]). Research systems have also shown that SMR can be employed with *Byzantine faults* with reasonable costs (e.g., [6, 9, 17, 21, 25]).

This paper addresses the problem of adding durability to SMR systems. *Durability* is defined as the capability of a SMR system to survive the crash or shutdown of all its replicas, without losing any operation acknowledged to the clients. Its relevance is justified not only by the need to support maintenance operations, but also by the many examples of significant failures that occur in data centers, causing thousands of servers to crash simultaneously [13, 15, 30, 33].

However, the integration of durability techniques – logging, checkpointing, and state transfer – with the SMR approach can be difficult [8]. First of all, these techniques can drastically decrease the performance of a service[1]. In particular, *synchronous logging* can make the system throughput as low as the number of appends that can be performed on the disk per second, typically just a few hundreds [24]. Although the use of SSDs can alleviate the problem, it cannot solve it completely (see §2.2). Additionally, *checkpointing* requires stopping the service during this operation [6], unless non-trivial optimizations are used at the application layer, such as copy-on-write [8, 9]. Moreover, recovering faulty replicas involves running a *state transfer protocol*, which can impact normal execution as correct replicas need to transmit their state.

Second, these durability techniques can complicate the programming model. In theory, SMR requires only that the service exposes an *execute()* method, called by the replication library when an operation is ready to be executed. However this leads to logs that grow forever, so in practice the interface has to support service state checkpointing. Two simple methods can be added to the interface, one to collect a snapshot of the state and another to install it during recovery. This basic setup defines a simple interface, which eases the programming of the service, and allows a complete separation between the replication management logic and the service implementation. However, this interface can become much more complex, if certain optimizations are used (see §2.2).

This paper presents new techniques for implementing data durability in crash and Byzantine fault-tolerant

[1]The performance results presented in the literature often exclude the impact of durability, as the authors intend to evaluate other aspects of the solutions, such as the behavior of the agreement protocol. Therefore, high throughput numbers can be observed (in req/sec) since the overheads of logging/checkpointing are not considered.

(BFT) SMR services. These techniques are transparent with respect to both the service being replicated and the replication protocol, so they do not impact the programming model; they greatly improve the performance in comparison to standard techniques; they can be used in commodity servers with ordinary hardware configurations (no need for extra hardware, such as disks, special memories or replicas); and, they can be implemented in a modular way, as a *durability layer* placed in between the SMR library and the service.

The techniques are three: *parallel logging*, for diluting the latency of synchronous logging; *sequential checkpointing*, to avoid stopping the replicated system during checkpoints; and *collaborative state transfer*, for reducing the effect of replica recoveries on the system performance. This is the first time that the durability of fault-tolerant SMR is tackled in a *principled* way with a set of algorithms organized in an *abstraction* to be used between SMR protocols and the application.

The proposed techniques were implemented in a durability layer on the BFT-SMaRt state machine replication library [1], on top of which we built two services: a consistent key-value store (SCKV-Store) and a non-trivial BFT coordination service (Durable DepSpace). Our experimental evaluation shows that the proposed techniques can remove most of the performance degradation due to the addition of durability.

This paper makes the following contributions:

1. A description of the performance problems affecting durable state machine replication, often overlooked in previous works (§2);
2. Three new algorithmic techniques for removing the negative effects of logging, checkpointing and faulty replica recovery from SMR, without requiring more resources, specialized hardware, or changing the service code (§3).
3. An analysis showing that exchanging disks by SSDs neither solves the identified problems nor improves our techniques beyond what is achieved with disks (§2 and §5);
4. The description of an implementation of our techniques (§4), and an experimental evaluation under write-intensive loads, highlighting the performance limitations of previous solutions and how our techniques mitigate them (§5).

2 Durable SMR Performance Limitations

This section presents a durable SMR model, and then analyzes the effect of durability mechanisms on the performance of the system.

2.1 System Model and Properties

We follow the standard SMR model [34]. Clients send requests to invoke operations on a service, which is implemented in a set of replicas (see Figure 1). Operations are executed in the same order by all replicas, by running some form of agreement protocol. Service operations are assumed to be deterministic, so an operation that updates the state (abstracted as a *write*) produces the same new state in all replicas. The state required for processing the operations is kept in main memory, just like in most practical applications for SMR [4, 8, 19].

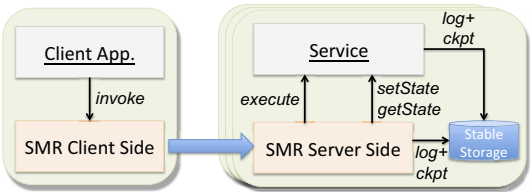


Figure 1: A durable state machine replication architecture.

The replication library implementing SMR has a client and a server side (layers at the bottom of the figure), which interact respectively with the client application and the service code. The library ensures standard safety and liveness properties [6, 27], such as correct clients eventually receive a response to their requests if enough synchrony exists in the system.

SMR is built under the assumption that at most f replicas fail out of a total of n replicas (we assume $n = 2f + 1$ on a crash fault-tolerant system and $n = 3f + 1$ on a BFT system). A crash of more than f replicas breaks this assumption, causing the system to stop processing requests as the necessary agreement quorums are no longer available. Furthermore, depending on which replicas were affected and on the number of crashes, some state changes may be lost. This behavior is undesirable, as clients may have already been informed about the changes in a response (i.e., the request completed) and there is the expectation that the execution of operations is persistent.

To address this limitation, the SMR system should also ensure the following property:

> *Durability:* Any request completed at a client is reflected in the service state after a recovery.

Traditional mechanisms for enforcing durability in SMR-based main memory databases are logging, checkpointing and state transfer [8, 16]. A replica can recover from a crash by using the information saved in stable storage and the state available in other replicas. It is important to notice that a recovering replica is considered faulty *until it obtains enough data to reconstruct the state* (which typically occurs after state transfer finishes).

Logging writes to stable storage information about the progress of the agreement protocol (e.g., when cer-

tain messages arrive in Paxos-like protocols [8, 20]) and about the operations executed on the service. Therefore, data is logged either by the replication library or the service itself, and a record describing the operation has to be stored before a reply is returned to the client.

The replication library and the service code synchronize the creation of checkpoints with the truncation of logs. The service is responsible for generating snapshots of its state (method *getState*) and for setting the state to a snapshot provided by the replication library (method *setState*). The replication library also implements a *state transfer* protocol to initiate replicas from an updated state (e.g., when recovering from a failure or if they are too late processing requests), akin to previous SMR works [6, 7, 8, 9, 32]. The state is fetched from the other replicas that are currently running.

2.2 Identifying Performance Problems

This section discusses performance problems caused by the use of logging, checkpointing and state transfer in SMR systems. We illustrate the problems with a consistent key-value store (SCKV-Store) implemented using BFT-SMaRt [1], a Java BFT SMR library. In any case, the results in the paper are mostly orthogonal to the fault model. We consider write-only workloads of 8-byte keys and 4kB values, in a key space of 250K keys, which creates a service state size of 1GB in 4 replicas. More details about this application and the experiments can be found in §4 and §5, respectively.

High latency of logging. As mentioned in §2.1, events related to the agreement protocol and operations that change the state of the service need to be logged in stable storage. Table 1 illustrates the effects of several logging approaches on the SCKV-Store, with a client load that keeps a high sustainable throughput:

Metric	No log	Async.	Sync. SSD	Sync. Disk
Min Lat. (ms)	1.98	2.16	2.89	19.61
Peak Thr. (ops/s)	4772	4312	1017	63

Table 1: Effect of logging on the SCKV-Store. Single-client minimum latency and peak throughput of 4kB-writes.

The table shows that *synchronous*[2] logging to disk can cripple the performance of such system. To address this issue, some works have suggested the use of faster non-volatile memory, such as flash memory solid state drives (SSDs) or/in NVCaches [32]. As the table demonstrates, there is a huge performance improvement when the log is written synchronously to SSD storage, but still only 23% of the "No log" throughput is achieved. Additionally, by employing specialized hardware, one arguably increases the costs and the management complexity of the nodes, especially in virtualized/cloud environments where such hardware may not be available in all machines.

There are works that avoid this penalty by using *asynchronous* writes to disk, allowing replicas to present a performance closer to the main memory system (e.g., Harp [28] and BFS [6]). The problem with this solution is that writing asynchronously does not give durability guarantees if all the replicas crash (and later recover), something that production systems need to address as correlated failures do happen [13, 15, 30, 33].

We would like to have a general solution that makes the performance of durable systems similar to pure memory systems, and that achieves this by *exploring the logging latency to process the requests* and by *optimizing log writes*.

Perturbations caused by checkpoints. Checkpoints are necessary to limit the log size, but their creation usually degrades the performance of the service. Figure 2 shows how the throughput of the SCKV-Store is affected by creating checkpoints at every 200K client requests. Taking a snapshot after processing a certain number of operations, as proposed in most works in SMR (e.g., [6, 27]), can make the system halt for a few seconds. This happens because requests are no longer processed while replicas save their state. Moreover, if the replicas are not fully synchronized, delays may also occur because the necessary agreement quorum might not be available.

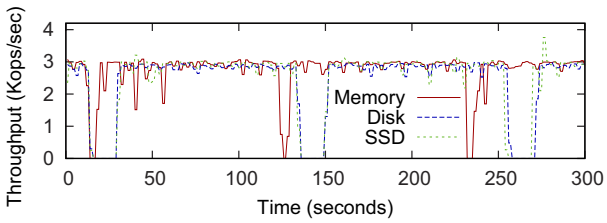


Figure 2: Throughput of a SCKV-Store with checkpoints in memory, disk and SSD considering a state of 1GB.

The figure indicates an equivalent performance degradation for checkpoints written in disk or SSD, meaning there is no extra benefit in using the latter (both require roughly the same amount of time to synchronously write the checkpoints). More importantly, the problem occurs even if the checkpoints are kept in memory, since the fundamental limitation is not due to storage accesses (as in logging), but to the cost to serialize a large state (1 GB).

Often, the performance decrease caused by checkpointing is not observed in the literature, either because no checkpoints were taken or because the service had a very small state (e.g., a counter with 8 bytes) [6, 10, 17, 21, 25]. Most of these works were focusing on ordering

[2]Synchronous writes are optimized to update only the file contents, and not the metadata, using the `rwd` mode in the Java' *RandomAccessFile* class (equivalent to using the `O_DSYNC` flag in POSIX *open*). This is important to avoid unnecessary disk head positioning.

requests efficiently, and therefore checkpointing could be disregarded as an orthogonal issue. Additionally, one could think that checkpoints need only to be created sporadically, and therefore, their impact is small on the overall execution. We argue that this is not true in many scenarios. For example, the SCKV-Store can process around 4700 4kB-writes per second (see §5), which means that the log can grow at the rate of more than 1.1 GB/min, and thus checkpoints need to be taken rather frequently to avoid outrageous log sizes. Leader-based protocols, such as those based on Paxos, have to log information about most of the exchanged messages, contributing to the log growth. Furthermore, recent SMR protocols require frequent checkpoints (every few hundred operations) to allow the service to recover efficiently from failed speculative request ordering attempts [17, 21, 25].

Some systems use *copy-on-write* techniques for doing checkpointing without stoping replicas (e.g., [9]), but this approach has two limitations. First, copy-on-write may be complicated to implement at application level in non-trivial services, as the service needs to keep track of which data objects were modified by the requests. Second, even if such techniques are employed, the creation of checkpoints still consumes resources and degrades the performance of the system. For example, writing a checkpoint to disk makes logging much slower since the disk head has to move between the log and checkpoint files, with the consequent disk seek times. In practice, this limitation could be addressed in part with extra hardware, such as by using two disks per server.

Another technique to deal with the problem is *fuzzy snapshots*, used in ZooKeeper [19]. A fuzzy snapshot is essentially a checkpoint that is done without stopping the execution of operations. The downside is that some operations may be executed more than once during recovery, an issue that ZooKeeper solves by forcing all operations to be idempotent. However, making operations idempotent requires non-trivial request pre-processing before they are ordered, and increases the difficulty of decoupling the replication library from the service [19, 20].

We aim to have a checkpointing mechanism that *minimizes performance degradation without requiring additional hardware and, at the same time, keeping the SMR programming model simple*.

Perturbations caused by state transfer. When a replica recovers, it needs to obtain an updated state to catch up with the other replicas. This state is usually composed of the last checkpoint plus the log up to some request defined by the recovering replica. Typically, (at least) another replica has to spend resources to send (part of) the state. If checkpoints and logs are stored in a disk, delays occur due to the transmission of the state through the network but also because of the disk accesses. Delta-checkpoint techniques based, for instance, on Merkle trees [6] can alleviate this problem, but cannot solve it completely since logs have always to be transferred. Moreover, implementing this kind of technique can add more complexity to the service code.

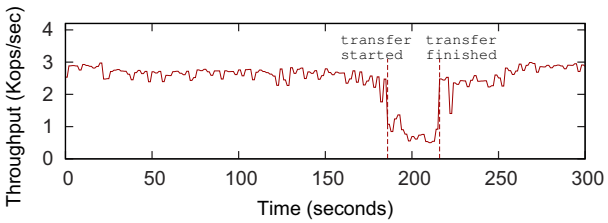


Figure 3: Throughput of a SCKV-Store when a failed replica recovers and asks for a state transfer.

Similarly to what is observed with checkpointing, there can be the temptation to disregard the state transfer impact on performance because it is perceived to occur rarely. However, techniques such as replica rejuvenation [18] and proactive recovery [6, 36] use state transfer to bring refreshed replicas up to date. Moreover, reconfigurations [29] and even leader change protocols (that need to be executed periodically for resilient BFT replication [10]) may require replicas to synchronize themselves [6, 35]. In conclusion, state transfer protocols may be invoked much more often than when there is a crash and a subsequent recovery.

Figure 3 illustrates the effect of state transmission during a replica recovery in a 4 node BFT system using the PBFT's state transfer protocol [6]. This protocol requires just one replica to send the state (checkpoint plus log) – similarly to crash FT Paxos-based systems – while others just provide authenticated hashes for state validation (as the sender of the state may suffer a Byzantine fault). The figure shows that the system performance drops to less than 1/3 of its normal performance during the 30 seconds required to complete state transfer. While one replica is recovering, another one is slowed because it is sending the state, and thus the remaining two are unable to order and execute requests (with $f = 1$, quorums of 3 replicas are needed to order requests).

One way to avoid this performance degradation is to ignore the state transfer requests until the load is low enough to process both the state transfers and normal request ordering [19]. However, this approach tends to delay the recovery of faulty replicas and makes the system vulnerable to extended unavailability periods (if more faults occur). Another possible solution is to add extra replicas to avoid interruptions on the service during recovery [36]. This solution is undesirable as it can increase the costs of deploying the system.

We would like to have a state transfer protocol that *minimizes the performance degradation due to state transfer without delaying the recovery of faulty replicas*.

3 Efficient Durability for SMR

In this section we present three techniques to solve the problems identified in the previous section.

3.1 Parallel Logging

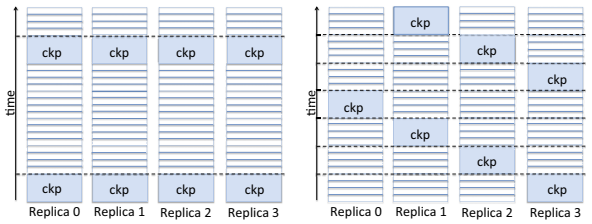
Parallel logging has the objective of hiding the high latency of logging. It is based on two ideas: (1) log groups of operations instead of single operations; and (2) process the operations in parallel with their storage.

The first idea explores the fact that disks have a high bandwidth, so the latency for writing 1 or 100 log entries can be similar, but the throughput would be naturally increased by a factor of roughly 100 in the second case. This technique requires the replication library to deliver groups of service operations (accumulated during the previous batch execution) to allow the whole batch to be logged at once, whereas previous solutions normally only provide single operations, one by one. Notice that this approach is different from the batching commonly used in SMR [6, 10, 25], where a group of operations is ordered together to amortize the costs of the agreement protocol (although many times these costs include logging a batch of requests to stable storage [27]). Here the aim is to pass batches of operations from the replication library to the service, and a batch may include (batches of) requests ordered in *different agreements*.

The second idea requires that the requests of a batch are processed while the corresponding log entries are being written to the secondary storage. Notice, however, that a reply can only be sent to the client after the corresponding request is executed *and logged*, ensuring that the result seen by the client will persist even if all replicas fail and later recover. Naturally, the effectiveness of this technique depends on the relation between the time for processing a batch and the time for logging it. More specifically, the interval T_k taken by a service to process a batch of k requests is given by $T_k = max(E_k, L_k)$, where E_k and L_k represent the latency of executing and logging the batch of k operations, respectively. This equation shows that the most expensive of the two operations (execution or logging) defines the delay for processing the batch. For example, in the case of the SCKV-Store, $E_k \ll L_k$ for any k, since inserting data in a hash table with chaining (an $\mathcal{O}(1)$ operation) is much faster than logging a 4kB-write (with or without batching). This is not the case for Durable DepSpace, which takes a much higher benefit from this technique (see §5).

3.2 Sequential Checkpointing

Sequential checkpointing aims at minimizing the performance impact of taking replica's state snapshots. The key principle is to exploit the natural redundancy that exists in asynchronous distributed systems based on SMR. Since these systems make progress as long as a quorum of $n-f$ replicas is available, there are f spare replicas in fault-free executions. The intuition here is to make each replica store its state at different times, to ensure that $n-f$ replicas can continue processing client requests.



(a) Synchronized. (b) Sequential.

Figure 4: Checkpointing strategies (4 replicas).

We define *global checkpointing period* P as the maximum number of (write) requests that a replica will execute before creating a new checkpoint. This parameter defines also the maximum size of a replica's log in number of requests. Although P is the same for all replicas, they checkpoint their state at different points of the execution. Moreover, all correct replicas will take at least one checkpoint within that period.

An instantiation of this model is for each replica $i = 0, ..., n-1$ to take a checkpoint after processing the k-th request where $k \bmod P = i \times \lfloor \frac{P}{n} \rfloor$, e.g., for $P = 1000$, $n = 4$, replica i takes a checkpoint after processing requests $i \times 250$, $1000 + i \times 250$, $2000 + i \times 250$, and so on.

Figure 4 compares a synchronous (or coordinated) checkpoint with our technique. Time grows from the bottom of the figure to the top. The shorter rectangles represent the logging of an operation, whereas the taller rectangles correspond to the creation of a checkpoint. It can be observed that synchronized checkpoints occur less frequently than sequential checkpoints, but they stop the system during their execution whereas for sequential checkpointing there is always an agreement quorum of 3 replicas available for continuing processing requests.

An important requirement of this scheme is to use values of P such that the chance of more than f overlapping checkpoints is negligible. Let C_{max} be the estimated maximum interval required for a replica to take a checkpoint and T_{max} the maximum throughput of the service. Two consecutive checkpoints will not overlap if:

$$\begin{aligned} C_{max} &< \frac{1}{T_{max}} \times \left\lfloor \frac{P}{n} \right\rfloor \Longrightarrow \\ P &> n \times C_{max} \times T_{max} \end{aligned} \quad (1)$$

Equation 1 defines the minimum value for P that can be used with sequential checkpoints. In our SCKV-Store example, for a state of 1GB and a 100% 4kB-write work-

load, we have $C_{max} \approx 15$s and $T_{max} \approx 4700$ ops/s, which means $P > 282000$. If more frequent checkpoints are required, the replicas can be organized in groups of at most f replicas to take checkpoints together.

3.3 Collaborative State Transfer

The state transfer protocol is used to update the state of a replica during recovery, by transmitting log records (L) and checkpoints (C) from other replicas (see Figure 5(a)). Typically only one of the replicas returns the full state and log, while the others may just send a hash of this data for validation (only required in the BFT case). As shown in §2, this approach can degrade performance during recoveries. Furthermore, it does not work with sequential checkpoints, as the received state can not be directly validated with hashes of other replicas' checkpoints (as they are different). These limitations are addressed with the *collaborative state transfer* (CST) protocol.

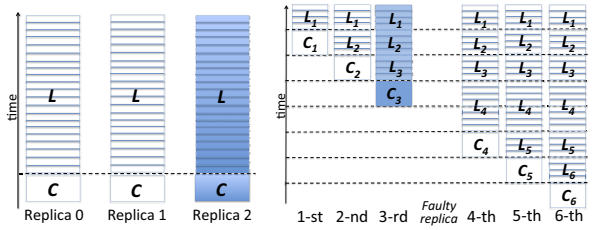
Although the two previous techniques work both with crash-tolerant and BFT SMR, the CST protocol is substantially more complex with Byzantine faults. Consequently, we start by describing a BFT version of the protocol (which also works for crash faults) and later, at the end of the section, we explain how CST can be simplified on a crash-tolerant system[3].

We designate by *leecher* the recovering replica and by *seeders* the replicas that send (parts of) their state. CST is triggered when a replica (leecher) starts (see Figure 6). Its first action is to use the local log and checkpoint to determine the last logged request and its sequence number (assigned by the ordering protocol), from now on called *agreement id*. The leecher then asks for the most recent logged agreement *id* of the other replicas, and waits for replies until $n - f$ of them are collected (including its own *id*). The *ids* are placed in a vector in descending order, and the largest *id* available in $f + 1$ replicas is selected, to ensure that such agreement *id* was logged by at least one correct replica (steps 1-3).

In BFT-SMaRt there is no parallel execution of agreements, so if one correct replica has ordered the *id*-th batch, it means with certainty that agreement *id* was already processed by at least $f + 1$ correct replicas[4]. The other correct replicas, which might be a bit late, will also eventually process this agreement, when they receive the necessary messages.

[3]Even though crash fault tolerance is by far more used in production systems, our choice is justified by two factors. First, the subtleties of BFT protocols require a more extensive discussion. Second, given the lack of a stable and widely-used open-source implementation of a crash fault tolerance SMR library, we choose to develop our techniques in a BFT SMR library, so the description is in accordance to our prototype.

[4]If one employs protocols such as Paxos/PBFT, low and high watermarks may need to considered.



(a) PBFT and others ($n = 4$). (b) CST ($n = 7$).

Figure 5: Data transfer in different state transfer strategies.

Next, the leecher proceeds to obtain the state up to *id* from a seeder and the associated validation data from f other replicas. The active replicas are ordered by the freshness of the checkpoints, from the most recent to the oldest (step 4). A leecher can make this calculation based on *id*, as replicas take checkpoints at deterministic points, as explained in §3.2. We call the replica with i-th oldest checkpoint the i-th replica and the checkpoint C_i. The log of a replica is divided in segments, and each segment L_i is the portion of the log required to update the state from C_i to the more recent state C_{i-1}. Therefore, we use the following notion of equivalence: $C_{i-1} \equiv C_i + L_i$. Notice that L_1 corresponds to the log records of the requests that were executed after the most recent checkpoint C_1 (see Figure 5(b) for $n = 7$).

The leecher fetches the state from the $(f + 1)$-th replica (seeder), which comprises the log segments L_1, ..., L_{f+1} and checkpoint C_{f+1} (step 8). To validate this state, it also gets hashes of the log segments and checkpoints from the other f replicas with more recent checkpoints (from the 1st until the f-th replica) (step 6a). Then, the leecher sets its state to the checkpoint and replays the log segments received from the seeder, in order to bring up to date its state (steps 10 and 12a).

The state validation is performed by comparing the hashes of the f replicas with the hashes of the log segments from the seeder and intermediate checkpoints. For each replica i, the leecher replays L_{i+1} to reach a state equivalent to the checkpoint of this replica. Then, it creates a intermediate checkpoint of its state and calculates the corresponding hash (steps 12a and 12b). The leecher finds out if the log segments sent by the seeder and the current state (after executing L_{i+1}) match the hashes provided by this replica (step 12c).

If the check succeeds for f replicas, the reached state is valid and the CST protocol can finish (step 13). If the validation fails, the leecher fetches the data from the $(f + 2)$-th replica, which includes the log segments L_1, ..., L_{f+2} and checkpoint C_{f+2} (step 13 goes back to step 8). Then, it re-executes the validation protocol, considering as extra validation information the hashes that were produced with the data from the $(f + 1)$-th replica (step 9). Notice that the validation still requires $f + 1$ matching

1. Look at the local log to discover the last executed agreement;
2. *Fetch* the id of the last executed agreement from $n-f$ replicas (including itself) and save the identifier of these replicas;
3. id = largest agreement id that is available in $f+1$ replicas;
4. Using id, P and n, order the replicas (including itself) with the ones with most recent checkpoints first;
5. $V \leftarrow \emptyset$; // the set containing state and log hashes
6. For $i = 1$ to f do:
   (a) *Fetch* $V_i = \langle HL_1, ..., HL_i, HC_i \rangle$ from i-th replica;
   (b) $V \leftarrow V \cup \{V_i\}$;
7. $r \leftarrow f+1$; // replica to fetch state
8. *Fetch* $S_r = \langle L_1, ..., L_r, C_r \rangle$ from r-th replica;
9. $V \leftarrow V \cup \{\langle \mathsf{H}(S_r.L_1), ..., \mathsf{H}(S_r.L_r), \mathsf{H}(S_r.C_r) \rangle\}$;
10. Update state using $S_r.C_r$;
11. $v \leftarrow 0$; // number of validations of S_r
12. For $i = r-1$ down to 1 do:
    (a) Replay log $S_r.L_{i+1}$;
    (b) Take checkpoint C'_i and calculate its hash HC'_i;
    (c) If $(V_i.HL_{1..i} = V_r.HL_{1..i}) \wedge (V_i.HC_i = HC'_i)$, $v{+}{+}$;
13. If $v \geq f$, replay log $S_r.L_1$ and return; Else, $r{+}{+}$ and go to 8;

Figure 6: The CST recovery protocol called by the leecher after a restart. *Fetch* commands wait for replies within a timeout and go back to step 2 if they do not complete.

log segments and checkpoints, but now there are $f+2$ replicas involved, and the validation is successful even with one Byzantine replica. In the worst case, f faulty replicas participate in the protocol, which requires $2f+1$ replicas to send some data, ensuring a correct majority and at least one valid state (log and checkpoint).

In the scenario of Figure 5(b), the 3rd replica (the $(f+1)$-th replica) sends L_1, L_2, L_3 and C_3, while the 2nd replica only transmits $HL_1 = \mathsf{H}(L_1)$, $HL_2 = \mathsf{H}(L_2)$ and $HC_2 = \mathsf{H}(C_2)$, and the 1st replica sends $HL_1 = \mathsf{H}(L_1)$ and $HC_1 = \mathsf{H}(C_1)$. The leecher next replays L_3 to get to state $C_3 + L_3$, and takes the intermediate checkpoint C'_2 and calculates the hash $HC'_2 = \mathsf{H}(C'_2)$. If HC'_2 matches HC_2 from the 2nd replica, and the hashes of log segments L_2 and L_1 from the 3rd replica are equal to HL_2 and HL_1 from the 2nd replica, then the first validation is successful. Next, a similar procedure is applied to replay L_2 and the validation data from the 1st replica. Now, the leecher only needs to replay L_1 to reach the state corresponding to the execution of request id.

While the state transfer protocol is running, replicas continue to create new checkpoints and logs since the recovery does not stop the processing of new requests. Therefore, they are required to keep old log segments and checkpoints to improve their chances to support the recovery of a slow leecher. However, to bound the required storage space, these old files are eventually removed, and the leecher might not be able to collect enough data to complete recovery. When this happens, it restarts the algorithm using a more recent request id (a similar solution exists in all other state state transfer protocols that we are aware of, e.g., [6, 8]).

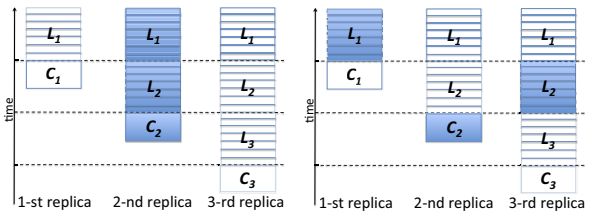
The leecher observes the execution of the other replicas while running CST, and stores all received messages concerning agreements more recent than id in an out-of-context buffer. At the end of CST, it uses this buffer to catch up with the other replicas, allowing it to be reintegrated in the state machine.

Correctness. We present here a brief correctness argument of the CST protocol. Assume that b is the actual number of faulty (Byzantine) replicas (lower or equal to f) and r the number of recovering replicas.

In terms of safety, the first thing to observe is that CST returns if and only if the state is validated by at least $f+1$ replicas. This implies that the state reached by the leecher at the end of the procedure is valid according to at least one correct replica. To ensure that this state is recent, the largest agreement id that is returned by $f+1$ replicas is used.

Regarding liveness, there are two cases to consider. If $b+r \leq f$, there are still $n-f$ correct replicas running and therefore the system could have made progress while the r replicas were crashed. A replica is able to recover as long as checkpoints and logs can be collected from the other replicas. Blocking is prevented because CST restarts if any of the *Fetch* commands fails or takes too much time. Consequently, the protocol is live if correct replicas keep the logs and checkpoints for a sufficiently long interval. This is a common assumption for state transfer protocols. If $b+r > f$, then there may not be enough replicas for the system to continue processing. In this case the recovering replica(s) will continuously try to fetch the most up to date agreement id from $n-f$ replicas (possibly including other recovering replicas) until such quorum exists. Notice that a total system crash is a special case of this scenario.

Optimizing CST for $f = 1$. When $f = 1$ (and thus $n = 4$), a single recovering replica can degrade the performance of the system because one of $n-f$ replicas will be transferring the checkpoint and logs, delaying the execution of the agreements (as illustrated in Figure 7(a)). To avoid this problem, we spread the data transfer between the active replicas through the following optimization in an *initial recovery round*: the 2nd replica ($f+1=2$) sends C_2 plus $\langle HL_1, HL_2 \rangle$ (instead of the checkpoint plus full log), while the 1st replica sends L_1 and HC_1 (instead of only hashes) and the 3rd replica sends L_2 (instead of not participating). If the validation of the received state



(a) General CST. (b) Optimized CST.

Figure 7: General and optimized CST with $f = 1$.

fails, then the normal CST protocol is executed. This optimization is represented in Figure 7(b), and in §5 we show the benefits of this strategy.

Simplifications for crash faults. When the SMR only needs to tolerate crash faults, a much simpler version of CST can be employed. The basic idea is to execute steps 1-4 of CST and then fetch and use the checkpoint and log from the 1st (most up to date) replica, since no validation needs to be performed. If $f = 1$, a analogous optimization can be used to spread the burden of data transfer among the two replicas: the 1st replica sends the checkpoint while the 2nd replica sends the log segment.

4 Implementation: Dura-SMaRt

In order to validate our techniques, we extended the open-source BFT-SMaRt replication library [1] with a durability layer, placed between the request ordering and the service. We named the resulting system *Dura-SMaRt*, and used it to implement two applications: a consistent key-value store and a coordination service.

Adding durability to BFT-SMaRt. BFT-SMaRt originally offered an API for invoking and executing state machine operations, and some callback operations to fetch and set the service state. The implemented protocols are described in [35] and follow the basic ideas introduced in PBFT and Aardvark [6, 10]. BFT-SMaRt is capable of ordering more than 100K 0-byte msg/s (the 0/0 microbenchmark used to evaluate BFT protocols [17, 25]) in our environment. However, this throughput drops to 20K and 5K msgs/s for 1kB and 4kB message sizes, respectively (the workloads we use – see §5).

We modified BFT-SMaRt to accommodate an intermediate *Durability layer* implementing our techniques at the server-side, as described in Figure 8, together with the following modifications on BFT-SMaRt. First, we added a new server side operation to deliver batches of requests instead of one by one. This operation supplies ordered but not delivered requests spanning one or more agreements, so they can be logged in a single write by the *Keeper* thread. Second, we implemented the parallel

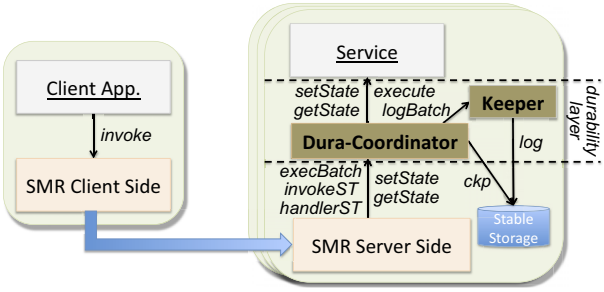


Figure 8: The Dura-SMaRt architecture.

checkpoints and collaborative state transfer in the *Dura-Coordinator* component, removing the old checkpoint and state transfer logic from BFT-SMaRt and defining an extensible API for implementing different state transfer strategies. Finally, we created a dedicated thread and socket to be used for state transfer in order to decrease its interference on request processing.

SCKV-store. The first system implemented with Dura-SMaRt was a *simple and consistent key-value store* (SCKV-Store) that supports the storage and retrieval of key-value pairs, alike to other services described in the literature, e.g., [11, 31]. The implementation of the SCKV-Store was greatly simplified, since consistency and availability come directly from SMR and durability is achieved with our new layer.

Durable DepSpace (DDS). The second use case is a durable extension of the DepSpace coordination service [2], which originally stored all data only in memory. The system, named Durable DepSpace (DDS), provides a tuple space interface in which tuples (variable-size sequences of typed fields) can be inserted, retrieved and removed. There are two important characteristics of DDS that differentiate it from similar services such as Chubby [4] and ZooKepper [19]: it does not follow a hierarchical data model, since tuple spaces are, by definition, unstructured; and it tolerates Byzantine faults, instead of only crash faults. The addition of durability to DepSpace basically required the replacement of its original replication layer by Dura-SMaRt.

5 Evaluation

This section evaluates the effectiveness of our techniques for implementing durable SMR services. In particular, we devised experiments to answer the following questions: (1) What is the cost of adding durability to SMR services? (2) How much does *parallel logging* improve the efficiency of durable SMR with synchronous disk and SSD writes? (3) Can *sequential checkpoints* remove the costs of taking checkpoints in durable SMR? (4) How

does *collaborative state transfer* affect replica recoveries for different values of f? Question 1 was addressed in §2, so we focus on questions 2-4.

Case studies and workloads. As already mentioned, we consider two SMR-based services implemented using Dura-SMaRt: the SCKV-Store and the DDS coordination service. Although in practice, these systems tend to serve mixed or read-intensive workloads [11, 19], we focus on write operations because they stress both the ordering protocol and the durable storage (disk or SSD). Reads, on the other hand, can be served from memory, without running the ordering protocol. Therefore, we consider a 100%-write workload, which has to be processed by an agreement, execution and logging. For the SCKV-Store, we use YCSB [11] with a new workload composed of 100% of replaces of 4kB-values, making our results comparable to other recent SMR-based storage systems [3, 32, 37]. For DDS, we consider the insertion of random tuples with four fields containing strings, with a total size of 1kB, creating a workload with a pattern equivalent to the ZooKeeper evaluation [19, 20].

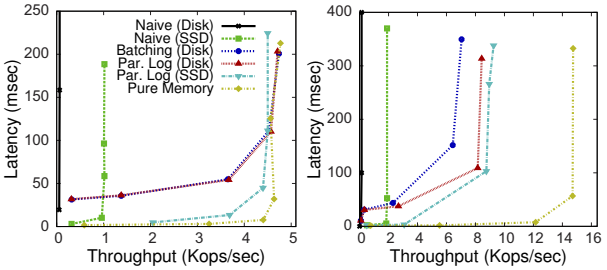
Experimental environment. All experiments, including the ones in §2, were executed in a cluster of 14 machines interconnected by a gigabit ethernet. Each machine has two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM memory, a 146 GB 15000 RPM SCSI disk and a 120 GB SATA Flash SSD. We ran the IOzone benchmark[5] on our disk and SSD to understand their performance under the kind of workload we are interested: rewrite (append) for records of 1MB and 4MB (the maximum size of the request batch to be logged in DDS and SCKV-Store, respectively). The results are:

Record length	Disk	SSD
1MB	96.1 MB/s	128.3 MB/s
4MB	135.6 MB/s	130.7 MB/s

Parallel logging. Figure 9(a) displays latency-throughput curves for the SCKV-Store considering several durability variants. The figure shows that naive (synchronous) disk and SSD logging achieve a throughput of 63 and 1017 ops/s, respectively, while a pure memory version with no durability reaches a throughput of around 4772 ops/s.

Parallel logging involves two ideas, the storage of batches of operations in a single write and the execution of operations in parallel with the secondary storage accesses. The use of batch delivery alone allowed for a throughput of 4739 ops/s with disks (a 75× improvement over naive disk logging). This roughly represents what would be achieved in Paxos [24, 27], ZooKeeper [19] or UpRight [9], with requests being logged during the agreement protocol. Interestingly, the addition of a separated thread to write the batch of operations, does not improve the throughput of this system. This occurs because a local put on SCKV-Store replica is very efficient, with almost no effect on the throughput.

[5] `http://www.iozone.org`.



(a) SCKV-Store. (b) Durable DepSpace.

Figure 9: Latency-throughput curves for several variants of the SCKV-Store and DDS considering 100%-write workloads of 4kB and 1kB, respectively. Disk and SSD logging are always done synchronously. The legend in (a) is valid also for (b).

The use of parallel logging with SSDs improves the latency of the system by 30-50ms when compared with disks until a load of 4400 ops/s. After this point, parallel logging with SSDs achieves a peak throughput of 4500 ops/s, 5% less than parallel logging with disk (4710 ops/s), with the same observed latency. This is consistent with the IOzone results. Overall, parallel logging with disk achieves 98% of the throughput of the pure memory solution, being the replication layer the main bottleneck of the system. Moreover, the use of SSDs neither solves the problem that parallel logging addresses, nor improves the performance of our technique, being thus not effective in eliminating the log bottleneck of durable SMR.

Figure 9(b) presents the results of a similar experiment, but now considering DDS with the same durability variants as in SCKV-Store. The figure shows that a version of DDS with naive logging in disk (resp. SSD) achieves a throughput of 143 ops/s (resp. 1900 ops/s), while a pure memory system (DepSpace), reaches 14739 ops/s. The use of batch delivery improves the performance of disk logging to 7153 ops/s (a 50× improvement). However, differently from what happens with SCKV-Store, the use of parallel logging in disk further improves the system throughput to 8430 ops/s, an improvement of 18% when compared with batching alone. This difference is due to the fact that inserting a tuple requires traversing many layers [2] and the update of an hierarchical index, which takes a non-negligible time (0.04 ms), and impacts the performance of the system if done sequentially with logging. The difference would be even bigger if the SMR service requires more processing. Finally, the use of SSDs with parallel logging in DDS was more effective than with the SCKV-Store, increasing the

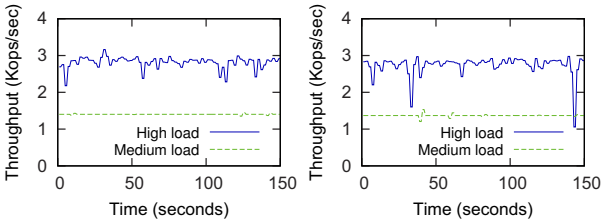
peak throughput of the system to 9250 ops/s (an improvement of 10% when compared with disks). Again, this is consistent with our IOzone results: we use 1kB requests here, so the batches are smaller than in SCKV-Store, and SSDs are more efficient with smaller writes.

Notice that DDS could not achieve a throughput near to pure memory. This happens because, as discussed in §3.1, the throughput of parallel logging will be closer to a pure memory system if the time required to process a batch of requests is akin to the time to log this batch. In the experiments, we observed that the workload makes BFT-SMaRt deliver batches of approximately 750 requests on average. The local execution of such batch takes around 30 ms, and the logging of this batch on disk entails 70 ms. This implies a maximum throughput of 10.750 ops/s, which is close to the obtained values. With this workload, the execution time matches the log time (around 500 ms) for batches of 30K operations. These batches require the replication library to reach a throughput of 60K 1kB msgs/s, three times more than what BFT-SMaRt achieves for this message size.

Sequential Checkpointing. Figure 10 illustrates the effect of executing sequential checkpoints in disks with SCKV-Store[6] during a 3-minute execution period.

When compared with the results of Figure 2 for synchronized checkpoints, one can observe that the unavailability periods no longer occur, as the 4 replicas take checkpoints separately. This is valid both when there is a high and medium load on the service and with disks and SSDs (not show). However, if the system is under stress (high load), it is possible to notice a periodic small decrease on the throughput happening with both 500MB and 1GB states (Figures 10(a) and 10(b)). This behavior is justified because at every $\lfloor \frac{P}{n} \rfloor$ requests one of the replicas takes a checkpoint. When this occurs, the replica stops executing the agreements, which causes it to become a bit late (once it resumes processing) when compared with the other replicas. While the replica is still catching up, another replica initiates the checkpoint, and therefore, a few agreements get delayed as the quorum is not immediately available. Notice that this effect does not exist if the system has less load or if there is sufficient time between sequential checkpoints to allow replicas to catch up ("Medium load" line in Figure 10).

[6]Although we do not show checkpoint and state transfer results for DDS due to space constraints, the use of our techniques bring the same advantage as on SCKV-Store. The only noticeable difference is due to the fact that DDS local tuple insertions are more costly than SCKV-Store local puts, which makes the variance on the throughput of sequential checkpoints even more noticeable (especially when the leader is taking its checkpoint). However, as in SCKV-Store, this effect is directly proportional to the load imposed to the system.



(a) 500MB state. (b) 1GB state.

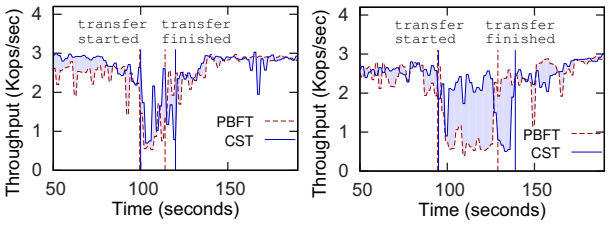
Figure 10: SCKV-Store throughput with sequential checkpoints with different write-only loads and state size.

Collaborative State Transfer. This section evaluates the benefits of CST when compared to a PBFT-like state transfer in the SCKV-Store with disks, with 4 and 7 replicas, considering two state sizes. In all experiments a single replica recovery is triggered when the log size is approximately twice the state size, to simulate the condition of Figure 7(b).

Figure 11 displays the observed throughput of some executions of a system with $n = 4$, running PBFT and the CST algorithm optimized for $f = 1$, for states of 500MB and 1GB, respectively. A PBFT-like state transfer takes 30 (resp. 16) seconds to deliver the whole 1 GB (resp. 500MB) of state with a sole replica transmitter. In this period, the system processes 741 (resp. 984) write ops/sec on average. CST optimized for $f = 1$ divides the state transfer by three replicas, where one sends the state and the other two up to half the log each. Overall, this operation takes 42 (resp. 20) seconds for a state of 1GB (resp. 500MB), 28% (resp. 20%) more than with the PBFT-like solution for the same state size. However, during this period the system processes 1809 (resp. 1426) ops/sec on average. Overall, the SCKV-Store with a state of 1GB achieves only 24% (or 32% for 500MB-state) of its normal throughput with a PBFT-like state transfer, while the use of CST raises this number to 60% (or 47% for 500MB-state).

Two observations can be made about this experiment. First, the benefit of CST might not be as good as expected for small states (47% of the normal throughput for a 500MB-state) due to the fact that when fetching state from different replicas we need to wait for the slowest one, which always brings some degradation in terms of time to fetch the state (20% more time). Second, when the state is bigger (1GB), the benefits of dividing the load among several replicas make state transfer much less damaging to the overall system throughput (60% of the normal throughput), even considering the extra time required for fetching the state (+28%).

We did an analogous experiment for $n = 7$ (not shown due to space constraints) and observed that, as expected, the state transfer no longer causes a degradation on the system throughput (both for CST and PBFT) since state is fetched from a single replica, which is available since



(a) 500MB and $n = 4$. (b) 1GB and $n = 4$.

Figure 11: Effect of a replica recovery on SCKV-Store throughput using CST with $f = 1$ and different state sizes.

$n = 7$ and there is only one faulty replica (see Figure 5). We repeated the experiment for $n = 7$ with the state of 1GB being fetched from the leader, and we noticed a 65% degradation on the throughput. A comparable effect occurs if the state is obtained from the leader in CST. As a cautionary note, we would like to remark that when using spare replicas for "cheap" faulty recovery, it is important to avoid fetching the state from the leader replica (as in [4, 8, 19, 32]) because this replica dictates the overall system performance.

6 Related Work

Over the years, there has been a reasonable amount of work about stable state management in main memory databases (see [16] for an early survey). In particular, parallel logging shares some ideas with classical techniques such as group commit and pre-committed transactions [14] and the creation of checkpoints in background has also been suggested [26]. Our techniques were however developed with the SMR model in mind, and therefore, they leverage the specific characteristics of these systems (e.g., log groups of requests while they are executed, and schedule checkpoints preserving the agreement quorums).

Durability management is a key aspect of practical crash-FT SMR-like systems [3, 8, 19, 20, 32, 37]. In particular, making the system use the disk efficiently usually requires several hacks and tricks (e.g., non-transparent copy-on-write, request throttling) on an otherwise small and simple protocol and service specification [8]. These systems usually resort to dedicated disks for logging, employ mostly synchronized checkpoints and fetch the state from a leader [8, 19, 32]. A few systems also delay state transfer during load-intensive periods to avoid a noticeable service degradation [19, 37]. All these approaches either hurt the SMR elegant programming model or lead to the problems described in §2.2. For instance, recent consistent storage systems such as Windows Azure Storage [5] and Spanner [12] use Paxos together with several extensions for ensuring durability. We believe works like ours can improve the modularity of future systems requiring durable SMR techniques.

BFT SMR systems use logging, checkpoints, and state transfer, but the associated performance penalties often do not appear in the papers because the state is very small (e.g., a counter) or the checkpoint period is too large (e.g., [6, 10, 17, 21, 25]). A notable exception is UpRight [9], which implements durable state machine replication, albeit without focusing on the efficiency of logging, checkpoints and state transfer. In any case, if one wants to sustain a high-throughput (as reported in the papers) for non-trivial states, the use of our techniques is fundamental. Moreover, any implementation of proactive recovery [6, 36] requires an efficient state transfer.

PBFT [6] was one of the few works that explicitly dealt with the problem of optimizing checkpoints and state transfer. The proposed mechanism was based on copy-on-write and delta-checkpoints to ensure that only pages modified since the previous checkpoint are stored. This mechanism is complementary to our techniques, as we could use it together with the sequential checkpoints and also to fetch checkpoint pages in parallel from different replicas to improve the state transfer. However, the use of copy-on-write may require the service definition to follow certain abstractions [7, 9], which can increase the complexity of the programming model. Additionally, this mechanism, which is referred in many subsequent works (e.g., [17, 25]), only alleviates but does not solve the problems discussed in §2.2.

A few works have described solutions for fetching different portions of a database state from several "donors" for fast replica recovery or database cluster reconfiguration (e.g., [23]). The same kind of techniques were employed for fast replica recovery in group communication systems [22] and, more recently, in main-memory-based storage [31]. There are three differences between these works and ours. First, these systems try to improve the recovery time of faulty replicas, while CST main objective is to minimize the effect of replica recovery on the system performance. Second, we are concerned with the interplay between logging and checkpoints, which is fundamental in SMR, while these works are more concerned with state snapshots. Finally, our work has a broader scope in the sense that it includes a set of complementary techniques for Byzantine and crash faults in SMR systems, while previous works address only crash faults.

7 Conclusion

This paper discusses several performance problems caused by the use of logging, checkpoints and state transfer on SMR systems, and proposes a set of techniques to mitigate them. The techniques – parallel logging, sequential checkpoints and collaborative state transfer – are purely algorithmic, and require no additional support (e.g., hardware) to be implemented in commodity

servers. Moreover, they preserve the simple state machine programming model, and thus can be integrated in any crash or Byzantine fault-tolerant library without impact on the supported services.

The techniques were implemented in a durability layer for the BFT-SMaRt library, which was used to develop two representative services: a KV-store and a coordination service. Our results show that these services can reach up to 98% of the throughput of pure memory systems, remove most of the negative effects of checkpoints and substantially decrease the throughput degradation during state transfer. We also show that the identified performance problems can not be solved by exchanging disks by SSDs, highlighting the need for techniques such as the ones presented here.

Acknowledgements. Thanks to the anonymous reviewers, John Howell and Lorenzo Alvisi, our shepherd, for the comments that helped improve the paper. This work was partially supported by the EC FP7 through project TCLOUDS (ICT-257243), by the FCT through project RC-Clouds (PTDC/EIA-EIA/115211/2009), the Multi-annual Program (LASIGE), and contract PEst-OE/EEI/LA0021/2011 (INESC-ID).

References

[1] BFT-SMaRt project page. `http://code.google.com/p/bftsmart`, 2012.

[2] A. Bessani, E. Alchieri, M. Correia, and J. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *EuroSys*, 2008.

[3] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[4] M. Burrows. The Chubby lock service. In *OSDI*, 2006.

[5] B. Calder et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[6] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[7] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, Aug. 2003.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live - An engineering perspective. In *PODC*, 2007.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *SOSP*, 2009.

[10] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[11] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[12] J. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[13] J. Dean. Google: Designs, lessons and advice from building large distributed systems. In *Keynote at LADIS*, Oct. 2009.

[14] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[15] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[16] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.

[17] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *EuroSys*, 2010.

[18] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. In *FTCS*, 1995.

[19] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale services. In *USENIX ATC*, 2010.

[20] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[21] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *EuroSys*, 2012.

[22] R. Kapitza, T. Zeman, F. Hauck, and H. P. Reiser. Parallel state transfer in object replication systems. In *DAIS*, 2007.

[23] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, 2001.

[24] J. Kirsh and Y. Amir. Paxos for system builders: An overview. In *LADIS*, 2008.

[25] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, Dec. 2009.

[26] K.-Y. Lam. An implementation for small databases with high availability. *SIGOPS Operating Systems Rev.*, 25(4), Oct. 1991.

[27] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[28] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp file system. In *SOSP*, 1991.

[29] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *EuroSys*, 2006.

[30] R. Miller. Explosion at The Planet causes major outage. *Data Center Knowledge*, June 2008.

[31] D. Ongaro, S. M. Ruble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[32] J. Rao, E. J. Shenkita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *VLDB*, 2011.

[33] M. Ricknäs. Lightning strike in Dublin downs Amazon, Microsoft clouds. *PC World*, Aug. 2011.

[34] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[35] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *EDCC*, 2012.

[36] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Veríssimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, Apr. 2010.

[37] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX ATC*, 2012.

Estimating Duplication by Content-based Sampling

Fei Xie Michael Condict Sandip Shete
{fei.xie, michael.condict, sandip.shete}@netapp.com
Advanced Technology Group, NetApp Inc.

Abstract

We define a new technique for accurately estimating the amount of duplication in a storage volume from a small sample and we analyze its performance and accuracy. The estimate is useful for determining whether it is worthwhile to incur the overhead of deduplication. The technique works by scanning the fingerprints of every block in the volume, but only including in the sample a single copy of each fingerprint that passes a filter. The selectivity of the filter is repeatedly increased while reading the fingerprints, to produce the target sample size. We show that the required sample size for a reasonable accuracy is small and independent of the size of the volume. In addition, we define and analyze an on-line technique that, once an initial scan of all fingerprints has been performed, efficiently maintains an up-to-date estimate of the duplication as the file system is modified. Experiments with various real data sets show that the accuracy is as predicted by theory. We also prototyped the proposed technique in an enterprise storage system and measured the performance overhead using the IOzone micro-benchmark.

1 Introduction

Deduplication detects and removes duplicate data blocks (blocks at different locations that have the same contents) from a storage system. In a system implementing perfect deduplication, only one copy of duplicate data blocks is stored, but in such a way that the user's view of the system remains unchanged. A. El-Shimi et al. [21] provide a nice overview of the recent research work in the area of data deduplication.

The benefit of deduplication in a primary storage system varies for different workloads. For certain workloads that have a low level of duplication, one would turn off the deduplication feature to avoid its effect on I/O performance and to avoid the metadata overhead of deduplication. It is desirable to have an efficient and effective deduplication estimator to allow customers to quickly estimate the deduplication benefit on their primary data sets before they turn on deduplication, and to allow the storage system to prioritize the scheduling of deduplication tasks for different data sets.

Existing deduplication estimators are either not fast enough or not accurate enough. A simple but intrusive and time-consuming way to discover the benefit of deduplication is to actually turn on deduplication. If the benefit is not satisfactory, deduplication can be reverted. Alternatively, one could roughly estimate the potential benefit of deduplication based on the type of workload. This approach often does not produce accurate estimates, since it does not look at the content of data.

Taking the content into account, one could attempt to estimate the level of duplication by reading a small random sample of the data set, and calculating the amount of duplication in it. This is much harder than it sounds, because of the large error in estimating the true number of occurrences of an item that only occurs once or twice in the sample. Furthermore, it has been proven that for any random-sampling-based estimation function, there are block-frequency distributions that cause it to be very inaccurate, unless the sample percentage is very large fraction of the data [1].

We defined and implemented an accurate and lightweight deduplication-estimation technique for a primary storage system. At a very high level, the technique samples the blocks based on the block fingerprint value, and only selects the fingerprints that satisfy some predicate on their value (i.e., a filter). That is, any two blocks with the same fingerprint are either both included in the sample or both excluded from the sample. This is why the sample is said to be content-based.

The remainder of this paper consists of a comparison to previous work (Section 2), an analysis of the algorithm (Section 3), a discussion of the design and implementation of the system (Section 4), a performance study (Section 5), and our conclusions (Section 6).

2 Related Work

Many commercial storage vendors provide deduplication estimators to allow customers to estimate potential space savings. A popular approach is to use rule-of-

thumb estimation methods which involve looking at metadata information like the type of data set, the frequency that data is changed, annual data growth rate, data retention, etc., which tend to influence deduplication ratios [11]. Many commercial estimators [9], [10] have adopted this method. Note that these estimators do not access the actual data in order to calculate the estimates, and so, can sometimes be extremely inaccurate.

The problem of estimating the duplication in a data set can be thought of as estimating the number of distinct block values in the set, given that the total number of blocks in use is known. This means that solutions to the latter problem can be applied to the former.

Distinct elements estimation is a well-studied problem, and frequently appears in literature concerning data-streaming algorithms, statistics, and databases. P.B. Gibbons [8] looks at many previous approaches to distinct value estimation and the difficulties with them. In the database world, the early literature has extensively studied sampling techniques, which involve gathering a uniform random sample of the data, and using it to approximately answer distinct-value queries on relational databases [5, 6, 7]. Although these estimators use sophisticated techniques to handle various input distributions, they are all unable to guarantee good accuracy for their estimates [1, 4]. Charikar et al. [1] proved this formally, establishing a strong negative result, namely that no estimator can guarantee a small error for all possible input distributions unless it examines a large fraction of the input data. Raskhodnikova et al. [2] and Valiant et al. [3] further provided near-linear and sub-linear lower bounds, respectively, on the sample size required for the estimate. The conclusion is that, in order to ensure high-accuracy, distribution-independent estimates, it is necessary to examine almost the entire data set.

Estimation algorithms that require scanning all the data once are referred to as single-pass algorithms. Flajolet and Martin, in their seminal work [12], presented the first single-pass algorithm for distinct values estimation in a large collection of data using small limited storage. Their probabilistic counting algorithm uses hash functions to map set of values to bitmap vectors, such that each distinct value maps to the i^{th} bit in the vector with $2^{-(i+1)}$ probability. Alon et al. [13] further build upon this work and proposed more practical hash functions, space bounds and provable error guarantees on their estimates. This line of research continues with more space/time efficient algorithms and better estimates of distinct values [14, 15]. Some similar approaches use adaptive sampling, which continuously maintain a bounded-size up-to-date sample of distinct values for the purpose of providing a very quick estimate of the cardinality of the data set [17, 18, 26]. Our sampling technique is in many respects similar to these.

D. Harnik et al. [19] are the first in the area of storage deduplication to provide a provably accurate two-phase algorithm for a one-time estimate of deduplication ratios, using very low storage space. Our technique differs from theirs in that, after a single pass over existing data for the initial estimate, it uses an adaptive technique to incrementally maintain an up-to-date estimate that takes into account any changes to the data.

3 Theory

Consider a data set consisting of a group of data blocks (of fixed size or variable size) with possible duplicates. We are interested in estimating the percentage of space that can be saved by deduplication. Thus, we define the deduplication ratio R as follows.

$$R = \frac{Size\ in\ bytes\ of\ distinct\ blocks\ in\ data\ set}{Size\ in\ bytes\ of\ the\ data\ set}$$

We assume a hash function that generates a fingerprint for each data block. The proposed *content-based sampling* applies a modulo-based filter to all the block fingerprints of a data set. A block fingerprint passes the filter and is added to the sample iff:

$$Fingerprint\ \mathbf{Mod}\ M = X$$

Where the divisor M is an integer greater than 1, and the remainder X is an integer between 0 and M - 1. Throughout this paper, we refer to M as the *filter divisor*. The idea is to split the fingerprint space into M partitions, and to use one of the partitions in the estimate. More specifically, the total size of the distinct (deduplicated) blocks in the sample is used to estimate the total size of the distinct blocks in the entire data set.

Assume there are K different block sizes in the data set. The sample can be partitioned into K groups of identical-sized blocks. Assume the ith ($i = 1 \ldots K$) block group in the sample has n_i distinct blocks of size s_i. Let N_i denote the total number of distinct blocks of size s_i in the data set. Let S be the size in byte of the distinct blocks in the data set. The estimate of S, denoted by S^*, is defined as:

$$S^* = M \cdot \sum_{i=1}^{K} n_i \cdot s_i.$$

The deduplication ratio can be estimated as $S^*/S_{\text{data\_set}}$, where $S_{\text{data\_set}}$ is the size of the data set before deduplication, which is known before the. In the case of fixed-size blocks, we can ignore the block size and count only the number of distinct blocks in the sample. Figure 1 illustrates the idea of content-based sampling.

The main theoretical result of this work is the relationship between the filter divisor and the accuracy of the estimate. Define the relative error of the estimate as

$$err = (S - S^*)/S$$

We show that if the fingerprinting algorithm has good uniformity and negligible collision probability, the relative error follows a normal distribution with zero mean (i.e., the estimate is unbiased) and known variance.

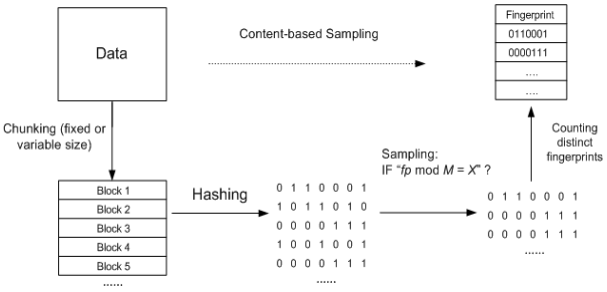


Figure 1. Illustration of the Content-based sampling

We assume a storage system that maintains a relatively strong fingerprint for data blocks, so that we can ignore the impact of collisions. A previous work in the networking domain [16] takes collisions into account in the estimate. As shown in that work, the expected error introduced by collisions (which always causes an underestimate) is computable from the size of the uncorrected estimate S^*. Thus, we could apply these results to correct for collisions, if necessary.

Theorem 1. Assume the fingerprinting has the uniformity property and negligible collision probability. For large N_i (i = 1,.., K), *err* has a normal distribution with zero as the mean and $(M-1)\cdot \bar{s}/S$ as the variance, where $\bar{s}$ is defined as $\bar{s} = \sum_{i=1}^{K} s_i \cdot (N_i \cdot s_i)/S$.

The proof of Theorem 1 is in Appendix A. The *α-β* accuracy of the estimate is defined as:

Definition of *α-β* accuracy. Given α and β ($\alpha, \beta \in [0, 1]$), the relative error *err* satisfies the following condition.

$$Prob(|err| \geq \alpha) \leq 1 - \beta$$

The relationship between M and the *α-β* accuracy is described in the following theorem.

Theorem 2. M satisfies the *α-β* accuracy if:

$$M \leq \frac{\alpha^2 \cdot S}{2\cdot(erf^{-1}(\beta))^2 \cdot \bar{s}} + 1 \qquad (1)$$

where $erf^{-1}()$ is the inverse of the Gauss error function and $\bar{s}$ is defined in Theorem 1 (proof in Appendix B).

To have the smallest possible sample size, one would choose the largest M that satisfies a given accuracy requirement. However, S and $\bar{s}$ are not known before the estimation. In practice, we could address this problem as follows. Variable-size chunking algorithms (e.g., [24]) typically have a known average block size, which can be used to approximate $\bar{s}$. Also, $\bar{s}$ is known for the fixed-size blocks case. Rewrite inequality (1) as:

$$\frac{S}{M-1} \geq \frac{2\cdot(erf^{-1}(\beta))^2 \cdot \bar{s}}{\alpha^2} \qquad (2)$$

The left side of (2) could be approximated by the total size of distinct blocks in sample (distinct block count in the fixed-size blocks case), which is countable during the sampling. The minimum "distinct sample size" that satisfies (2) is called the *target sample size*. Table 1 gives some α, β values and the corresponding target sample size in number of blocks. As long as there are enough distinct blocks in the sample, we can increase the selectiveness of the filter to reduce sample size in the following *Adaptive Sampling* Approach.

During the sampling process, the ratio of the distinct block count in sample to the *target sample size* is periodically monitored. If the ratio is greater than 2, we find the largest power of two that is less than or equal to the ratio (denoted as *f*). M and X are updated as follows:

Step 1: $X = X + M \cdot rand(f)$
Step 2: $M = M \cdot f$

The function rand(*f*) generates a random integer between 0 and $f - 1$. This allows us to randomly choose a fingerprint partition while we aggressively divide the fingerprint space. Finally, we remove the unqualified blocks from the existing sample, and continue the sampling with the new, more restrictive filter.

TABLE 1. Target sample size vs *α-β* accuracy

Target sample size	α	β
270	0.1	0.9
1843	0.06	0.99
12030	0.03	0.999
1513670	0.01	0.9999

4 Design and Implementation

We chose an enterprise-class network-attached storage system as the reference system in which to implement the estimation technique. The system uses a log-structured file system [22] with 4KB blocks. Individual data blocks of a file can be identified by a file handle together with an offset within the file (called a file block number). Our implementation estimates the number of unique data blocks in a volume. Ignoring the initialization delay due to the one-time full-volume scan, an up-to-date estimate can be returned on-demand with-

in a couple of minutes, no matter how large the volume is. A large part of the recurring maintenance runs in the background, in a non-intrusive fashion.

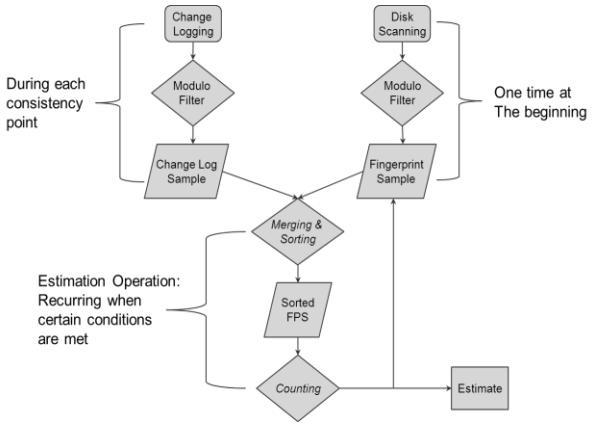


Figure 2. System Components

The major modules built into the reference storage system are depicted in Figure 2. The ***change logging*** is a software module that samples *data blocks* during the consistency point. Metadata blocks are not sampled. The storage system computes a 64-bit variant of the Adler checksum [23] for each block as the RAID checksum. This checksum, available at the time of a consistency point, is used as the fingerprint. Existing blocks in the volume are sampled by a ***scanner*** module. The sample is stored in a fingerprint sample file (FPS). The FPS contains a header and a sequence of entries (20 bytes per entry) in the format of {file handle, file block number, fingerprint}.

The ***estimation operation*** merges the sample from change logging to the FPS, and updates the estimate accordingly. File swapping is used in the merging, so that the ongoing change logging process is not affected. After merging, we count the distinct blocks by sorting the FPS. We remove stale entries (i.e., blocks removed or over-written) before counting. This is done by comparing the fingerprint in the entry with the fingerprint of the real block. We maintain a stale inode cache during the validation, to reduce the number of unnecessary block-read attempts.

Adaptive sampling is triggered at the end of the estimation operation. Once the filter divisor increases, the change logging produces a smaller sample. The FPS is shrunk as well. The initial value of M can be set by the user. The initial value of X is chosen randomly.

The estimation operation is triggered if we have enough data in the change logs or there is file deletion in the volume (i.e., volume size decrease). These conditions are checked periodically. This ensures minimum impact to read-intensive workloads.

5 Performance Study

We studied the accuracy of our estimate using real-world data sets (see TABLE 2). Given a data set, we compared the empirical error's standard deviation and the theoretical ones, for various values of M. There were 1000 data points for each empirical statistics, generated by varying the remainder X from 0 to 999. We tested estimations over both 4KB fixed-size blocks, and variable-size blocks [24]. The variable block size is between 2KB and 8KB. The true deduplication ratio was obtained as follows. In the case of variable-size blocks, we trust the results of deduplicating over the MD5 hash values of the blocks. For fixed-size blocks, we deduplicated the data set in a NetApp® system.

TABLE 2. Information of the data sets

Names	*Size*	*Dedupe Ratio*	*Description*
Corp. Web	1.5TB	~50%	Corporate web directory
Debian	260GB	~60%	2-month Debian build
Sharepoint	29GB	~18%	Corporate Sharepoint

We consistently saw good matches between the empirical results and the theoretical results, for both the variables-size and fixed-size cases (see Figure 3 for details). For the sake of space, we only report selected results in this paper.

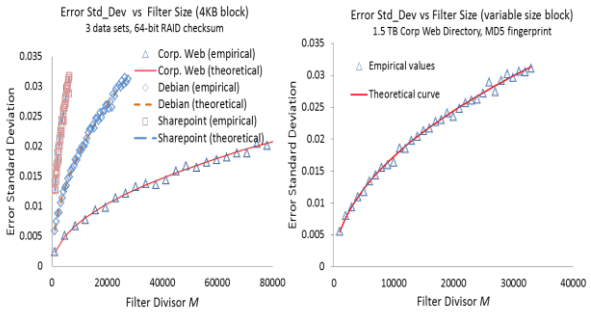


Figure 3. Accuracy test result

The evaluations of the prototype were done using a NetApp FAS 3070 storage system running Data ONTAP® 8.1 [25]. IOzone [20] was chosen as the synthetic I/O trace generator, since it can generate traces with duplicated content. The storage system exported a NFS v3 volume to the trace-generating client.

There are two major types of test, namely *64KB sequential write* and *64KB sequential read*. All the tests were set to 50% inter-file duplication and 0% intra-file duplication. Five files are accessed in parallel in the tests, which saturated a 1GB network link. The deduplication ratio of the synthetic data set is 0.6. Every 4

minutes, the system checked whether the estimation operation should be triggered. Every 5 seconds, the storage system sampled the CPU usage, number of 64KB I/O per second (IPOS), disk read rate, and disk write rate. A single test run lasted for about 80 minutes. There were 15 test runs for a single setting.

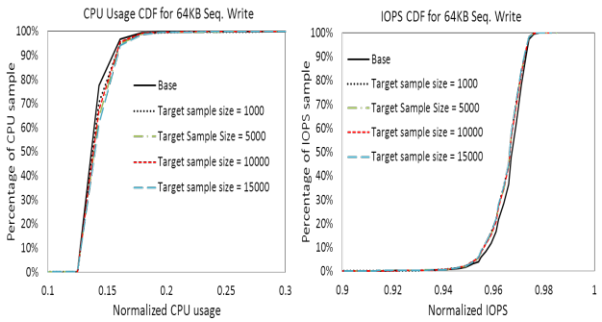


Figure 4. CDF for normalized CPU usage and IOPS

We first look at the results for 64KB sequential writes. We only report the cumulative distribution function (CDF) curves of CPU usage and IPOS in Figure 4. As expected, the estimation test consumes more CPU and has less IOPS, compared to the base case. Besides these results, our data shows that the average changes in CPU usage, IOPS, and disk reads are less than 0.5% for all the tested target sample sizes. The average disk read rate increases from 1% to 4% as the target sample size increases from 1000 to 15000. The additional disk reads are mainly contributed by the random disk access from the counting module.

Figure 5. Latency histogram for both read and write tests

There is no significant impact to the system in the sequential read test. This is because the estimation code has negligible overhead for a read-only workload. The CDF results are not reported due to space limitations. Figure 5 plots the client-side latency histogram reported by IOzone. This plot shows that estimation's impact to the I/O latency is also negligible.

We also studied the estimation accuracy of our adaptive sampling in the 64KB sequential write case. Figure 6 plots the results in one test run. As the volume size grows, the filter divisor is doubled while the corresponding distinct block count in the sample floats between 2500 and 1000. *M* was initially set to 4096. The error is high at the beginning due to the small sample size, and remains below 5% later.

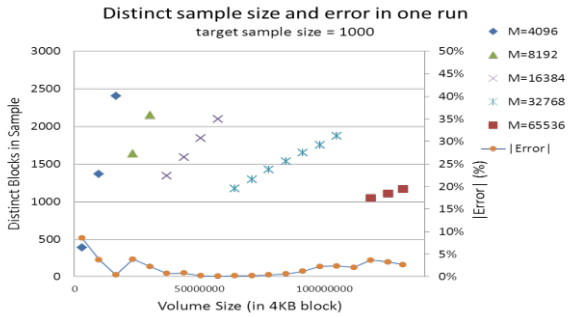


Figure 6. Adaptive sampling in one experiment run.

6 Concluding Remarks

The main contribution of this work is a method that estimates duplication in a storage system with statistically guaranteed accuracy using a single scan of the data set. It is also notable for requiring only a small, fixed amount of memory resources for a given level of accuracy, independent of the size of the volume. This makes it amenable to efficient in-line use and to maintain an accurate estimate in the face of a rapidly changing data set, which allows storage users to better assess the utility of data deduplication. We implemented the technique as a practical enhancement to a commercial storage system, and confirmed that the accuracy was within the statistically expected range for a variety of real-world data sets. The performance impact of our technique was found to be less than a few percent.

7 References

[1] M. Charikar, et al. Towards Estimation Error Guarantees for Distinct Values. Proceedings of the 19th ACM Symposium on Principles of Database Systems. ACM, New York, 2000.

[2] S. Raskhodnikova, et al. Strong Lower Bounds for Approximating Distribution Support Size and the Distinct Elements Problem. *SIAM Journal on Computing*, pages 813-842, 2009.

[3] G. Valiant, and P. Valiant. Estimating the Unseen: An n/log(n)-sample Estimator for Entropy and Support Size, Shown Optimal via New CLTs. In the *43rd ACM Symposium on Theory of Computing,* STOC, pages 685-694, 2011.

[4] S. Chaudhuri et al. Random sampling for histogram construction: How much is enough? In Proc. ACM SIGMOD International Conf. on Management of Data, pages 436–447, June 1998.

[5] P. J. Haas, et al. Sampling-based estimation of the number of distinct values of an attribute. In Proc. 21st International Conf. on Very Large Data Bases, pages 311–322, September 1995.

[6] G. Ozsoyoglu, et al. On estimating COUNT, SUM, and AVERAGE relational algebra queries. In Proc. Conf. on Database and Expert Systems Applications , pages 406–412, 1991.

[7] F. Olken. Random Sampling from Databases. PhD thesis, Computer Science, U.C. Berkeley, April 1993.

[8] P. B. Gibbons. Distinct-values estimation over data streams. In Manuscript, 2009.

[9] http://www.emcemearegistration.com/tapereplace/esquare/calculator.php - EMC Data Domain

[10] http://www.itcalc.com/ - NetApp Inc.

[11] https://www.snia.org/sites/default/files/Understanding_Data_Deduplication_Ratios-20080718.pdf

[12] P. Flajolet et al. Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. 31(2): 182-209 (1985).

[13] N. Alon et al. The space complexity of approximating the frequency moments. ACM STOC, 1996, pp. 20–29.

[14] P. Flajolet, et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In AOFA 2007.

[15] D. M. Kane et al. An optimal algorithm for the distinct elements problem. In PODS, pp. 41–52, 2010.

[16] C. Estan, G. Varghese, and M. E. Fisk. Bitmap algorithms for counting active flows on high-speed links. IEEE/ACM Transactions on Networking, 14(5):925-937, 2006.

[17] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. VLDB'01, pp 541–550

[18] Counting distinct items over update streams. ACM Journal Theoretical Computer Science 378(3):211-222, 2007

[19] D. Harnik et al. Estimation of deduplication ratios in large data sets. In Mass Storage Systems and Technologies (MSST), 2012

[20] IOzone Filesystem Benchmark http://www.IOzone.org/

[21] A. El-Shimi et al. Primary Data Deduplication -- Large Scale Study and System Design. USENIX ATC '12.

[22] D. Hitz et al. File system design for a file server appliance. In USENIX Technical Conference, 1994, pages 235–245

[23] P. Corbett el al. Row-Diagonal Parity for Double Disk Failure Correction. In Proceedings of the 2004 Usenix FAST, pp: 1- 14.

[24] E. Kave et al. A Framework for Analyzing and Improving Content-Based Chunking Algorithms No. HPL-2005-30R1.

[25] www.netapp.com/us/library/technical-reports/tr-3982.html

[26] A. Chen & A. Cao. Distinct counting with a self-learning bitmap. IEEE ICDE '09. Pages 1171–1174.

Appendix A

Denote the total number of distinct *fixed-size blocks* in the sample and data set as n_d and N_d , respectively.

Lemma 1. Assume the fingerprint has uniformity property and negligible collision probability. For large N_d , *err* has a normal distribution with zero as the mean and $(M-1)/N_d$ as the variance.

Proof: Because of good uniformity, any fingerprint in the data sets has $1/M$ probability to be sampled. Since the collision is negligible, the number of distinct fingerprints is approximately equal to the number of distinct blocks in the sample. The sampling can be treated as a Bernoulli trail of length N_d and successful rate $1/M$. The number of successes in the trail is equal to n_d. Therefore n_d follows a binomial distribution. Based on the definition of *err*, it can be represented as:

$$err = (R - R^*)/R = 1 - n_d \cdot M/N_d$$

When N_d is large, the distribution of n_d can be approximated as a normal distribution with $\mathrm{E}[n_d] = N_d / M$ and $\mathrm{Var}[n_d] = N_d \cdot (M-1)/M^2$. Therefore *err* follows a normal distribution of zero mean and $(M-1)/N_d$ as the variance. ■

Proof of Theorem 1: The sample in the variable-size blocks case can be seen as a group of K fixed-size block samples. According to Lemma 1, when N_i is large, n_i has a normal distribution: $\mathrm{E}[n_i] = N_i / M$ and $\mathrm{Var}[n_i] = N_i \cdot (M-1)/M^2$. Since S^* is a linear combination of n_i ($i = 1,\ldots,K$), S^* also follows a normal distribution with

$E[S^*] = \sum_{i=1}^{K} s_i \cdot N_i = S$ and $Var[S^*] = (M-1) \cdot \sum_{i=1}^{K} N_i \cdot {s_i}^2$.

Since $err = (S - \mathrm{S}^*)/\mathrm{S}$, it also has a normal distribution. E[*err*] is simply zero. The variance is calculated as:

$$Var[err] = \frac{Var[S^*]}{S^2} = (M-1) \cdot \sum_{i=1}^{K} N_i \cdot \frac{{s_i}^2}{S^2}$$

$$= \frac{(M-1)}{S} \cdot \sum_{i=1}^{K} s_i \cdot \frac{N_i \cdot s_i}{S} = \frac{(M-1) \cdot \bar{s}}{S}$$

This proves the theorem. ■

Appendix B

Proof: The proof of this theorem is simply based on the Empirical Rule of the normal distribution. Since the error has a normal distribution, the accuracy of the estimation has the following property.

$$Prob\left(|err| \geq \varepsilon \cdot \sqrt{Var[err]}\right) \leq 1 - erf\left(\varepsilon/\sqrt{2}\right)$$

, where *erf*() is the error function and ε is a constant. Substitute the variance of *err* from Theorem 1, we have:

$$Prob\left(|err| \geq \varepsilon \cdot \sqrt{(M-1) \cdot \bar{s}/S}\right) \leq 1 - erf\left(\varepsilon/\sqrt{2}\right)$$

We substitute ε with $\mathrm{erf}^{-1}(\beta) \cdot \sqrt{2}$ in the above inequality, which yields:

$$Prob\left(|err| \geq erf^{-1}(\beta) \cdot \sqrt{2} \cdot \sqrt{(M-1) \cdot \bar{s}/S}\right) \leq 1 - \beta$$

This means that as long as:

$$M \leq \frac{\alpha^2 \cdot S}{2 \cdot (erf^{-1}(\beta))^2 \cdot \bar{s}} + 1$$

the estimation satisfies α-β accuracy. ■

MutantX-S: Scalable Malware Clustering Based on Static Features

Xin Hu[1] Sandeep Bhatkar[2] Kent Griffin[2] Kang G. Shin[3]
[1] *IBM T.J. Waston Research Center* [2] *Symantec Research Labs* [3] *University of Michigan, Ann Arbor*

Abstract

The current lack of automatic and speedy labeling of a large number (thousands) of malware samples seen everyday delays the generation of malware signatures and has become a major challenge for anti-virus industries. In this paper, we design, implement and evaluate a novel, scalable framework, called `MutantX-S`, that can efficiently cluster a large number of samples into families based on programs' static features, i.e., code instruction sequences. `MutantX-S` is a unique combination of several novel techniques to address the practical challenges of malware clustering. Specifically, it exploits the instruction format of x86 architecture and represents a program as a sequence of opcodes, facilitating the extraction of *N*-gram features. It also exploits the hashing trick recently developed in the machine learning community to reduce the dimensionality of extracted feature vectors, thus significantly lowering the memory requirement and computation costs. Our comprehensive evaluation on a `MutantX-S` prototype using a database of more than 130,000 malware samples has shown its ability to correctly cluster over 80% of samples within 2 hours, achieving a good balance between accuracy and scalability. Applying `MutantX-S` on malware samples created at different times, we also demonstrate that `MutantX-S` achieves high accuracy in predicting labels for previously unknown malware.

1 Introduction

According to the Symantec's latest Internet Threat Report, 403 million new variants of malware were created in 2011, a 41% increase from 2010. This exponential growth of malware samples has created a major challenge for anti-virus (AV) companies: how to efficiently process this huge influx of new samples and accurately labels them? It is practically impossible to manually analyze several thousands of suspicious samples received every day. As a result, a large fraction of samples are left unlabeled, which delays the signature generation. One possible solution is to *automatically* cluster malware samples and assign them labels according to their similarities. The intuition is that malware programs bearing significant similarities are likely to have been derived from the same code base, and hence from the same malware family. One can thus group similar malware and label them with high accuracy by analyzing only a few representative samples. Moreover, the label of a new sample can be automatically derived and previous mitigation techniques can be re-used if it is determined to belong to an existing family. Therefore, accurate clustering plays a crucial role in helping AV companies categorize large amount of incoming samples by avoiding duplicate work and enabling malware analysts to prioritize limited resources on novel and representative samples [17, 12, 7]. In this paper, we design, implement and evaluate `MutantX-S`, a novel and scalable system, that can efficiently cluster a large number of malware samples into families based on their static features, i.e., code instruction sequences.

Many existing malware clustering/classification systems are based on dynamic behavioral features such as runtime API or system call traces [6, 7, 24]. The major benefit of using dynamic behavioral features is that they are less susceptible to mutation schemes frequently employed by malware writers to evade binary analysis, e.g., packing or obfuscation. However dynamic-feature-based approaches also suffer from several limitations. First, they may have only limited coverage of an application's behavior, failing to reveal the entire capabilities of a given malware program. This is because a dynamic analysis can only capture API or system call traces corresponding to the code path that was taken during a particular execution. Different code paths may be taken in different runs, depending on the program's internal logics and/or external environments. Also, malware often include triggers in their programs and exhibit an interesting behavior only when certain conditions are met. For example bot programs wait for commands from botmasters and some malware are designed to launch attacks on a certain time. Although there exists work that forces a program to run all code paths [21], they are too expensive to analyze large amount of malware. Second, dynamic analysis is inherently resource-intensive and doesn't scale well. With limited resource and the sheer number of malware, a dynamic-analysis system can execute and monitor each sample only for a short period of time, e.g., a couple of minutes. Unfortunately, this time is often too short for typical malware to reveal all their true behavior.

In this paper, we present `MutantX-S`, a new and practical system that exploits static features of code instruction sequences for efficient and automatic malware clus-

tering and labeling. MutantX-S is motivated by the common observation that majority of today's malicious programs are variations of a relative small number of malware families and thus share similar instruction sequences. Analyzing static features of malware offers several unique benefits. First, it has the potential to cover all possible code paths, yielding more accurate representations of the entire functionalities of the program. Moreover, approaches based on static features are much more scalable than their dynamic counterparts, as they do not require resource-intensive and time-consuming monitoring of program behavior. This is particularly important for AV companies to process a rapidly-increasing number of new malware samples. Unfortunately, static analysis is well-known to suffer from run-time packing and obfuscation techniques. Therefore, the goal of MutantX-S is not to replace existing dynamic-behavior-based systems, but to complement them to achieve higher clustering accuracy and better coverage of malware programs.

MutantX-S features a unique combination of techniques to address the deficiencies of static malware analysis. First, it tailors a generic unpacking technique to handle run-time packers without needs to know its specific packing algorithm. Second, it employs an efficient encoding mechanism that exploits the IA32 instruction format to encode a program into opcode sequences that are resilient to low level mutations. In addition, it applies a hashing-trick and a close-to-linear clustering algorithm to allow MutantX-S to efficiently handle large number of malware with very high dimensional features. We have successfully implemented a fully-automated prototype of MutantX-S and evaluated its performance using over 130,000 distinct malicious programs. Our evaluation demonstrates MutantX-S' efficiency and efficacy of creating clusters corresponding to malware families and accurately predicting labels for new malware.

The rest of the paper is organized as follows. Section 2 surveys related work of malware analysis. Section 3 describes the architecture of MutantX-S followed by elaboration of all subcomponents including unpacking (Section 4), feature extraction (Section 5) and clustering (Section 6). The performance evaluation is presented in Section 7. Section 8 discusses the limitation and potential improvement, and Section 9 concludes the paper.

2 Related Work

Malware pose one of the severest threats to computer systems and the Internet. Various schemes have been proposed to automatically cluster/classify malware based on either dynamic behavior or features.

Dynamic-analysis approaches have the major benefit of handling obfuscated malware samples based on their runtime system or API calls. Lee and Mody [18] used a sequence of runtime events (e.g., registry and file system modifications) to cluster similar malware programs. Rieck *et al.* [23] applied SVM (Support Vector Machine) to learn the frequency of run-time behavior, and classified unknown samples to their closest kin. Later, Bailey *et al.* [6] applied a hierarchical clustering algorithm to group similarly-behaving malware samples. Unfortunately, the complexity of this clustering algorithm is $O(n^2)$, limiting its applicability only to a small number of samples. To address this problem, Bayer *et al.* [7] and Rieck *et al.* [24] developed different methods to scale the clustering. Bayer *et al.* [7] applies locality-sensitive hashing (LSH) to efficiently compute an approximate hierarchical clustering with a significantly smaller number of distance computations. By contrast, Rieck *et al.* [24] applied a prototype-based clustering algorithm that reduces the runtime complexity by performing clustering only on representative samples. Comparing to LSH clustering, a prototype-based algorithm facilitates the analysis of behavior groups because each prototype represents a particular malware group [24]. In MutantX-S, we adopt the same prototype-based algorithm as in [24] because of its efficiency and explicit expression of malware features.

Static analysis, on the other hand, uses features extracted directly from malware binaries. Christodorescu *et al.* [8] discovered malicious patterns from disassembled malware that are resilient to obfuscation. Wicherski [30] utilizes static features from PE headers, e.g., entry point, import table, etc., to group malware programs. Karim *et al.* [13] demonstrates the effectiveness of *N*-gram and *N*-perm on assembly instructions by using them to study the malware evolution. Similar features have also been used in [15] to validate various learning methods. MutantX-S falls into the static-analysis category since it relies on features extracted from the malware instructions. MutantX-S differs from previous approaches in its unique combination of novel techniques to improve its scalability in handling very large malware datasets. Another independently developed system similar to MutantX-S is BitShred [12] which also focuses on malware comparison and triage on a large scale. However, BitShred compares malware using their byte sequences which is susceptible to binary level obfuscation.

3 Architecture

Figure 1 shows an overview of MutantX-S. At a high level, MutantX-S takes a set of malicious or suspicious samples as input and extracts their features using static analysis to avoid the computational overhead and maximize code coverage. Specifically, MutantX-S first uses existing tools (e.g., PeID[1] [3]) to identify malware files that are likely processed by packing tools such as UPX

[1] a popular packer detection tool that currently detect more than 470 different packer signatures in executables

[28], ASPack [5]. These files will be unpacked with a generic unpacking technique tailored for MutantX-S. Together with samples that are in their original binary (not packed), they are disassembled to code instructions. These pre-processing steps ensure that features inherent to malware families can be successfully extracted without influence of encryption or compression. Then, all malware samples are processed with three steps to extract representative features: (1) *Instruction Encoding* for converting each instruction to a sequence of operation codes that capture the underlying semantics of the programs, (2) *N-gram analysis* for constructing feature vectors used to compute program similarities, and (3) *Hashing Trick* for compressing the feature vectors, which significantly improves the speed of similarity computation with only a small penalty in accuracy. Finally, a prototype-based clustering algorithm is applied on compressed feature vectors and partitions samples into clusters, each representing a group of similar malware programs.

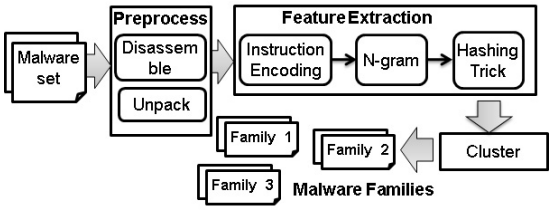


Figure 1: A system overview of MutantX-S

4 Generic Unpacking Algorithm

Run-time packing is arguably the most popular techniques used by malware writers to circumvent anti-virus detection. More than 80% of malware programs are estimated to be packed [10]. A typical packer like UPX works as follows. UPX first compresses all the code and data sections of a portable executable (PE) binary[2] into a single section. Then, it creates a new PE binary containing the compressed data followed by the unpacker code. The entry point in the new PE header is altered to point to the unpacker code such that when the packed program runs the unpacker will first be executed . The unpacker decompresses the original program codes into memory and then jump to the first instruction of the restored codes (i.e., the original entry point) to resume execution. This packing process enables malware programs to disguise their malicious instructions as random-looking data while keeping the original functionality intact. Since all static analysis tools including MutantX-S rely on features extracted from original instructions, it is imperative for them to handle packing correctly and efficiently.

While there exist unpacking tools such as UPX itself, ArmaGedoon, etc., they are often targeted specifically at one or a few packers. As more packers appear in the wild, the cost of manually reverse-engineering packers and continually updating unpacking tools is expected to grow over time. In addition, unpacking tools often have to perform expensive processing to ensure that the unpacked program can be successfully executed (e.g., the PE headers and imported tables must be correctly reconstructed), making them too slow for large scale processing. MutantX-S, on the other hand, need not guarantee the executability of unpacked programs as long as the original instructions can be inspected and features extracted. MutantX-S thus exploits this advantage and tailors a generic unpacking mechanism to meet the particular need for efficient malware clustering. The basic idea is to use the inherent property of the unpacking procedure, i.e., a packed binary has to write the unpacked code into some memory space and transfer control to the modified memory locations to continue execution. By continuously monitoring memory accesses, we can learn the occurrence of some form of unpacking, self-modification or on-the-fly code generation if the program executes code at a memory address after writing into it. These written-then-executed memory pages likely contain the original instructions and thus are the targets of MutantX-S.

The unpacking component of MutantX-S exploits the physical non-execution (NX) support in modern x86 CPUs to track memory page status. It consists of a kernel driver responsible for tracking system calls and a user-level component that is injected as a remote thread into a program's address space. The unpacking component does two things: (1) runs the packed binary and dumps the memory image of the running process at an appropriate time when the binary is likely to finish unpacking, and (2) determines the correct original entry point (OEP) for the dumped image. Finding the correct OEP is critical for correct program disassembling and feature extraction. A wrong entry point may cause a disassembler to miss all the instructions between the original and the misidentified entry points (if there is no other reference to this portion of codes). In addition, if the entry point is incorrectly set in the middle of an instruction, the disassembler will fail or generate completely wrong assembly codes. The unpacking process is summarized in Algorithm 1 and elaborated below:

1. MutantX-S loads the packed program, suspends its execution and injects the user-level hooking DLL into the process' memory space. It marks all the memory pages as *executable* but *non-writable*, and resumes its execution.

2. During the execution, when the unpacker attempts to write unpacked codes into memory (which has been marked as *non-writable*), a write exception will occur. MutantX-S marks the page as dirty and changes its permission to *writable* but *non-executable*.

3. When the unpacker jumps to the the newly-generated

[2]PE is the executable file format used by Windows OS

code for execution (e.g., after finishing unpacking), the *non-executable* permission on these pages triggers an execution exception. `MutantX-S` intercepts the exception and records the memory address where it occurred. For simple packers (e.g. UPX) that first unpack the entire program and then jump to the restored codes for execution, the memory address where the exception occurs is the OEP (original entry point). However, this is not necessarily true for more sophisticated packers (e.g., self-modifying code that rewrite to the same memory location). Hence, `MutantX-S` removes the *write* permission from these memory pages again, grants execution privilege and continues execution. `MutantX-S` also monitors dynamic allocation of memory pages and removes their write permission to track unpacking on these pages.

4. `MutantX-S` dumps the process memory image either at the end of program execution or after certain time. The rationale is that after the program has been running for a sufficient amount of time (e.g., 1 minute), it is fairly safe to assume that the program has finished unpacking and the original codes are placed in the memory.

Algorithm 1 `MutantX-S` unpacking algorithm

Input: A packed binary program B
Output: A unpacked PE file with original program codes

Load the packed program into memory
for all p in the program's memory pages **do**
  $Permission(p)| = \tilde{W}$ //remove write permission
end for

while B is running **and** $T_{runtime} < T_{thresh}$ **do**
  a: The address of the page fault
  t: The page fault type $t \in \{$WRITE, EXECUTE$\}$
  $p \leftarrow Page(a)$
  if t = WRITE **then**
    $Permission(p)| = (W|\tilde{X})$ // Writable but non-executable
    $last\_written(p) \leftarrow$ current time
  end if
  if t = EXECUTE **then**
    $Permission(p)| = (\tilde{W}|X)$ //non-writable but executable
    $last\_exec(p) \leftarrow$ current time
    $addr\_exec(k) \leftarrow a$
  end if
end while

Dump process memory
reconstruct B' by setting OEP to be $addr\_exec(k)$ where:
$k = \arg\min_k(last\_exec(k) > \max(last\_written(i))$
return B'

With the dumped memory image, `MutantX-S` creates a valid PE file from which a standard disassembler can disassemble instructions. As mentioned earlier, the major challenge in creating a valid PE file is to identify the correct entry point in the PE header. For simple packers like UPX, the entry point is simply the start address of the dirty memory page where the first execution exception occurs. Unfortunately, as adversaries become increasingly sophisticated, various evasion schemes have been developed to obfuscate OEP. A typical method is to fake end-of-unpacking by writing rouge instructions into a reserved memory page, transfer control to it, and jump back to the unpacker code. More advanced packers use incremental unpacking that decrypts only part of the payload and executes them before decrypting more instructions. In such cases, detecting the first execution exception is not enough because only rouge instructions or part of the original program is visible. To address these problems, we developed a new heuristic called LMFE (*Last Modification First Execution*).

The idea is to keep track of time when the last write exception and a subsequent execution exception occur on each memory page, so `MutantX-S` can identify the unpacker's attempts to write to the same memory page multiple times, in which case, the previous modification and execution on the page are likely to be spurious. More specifically, for each memory page, `MutantX-S` keeps a record of: 1) last modification time (i.e., a write exception occurred), denoted as $last\_written$; 2) last execution exception time, denoted as $last\_exec$; and 3) the address $addr\_exec$ where the exception had occurred. At any point of execution, there are 3 types of memory pages:

Type I: memory pages that have valid $last\_written$ and $last\_exec$, i.e., pages that have been both modified and executed.

Type II: memory pages that have valid $last\_written$ but not $last\_exec$, i.e., pages that have been modified but not executed. They could either be page containing temporary data or code pages that have not yet been executed.

Type III: memory pages that have neither valid $last\_written$ nor valid $last\_exec$. These could be initialized data-section pages or unpacker-code pages.

Essentially, type-I memory pages are those that hold the unpacked instructions and thus contain the OEP. When dumping the process memory, `MutantX-S` uses the following algorithm to pinpoint the correct OEP. Let $P(i)$, $i = 1..n$ represent all type-I memory pages and $last\_written(i)$, $last\_exec(i)$ and $addr\_exec(i)$ respectively represent the time of the last write exception, last execution exception and address where the exception occurred for page $P(i)$. Then, the OEP is $addr\_exec(k)$ in the memory page $P(k)$ where

$$k = \arg\min_k(last\_exec(k) > \max(last\_written(i)) \quad (1)$$

where $i = 1..n$. In other words, $P(k)$ is the first memory page that is executed after all type-I memory pages have been written. Below we show that the heuristic is able

to find the correct OEP (i.e. $addr\_exec(k)$) even when the packers try to fool the MutantX-S using spurious write-and-execute sequences or multi-layer packing.

Proposition. *For k satisfying Eq. (1), $addr\_exec(k)$ is the correct OEP of the original program no matter whether the program is packed with simple packers or more advanced packers that fake the end of unpacking.*

Proof. First, for simple packers like UPX that restore the entire program into memory before executing it, let $P(j),\ j = 1..m$ denote memory pages where the unpacker writes the original program codes. Without loss of generality, we can assume that the contents are written sequentially from $P(1)$ to $P(k)$, meaning that $last\_written(1) < last\_written(2) < \ldots < last\_written(m)$. When the packer finishes unpacking and starts executing the restored program by jumping to the OEP, an execution exception will first occur in the page $P(k)$ that contains the OEP, i.e., $last\_exec(k) > last\_written(m)$ and $last\_exec(k) < last\_exec(j) \forall j \neq k$. As a result, $addr\_exec(k)$ is the correct OEP.

Second, assume a more advanced packer with the following spurious unpacking sequence: it writes arbitrary instructions into some memory page, executes them and, at the end of execution, returns to the unpacker code. Such a routine may be called multiple times during the entire unpacking process. As a result, an unpacking tool cannot assumes that unpacking ends at the first (or first few) execution exception. MutantX-S is resilient to this type of evasion by enforcing the invariant that the execution exception on the OEP must succeed all the write exceptions. For example, when the spurious unpacking routing touches memory page $P(s)$, MutantX-S records $last\_exec(s)$ and marks $P(s)$ as executable but un-writable. Then, the unpacker resumes the normal unpacking and writes more decrypted instructions to memory page $P(t)$ (t could be any page including s). This creates a new write exception on $P(t)$ at timestamp $last\_written(t)$. Note that because $last\_exec(s) < last\_written(t)$, the heuristic determines s to not contain the OEP. In contrast, after the packer finishes unpacking and transfers control to the real OEP, the execution exception satisfies Eq. (1). By keeping $addr\_exec$ up-to-date and pointing to a valid instruction, MutantX-S is able to keep track of the real OEP accurately. Same arguments hold for multi-layer packing because the write exceptions of code pages will always precede the executable exceptions caused by jumping to the OEP.

5 Feature Extraction

With the correct OEP identified, MutantX-S reconstructs a new PE file with dumped memory images. The correct OEP ensures a proper starting point to disassemble instructions and MutantX-S uses the IDA Pro[3] to disassemble a malware program into a sequence of machine instructions that are then used for feature extraction. The key step in MutantX-S is the similarity comparison between malware samples based on the disassembled instruction sequences, e.g., move eax, ebx; cmp eax, 1h. The main challenge in similarity comparison lies in handling the variations of machine instructions. Malware often undergo changes for many reasons, such as mutation, polymorphism, and obfuscation where semantically-equivalent instruction sequences are used to replace each other. Hence, ensuring exactness in comparing instructions will not tolerate any variation in the syntax. At the other extreme, correctness is compromised if all forms of variation are tolerated. MutantX-S strikes a balance between these two extremes by exploiting the x86 instruction format (Fig. 2) and uses the opcode as a succinct representation of the instruction semantics.

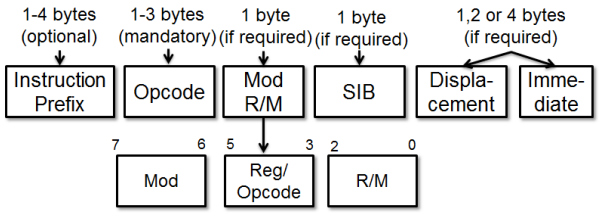


Figure 2: x86 instruction format

Using opcodes—instead of other features used before, such as control flow graphs, binary sequences or mnemonic sequences—offers several benefits. First, opcodes generalize well to represent variants of a malware family. Malware in the same family are often derived from the same code base and thus share similarities in their instructions. However, due to relinking, rebinding and rebasing, the operands (e.g., registers, memory addresses) of instructions tend to change across the variants. Using opcodes and ignoring the operands (i) make MutantX-S more resilient to low-level mutations while providing a meaningful characterization of semantics and (ii) reflect the functionality of the malware programs. Second, previous approaches often use mnemonic sequences (e.g., mov, push) to represent the instruction functionalities and address the variability of operands. From the evaluation of MutantX-S, we discover that the opcode sequence offers a better representation of instruction semantics. Mnemonics sometimes *overly* generalize the underlying CPU operations, causing instructions with distinct semantics to appear similar. To illustrate this, consider all the instructions in Table 1. Although all of them have the same mnemonic (i.e., mov), the underlying functionalities are drastically different. For instance, moving a value to a control or debug register of-

[3] the de-facto disassembler for the analysis of hostile code

ten indicates a critical OS operation, such as interrupt control, switch addressing mode or enable/disable debugging, etc., which should not be treated the same as moving a value between registers. Ideally, moving data from memory to a register (memory load operation) should also be considered as a distinct operation from that of moving from a register to memory (memory store operation). Unfortunately, using mnemonics would cause all these distinct instructions to be represented with a single feature (i.e., mov), which may lead to an accidental similarity between code sequences. As illustrated in Fig. 3, however, features based on opcode provide higher distinguishability between semantically different instructions, thus yielding better clustering accuracy.

Op	Instruction	Description
89	MOV r/m32, r32	Move from reg to mem/reg
8B	MOV r32, r/m32	Move from mem/reg to reg
B8	MOV r32, imm32	Move immediate val to reg
0F 20	MOV r32, CR0-CR4	Move from control reg to reg
0F 22	MOV CR0-CR4,r32	Move from reg to control reg
0F 21	MOV r32, DR0-DR7	Move from debug reg to reg
0F 23	MOV DR0-DR7,r32	Move from reg to debug reg

Table 1: Op (Opcodes) provides fine-grain representations of instruction semantics

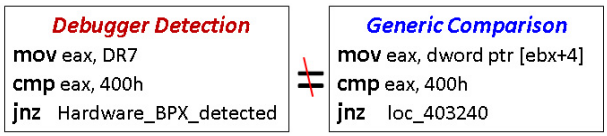


Figure 3: Two code pieces with different semantics share same mnemonic representation (i.e., move, cmp, jnz). However, they can be differentiated by their opcode representation: ”0F 21 3D 75” vs ”8B 3D 75”

.

With this encoding scheme, a program is represented as a sequence of opcodes (Fig. 4). `MutantX-S` uses the standard N-gram analysis to characterize the content of a malware program, i.e., moving a fixed-length window over the sequence and considering a subsequence of length N at each position. The resulting N-gram of opcodes reflects short instruction patterns and implicitly captures the underlying program semantics. Then, `MutantX-S` constructs a feature vector V in an $|S|$-dimensional space ($|S| = |\mathscr{O}|^N$ where $\mathscr{O}$ is the set of all possible opcodes). Each dimension of V is the number of *occurrences* of a particular opcode N-gram. Then, `MutantX-S` can geometrically calculate the similarity between two malware programs (m, v) as the Euclidean distance between their feature vectors in the vector space: $d(m,n) = \|V_m - V_n\| = \sqrt{\sum_{i=1}^{|S|}(V_m(i) - V_n(i))^2}$. Compared to the other similarity metrics (e.g., locality-based hashing), geometric calculation of similarity in the vector space provides *explicit feature representation* [24] where the importance or contribution of each N-gram in clustering malware can be traced back to its original code patterns. For N-grams that may correspond to inherent characteristics of a malware family (e.g., those that appear frequently within a family but rarely in others), their original code segments can be traced back and used as signatures to detect variants.

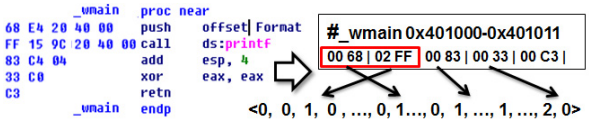


Figure 4: Encoding a function into a feature vector

6 Clustering Algorithm

The next step in `MutantX-S` is clustering malware samples into groups that share common traits. Considering the enormous amount of malware in the wild, the goal of `MutantX-S` is to process hundreds of thousands malware files sufficiently fast. Unfortunately, classic clustering algorithms such as hierarchical and partitioning-based clustering, e.g., K-Means or K-Medoids—although they have been successfully applied to cluster malware behaviors and programs [6, 13]—incur a time complexity at least quadratic in the number of samples, which in practice, does not scale to the `MutantX-S`' target. `MutantX-S` exploits two approaches to address the scalability issue: (1) a hash kernel that compresses the high dimensional feature vector into a low dimensional space, and (2) a prototype-based clustering algorithm that has close-to-linear runtime complexity.

6.1 Hashing Kernel

Kernel methods [25] are powerful tools used in machine learning to allow operation in the high-dimensional feature space without having to compute the coordinates of the data in that space. This is particularly useful when the input data has a non-linear decision boundary but can be linearly separated in a high dimensional feature space. In `MutantX-S`, however, we have encountered the opposite problem: the original space is very high-dimensional[4]. The number of dimensions D determines the complexity when computing the vector distance and D increases exponentially with N in N-gram (i.e. $D = |\mathscr{O}|^N$ where $|\mathscr{O}|$ is the number of different opcodes and in practice $|\mathscr{O}| > 200$). Therefore even a small N like 3 will result in a (very sparse) feature vector with more than 8 million dimensions, which is computationally prohibitive when calculating similarities for large amount of malware

[4]thus, the input data are likely already linearly separable

samples. Unfortunately, N has to be at least 3 or 4 to be descriptive enough for capturing the program semantics.

MutantX-S addresses the problem by exploiting the *hashing-trick* recently developed in the machine learning community[26], which *hashes* the high dimensional input vector $x \in \mathbb{R}^n$ into a lower dimensional feature space $\mathbb{R}^m$ with the mapping function $\phi : \mathscr{X} \to \mathbb{R}^m$. Since $m \ll n$, the hashing trick reduces a feature vector to a more compact representation, allowing the clustering algorithm to handle a large volume of data, and save both computation and memory requirements. Previous research has shown that the hash kernel approximately preserves the vector distance and the penalty incurred from using a hash for reducing dimensionality only grows *logarithmically* with the number of samples and groups [26, 14].

Specifically, MutantX-S applies a *uniform* hash function $H : \{\text{N-gram}\} \to [1..m]$ that hashes N-gram directly into a position in the feature vector of length m. In case of a collision where two or more N-grams map to the same position, the sum of their counts is used as the value in the new vector. More formally, for malware M and M', let v and v' represent their original feature vector extracted from the encoded opcode sequences and ξ denote the mapping from the N-gram $(o_1, o2, \ldots, o_N) \in S$ to the index in v. We define the hash feature map ϕ as

$$\phi_i(v) = \sum_{l:H(o)=i, l \in S} v(\xi(o))$$

and the distance between M and M' as

$$d_\phi(M, M') = \|v - v'\|_\phi = \|\phi(v), \phi(v')\|.$$

The choice of m, the length of the low dimensional vector, is a trade-off between clustering accuracy and storage overhead plus computation complexity. Choosing a smaller m means shorter vector length, thus, faster distance computation and smaller memory footprint to store malware features. However, decreasing m increases the collision possibility, leading to over-compression of features and negative impact of the clustering accuracy.

6.2 Prototype-Based Clustering

Classic clustering algorithms typically incur a complexity that is super-linear in the size of the input data. For example, the running time for two most widely-used clustering algorithms k-means and hierarchical clustering are $O(n^{kd})$ [4] and $O(n^2 \log n)$ [19], resulting in the computation time that is prohibitively large for the number of malware we have to deal with. Instead, MutantX-S adopts the prototype-based close-to-linear clustering algorithm[24].

Despite their simplicity, Prototype-based algorithms have been empirically shown to be very effective and often one of the best performers in real data [11]. Prototype-based clustering extracts a set of prototypes each of which serves as the representative for a small group. The remaining data points are associated with their closest prototype in the feature space. The key idea of Prototype-based algorithm is to perform computation (e.g. clustering) only on the prototypes which are a small subset of original data points, thus reducing the computation time significantly. The algorithm comprises 2 steps.

Prototype extraction: The quality of final clusters depends on the choice of the prototypes. Well-positioned prototypes can accurately capture the distribution of input data and allows creating accurate class boundaries in the feature space. Unfortunately, determining the optimal number and positions of prototypes is NP-hard and an approximate algorithm by Gonzàlez [9] was commonly used to iteratively select prototypes. During each iteration, the data point with the largest distance to existent prototypes is selected as the next prototype (the first prototype is selected randomly). The process repeats until the distance from all the data points to their nearest prototype is smaller than a predefined threshold P_{max}, i.e., all the data points are located within a certain radius from their closest prototypes. The run-time complexity of this algorithm is $O(kn)$ where k is the number of prototypes selected. Since k only depends on the distribution of the data (in this case, k is proportional to the amount of malware families), with a reasonable choice of P_{max} the algorithm is linear in the number of input data n.

Clustering with prototypes. Instead of working on the huge number of original data, the algorithm performs agglomerative hierarchical clustering only on the prototypes. Specifically, the algorithm starts with individual prototypes as singleton clusters, successively merges two closest clusters, and terminates when the distance between the closest clusters is larger than a predefined distance threshold Min_d. Then, prototypes within the same cluster are assigned the same cluster label and subsequently propagate the label to their associated data points. Because each prototype is a good representation of its associated data points (all within a radius of P_{max}), the algorithm avoids expensive distance computation between the original data points without too much loss in the overall accuracy. The respective run-time complexities of clustering and propagation steps are $O(k^2 log\ k)$ and $O(n)$. Compared to the $O(n^2 log\ n)$ complexity of applying an hierarchical clustering algorithm on the original data points, this algorithm achieves a speed-up with a factor of at least $(n/k)^2$.

7 Experimental Evaluation

We now evaluate MutantX-S' efficiency and accuracy using two data sets: (1) a reference data set containing 4821 malware files whose labels are generated by security experts from a large anti-virus company and thus

more reliable; and (2) a large malware data set collected from an online malware archive [29] which comprises 132,234 malware samples with unreliable labels derived from AV scanners. The reference data set includes malware samples from 20 different families and their distributions are given in Table 2. Considering its reliable labeling, the reference set is used to evaluate and fine-tune the empirical parameters for the MutantX-S' clustering engine while the larger set is used to assess its scalability.

Family	#	Family	#	Family	#
Pilleuz	500	Bredolab	301	Tidserv	59
Koobface	496	Vundo	249	Waledac	34
Silly	489	Almanahe	241	Ackantta	32
Fakeav	489	Sasfis	199	Mebroot	26
Zbot	459	Graybird	166	Hotbar	21
Banker	449	Gammima	126	Qakbot	17
Virut	361	Mabezat	107		

Table 2: Malware families of the reference data set

7.1 Effectiveness of Unpacking Engine

To evaluate the effectiveness of MutantX-S's unpacking component, we select and then pack a malware program with 8 popular packers. We then unpack them with MutantX-S and compare unpacked files with the original version. Ideally, the unpacked binary should be byte-to-byte identical to the original file. However, this is neither possible (MutantX-S does not reconstruct the import table, and the unpacker code is also dumped from the memory), nor necessary for the purpose of malware clustering. As a result, we compared the unpacked files with the original one using two metrics: (i) the difference in their *instruction count* (IC), and (ii) the distance between their N-gram feature vectors (NG), because they are directly related to clustering accuracy. Table 3 summarized the results. For most packers, the MutantX-S successfully recovered their original binaries with only a 1–6% increase of ICs which is often due to the inclusion of unpacker routines in the dumped memory. Besides, the feature vectors of unpacked binaries are very similar to those of the original binary with most normalized distance measurements below 0.1, where 0 means identical and 1 means completely different. However, MutantX-S also failed on certain packers. In particular, the memory dump of Armadillo-packed malware sample still contains a packed version of the binary. A further investigation showed that Armadillo works by unpacking an intermediate executable on disk and creating another process to run this executable [20]. Therefore, memory dump of an Armadillo-packed file does not contain original instructions. After running MutantX-S on the larger data set, we have also observed other causes of unsuccessful unpacking, such as malware samples refusing to run in a virtual machine or the time required for unpacking is longer than the threshold. Nevertheless. MutantX-S' generic unpacking technique is still effective against popular packers without requiring any specialized knolwedge of packing algorithms.

Packer	Diff in IC (%)	NG Dist	Packer	Diff in IC (%)	NG Dist
PEcompact	0.88%	0.068	ASprotect	6.70%	0.133
EXECryptor	3.20%	0.176	UPX	0.88%	0.068
EXEStealth	0.88%	0.071	NSPack	0.87%	0.069
VMprotect	2.50%	0.10	Armadillo	-	-

Table 3: Unpacking effectiveness (IC: Instruction Count; NG Dist: N-gram Difference)

7.2 Malware Clustering Accuracy

We first evaluate and calibrate MutantX-S against the reference data set. All of our evaluations were done on a Ubuntu 10.4 machine with Core i7 3.0G CPU and 12GB memory. We use *precision* and *recall* as the main metrics to assess the accuracy of MutantX-S. Suppose that with respect to the original labels (i.e., family names in Table 2), n input malware samples can be grouped into a set of clusters $O = \{O_1, O_2, \ldots, O_o\}$. Assume MutantX-S outputs a set of clusters $C = \{C_1, C_2, \ldots, C_c\}$. Then, precision P measures how well individual clusters agree with the original classes (i.e., the *exactness* of clusters), and recall R measures how much the malware classes are scattered across the clusters (i.e., the *completeness* of each cluster). Formally, we define

$$P = \frac{1}{n}\sum_{i=1}^{c} max(|C_i \cap O_1|, |C_i \cap O_2|, \ldots, |C_i \cap O_o|)$$

$$R = \frac{1}{n}\sum_{j=1}^{o} max(|O_j \cap C_1|, |O_j \cap C_2|, \ldots, |O_i \cap C_c|)$$

P will be 1 if all the samples in every cluster C_i are from the same family and R will be 1 if all malware samples from the same family fall into a single cluster (but not necessarily the only family in this cluster). Fig. 5 shows the precision and recall of MutantX-S's clustering with varying thresholds P_{max} and Min_d (defined in Section 6). The experiment uses 4-gram and 12 hash bits (i.e., the 4-gram is mapped into 2^{12} hash bins).

From the figure, we observe that MutantX-S is able to cluster the samples with the precision ranging from 0.72 to 0.89 (average=0.80). The precision number is smaller than those reported in previous dynamic-behavior approaches, e.g., 0.996 in [24] and 0.984 in [7]. We conjecture that this difference may be due to different malware sets (and possibly incorrect labeling) used in our experiments and the reason for the higher accuracy of dynamic-behavior approaches is also likely due to their high-level generalization of behavior at the cost of longer

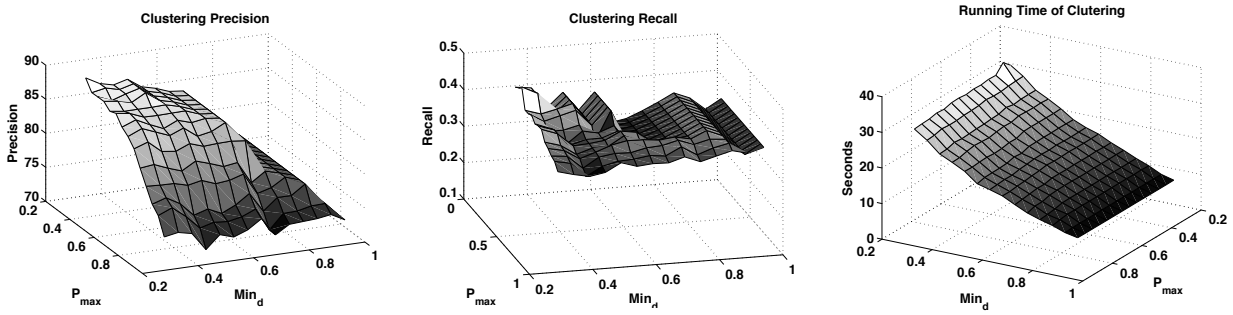


Figure 5: Precision, recall and running time of MutantX-S

running time and limited coverage as discussed in Section 1. Therefore, MutantX-S can provide an alternative way of categorizing malware and is complementary to the behavior-based analysis with better scalability while maintaining reasonably good accuracy. Indeed, Fig. 5 shows that it takes only less than 30 seconds to complete the clustering for the entire reference dataset (we also ran the K-mean and hierarchical clustering on the same dataset which respectively took 32.3 seconds with precision 0.75 and 51.3 seconds with precision 0.82). In addition, we observe that the recall of MutantX-S is around 0.3 and 0.4. However, this low value of recall is not surprising, because there often exists significant diversity across malware variants. For instance, we observed that one variant in Vundo family is 10 times larger in terms of file size than the other Vundo variant. This is possibly due to mislabeled samples, unidentified packers or heavily-obfuscated binaries. Because of the highly diverse variants, MutantX-S often breaks one family into several sub-families, resulting in a low recall, e.g., MutantX-S creates more than 50 clusters for the reference dataset which contains 20 families according to the labels. Albeit less ideal, a breakdown into sub-families is acceptable in practice, e.g., predicting labels for unknown samples as we will show later.

Another observation from these results is that P_{max} (the threshold for distances from data points to their nearest prototypes) has a greater influence on the clustering speed, since a smaller P_{max} forces the algorithm to find more prototypes to cover all the data points, thus requiring more computation. On the other hand, Min_d has a major impact on the clustering accuracy. Increasing Min_d reduces the precision, because a smaller inter-cluster distance threshold will stop the prototype-merging process earlier which reduces the probability of combining unrelated prototypes into a larger cluster. However, the price for this is the over-fitting of clustering, i.e., the algorithm tends to create several small clusters. Hence, a trade-off has to be empirically made, as in our later experiments.

7.3 Validity of the Hashing Trick

The main concern in using the hashing trick is the possible loss of information due to the compression of high dimensional features into a lower dimensional space. To evaluate the efficacy of hashing trick, we use different number of hash bins to cluster the reference data set. The hash function used in MutantX-S is MurmurHash 2.0 [1], a simple hash implementation with uniform value distribution, high throughput, and good collision resistance. As comparison, we also ran MutantX-S on the original feature vectors without the hashing trick, which serves as the baseline benchmark and best-possible result achievable without information loss.

Figure 6 compares the precision, clustering time and peak memory requirements with different hash sizes (the number of hash bins ranging from 2^8 to 2^{16} and no hash). Different bars represent the results generated by different parameter combinations. From the left figure, we find that as the hash size increases, the precision improves because the collision probability reduces. In fact, when the hash size is large enough, the probability of collision becomes so negligible that the hashed features vector perform the same as the original ones. For instance, with more than 2^{12} hash bins, the clustering achieves almost the same precision, 0.864, as the original features $P = 0.868$. However, as the hash size becomes smaller, the impact of collision starts to surface. When the number of hash bins reduces to $2^8 = 256$, the precision drops significantly (less than 0.5) for some parameter combinations, due to collision of many critical features (e.g., features indicative of different families are now mapped to the same hash bin) In this regard, a larger hash size is preferable. On the other hand, the middle and right figures in Figures 6 show that a small hash size is very effective in reducing the algorithm's running time and memory footprints. This is because smaller number of hash bins means shorter feature vectors which require less memory for storage and fewer CPU cycles to compute the distance. For instance, as the hash size decreases from 16 bits to 8 bits, the required running time drops from almost 2 minutes to less than 10 seconds and memory requirement from 800 Mbytes to less than 100 Mbytes, at

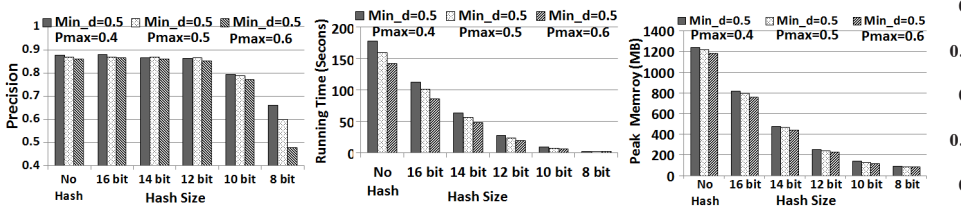


Figure 6: Precision, time, and peak memory with hash bin number ranging from 2^8 to 2^{16} and with no hash trick .

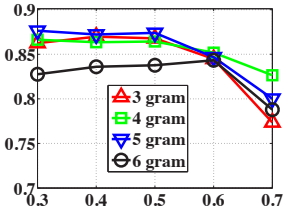


Figure 7: Precision of clustering with different N.

the cost of precision. In practice, a 12-bit hash function is found to be a good compromise, reducing the time and memory requirements by over 80% while still keeping good accuracy[5]. Figure 6 also shows that as the number of malware increases, the hashing trick becomes critical. Without it, the memory requirement could quickly become prohibitively high.

7.4 Impact of N-gram on Performance

Intuitively, as N increases, N-gram becomes more descriptive, providing better distinguishability. However, this comes at the cost of exponential increase in the dimensionality of the resulting feature vectors (m^N where m is total number of different opcodes), as well as the required storage and computation time. Therefore, previous work that uses N-gram based approaches commonly chose small N (3 or 4). Fortunately, the hashing trick enables us to compress the feature vectors and evaluate the performance of large N. Figure 7 summarizes the result.

From Figure 7, one can observe that use of a larger N value indeed improves the precision, e.g., 4- and 5-grams achieve better precision than 3-gram since larger grams can better capture the underlying instruction semantics. However, the figure also shows that 6-gram performs the worst. This is because the number different 6-grams (i.e., over $6.4 * 10^{12}$) is too large for the 12-bit hash function (4096 hash bins), leading to a large number of collisions between irrelevant features. In `MutantX-S`, we have chosen 4-gram, because the improvement provided by 5-gram is not large enough to warrant the additional storage and computation overheads.

7.5 Scalability of `MutantX-S`

In this subsection, we evaluate the scalability and accuracy of `MutantX-S` on the large malware data set with over 130,000 samples. We ran `MutantX-S` on the entire set with different parameters and plotted the results in Fig. 8. The right figure shows the amount of time for clustering the entire set. the value P_{max} seems to have a more significant impact on the running time. For example, when P_{max} is set to 0.5, the clustering takes less than 1 hour which is almost half of the time when P_{max} is set to 0.2. As mentioned before, P_{max} determines the number of prototypes extracted from the input data which determines the total number of distance computations required for clustering. Although a larger P_{max} leads to a shorter running time, the left plot in Fig. 8 illustrates the correlation between a large P_{max} and the reduced clustering precision, i.e., increasing P_{max} from 0.2 to 0.5 reduces the precision by almost 10%. This can be explained as follows: a large P_{max} allows each prototype to cover a large portion of the space, thus increasing the possibility of including samples from irrelevant families. With a reasonable setting (e.g., $Min_d = 0.5$ and $P_max = 0.4$), `MutantX-S` is able to complete the clustering of over 130K malware in less than 1.5 hours with the precision close to 0.82.[6] The peak memory usage is around 3.6GB. These results indicate that `MutantX-S` is very efficient in handling a large number of samples and thus has the potential to keep up with the huge influx of malware variants received nowadays.

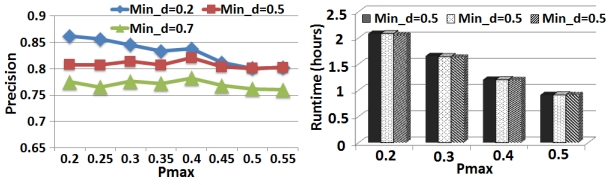


Figure 8: Precision and running time of `MutantX-S`'s clustering over 130K samples

7.6 Predicting Labels of Unknown Malware

So far, we have evaluated `MutantX-S` using the data set of known malware families. In a realistic scenario, e.g., in AV companies, `MutantX-S` is more likely to be used to analyze new incoming malware and predict their family labels. In such a scenario, incoming malware

[5]Hence, unless specified otherwise, throughout the paper, the experiments are performed with a 12-bit hash function

[6]The recall for the large data set is around 0.25 because of breaking the samples from large malware families into relatively small groups.

are analyzed and labeled according to their association with the closest kin in the previously-analyzed samples. To simulate this situation, we need a chronological order of malware samples according to their creation time. We extract the creation time for each malware from their IMAGE_FILE_HEADER. IMAGE_FILE_HEADER is a standard header in the PE file and contains a timestamp field that is set by the compiler at the compilation time. We use this timestamp to bucket malware programs into months and select one year worth of malware (more than 40,000 unique samples). Fig. 9 shows the distribution of the number of new malware samples across all months. Next, we use these malware to simulate the process of determining the labels for new incoming samples.

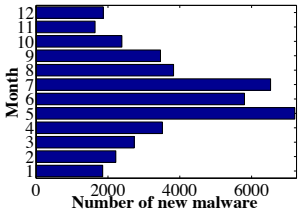


Figure 9: Number of new samples in each month used to evaluate prediction capability

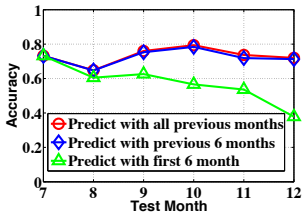


Figure 10: Accuracy in applying MutantX-S to predict family labels for unknown malware

Specifically, we separate the malware program into *training set* and *testing set* based on their creation time in order to simulate the scenario where AV companies have analyzed the malware from the training set and try to predict the labels for newly-received malware (test set). The test set consists of malware samples from each month between July and December (these months are "test months"). Then, we choose 3 different training sets. For the first training set, we use samples from all months from January to one month before the test month. For instance, if the test month is September, the training months are January through August. For the second training set, we use 6 months prior to the test month, i.e., if September is the test month, March through August will be the training months. Finally, as a controlled experiment, we keep the the first 6 months (i.e., January through June) as the training month regardless of test months. Given any training set, MutantX-S creates a set of clusters C_i ($i = 0, 1, \ldots, n$). Each cluster has a label $L(C_i)$ determined by the majority family labels of the constituent malware samples. Then MutantX-S determines the family label $L(x_j)$ of the new sample x_j in the test month based on the label of the cluster that is closest to x_j, i.e., $L(x_j) = L(C_i)$ where $d(x_j, C_i) = \min(d(x_j, C_k))\forall k = 0, 1, \ldots, n$. We then compare this predicted family label with the original label of x_j, and plot the percentage of correctly predicted samples in Fig. 10. The first observation from the figure is that malware are constantly evolving and the information obtained from previous clustering can become obsolete quickly, as shown by the bottom green line where we kept on using the same first 6 months as the training data and the prediction accuracy degraded rapidly from 0.7 in July to below 0.4 in December. In contrast, if we use the full history as the training data, the accuracy stays consistently in the 0.7–0.8 range (the top red line in Fig. 10), thanks to the up-to-date information from the recent malware. However, in reality, due to the resource (e.g., storage) constraints, it may not be possible to keep the entire history of previous malware samples. It is more efficient to use only the most recent history, e.g., 6 months as in the middle blue line. From Fig. 10, one can see that the result is very close to that of using the full history, with only a small decrease, about 2 to 3%. These results imply that there exists a strong temporal correlation among malware variants which can be exploited by MutantX-S in predicting the labels for unknown malware samples.

8 Limitations and Improvements

Here we discuss limitations of the current prototype of MutantX-S that could be exploited by adversaries to degrade its effectiveness in clustering. As a static-analysis approach, MutantX-S is vulnerable to binary/instruction-level obfuscation. First, even with a generic unpacking algorithm, MutantX-S is less effective against advanced packers that employ sophisticated protection mechanisms, e.g., driver-level protection, anti-debug, anti-emulation, etc. Specialized unpacking tools [2] have been developed for these packers and they can be incorporated into MutantX-S to combat sophisticated packers. Second, MutantX-S extracts features from disassembled malware code. Unfortunately, producing correct disassembly is often very challenging and many anti-disassembly tricks [31] can be used to confuse a disassembler, such as mixture of code and data, making an infeasible conditional jump to the middle of next instruction, etc. Although the current prototype does not handle these types of obfuscation for simplicity, there are a variety of techniques [16] proposed to mitigate these problems. Third, MutantX-S relies on the similarity of code instructions to cluster malware samples. It is possible to create syntactically distinct but semantically similar variants through heavy instruction-level obfuscation. To address these problems, MutantX-S could incorporate more advanced de-obfuscation techniques [27, 22] and normalize the malware codes before clustering them. Note that dynamic-behavior-based approaches do not suffer from this limitation, but they come with their own deficiencies—limited coverage, scalability and specific evasion techniques. Therefore, MutantX-S' goal is not to replace the dynamic approaches, but to complement

them and collaboratively mitigate their weaknesses (e.g. apply dynamic analysis only on representative samples or outliers of static analysis). Finally, MutantX-S cannot handle file infector or parasitic malware types which inject themselves into host executables. This is a limitation for *any* similarity based clustering, regardless static or dynamic approaches, because most features are from the host executables rather than the maleware. Such parasitic malware are a matter of our future inquiry.

9 Conclusion

In this paper, we have presented the design, implementation and evaluation of a malware clustering system based on static features, called MutantX-S. MutantX-S can accurately and efficiently group malware variants according to the similarity in their code instructions. It converts each malware program into a compact but effective opcode representation and performs prototype-based clustering on the corresponding N-gram feature vectors. It also incorporates a generic unpacking technique to maximize the capability of analyzing the malware's original instructions. To ensure the scalability, MutantX-S uses a combination of a hashing kernel that reduces the dimensionality of feature vectors and a close-to-linear time prototype-based clustering that uses a small set of representative samples for fast data organization. Equipped with these techniques, MutantX-S is experimentally shown to be able to process more than 130,000 malware samples within a few hours. As a static-analysis approach, MutantX-S is expected to be very effective and can be combined with existing dynamic-behavior-based system to provide the level of accuracy and coverage required to pace with the current malware sample submission rate.

References

[1] Murmurhash 2.0. http://sites.google.com/site/murmurhash/.

[2] Unpackers. http://www.exetools.com/unpackers.htm.

[3] Peid 0.95. http://www.peid.info/, 2008.

[4] ARTHUR, D., AND VASSILVITSKII, S. How slow is the k-means method? In *Proceedings of the twenty-second annual symposium on Computational geometry* (2006).

[5] ASPACK SOFTWARE. Aspack. http://www.aspack.com/.

[6] BAILEY, M., ANDERSEN, J., MAO, Z. M., AND JAHANIAN, F. Automated classification and analysis of internet malware. Tech. rep., Proceedings of RAID, 2007.

[7] BAYER, U., COMPARETTI, P., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Proc. of the 16th NDSS* (2009).

[8] CHRISTODORESCU, M., AND JHA, S. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium* (2003).

[9] GONZALEZ, T. Clustering to minimize the maximum intercluster distance. In *Theoretical Computer Science* (1985), vol. 38, pp. 293–306.

[10] GUO, F., FERRIE, P., AND CHIUEH, T.-C. A study of the packer problem and its solutions. In *Proceedings of RAID* (2008).

[11] HASTIE, T., TIBSHIRANI, R., AND FRIEDMAN, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, 2009.

[12] JANG, J., BRUMLEY, D., AND VENKATARAMAN, S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of CCS'11* (2011).

[13] KARIM, M. E., WALENSTEIN, A., LAKHOTIA, A., AND PARIDA, L. Malware phylogeny generation using permutations of code. *JOURNAL IN COMPUTER VIROLOGY 1* (2005), 13–23.

[14] KILIANWEINBERGER, DASGUPTA, A., LANGFORD, J., SMOLA, A., AND ATTENBERG, J. Feature hashing for large scale multitask learning. In *Proceedings of the 26th ICML* (2009).

[15] KOLTER, J. Z., AND MALOOF, M. A. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research 7* (2006), 2006.

[16] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium* (2004).

[17] LABS, A. R. New toy in the avast research lab. https://blog.avast.com/2012/12/03/new-toy-research-lab/.

[18] LEE, T., AND J.MODY, J. An automated virus classification system. In *Proceedings of VIRUS BULLETIN CONFERENCE OCTOBER 2005* (2005).

[19] MANNING, C. D., RAGHAVAN, P., AND SCHUTZE, H. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[20] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of ACSAC* (2007).

[21] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007).

[22] RABER, J., AND LASPE, E. Deobfuscator: An automated approach to the identification and removal of code obfuscation. *Reverse Engineering, Working Conference on* (2007).

[23] RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *Proc. of the DIMVA'08* (2008).

[24] RIECK, K., TRINIUS, P., WILLEMS, C., AND HOLZ, T. Automatic analysis of malware behavior using machine learning. tech report, Berlin Institute of Technology, 2009.

[25] SHAWE-TAYLOR, J., AND CRISTIANINI., N. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[26] SHI, Q., PETTERSON, J., DROR, G., LANGFORD, J., SMOLA, A., STREHL, A., AND VISHWANATHAN, V. Hash kernels. In *Proc. of the 12th International Conference on Artificial Intelligence and Statistics* (2009).

[27] UDUPA, S. K., DEBRAY, S. K., AND MADOU, M. Deobfuscation: Reverse engineering obfuscated code. *Reverse Engineering, Working Conference on* (2005).

[28] UPX. http://upx.sourceforge.net/.

[29] VXHEAVEN. Vxheaven virus collection. http://vx.netlux.org/, 2010.

[30] WICHERSKI, G. pehash: A novel approach to fast malware clustering. In *2nd Usenix LEET Workshop* (2009).

[31] YASON, M. The art of unpacking, https://www.blackhat.com/presentations/bh-usa-07/yason/whitepaper/bh-usa-07-yason-wp.pdf.

Redundant State Detection for Dynamic Symbolic Execution

Suhabe Bugrara
Stanford University

Dawson Engler
Stanford University

Abstract

Many recent tools use dynamic symbolic execution to perform tasks ranging from automatic test generation, finding security flaws, equivalence verification, and exploit generation. However, while symbolic execution is promising, it perennially struggles with the fact that the number of paths in a program increases roughly exponentially with both code and input size. This paper presents a technique that attacks this problem by eliminating paths that cannot reach new code before they are executed and evaluates it on 66 system intensive, complicated, and widely-used programs. Our experiments demonstrate that the analysis speeds up dynamic symbolic execution by an average of 50.5 X, with a median of 10 X, and increases coverage by an average of 3.8 %.

1 Introduction

Dynamic symbolic execution has enabled many recent advances in program analysis such as high-coverage test input generation, patch checking, equivalence verification, malware signature generation, assertion checking, and debugging [7]. The technique's power comes from its ability to systematically and precisely enumerate program paths automatically. Further, in many cases it can eliminate false positives by producing a concrete test case to demonstrate a bug or specific path execution.

There are many variations in modern symbolic execution interpreters, but broadly speaking they work as follows. First, they start from some initial program state in which program inputs are represented by "unknowns" that take on any value. The interpreter symbolically executes the program by updating the state with the effect of each instruction. When it reaches a branch statement with condition C, the interpreter forks the state into two states, adding the constraint C to one and $\neg C$ to the other. This forking is skipped if one branch direction is infeasible. The interpreter repeatedly executes and forks states until either it generates every possible state of the program, thereby exploring every possible path with respect to the inputs or, more typically, it exhausts memory or exceeds a time limit.

While powerful, naive symbolic execution faces the significant challenge in practice that the number of paths (and thus states) increases roughly exponentially both with the size of the tested program and with the size of the program inputs. Programs with fewer than ten thousand lines of code routinely generate millions of states, each consisting of tens of thousands of memory locations. Thus, under realistic time and memory limits, state-of-the-art dynamic symbolic execution tools only explore a small percentage of paths.

As a result, while capable of deep reasoning, these tools have had limited applicability. They often fail when used to verify a property when doing so requires exploring every feasible path involving the property. Even when used purely for bug-finding, they often quickly get lost in an exponential number of superficially different but essentially identical states. Many tools [4, 5, 10, 12] counter this problem by using heuristic search strategies, but these have proven notoriously fragile.

This paper presents a novel, complementary approach that exploits a key observation: for many program analysis applications, most paths are redundant with respect to the goal of the symbolic execution and thus do not need to be explored. For example, if the goal is to generate a suite of program inputs that covers every line of code, then the symbolic execution only needs to explore paths that will reach lines which have not been covered by previously explored paths.

The contributions of this paper are (1) the design and implementation of a sound, redundant state detector capable of scaling up to handle real programs and (2) a thorough experimental evaluation on 66 system-intensive, complicated, and widely used programs which demonstrates that the detector yields dramatic performance improvements. Our technique speeds up dynamic

symbolic execution by an average of 50.5 X, with a median of 10 X, and increases coverage by an average of 3.8 %. On 12 of the benchmarks, the analysis reduces the state space sufficiently that the tool exhaustively explores all remaining states.

2 Overview

In this section, we give an overview of our analysis using the following program, which contains two variables: `w` and `m`.

```
1  if (w)
2    printf("X");
3  else
4    printf("Y");
5  if (m)
6    exit(0);
7  else
8    exit(1);
```

The program is symbolically executed from four initial states. States consists of a program counter and a set of constraints over program variables. Each of these four initial states starts at line 1. Suppose that state A has the constraints {`w = 0, m = 1`}, state B has constraints {`w = 1, m = 1`}, state C has constraints {`w = 2, m = 1`}, and state D has constraints {`w = 3, m = 0`}. When state A is symbolically executed through the program, it covers the lines 1, 3, 4, 5, and 6; state B covers 1, 2, 5 and 6; state C covers 1, 2, 5, and 6; and state D covers 1, 2, 5, 7, and 8. Figure 1 shows a diagram of the *state tree* produced by symbolically executing the program from these initial states.

As these four initial states are executed over the program, the same lines are covered over and over again. Since our goal is to produce high-coverage test suites and to explore different parts of the program to find bugs, we can avoid a significant amount of redundant work if we discard states that behave similarly to previously executed states. For example, after states A and B are executed to completion, state C will only cover a subset of the lines that A and B covered, so C is redundant and can be eliminated. The challenge is to detect that C is redundant without actually executing it. One simple approach is to compare the constraints of C to the constraints of A and B: if they have the same constraints, then they must follow the same paths, and thus will cover exactly the same lines. However, requiring that two states have identical constraints is unnecessarily conservative.

In this paper, we present an approach that precisely determines which constraints affect whether a state will cover the same lines as a previously executed state. We use *dynamic slicing* [1], a program analysis technique that finds all program statements that affected the value of a variable occurrence for given program inputs. Our

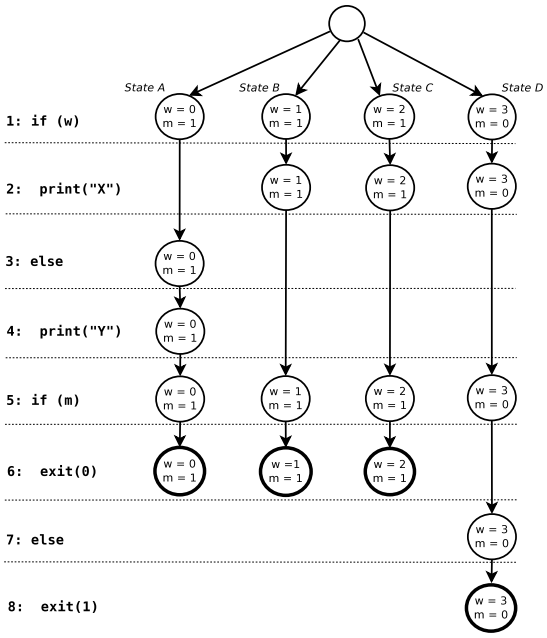


Figure 1: A diagram illustrating the *state tree* produced by symbolically executing the program in Section 2 from the initial states A, B, C, and D. Each state is represented by a circle that is labeled with the state's constraints. Each row of the diagram corresponds to a line of the program. States at exit instructions are denoted by thick borders. Let pc_1 be the program counter of state σ_1. An arrow from state σ_1 to state σ_2 means that σ_2 is the result of executing pc_1 on σ_1. The initial states A, B, C, and D appear in the top row, which corresponds to line 1. Note that the path starting from state A does not reach lines 2, 7, and 8, which is reflected in the diagram by missing circles in those rows along the path.

approach detects which variables affect branch instructions that control uncovered lines and restrict state comparison to constraints that involve these variables. By minimizing the set of constraints used to compare states, the analysis can find more opportunities to eliminate redundant states.

In the remainder of this section, we describe how our analysis works on the example program. As described above, symbolic execution of the program begins with four initial states A, B, C, and D whose program counters are set to line 1. First, state A is selected and a path through lines 1, 3, 4, 5, and 6 is explored until it terminates by executing the exit instruction on line 6. Figure 2 shows four different state trees at various stages of symbolically executing the program using our analysis. The leftmost diagram shows the state tree immediately after the path starting from A terminates. Note that the diagram indicates that the path starting from state B has not

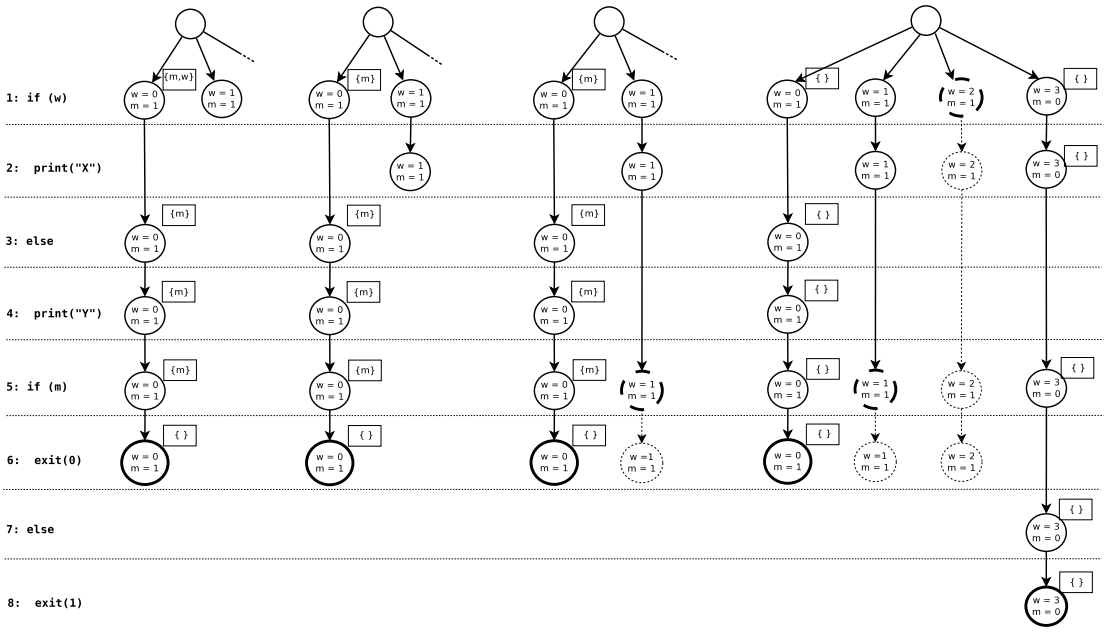


Figure 2: The state tree at four different stages of symbolic execution of the example program in Section 2. The box next to each state contains its *relevant location set*.

been explored yet since B has no successor states. At this stage, the uncovered lines are 2, 7 and 8.

As soon as the path starting from initial state A terminates, the analysis determines which variables affected the decisions of branch instructions along the path that controlled each uncovered line. We call these variables *relevant locations*. A set of relevant locations at each point along the terminated path is computed backwards starting from the termination point. For example, the set of relevant locations at line 1 along this path is {`m`,`w`} because, at this stage of symbolic execution, lines 2, 7, and 8 are uncovered, and both `m` and `w` affect the decisions of the branches that control these lines. Similarly, the relevant location set at line 5 is {`m`}, because, from line 5 onward, the only variable that affects the decision of a branch that controls an uncovered line is `m`.

After the analysis has computed a relevant location set for each of the previously executed states along this terminated path, it can now try to *match* any of the currently running states to the previously executed ones that have the same program counter. For example, our analysis tries to *match* the currently running state B to the previously executed state A. This matching is performed by looking up the relevant location set of state A, which is {`m`,`w`}, and checking whether the constraints of state A that involve `m` and `w` *imply* the constraints of state B that involve `m` and `w`. Because the constraints differ on variable `w`, the analysis determines that no match exists, and thus, it cannot eliminate B.

Next, the path starting at state B executes line 1 and moves to line 2. The second diagram from the left in Figure 2 shows the state tree at this stage of symbolic execution, which illustrates an important aspect of our analysis. Now, lines 1, 2, 3, 4, 5, and 6 are all covered, which means that the branch on line 1 no longer controls an uncovered line. Our analysis immediately detects this change and *refines* the relevant location sets along the path starting from state A by removing `w` because `w` no longer affects the decision of a branch that controls an uncovered line. Dynamically adjusting the relevant location sets as more lines become covered increases our analysis's ability to eliminate redundant states. As a result, in many cases, the analysis can eliminate all states, exhaustively exploring the state space and *soundly proving* that the remaining uncovered lines are dead code with respect to the modeled environment.

After the refinement of the relevant location sets, the analysis tries to match the currently running state on line 2 to other previously executed states along terminated paths. The analysis fails to find a match because, so far, no other states have reached line 2, so there's nothing to compare it to. Thus, the path continues executing until it reaches line 5 as illustrated in the third state tree from the left in Figure 2. Let this state be called $\sigma_{B,5}$ because it is generated on the path that started from state B and its program counter is line 5. Again, the analysis tries to find a match for $\sigma_{B,5}$ and sees that a previously executed state on the terminated path starting from A has the same program counter. Let this state be called $\sigma_{A,5}$.

Now, the analysis checks to see if $\sigma_{A,5}$'s constraint set implies $\sigma_{B,5}$'s constraint set with respect to the relevant location set associated with $\sigma_{A,5}$. In this case, a match does exist: $\sigma_{A,5}$'s constraint set is {w=0,m=1}, $\sigma_{B,5}$'s constraint set is {w=1,m=1}, and $\sigma_{A,5}$'s relevant location set is m. Note that even though their constraint sets differ on w, a match still exists because w is not relevant to covering the remaining lines 7 and 8. After finding the match, the analysis *prunes* $\sigma_{B,5}$ by eliminating it from symbolic execution and deallocating it. Figure 2 illustrates that $\sigma_{B,5}$ has been pruned by giving it a thick, dashed border.

Next, the analysis performs the same process on the paths starting from states C and D. State C is pruned immediately because it matches state A on m. State D is never pruned. The analysis tries to match states along the path starting from D to the previously executed states, but it finds that they always differ on m: in state D, m is 0 and in the other three states, it is 1. The path starting from D proceeds until it reaches the uncovered lines 7 and 8 and then terminates when it reaches the exit instruction on line 8.

The fourth state tree in Figure 2 illustrates the final state tree generated at the end of the symbolic execution. Note that because all lines are now covered, the relevant location sets for every state have been *refined* to the empty set because there are no longer any more branch instructions that control uncovered lines.

3 Redundant State Detector

This section explains how the redundant state detector dynamically monitors symbolic execution and eliminates redundant states. Algorithm 1 gives a high-level description of the different parts of the detector and how they interact with symbolic execution. The boxed lines are performed by the detector, and the non-boxed ones are performed by normal symbolic execution, which we discuss first.

Symbolic execution uses a worklist algorithm to generate and execute states until the entire state space is explored. A state consists of both a program counter and a constraint set that encodes the sequence of branch decisions made by the state thus far in terms of program memory locations. Line 2 initializes the worklist with special states in which global variables are initialized and program arguments have been assigned to arrays of "unknowns" which can take on any value. On each iteration of the worklist algorithm, line 4 selects and removes a state with constraint set C from the worklist. If the state is at a branch, lines 6 - 10 create two copies of the state and place them on the worklist if their paths are feasible. Otherwise, line 12 updates the state with the effects of the instruction specified by its program counter, which is then incremented on the following line. If the state has reached a program exit, line 21 generates concrete program inputs that drive program execution down the corresponding path, and then line 22 deletes the state. Otherwise, the state is inserted into the worklist on line 24.

```
   input: A program and a set of initial states
1  ConstructStaticControlDepGraph()
2  worklist ← InitialStates
3  while worklist ≠ ∅ do
4      state_C ← worklist.pop()
5      if state_C at branch condition B then
6          (state_B, state_¬B) ← Fork(state_C)
7          if C ∧ B is satisfiable then
8              worklist.insert(state_B)
9          if C ∧ ¬B is satisfiable then
10             worklist.insert(state_¬B);
11     else
12         Execute(state_C)
13         increment state_C's program counter
14         UpdateDynamicDepGraph(state_C)
15         if state_C.pc was previously uncovered then
16             UpdateRelBranchSet(state_C)
17             RefineRelLocSets()
18         FindMatch(state_C)
19         if state_C has match or is at program exit then
20             ConstructRelLocSets(state_C)
21             compute test inputs for state_C
22             delete state_C
23         else
24             worklist.insert(state_C)
```

Algorithm 1: High-level description of how the redundant state detector dynamically monitors symbolic execution to eliminate redundant states.

We now discuss the statements of Algorithm 1 performed by the redundant state detector. On line 1, the detector constructs a *static control dependence graph* to keep track of which static branches are *relevant* in the sense that they control a line that has not yet been covered by symbolic execution as described in Section 3.1. Line 14 updates the *dynamic dependence graph* immediately after every call to Execute with the effects of the recently executed instruction. This graph, which is described in Section 3.2.1, tracks the dynamic data dependencies between two writes as well as the dynamic control dependencies between a dynamic branch and the writes it controls.

On line 15, the algorithm checks whether the state has reached a previously uncovered instruction. If so, on line 16, the detector uses the static control dependence graph to update the current set of relevant static branches. Then, on line 17, it *refines* all relevant constraint sets constructed thus far because they may contain reads that are

no longer relevant as described in Section 3.4.

On line 18, as described in Section 3.3, the detector searches for a relevant constraint set constructed along a previously explored path that matches the state. A successful match implies that the state is redundant. As soon as a state reaches a program exit or is pruned, the detector dynamically slices the part of the dynamic dependence graph corresponding to the path followed by the state to determine which reads along the path were *relevant* in the sense that they potentially affected whether the state reached an uncovered instruction. Section 3.2 describes how the call on line 20 extracts the dynamic slice and how the slice is used to construct relevant constraint sets.

3.1 Relevant Static Branches

The most basic component of the redundant state detector is identifying which static branches of the program are *relevant*. A static branch is *relevant* if the outcome of its condition may affect whether an uncovered instruction is reachable. Consider the simplified code snippet from the Unix utility `chown` below. Suppose for now that the program never exits before line 11—we explain how we handle *embedded halts* in Section 3.2.3.

```
1  if (reference_file) {
2    if (stat (...))
3      error (...); //uncovered
4    ...;
5  } else {
6    if (parse_user_spec (...))
7      error (...);
8    ...;
9  }
10 if (chopt.recurse & preserve_root)
11   ...; //uncovered
```

Because lines 3 and 11 are uncovered, the branches `if(reference_file)` on line 1, `if(stat(...))` on line 2 and `if(chopt.recurse & preserve_root)` on line 10 are the relevant static branches since they affect whether an uncovered instruction is reachable.

The detector identifies the relevant static branches of a program by constructing a *static control dependence graph* [17] whose nodes are the static instructions of the program and whose edges (b,i) signify that branch b *statically controls* instruction i, so the outcome of b's condition affects whether i is executed. A static branch b is relevant if there exists a path in the static control dependence graph from b to some uncovered instruction i. For example, the static control dependence graph for this code snippet contains four nodes: one for each branch condition on lines 1, 2, 6, 10. The static branch `if(reference_file)` on line 1 is relevant because line 3 is uncovered, and the graph has a path from line 1 to line 3 via line 2.

Note that the set of relevant static branches gets smaller and smaller as more instructions are reached and become covered. Thus, every time a state reaches an uncovered instruction, the detector updates the set of relevant static branches as shown in Algorithm 1, line 16.

3.2 Relevant Locations

The detector determines that a state is redundant by comparing it to previously executed states. Conceptually, to perform this comparison, it records a complete history of the symbolic execution by taking a *snapshot* of each state every time it executes an instruction. The kth *snapshot* of a state is simply a copy of the state's constraint set immediately after it executes the kth instruction along its path.

If a snapshot's constraint set is equivalent to a state's constraint set, then the detector concludes that the state is redundant and eliminates it. However, this condition is unnecessarily conservative in the sense that a state may be redundant even if its constraint set is not entirely equivalent to a snapshot's. For example, if a snapshot's constraint set is a *subset* of a state's constraint set, then the detector can also conclude that the state is redundant because the state is "more constrained" than the snapshot and thus will not explore any new relevant behaviors of the program.

We say that this subset condition is more *precise* than the equivalence condition because its weaker and thus allows the detector to find more redundant states. We also say that it is *sound* because, even though it is weaker, it never concludes that a state is redundant when it is not.

This approach faces two practical challenges: how to incrementally encode every snapshot of the symbolic execution efficiently, and how to compare a state to a snapshot precisely. The detector addresses both challenges simultaneously using *dynamic slicing*, a program analysis technique that identifies which instructions along a path affect the value of a given memory location [1].

As soon as a state reaches a program exit, the detector dynamically slices the path taken by the state to identify, at every kth instruction along the path, which locations are *relevant* in the sense that they potentially affected the outcome of the decision of a relevant static branch further down the path. We refer to the relevant locations at the kth instruction along a path taken by a state as the state's kth *relevant location set*. Note that the relevant location sets for a state are not constructed until it has reached a program exit to ensure that no relevant locations are missed.

We use relevant location sets to devise an even more precise, yet sound, condition for detecting that a state is redundant: if the constraints of a snapshot that *depend on a relevant location* is a subset of the set of constraints of a state that *depend on a relevant location*, then the state is redundant. Conceptually, the constraints in constraint

set Δ that depend on a location l are those that limit the possible values that l can have. Formally, let L be a set of relevant locations. The subset $\Delta|_L$ of Δ that depends on locations in L is recursively defined as the constraints in Δ that use locations in L or locations that appear in some constraint in $\Delta|_L$. For example, the constraints in the set $\{a > b,\ b = 0,\ c = 1,\ d = 2\}$ that depend on the locations $\{a, c\}$ are $\{a > b,\ b = 0,\ c = 1\}$.

We use the following abridged code snippet from the Unix utility `chown` to demonstrate relevant location sets. The symbols `TRUE` and `FALSE` in the code below are `int` constant variables assigned the values 1 and 0, respectively. Suppose the initial state consists of three command line arguments: the program name in `argv[0]`, a three-byte array of "unknowns" in `argv[1]`, and a sixteen-byte array of "unknowns" in `argv[2]`. Further, suppose that lines 7 and 21 are the only uncovered lines.

```
1  chopt.recurse = FALSE;
2  preserve_root = FALSE;
3  ...
4  if (!strcmp(argv[1], "-R"))
5    chopt.recurse = TRUE;
6  if (!strcmp(argv[2], "--preserve-root"))
7    preserve_root = TRUE; //uncovered
8  ...
9  if (reference_file) {
10   ...
11   chopt.user_name = uid_to_name(...);
12   chopt.group_name = gid_to_name(...);
13 } else {
14   ...
15   if(!chopt.username && chopt.group_name)
16     chopt.username = "";
17   ...
18 }
19 if (chopt.recurse)
20   if (preserve_root)
21     ...; //uncovered
22 ok = chown_file(...,&chopt);
23 exit(ok);
```

The branches on lines 6, 19, 20 are the only relevant static branches because they control the only uncovered instructions as discussed in Section 3.1. Consider the state that takes the path through lines 1-6, 8-12, 19, 22, 23. As soon as it reaches the program exit on line 23, the detector constructs relevant location sets at every point along the path, which are shown in the table below.

Line	Relevant Location Set
1-3	{ `TRUE`, `argv[2][0]`, `argv[1][0..2]` }
4	{ `TRUE`, `argv[2][0]`, `argv[1][0..2]` }
5	{ `TRUE`, `argv[2][0]` }
6	{ `chopt.recurse`, `argv[2][0]` }
8-12	{ `chopt.recurse` }
19	{ `chopt.recurse` }
22	{}
23	{}

On line 23, the relevant location set is trivially empty because it is a program exit. On line 22, it is empty because no relevant static branches are reachable on the path from this point onward. Now on line 19, the detector determines that `chopt.recurse` is a relevant location at this point because it affects the decision of the relevant static branch on line 19. On lines 10-12, `chopt.recurse` is also relevant for the same reason. Note that `preserve_root` is not relevant on these lines because even though it *statically* controls an uncovered instruction, it does not *dynamically* control the uncovered instruction, since it is not read along the path taken by the state.

Now, on lines 8 and 9, the relevant location set is still {`chopt.recurse`} and does not include `reference_file`. Even though `reference_file` is read by a branch condition along the path, it does not affect the decision of a *relevant* branch. The power of the detector is demonstrated here: it is capable of reasoning that the entire `if-else` block on lines 9-18 does not affect whether the uncovered instruction on line 21 is reachable and can thus be ignored, keeping the number of relevant locations small, which is valuable when comparing a snapshot to a state: the fewer of a snapshot's constraints that need to appear in the state's constraints, the greater the chance of soundly concluding that the state is redundant.

On line 6, `argv[2][0]` is included in the relevant location set because this location is used by the condition of the relevant static branch on line 6 via the call to `strcmp`.

Now, on line 5, `chopt.recurse` is replaced by the constant `TRUE` in the relevant location set. Conceptually, the location `chopt.recurse` is not relevant at this point because the write on line 5 which affects the decision of the relevant static branch on line 19 has not occurred on the path yet. The location `TRUE`, however, is now relevant because it is used by the assignment to compute a value that is written to a relevant location. In Section 3.5 we describe a way to handle locations that have the same value across all paths, such as `TRUE`, in a special way that reduces the overhead of the detector.

Finally, on line 4, `argv[1][0..2]` is included in the relevant location set because it is used by the condition of the branch on line 4 that dynamically controls the write of the relevant location `chopt.recurse`. In the remainder of this section, we describe how the relevant location sets are constructed in the general case. First we describe how the *dynamic dependence graph* is constructed in Section 3.2.1 and then, in Section 3.2.2, how relevant locations are inferred by slicing the graph with respect to relevant static branches.

3.2.1 Dynamic Dependence Graph

Throughout symbolic execution, the detector records the dependencies between executed instructions along each path by incrementally updating the *dynamic dependence graph* whose nodes are byte-level writes and whose edges are either *data*, *control*, or *potential* dependencies [1, 2]. A write w_2 performed by instruction i is *data-dependent* on another write w_1 if i reads w_1. A write w_2 is *control-dependent* on a write w_1 if a branch that reads w_1 dynamically controls w_2. A write w_2 is *potentially-dependent* on a write w_1 if a branch that reads w_1 statically controls assignment $e_2 := e$ not along the path and e_2 aliases the static location of w_2.

Inferring data and control dependencies is straightforward because it requires reasoning about parts of the executed path only, and not other parts of the program. In contrast, inferring potential dependencies is challenging because it requires reasoning about both the executed path as well as static locations on non-executed paths. Consequently, potential dependencies require a sound, interprocedural aliasing analysis. Our implementation uses a sound, highly precise intraprocedural pointer analysis to resolve non-escaping aliases and a sound, scalable, yet coarse interprocedural alias analysis to resolve escaping aliases [13]. While state-of-the-art, sound aliasing analysis is notoriously imprecise, in Section 3.6, we describe a technique that allows the detector to recover from some of the precision loss. Note that this imprecision only affects the detector's ability to identify redundant states; it does not affect the completeness of the symbolic execution and does not cause it to report false positives.

3.2.2 Dynamic Slicing

This section describes how, in general, the dynamic dependence graph is sliced with respect to relevant static branches in order to infer relevant location sets. One slight complication, however, is that symbolic execution paths are not linear, but rather form trees, because a state may fork into several states. Thus, the detector cannot construct the kth relevant location set until *all* states generated by forks *after* the kth instruction have reached a program exit.

When a state reaches a program exit, the detector starts from the end of the path taken by the state and constructs the relevant location set L_k at the kth instruction along the path. Let C_k be the *relevant control set* used as an intermediate set for constructing relevant location sets that consist of the dynamic branches that control the kth instruction. The sets L_k and C_k are constructed as follows:

1. Compute $L_k^{\cup}$, which is the union of the relevant location sets at the immediate successor instructions of the kth instruction. Recall that an instruction along a path may have multiple immediate successor instructions if symbolic execution forked at that point.
2. Compute $C_k^{\cup}$, which is the union of the relevant control sets at the immediate successor instructions of the kth instruction.
3. If the kth instruction is a program exit, both L_k and C_k are empty.
4. If the kth instruction is a dynamic branch b, then add to C_k all of $C_k^{\cup} - \{b\}$, and also add to C_k the dynamic branch that controls b. If at least one of the following three situations hold, then add to L_k the locations used in the branch condition:
   (a) b corresponds to a relevant static branch,
   (b) b is in $C_k^{\cup}$,
   (c) some location in $L_k^{\cup}$ potentially depends on b.
5. If the kth instruction is a dynamic assignment $e_2 := e_1$ and the location of e_2 is in $L_k^{\cup}$, then add to C_k the dynamic branch that controls the assignment, and add to L_k all the locations read by the expression e_1.

3.2.3 Irregular Control Dependence

The explanation of the detector thus far is not sound for programs that have irregular control flow introduced by *embedded halts* [17] which are instructions that terminate execution of the program, such as calls to `exit` and `abort`, that are distinct from the normal termination point. An embedded halt induces interprocedural control dependence from the branch that controls the halt to *any* statement statically reachable from that branch. These interprocedural control dependencies must be handled by the detector because, otherwise, it may not recognize a relevant location as relevant. By expanding the third step of the dynamic slicing algorithm in Section 3.2.2 as follows, the detector will soundly handle irregular control flow:

3. If the kth instruction is a program exit, then L_k is empty and C_k is the set of dynamic branches that control it.

3.2.4 Inter-Path Dynamic Slicing

So far in this paper, relevant location sets have been constructed along a path only when the state reaches a program exit. Because the detector eventually prunes almost every state before the state can reach a program exit, very few paths have relevant location sets constructed along them. Consequently, the precision of the detector is severely limited: in general, the more relevant location sets that are constructed, the more opportunities the detector has to find a match for state and thus to prune it.

We add an additional step to the dynamic slicing algorithm in Section 3.2.2 to give the detector the ability to construct relevant location sets along paths taken by states that are pruned before they reach a program exit:

6. If the kth instruction is the last instruction along the path of the state that was matched and pruned by the relevant location set L'_j, then add to L_k all of L'_j and add to C_k all the dynamic branches that control the kth instruction.

The effect of this additional step allows the dynamic slicing algorithm to incorporate the slicing result of previously explored paths to construct relevant location sets along new paths of pruned states.

3.3 State Matching

A state is redundant if it *matches* any snapshot, which occurs when the following conditions hold. Let L be the relevant location set that corresponds to the snapshot.

1. The state and the snapshot are at the same *context-sensitive static instruction*. A state's *context-sensitive static instruction* is a pair consisting of (1) the state's program counter and (2) the sequence of call instructions on its call stack.
2. The snapshot constraints that depend on L are a subset of the state constraints that depend on L.

Recall from Section 3.2 that the constraints of a constraint set that depend on a location l are those that limit the possible values that l can have in a satisfying assignment.

Throughout Section 3, we use the concept of snapshots to explain how the detector works. However, our implementation never explicitly constructs these snapshots because the time and space overhead of allocating millions of constraint sets, each of which contains tens of thousands of constraints is prohibitively expensive. Instead, the detector exploits the fact that the matching conditions above only need the constraints of a snapshot that *depend on relevant locations*, called the *relevant constraints* of the snapshot. The set of relevant constraints of a snapshot is 100X smaller than its entire set of constraints.

3.4 Dynamic Refinement

In this section we describe a technique that increases the detector's precision as symbolic execution covers more and more of the program. Recall from Section 3.1 that relevant static branches are those that control uncovered instructions, and a location is relevant if it affects the decision of a relevant static branch. Thus, as symbolic execution covers more uncovered instructions, the fewer the relevant static branches, and the fewer the relevant locations, and thus the more chances the detector has to find a match for a state.

As soon as symbolic execution reaches an uncovered instruction, the detector iterates over all the relevant location sets and removes any locations that were included because that line was previously uncovered.

This dynamic refinement of relevant location sets substantially improves the precision of the detector, making it possible to exhaustively explore the entire state space on some benchmarks, *proving* that any remaining uncovered instructions are dead code with respect to the environment without any false positives or false negatives.

3.5 Single-Valued Locations

In this section, we describe an optimization that reduces the overhead of the detector by exploiting the fact that, in practice, more than half of the locations that appear in relevant constraint sets are *single-valued*, meaning they have the same value written to them along every path explored thus far by symbolic execution. Note that a single-valued location is not necessarily constant; it may be that the symbolic execution will eventually explore a path that writes a different value to the location. Thus, a location that is *single-valued* for the first few minutes of symbolic execution may not be single-valued thereafter. These locations are prevalent in programs that extensively use libraries because the majority of locations are typically initialized to a single value and then mostly read and rarely overwritten with a different one.

The detector exploits this observation by removing single-valued locations from relevant constraint sets, thus reducing the sizes of the sets, and consequently, reducing overhead substantially. The optimization is sound and does not introduce a loss in precision.

One difficulty with implementing this optimization is that a location l may be single-valued for the first n symbolically executed instructions, but on the $n+1$ instruction, a different value may be written to it. At this point, the detector may become unsound and prune a non-redundant state because a relevant constraint set constructed before this point may have previously removed l. To ensure that soundness is maintained, as soon as l is written-to with a different value, the detector identifies which relevant constraint sets previously removed l and re-adds l to them before executing the $n+2$ instruction. This refinement step requires re-slicing the paths along which l was removed.

3.6 Relevant Search Heuristic

One source of imprecision in the detector is the use of a sound, coarse, interprocedural alias analysis to infer po-

tential dependencies as described in Section 3.2.1. Consequently, locations that are not actually relevant may be inferred as such. To recover from this loss of precision, the detector keeps *two* relevant location sets at each instruction along a path: one relevant location set is constructed *soundly* by slicing through data, control and potential dependencies in the dynamic dependence graph, and the other is constructed *unsoundly* by only slicing through data and control dependencies.

Throughout the entire symbolic execution, the detector uses only sound relevant location set to prune states. However, simultaneously, the detector incorporates the unsound relevant location sets into its search strategy by giving priority to states that do *not* match any unsound relevant location set. This technique substantially decreases the time it takes to reach the maximum coverage of a program. Intuitively, it is effective because it gives preference to states that are the most "different", in a relevant way, from any other previously explored state, and thus are more likely to reach an uncovered instruction.

3.7 Efficient State Matching

As discussed in Section 3.3, after each executed instruction, the detector searches for a match of a state by comparing it to all the relevant constraint sets at the same context-sensitive static instruction. A straightforward implementation will perform this search by comparing the state to each relevant constraint set individually.

Unfortunately, even programs with fewer than ten thousand lines of code will have thousands of unique relevant constraint sets at each program point, and each set may contain hundreds of constraints. Thus, comparing a state to each relevant constraint set individually after each executed instruction is prohibitively expensive. To address this challenge, the detector constructs, at each context-sensitive static instruction *pc*, a *decision tree* [16] that organizes relevant constraint sets at *pc* in a manner that makes searching for a match efficient. A decision tree is a highly effective data structure for this search problem because it can exploit the fact that the relevant constraint sets at a program point share many common locations.

Consider the following seven relevant constraint sets at context-sensitive static instruction *pc*.

1. { `x = 2, y = 4, u = 10` }
2. { `x = 2, y = 5, u = 10` }
3. { `x = 1, u = 10` }
4. { `x = 3, u = 11` }
5. { `x = 2, u = 11` }
6. { `x = 2, u = 10, w = 6` }
7. { `x = 2, u = 10, w = 7` }

Suppose that the current state is also at *pc* and has constraints {`x = 2, y = 4, z = 8, u = 11`}. If the state is compared to each relevant constraint set individually, the detector would need to perform eighteen lookups in the state's constraint set, one for each relevant constraint. However, by organizing these relevant constraints sets into a decision tree, only four lookups are required.

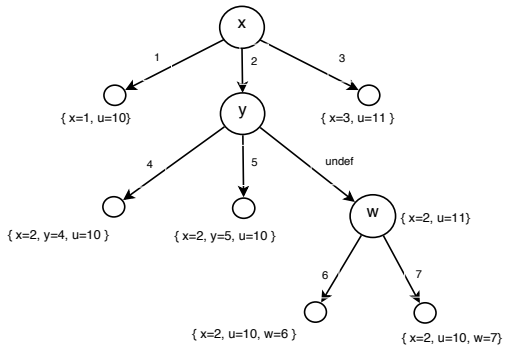


Figure 3: The decision tree for the seven relevant constraint sets in Section 3.7.

Figure 3 shows the decision tree for the seven relevant constraint sets above. Each of the tree's non-leaf nodes is labeled with a location l, and its outgoing edges are labeled with the possible values associated with l in the relevant constraint sets. If l does not appear in every relevant constraint set, then it may have an outgoing edge labeled `undef`. Each relevant constraint set is associated with exactly one of the nodes. In the figure, the node for location `x` has three outgoing edges, one for each of the possible values that `x` is constrained to. The node for location `y` has an outgoing edge labeled `undef` because it is not constrained in some relevant constraint sets.

The decision tree is used to search for a relevant constraint set that matches the state. The search process starts at root node `x`, so `x` is looked up in the state and is found to have the value 2. Thus, the search process moves to the child whose incoming edge has the value 2, which is node `y`. Then, `y` is looked up in the state and is found to have the value 4. Thus, the search process moves to the child whose incoming edge has the value 4 and contains the relevant constraint set {`x = 2, y = 4, u = 10`}. Now, the search process checks if this relevant constraint set matches the state entirely. The locations `x`, `y` and `u` are looked up in the state to see if they have the same values in the relevant constraint set, that is, 2 and 11, respectively. They do not match on the location u, so the state does not match this relevant constraint set. Because the current node is a leaf, the search process ends, which means that the state does not match any of these relevant constraint sets. Thus, by using a decision tree for the search process, only four lookups were necessary in this example, compared to eighteen lookups if the relevant constraint sets were compared individually.

In general, the search process starts at the root node of

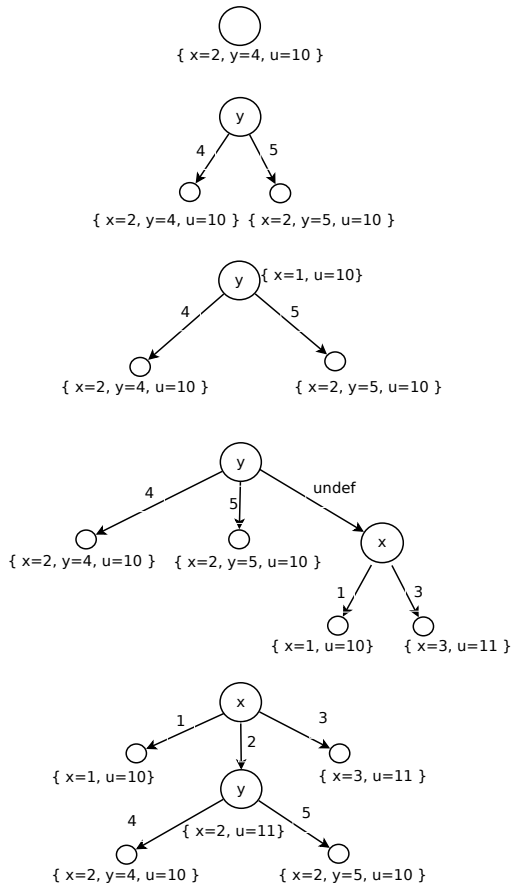


Figure 4: The different stages of the decision tree in Figure 3 as each of the first five relevant constraint sets in Section 3.7 is incorporated.

the decision tree and performs the following steps until either a match or a conflict is found. Let t be a non-leaf node representing a location l. If l is defined by the state, let v be its value. There are three possibilities:

1. If l is defined in the state and if t has an outgoing edge labeled v to a node s, then the search proceeds to s.

2. If l is defined in the state and if t has no outgoing edge labeled with v, then a conflict is found and the search process at this node ends without finding a match.

3. If l is undefined in the state, the search proceeds to each child node of t individually.

Construction. Even for programs with fewer than ten thousand lines of code, thousands of new, relevant constraint sets are generated during each minute of symbolic execution. Thus, the construction of these decision trees must be *incremental*: every time a relevant constraint set is generated, it must be efficiently incorporated into the already existing decision tree.

Figure 4 shows how the decision tree in Figure 3 is constructed incrementally as each of the first five relevant constraint sets above are added. Initially, the decision tree contains a single, empty root node. The first set { `x = 2, y = 4, u = 10` } is simply added to the root node. When the second relevant constraint set { `x = 2, y = 5, u = 10` } is added, the root node is labeled with the location that has most number of distinct values, which is `y` because it takes on the two values, 4 and 5, whereas locations `x` and `u` each only take on a single value. Then, two child nodes are added, one for each distinct value of `y`. The relevant constraint set { `x = 2, y = 4, u = 10` } is placed at the child node for value 4 and { `x = 2, y = 5, u = 10` } at the child node for value 5.

Conceptually, `y` is chosen because it splits the relevant constraint sets into as many partitions as possible, thus minimizing the number of lookups needed to find matching relevant constraint sets. If either `x` or `u` was chosen, no partitioning would have been possible. In general, the split heuristic used by the decision trees to minimize the number of lookups is to select locations that maximize the number of partitions at each level of the tree.

Next, the third set { `x = 1, u = 10` } is added to the root node because it does not have a constraint for `y`, and it is the only such set. Once the fourth set { `x = 3, u = 11` } is added, a new child node labeled `x` is created whose edge from the root node is labeled `undef`. Both the third and fourth sets are placed at separate child nodes of this `x` node.

Now, once the fifth set { `x = 2, u = 11` } is added, the location `x` takes on three distinct values whereas `y` only takes on two distinct values. Consequently, the split heuristic is violated at the first level of the tree. As a result, the tree is rebuilt from scratch to have location `x` at the root node and a node labeled `y` as one of its children.

We use two additional optimizations to further reduce construction overhead:

1. Relevant constraint sets are added to a decision tree *lazily*. Instead of incorporating the constraint set immediately after it is generated, the detector waits until a state actually needs to match itself against the corresponding decision tree.

2. Only the specific subtree of the decision tree that violates the split heuristic is rebuilt from scratch. The remaining portion of the tree remains intact.

4 Evaluation

We implemented our state space reduction analysis by augmenting a copy of the open source version of

KLEE [5], a symbolic virtual machine capable of automatically generating test inputs for complex programs such as device drivers, network drivers, and utility programs written in C. For the remainder of this paper, we refer to this unmodified open source version of KLEE as KLEE-BASE, and we refer to our augmentation of KLEE with our state space reduction analysis as KLEE-REDUCE. In this section, we discuss the details of our implementation that enabled us to achieve the orders of magnitude improvement in performance.

KLEE-BASE performs symbolic execution over LLVM bytecode instructions starting from several initial states, each with different numbers and sizes of symbolic objects representing the program's arguments. The initial states were generated using the following KLEE flags: `--sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdout`. We configured KLEE-BASE to use KLEE's best, built-in heuristic search strategy for line coverage, which is specified using the flags: `--use-random-path --use-interleaved-covnew-NURS --use-batching-search --batch-instructions=10000`.

This section measures how KLEE-REDUCE performs compared to KLEE-BASE on three metrics: (1) how much faster it reaches the same statement coverage, (2) how much more statement coverage it gets, and (3) how many more programs have their state spaces exhausted.

Our benchmarks consist of 66 programs from the GNU Coreutils utility suite, a diverse set of system-intensive, complicated programs that form the core user-level environment installed on millions of computer systems. They are the same benchmarks used in the original KLEE paper [5] and are among the largest and most complex benchmarks that constraint-based automatic input generation has been shown to run on for code coverage. The reader is referred to Figure 4 in [5] for the distribution of program sizes.

We view Coreutils as a fair test for KLEE-BASE since it was used in the original paper [5]. For similar reasons, our experiments follow the original paper's methodology: each program was checked for one hour each with the same flags and with the same number (and sizes) of symbolic inputs.

Our experiments have two changes from the originals, which we do not expect to have substantive impact. First, the open source version of KLEE eliminated some flags used by the original system, so we obviously could not use them. Second, we only checked 66 of the 89 possible utilities since the others either had errors when run on our 64-bit machine (the original KLEE-BASE results were on 32-bit) or were so small that they reached the maximum possible coverage in under ten seconds. The sizes of these 66 utilities varied from 6.6 K to 29 K instructions of *optimized* LLVM bytecode (using the `--optimize` KLEE flag) with a median size of 12.6 K. All together, they sum to 905 K instructions.

Speedups. We calculated speedup as follows:

1. Ran each program for one hour with KLEE-BASE and one hour with KLEE-REDUCE .
2. Recorded the maximum coverage C_{max} that both KLEE-BASE and KLEE-REDUCE were able to reach.
3. Recorded the times T_{base} and T_{red} at which KLEE-BASE and KLEE-REDUCE reached C_{max}, respectively.
4. Calculated the speedup as T_{base}/T_{red}.

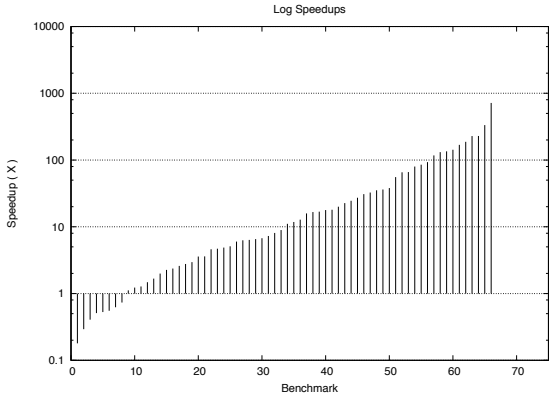


Figure 5: Log-scale speedups of how many times faster KLEE-REDUCE reaches the same statement coverage as KLEE-BASE.

Figure 5 uses a log scale to show the relative speedup, sorted from least to most. A bar below 1.0 means KLEE-REDUCE ran slower than KLEE-BASE and a bar above 1.0 means that it ran faster. As can be seen from the results, KLEE-REDUCE gives enormous speedups on the vast majority of benchmarks. KLEE-REDUCE 's average speedup is 50.5 X , that is, 50.5 times faster than KLEE-BASE. Its median speedup is 10 X , the maximum is 717 X and its maximum slow down is 0.2 X . And, 54 out of 66 (82 %) benchmarks had speedups greater than 1 X.

Table 1 shows the impact of KLEE-REDUCE on several metrics. Table 2 shows individual results from running KLEE-BASE and KLEE-REDUCE on the ten largest coreutils benchmarks.

Coverage. On 55 of the 66 (83 %) benchmarks, KLEE-REDUCE reaches at least the same coverage

Statistic	BASE	REDUCE	Reduction
Instructions	1.8 billion	222 million	8 X
Paths	5,285,596	391,057	14 X
Queries	266,286	77,614	3.4 X
Query Constructs	24.3 million	9.8 million	2.5 X
Solver Time	20.6 hours	4.1 hours	5 X
Solver Overhead	73 %	37 %	1.9 X

Table 1: Impact of KLEE-REDUCE on the number of instructions executed, paths generated, queries made, solver time, and solver overhead across all benchmarks.

	KLEE-BASE			KLEE-REDUCE				
	T	cov	T_c	T	cov	T_c	X	cov_{inc}
join	3,587	83.1	3,587	3,189	87.5	1,002	3.6	4.4
csplit	1,686	70.4	1,686	3,163	77.8	1,322	1.3	7.4
stty	2,340	58.2	2,340	694	86.5	66	35.2	28.3
dd	3,353	40.3	3,353	1,999	44.5	137	24.5	4.2
tail	3,230	70.1	3,230	2,305	76.8	143	22.7	6.7
od	3,599	74.7	3,599	1,250	84.9	200	18	10.2
tr	2,839	60.6	2,839	1,164	64	962	3	3.4
ptx	1,541	18.5	1,541	3,461	40.6	18	85.1	22.1
pr	234	57.2	29	54	39.4	54	0.5	-17.8
ls	298	34	92	82	29.5	82	1.1	-4.5

Table 2: Individual results from running KLEE-BASE and KLEE-REDUCE on the ten largest coreutils benchmarks. T is time to reach the maximum coverage, cov is maximum coverage, T_c is time to reach the same coverage, X is relative speedup, and cov_{inc} is coverage increase.

as KLEE-BASE, and on 30 (45 %) KLEE-REDUCE reaches higher coverage. The average coverage increase is 3.8 % with a median of 2.5 % on the 54 benchmarks on which either KLEE-BASE or KLEE-REDUCE did not reach the maximum coverage possible given the modeled environment. Figure 6 shows the coverage increases for each benchmark.

Exhaustion. Our analysis eliminates states to the extent that on 12 out of 66 benchmarks (18 %), KLEE-REDUCE explores every state in the state space exhaustively, *proving* that any remaining uncovered lines are dead code with respect to the modeled environment. KLEE-BASE , on the other hand, was not able to explore the state space exhaustively on any of these benchmarks.

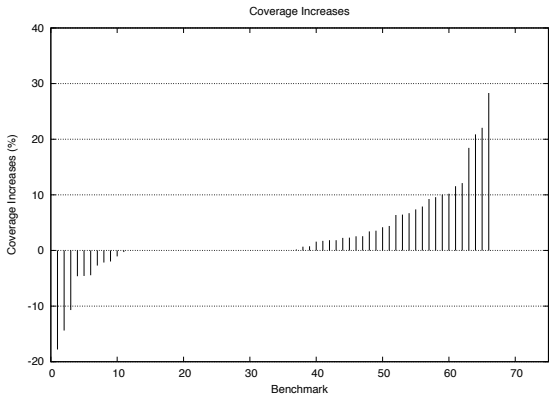


Figure 6: Coverage increases

Challenges. Now, we discuss the 14 out of 66 (21 %) benchmarks that were either slower than KLEE-BASE or had lower coverage. On eleven of these 14 benchmarks, KLEE-REDUCE did not reach the maximum coverage of KLEE-BASE, and on the remaining three, it reached the maximum coverage but was slower than KLEE-BASE.

On seven of these 14 benchmarks, KLEE-REDUCE did not outperform KLEE-BASE because these particular benchmarks had especially high solver overhead, as they spent more than 90% of their time solving queries. Thus, there was less opportunity for KLEE-REDUCE to improve performance. Furthermore, KLEE-REDUCE is biased, by design, to explore deeper parts of the program which causes it to encounter harder queries.

On the other seven of these 14 benchmarks, KLEE-REDUCE did not outperform KLEE-BASE because it does not reason precisely enough about constraints that are inconsistent in general, yet, with respect to reaching uncovered lines of code, they are equivalent. For example, consider the following common code pattern that uses the libc `read` function, which returns either -1 on error, 0 on reaching the end-of-file, or a positive value denoting the number of bytes read into `buf`.

```
1  while (1) {
2    bytes_read = read(fd, buf, sizeof buf);
3    if (bytes_read <= 0)
4      break; //uncovered
5    ...
6  }
```

Suppose a state reaches line 3 with the constraint `bytes_read = 5`, and the detector tries to match it against a relevant constraint set with `bytes_read = 7`. The detector will conclude that no match exists because it will see that these two constraints have different values for `bytes_read`. However, with respect to reaching the uncovered line, they essentially match and thus the state should be pruned.

5 Related Work

The closest antecedent to our work is Boonstoppel et al. [3]. They detect redundant states by comparing a state's *live* constraints against those of previous states that have reached the same context-sensitive program point, where a constraint is *live* if it involves a location that is read anywhere along the path taken by the previous state. They require that a depth-first search strategy is used for exploring states, which severely limits its ability to achieve high coverage. In contrast, our analysis works with any search strategy and only compares

constraints that are specifically *relevant* to uncovered instructions, thus making it significantly more precise. Inferring relevant locations is substantially more challenging because it requires slicing the dynamic data, control, and potential dependencies between locations (§ 3.2.2) and handling irregular control flow (§ 3.2.3). Furthermore, in our paper, we introduce the novel techniques of inter-path slicing (§ 3.2.4), dynamic refinement (§ 3.4), single-valued locations (§ 3.5) and using unsound relevant location sets to enhance search strategies (§ 3.6).

It is difficult to directly compare the two approaches. This previous system was built on EXE [6], which typically handles three to five orders of magnitude fewer states than KLEE, in large part because EXE created a new kernel process using (`fork`) at each branch point (e.g., each if-statement with two feasible branches). As a result, they did not solve many of the non-trivial engineering challenges we had to handle in order to successfully beat a much faster base system (KLEE). As a crude method to ignore this important engineering aspect and just compare the additional benefit of only the new refinement and propagation ideas in our approach, we disabled all techniques we added that were not in the original paper, and re-ran our system over the same 66 benchmarks in Section 4. On more than 90% of these benchmarks, the previous system either did not reach the same coverage as KLEE-BASE or was slower than it. On average, it had a coverage *decrease* of -15.5%.

In [15], the authors use dynamic slicing as part of a DART-based path exploration technique that helps avoid exploring redundant paths. They evaluate their approach on five very simple benchmarks. On the two smallest benchmarks each of which is less than 120 lines of code, they show that their approach saves a total of 60 seconds. On the other three benchmarks each of which is less than 260 lines, their approach either does not reach the same coverage as the base system or shows no improvement. We handle much larger programs, a much larger variety of them, and get much larger speedups.

Finally, there are a set of interesting state space reduction techniques for dynamic symbolic execution that improve scalability that are complementary to our work. Collingbourne et al. [8] use phi-node folding to replace control-flow forking with predicated select instructions in order to reduce the number of paths explored by symbolic execution. Kuznetsov et al. [14] propose a technique to merge states obtained on different paths to reduce the state space that a dynamic symbolic execution system needs to explore. The challenge they tackle is that merging states introduces disjunctions into the path condition and increases its complexity thereby stressing the underlying constraint solver. They demonstrate that their technique allows a symbolic execution system to explore substantially more paths. Godefroid et al. [9, 11] propose constructing function summaries for dynamic symbolic execution represented as input-output constraints. We believe using our techniques would allow this prior work to achieve even greater improvements (and vice versa).

6 Acknowledgments

We would like to thank Cristian Cadar, David Ramos, Philip Guo, and the anonymous reviewers for their insightful comments and feedback.

References

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic Program Slicing. In *PLDI* (1990).

[2] AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. Incremental Regression Testing. In *CSM* (1993).

[3] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking Path Explosion in Constraint-based Test Generation. In *TACAS* (2008).

[4] BURNIM, J., AND SEN, K. Heuristics for Scalable Dynamic Test Generation. In *ASE* (2008).

[5] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI* (2008).

[6] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *CCS* (2006).

[7] CADAR, C., GODEFROID, P., KHURSHID, S., PĂSĂREANU, C. S., SEN, K., TILLMANN, N., AND VISSER, W. Symbolic Execution for Software Testing in Practice: A Preliminary Assessment. In *ICSE* (2011).

[8] COLLINGBOURNE, P., CADAR, C., AND KELLY, P. H. Symbolic Crosschecking of Floating-point and SIMD Code. In *Eurosys* (2011).

[9] GODEFROID, P. Compositional Dynamic Test Generation. In *POPL* (2007).

[10] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *PLDI* (2005), V. Sarkar and M. W. Hall, Eds.

[11] GODEFROID, P., NORI, A. V., RAJAMANI, S. K., AND TETALI, S. D. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *POPL* (2010).

[12] GROCE, A., AND VISSER, W. Heuristic Model Checking for Java Programs. In *STTT* (2004).

[13] HACKETT, B., AND AIKEN, A. How Is Aliasing Used in Systems Software? In *FSE* (2006).

[14] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient State Merging in Symbolic Execution. In *PLDI* (2012).

[15] QI, D., NGUYEN, H. D., AND ROYCHOUDHURY, A. Path Exploration Based on Symbolic Output. In *FSE* (2011).

[16] QUINLAN, J. R. Induction of Decision Trees. *Machine Learning 1*, 1 (Mar. 1986).

[17] SINHA, S., HARROLD, M. J., AND ROTHERMEL, G. Computation of Interprocedural Control Dependence. In *ISSTA* (1998).

packetdrill: Scriptable Network Stack Testing, from Sockets to Packets

Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan,
Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert
Google

Abstract

Testing today's increasingly complex network protocol implementations can be a painstaking process. To help meet this challenge, we developed packetdrill, a portable, open-source scripting tool that enables testing the correctness and performance of entire TCP/UDP/IP network stack implementations, from the system call layer to the hardware network interface, for both IPv4 and IPv6. We describe the design and implementation of the tool, and our experiences using it to execute 657 test cases. The tool was instrumental in our development of three new features for Linux TCP—Early Retransmit, Fast Open, and Loss Probes—and allowed us to find and fix 10 bugs in Linux. Our team uses packetdrill in all phases of the development process for the kernel used in one of the world's largest Linux installations.

1 Introduction

Despite their importance in modern computer systems, network protocols often undergo only ad hoc testing before their deployment, and thus they often disappoint us. In large part this is due to their complexity. For example, the TCP roadmap RFC [19] from 2006 lists 32 RFCs. Linux implements many of these, along with a few post-2006 Internet drafts, the sockets API, a dozen congestion control modules, SYN cookies, numerous software and hardware offload mechanisms, and socket buffer management. Furthermore, new algorithms have unforeseen interactions with other features, so testing has only become more daunting as TCP has evolved. In particular, we have made a number of changes to Linux TCP [14, 15, 17, 20–22, 30] and have faced significant difficulty in testing these features. The difficulties are exacerbated by the number of components interacting, including the application, kernel, driver, network interface, and network. We found we needed a testing tool for three reasons:

New feature development. Development testing of new TCP features has often relied either on testing patches on production machines or in emulated or simulated network scenarios. Both approaches are time-consuming. The former is risky and impossible to automate or reproduce; the latter is susceptible to unrealistic modeling.

Regression testing. While valuable for measuring overall performance, TCP regression testing with netperf, application load tests [16], or production workloads can fail to reveal significant functional bugs in congestion control, loss recovery, flow control, security, DoS hardening, and protocol state machines. Such approaches suffer from noise due to variations in site/network conditions or content, and a lack of precision and isolation; thus bugs in these areas can go unnoticed (e.g. the bugs discussed in Section 4.2 were only discovered with packetdrill tests).

Troubleshooting. Reproducing TCP bugs is often challenging, and can require developers to instrument a production kernel to collect clues and identify the culprit. But production changes risk regressions, and it can take many iterations to resolve the issue. Thus we need a tool to replay traces to reproduce problems on non-production machines.

To meet these challenges, we built packetdrill, a tool that enables developers to easily write precise, reproducible, automated test scripts for entire TCP/UDP/IP network stacks. We find that it meets our design goals:

Convenient. Developers can quickly learn the syntax of packetdrill and need not understand the internals of protocols or packetdrill itself. The syntax also makes it easy for the script writer to translate packet traces into test scripts. The tool runs in real time so tests often complete in under one second, enabling quick iteration.

Realistic. packetdrill works with packets and system calls, testing precise sequences of real events. packetdrill tests the exact kernel image used in production, running in real time on a physical machine. It

can run with real drivers and a physical network interface card (NIC), wire, and switch, or a TUN virtual NIC. It does not rely on virtual machines, user-mode Linux, emulated networks, or approximate models of TCP.

Reproducible. `packetdrill` can reliably reproduce test script timing with less than one spurious failure per 2500 test runs (see Section 4.4).

General. `packetdrill` allows a script to run in IPv4, IPv6, or IPv4-mapped IPv6 mode without modification. It runs on Linux, FreeBSD, OpenBSD, and NetBSD, and is portable across POSIX-compliant operating systems that support the `libpcap` packet capture/injection library. Since it is open source, it can be extended by protocol implementors to work with new algorithms, features, and packet formats, including TCP options.

We find `packetdrill` useful in feature development, regression testing, and production troubleshooting. During feature development, we use it to unit test implementations, thereby enabling test-driven development—we have found it vital for incrementally testing complex new TCP features on both the server and client side during development. Then we use it for easy regression testing. Finally, once code is in production, we use it to isolate and reproduce bugs. Throughout the process, `packetdrill` provides a succinct but precise language for discussing TCP scenarios in bug reports and email discussions.

In the rest of the paper, we discuss the design and implementation of `packetdrill`, our experiences using it, and related work.

2 Design

2.1 Scripting Language

`packetdrill` is entirely script-driven, to ease interactive use. `packetdrill` scripts use a language we designed to closely mirror two syntaxes familiar to networking engineers: `tcpdump` and `strace`. The language has four types of statements:

- Packets, using a `tcpdump`-like syntax, including TCP, UDP, and ICMP packets, and common TCP options: SACK, Timestamp, MSS, window scale, and Fast Open.
- System calls, using an `strace`-like syntax.
- Shell commands enclosed in `` `` `` backticks, which allow system configuration or assertions about network stack state using commands like `ss`.
- Python scripts enclosed in `%{}%` braces, which enable output or assertions about the `tcp_info` state that Linux and FreeBSD expose for TCP sockets.

2.2 Execution Model

`packetdrill` parses an entire test script, and then executes each timestamped line in real time—at the pace described by the timestamps—to replay and verify the scenario. For each system call line, `packetdrill` executes the system call and verifies that it returns the expected result. For each command line, `packetdrill` executes the shell command. For each incoming packet (denoted by a leading `<` on the line), `packetdrill` constructs a packet and injects it into the kernel. For each outgoing packet (denoted by a leading `>` on the line), `packetdrill` sniffs the next outgoing packet and verifies that the packet's timing and contents match the script.

Consider the example script in Figure 1, which shows a `packetdrill` script that tests TCP fast retransmit. This test passes as-is on Linux, FreeBSD, OpenBSD, and NetBSD, using a real NIC. As is typical, this script starts by setting up a socket (lines 1–4) and establishing a connection (lines 5–8). After writing data to a socket (line 9), the script expects the network stack under test to send a data packet (line 10) and then directs `packetdrill` to inject an acknowledgement (ACK) packet (line 11) that the stack will process. The script ultimately verifies that a fast retransmit occurs after three duplicate acknowledgements arrive.

2.3 Local and Remote Testing

`packetdrill` enables two modes of testing: *local* mode, using a TUN virtual network device, or in *remote* mode, using a physical NIC. In local mode, `packetdrill` uses a single machine and a TUN virtual network device as a source and sink for packets. This tests the system call, sockets, TCP, and IP layers, and is easier to use since there is less timing variation, and users need not coordinate access to multiple machines. In remote mode, users run two `packetdrill` processes, one of which is on a remote machine and speaks to the system under test over a LAN. This approach tests the full networking system: system calls, sockets, TCP, IP, software and hardware offload mechanisms, the NIC driver, NIC hardware, wire, and switch. However, due to the inherent variability in the many components under test, remote mode can result in larger timing variations, which can cause spurious test failures.

3 Implementation

`packetdrill` is a user-level application written entirely in C, adhering to Linux kernel code style to ease use in kernel testing environments. In this section we delve into the implementation of the tool.

```
0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3           // Create a socket.
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0  // Avoid binding issues.
+0  bind(3, ..., ...) = 0                               // Bind the socket.
+0  listen(3, 1) = 0                                    // Start listening.

+0  < S 0:0(0) win 32792 <mss 1000,sackOK,nop,nop,nop,wscale 7>   // Inject a SYN.
+0  > S. 0:0(0) ack 1 <...>                                       // Expect a SYN/ACK.
+.1 < . 1:1(0) ack 1 win 257                                      // Inject an ACK.
+0  accept(3, ..., ...) = 4                                       // Accept connection.

+0  write(4, ..., 1000) = 1000   // Write 1 MSS of data.
+0  > P. 1:1001(1000) ack 1      // Expect it to be sent immediately.
+.1 < . 1:1(0) ack 1001 win 257  // Inject an ACK after 100ms.

+0  write(4, ..., 4000) = 4000   // Write 4 MSS of data.
+0  > .  1001:2001(1000) ack 1   // Expect immediate transmission.
+0  > .  2001:3001(1000) ack 1
+0  > .  3001:4001(1000) ack 1
+0  > P. 4001:5001(1000) ack 1

+.1 < . 1:1(0) ack 1001 win 257 <sack 2001:3001,nop,nop>  // Inject 3 ACKs with SACKs.
+0  < . 1:1(0) ack 1001 win 257 <sack 2001:4001,nop,nop>
+0  < . 1:1(0) ack 1001 win 257 <sack 2001:5001,nop,nop>

+0  > . 1001:2001(1000) ack 1    // Expect a fast retransmit.
+.1 < . 1:1(0) ack 6001 win 257  // Inject an ACK for all data.
```

Figure 1: A `packetdrill` script for TCP fast retransmit. Scripts use `...` to omit irrelevancies.

3.1 Components

3.1.1 Lexer and Parser

For generality and extensibility, we use `flex` and `bison` to generate `packetdrill`'s lexer and parser, respectively. The structure of the script language is fairly simple, and includes C/C++ style comments.

3.1.2 Interpreter

The `packetdrill` interpreter has one thread for the main flow of events and another for executing any system calls that the script expects to block (e.g. `poll()`).[1]

Packet events. For convenience, scripts use an abstracted notation for packets. Internally, `packetdrill` models aspects of TCP and UDP behavior; to do this, it maintains mappings to translate between the values in the script and those in the live packet. The translation includes IP, UDP, and TCP header fields, including TCP options such as SACK and timestamps. Thus we track each socket and its IP addresses, port numbers, TCP sequence numbers, and TCP timestamps.

For outbound packet events we start sniffing immediately, in order to detect any packets that go out earlier than the script specifies. When we sniff an outbound live packet we find the socket that sent it, and verify that the packet was sent at the expected time. Then we translate the live packet to its script equivalent and verify that the bits the kernel sent match what the script expected.

For inbound packet events we pause until the specified time, then translate the script values to their live equivalents so the network stack under test can process them, and then inject the packet into the kernel.

To capture outgoing packets we use a packet socket (on Linux) or `libpcap` (on BSD-derived OSes). To inject packets locally we use a TUN device. To inject packets over the physical network in remote mode we use `libpcap`. To consume test packets in local mode we use a TUN device; remotely, packets go over the physical network and the remote kernel drops them, since it has no interface with the test's remote IP address.

In `packetdrill` scripts, several aspects of outgoing TCP packets are optional. This simplifies tests, allows them to focus on a single area of behavior, eases maintenance, and facilitates cross-platform testing by avoiding test failures due to irrelevant differences in protocol stack behavior over time or between different OSes. For example, scripts may omit the TCP receive window, or use a `<...>` notation for TCP options. If specified, they are checked; otherwise they are ignored. For example, the `<...>` on the SYN/ACK packet in Figure 1 ignores the only difference between the four OSes in this test.

System calls. For non-blocking system call events, we invoke the system call directly in the main thread. For

[1] Currently, for simplicity of both understanding and implementation, we support only one blocking system call at a time.

blocking calls, we enqueue the event on an event queue and signal the system call thread. The main thread then waits for the system call thread to block or finish the call.

When executing system calls we evaluate script symbolic expressions and translate to live equivalents to get inputs for the call. Then we invoke the system call; when it returns we verify that the actual output, including `errno`, matches the script's expected output.

Shell commands. `packetdrill` executes command strings with `system()`.

Python scripts. `packetdrill` runs Python snippets by recording the socket's `tcp_info` struct at the time of the script event, and then emitting Python code to export the data, followed by the Python snippet itself, for the Python interpreter to run after test execution completes.

3.2 Handling Variation

3.2.1 Network protocol features

`packetdrill` supports a wide array of protocol features. Developers can use the same script unmodified across IPv4, IPv6, and IPv4-mapped IPv6 modes by using command line flags to select the address mode and MTU size. Beyond IPv4, IPv6, TCP, and UDP, we support ECN and inbound ICMP (for path MTU discovery). It would be straightforward to add support for other IP-based protocols, such as DCCP or SCTP.

3.2.2 Machine configuration

We have found that most scripts share machine settings, and thus most scripts start by invoking a default shell script to configure the machine. Also, since script system calls do not specify aspects of the test machine's configuration, the interpreter substitutes these values in during test execution. For example, we select a default IP address that will be used for `bind` system calls based upon the choice of IPv4, IPv6, or IPv4-mapped IPv6.

3.2.3 Timing Models

Since many protocols are very sensitive to timing, we added support for significant timing flexibility in scripts. Each statement has a timestamp, enforced by `packetdrill`: if an event does not occur at the specified time, `packetdrill` flags an error and reports the actual time. Table 1 shows the `packetdrill` timing models.

3.2.4 Avoiding Spurious Failures

For over a year, we have used a `--tolerance_usecs` value of 4 ms, so a test will pass as long as events happen within 4 ms of the expected time. This allows the most common variation: a 1-ms deviation in RTT leads to a 3-ms deviation in retransmission timeout (RTO), initialized to $3 \cdot RTT$ per RFC 6298. We have found this to be a practical trade-off between precision and maintenance overhead, catching most significant timing bugs while usually allowing a full run of all `packetdrill` scenarios without a single spurious failure.

`packetdrill` also takes steps internally to reduce timing variation and spurious failures, including aligning the start of test execution at a fixed phase offset relative to the kernel scheduler tick, leveraging sleep wake-up events to obtain fresh tick values on "tick-less" Linux kernels lacking a regular scheduler tick, using a real-time scheduling priority, using `mlockall()` to attempt to pin its memory pages into RAM, precomputing data where possible, and automatically sending a TCP RST segment to all test connections at the end of a test to avoid interference from retransmissions.

4 Experiences and results

For over 18 months we have used `packetdrill` to test the Linux kernel used on Google production machines. Next we discuss how we've found it useful.

4.1 Features developed with `packetdrill`

Our team has used `packetdrill` to test the features that we have implemented in Linux and have published. We avoided pushing into production numerous bugs by using `packetdrill` during development to test TCP Early Retransmit [14], TCP Fast Open [30], TCP Loss Probe [20], and a complete rewrite of the Linux F-RTO implementation [15]; we also used it to test forward error correction for TCP [24]. The TCP features we developed before `packetdrill`, and thus for which we wrote `packetdrill` tests afterward, include increasing TCP's initial congestion window to ten packets [22], reducing TCP's initial retransmission timeout to 1 second [17], and Proportional Rate Reduction [21].

4.2 Linux bugs found with `packetdrill`

In the process of writing tests for the Linux TCP stack, our team found and fixed 10 bugs in the official version of Linux maintained by Linus Torvalds.

DSACK undo. Linux TCP can use duplicate selective acknowledgements, or DSACKs, to undo congestion window reductions. There was a bug where DSACKs were ignored if there were no outstanding unacknowledged packets at the time the sender receives the DSACK—this is actually the most common case [4]. Also, Linux was not allowing DSACK-based undo in some cases where ACK reordering occurred [5].

CUBIC and BIC RTO undo. CUBIC, the default TCP congestion control module for Linux, and the related BIC module had bugs preventing them from undoing a congestion window reduction that resulted from an RTO [6, 7]; RTOs are the most common form of loss recovery in web sites with short flows [21].

Model	Syntax	Description
Absolute	`0.750`	Specifies the specific time at which an event should occur.
Relative	`+0.2`	Specifies the interval after the last event at which an event should occur.
Wildcard	`*`	Allows an event to occur at any time.
Range	`0.750~0.9`	Requires the given event to occur within the time range.
Loose	`--tolerance_usecs=800`	Allows all events to happen within a range (from the command line).
Blocking	`0.750...0.9`	Specifies a blocking system call that starts/returns at the given times.

Table 1: Timing models supported in `packetdrill`.

TCP Fast Open server. We used `packetdrill` to find and fix several minor bugs in the TCP Fast Open server code: the RTT sample taken using the server's TCP SYN/ACK packet was incorrect [8, 9], TFO servers failed to process the TCP timestamp value on the incoming receiver ACK that completed the three-way handshake [10], and TFO servers failed to count retransmits that happened during the three-way handshake [11].

Receiver RTT estimation. We found and fixed a bug in which receiver-side RTT estimates were broken due to a path in which the code was directly comparing a raw RTT sample with one that had already been shifted into a fixed point representation [12].

Scheduler jiffies update. Jitter in `packetdrill` test RTT estimates hinted at a Linux kernel code path in which tick-less jiffies could be stale. Our audit of the jiffies code revealed such a bug, which we fixed [13].

4.3 Catching external behavior changes

`packetdrill` scripts brought to our team's attention external Linux kernel changes that were not bugs, but still had significant impacts in our environment, including timer slack [32] and recent fixes in packet size accounting [23]. For these changes we ended up adjusting our production kernel's behavior.

4.4 Test Suite

Coverage. Our team of nine developers has written 266 `packetdrill` scripts to test the Google production Linux kernel and 92 scripts to test `packetdrill` itself. Because `packetdrill` enables developers to run a given test script in IPv4, IPv6, or IPv4-mapped IPv6 modes, the number of total test case scenarios is even greater: 657. Table 2 summarizes the areas of TCP functionality covered by our `packetdrill` scripts.

Reproducibility. To quantify the reproducibility of our test results, we examined the spurious failure rate for two days of recent test runs on a 2.2GHz 64-bit multiprocessor PC running a recent Google production Linux kernel. We examined the most recent 54 complete runs of all 657 `packetdrill` test cases relevant for that release of the kernel, and found 14 test case failures, all of which were spurious. This implies an overall spurious test case failure rate of just under 0.0004, or 1 in 2500. Since fewer than a quarter of full test runs suffer from spurious failures, we find this to be an acceptable overhead on our kernel team. However, we continue to refine scripts to further reduce the spurious failure rate.

Feature	Description	Tests
Socket API	listen, connect, write, close, etc.	11
RFC 793	Core functionality	21
RFC 1122	Keep-alive	4
RFC 1191	Path MTU discovery	4
RFC 1323	Timestamps	1
RFC 2018	SACK (Selective Acknowledgement)	12
RFC 3168	Explicit Congestion Notification	3
RFC 3708	DSACK-based undo	10
RFC 5681	Congestion control	10
RFC 5827	Early retransmit	11
RFC 5682	F-RTO (Forward RTO-Recovery)	14
RFC 6298	Retransmission timer	13
RFC 6928	Initial congestion window	5
RFC 6937	Proportional rate reduction	10
IETF draft	Fast open	44
IETF draft	Loss probe	9
IETF draft	CUBIC congestion control	1
n/a	TSO (TCP segmentation offload)	3
n/a	Receive buffer autotuning	2
n/a	Linux `inet_diag` sockets	3
n/a	Miscellaneous	75
	Total test scripts	266

Table 2: Areas of TCP tested by `packetdrill` scripts.

Execution Time. `packetdrill` scripts execute quickly, so we run all `packetdrill` scripts before sending for review any commit that modifies the Google production TCP code. For the 54 test runs mentioned above, the total time to execute all 657 test cases was 25–26 minutes in all 54 test runs, an average of 2.4 seconds per test case.

5 Related Work

There are many tools to debug and test protocol implementations. RFC2398 [28] categorizes late-90s tools. The Packet Shell [27] seems to be the closest to `packetdrill` in design. It allowed scripts to send and receive packets to test a TCP peer's responses, but it was developed specifically for Solaris, is no longer available publicly, was more labor-intensive (e.g. it took 8

lines of `Tcl` commands to inject a single TCP SYN), and had no support for the sockets API, specifying packet arrival times, or handling timers. Orchestra [18] is a fault-injection library to check the conformance of TCP implementations to basic TCP RFCs. It places a layer below the X-kernel TCP stack and executes user-specified actions to delay, drop, reorder, and modify packets. Results require manual inspection, and the tests are not automated to check newer TCP stacks. While not developed for testing, TCPanaly [29] analyzes TCP traces to identify TCP implementations and diagnose RFC violations or performance issues. In `packetdrill` such domain knowledge is constructed through scripts; in TCPanaly its built directly into the software itself, which is harder to revise and expand.

The tools above were developed in the late 1990s, and to our knowledge none of them is being actively used to test modern TCP stacks. By contrast, IxANVL [1] is a modern commercial protocol conformance testing tool that covers core TCP RFCs and a few other networking protocols, but unlike `packetdrill` it can not be easily extended or scripted to troubleshoot bugs or test new changes, and is not open source.

Other research efforts test protocols by manually writing a model in a formal language, and then using automated tools to check for bugs [2, 3, 26, 31]. While these models are rigorous, their high maintenance cost is unsustainable, since they diverge from the rapidly-evolving code they try to model. Other tools automatically find bugs, but only within narrow classes, or in user-level code [25]. These approaches are complementary to ours.

6 Conclusion

`packetdrill` enables quick, precise, reproducible scripts for testing entire TCP/UDP/IP network stacks. We find `packetdrill` indispensable in verifying protocol correctness, performance, and security during development, regression testing, and troubleshooting. We have released `packetdrill` as open source in the hope that sharing it with the community will make the process of improving Internet protocols an easier one.

The source code and test scripts for `packetdrill` are available at: http://code.google.com/p/packetdrill/.

Acknowledgements

We would like to thank Bill Sommerfeld, Mahesh Bandewar, Chema Gonzalez, Laurent Chavey, Willem de Bruijn, Eric Dumazet, Abhijit Vaidya, Cosmos Nicolaou, and Michael F. Nowlan for their help and feedback.

References

[1] IxANVL. http://goo.gl/SV6ia.

[2] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proc. of SIGCOMM* (2005), ACM.

[3] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proc. of ACM POPL* (2006), ACM.

[4] CARDWELL, N. *Linux commit 5628adf*. http://goo.gl/Fnj46.

[5] CARDWELL, N. *Linux commit e95ae2f*. http://goo.gl/uyRUp.

[6] CARDWELL, N. *Linux commit fc16dcd*. http://goo.gl/xv6xB.

[7] CARDWELL, N. *Linux commit 5a45f00*. http://goo.gl/cHYiw.

[8] CARDWELL, N. *Linux commit 0725398*. http://goo.gl/jWu0S.

[9] CARDWELL, N. *Linux commit 016818d*. http://goo.gl/axR97.

[10] CARDWELL, N. *Linux commit e69bebd*. http://goo.gl/Rh2J1.

[11] CARDWELL, N. *Linux commit 30099b2*. http://goo.gl/BKZWH.

[12] CARDWELL, N. *Linux commit 18a223e*. http://goo.gl/BL4O5.

[13] CARDWELL, N. *Linux commit 6f10392*. http://goo.gl/IFQ4D.

[14] CHENG, Y. *Linux commit eed530b*. http://goo.gl/MPmF0.

[15] CHENG, Y. *Linux commit e33099f*. http://goo.gl/hhIfU.

[16] CHENG, Y., HÖLZLE, U., CARDWELL, N., SAVAGE, S., AND VOELKER, G. Monkey see, monkey do: A tool for TCP tracing and replaying. In *Proc. of USENIX ATC* (2004).

[17] CHU, J. *Linux commit 9ad7c04*. http://goo.gl/gxiFT.

[18] DAWSON, S., JAHANIAN, F., AND MITTON, T. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience 27*, 12 (1997), 1385–1410.

[19] DUKE, M., BRADEN, R., EDDY, W., AND BLANTON, E. A Roadmap for Transmission Control Protocol (TCP) Specification Documents, September 2006. RFC 4614.

[20] DUKKIPATI, N., CARDWELL, N., CHENG, Y., AND MATHIS, M. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses, Feb. 2013. IETF Draft, draft-dukkipati-tcpm-tcp-loss-probe-01.

[21] DUKKIPATI, N., MATHIS, M., CHENG, Y., AND GHOBADI, M. Proportional rate reduction for TCP. In *Proc. of IMC* (2011).

[22] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An Argument for Increasing TCP's Initial Congestion Window. *ACM Comput. Commun. Rev. 40* (2010).

[23] DUMAZET, E. *Linux commit 87fb4b7*. http://goo.gl/MgRWi.

[24] FLACH, T., DUKKIPATI, N., TERZIS, A., RAGHAVAN, B., CARDWELL, N., CHENG, Y., JAIN, A., HAO, S., KATZ-BASSETT, E., AND GOVINDAN, R. Reducing Web Latency: the Virtue of Gentle Aggression. In *SIGCOMM* (2013).

[25] KOTHARI, N., MAHAJAN, R., MILLSTEIN, T. D., GOVINDAN, R., AND MUSUVATHI, M. Finding protocol manipulation attacks. In *SIGCOMM* (2011), pp. 26–37.

[26] MUSUVATHI, M., ENGLER, D., ET AL. Model checking large network protocol implementations. In *Proc. of NSDI* (2004).

[27] PARKER, S., AND SCHMECHEL, C. The packet shell protocol testing tool. http://goo.gl/CS4kf.

[28] PARKER, S., AND SCHMECHEL, C. RFC2398: Some testing tools for TCP implementors, August 1998.

[29] PAXSON, V. Automated packet trace analysis of TCP implementations. In *Proc. of ACM SIGCOMM* (1997), ACM.

[30] RADHAKRISHNAN, S., CHENG, Y., CHU, J., JAIN, A., AND RAGHAVAN, B. TCP Fast Open. In *Proc. of CoNEXT* (2011).

[31] SMITH, M., AND RAMAKRISHNAN, K. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM ToN 10*, 2 (2002).

[32] VAN DE VEN, A. *Linux commit 3bbb9ec*. http://goo.gl/w18r6.

DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments

Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić[†], and Ricardo Bianchini[‡]
EPFL, Switzerland [†]*Institute IMDEA Networks, Spain* [‡]*Rutgers University, USA*

Abstract

We describe the design and implementation of DeepDive, a system for transparently identifying and managing performance interference between virtual machines (VMs) co-located on the same physical machine in Infrastructure-as-a-Service cloud environments. DeepDive successfully addresses several important challenges, including the lack of performance information from applications, and the large overhead of detailed interference analysis. We first show that it is possible to use easily-obtainable, low-level metrics to clearly discern when interference is occurring and what resource is causing it. Next, using realistic workloads, we show that DeepDive quickly learns about interference across co-located VMs. Finally, we show DeepDive's ability to deal efficiently with interference when it is detected, by using a low-overhead approach to identifying a VM placement that alleviates interference.

1 Introduction

Many enterprises and individuals have been offloading their workloads to Infrastructure-as-a-Service (IaaS) providers, such as Amazon and Rackspace. A key enabling factor in the expansion of cloud computing is virtualization technology. IaaS providers use virtualization to (1) package each customer's application into one or more virtual machines (VMs), (2) isolate misbehaving applications, (3) lower operating costs by multiplexing their physical machines (PMs) across many VMs, and (4) simplify VM placement and migration across PMs.

Despite the benefits of virtualization, including its ability to slice a PM well in terms of CPU and memory space allocation, performance isolation is far from perfect in these environments. Specifically, a challenging problem for providers is identifying (and managing) *performance interference* between the VMs that are co-located at each PM. For example, two VMs may thrash in the shared hardware cache when running together, but fit nicely in it when each is running in isolation. As another example, two VMs, each with sequential disk I/O when running in isolation, may produce a random access pattern on a shared disk when running together. To make things worse, technology trends point to manycore PMs with hundreds or even thousands of cores. On these PMs, the chance of experiencing interference will increase.

Interference can severely diminish the trust of customers in the cloud's ability to deliver predictable performance. Thus, interference might become a stumbling block in attracting performance-sensitive customers.

Effectively dealing with interference is challenging for many reasons. First, the IaaS provider is oblivious to its customers' applications and workloads, and it cannot easily determine that interference is occurring. Moreover, the IaaS provider cannot rely on applications to report their performance levels (and therefore know when interference is occurring), because this might overburden application developers who moreover cannot be trusted. This challenge speaks against non-transparent approaches [12, 18, 25, 26, 27, 33, 37]. Second, interference is complex in nature and may be due to any server component (e.g., shared hardware cache, memory, I/O). An effective solution has to account for all components. Further, interference might only manifest when the co-located VMs are concurrently competing for hardware resources. The existing approaches for predicting performance degradation [12, 18, 25, 26, 37] are not applicable, as they require the provider to have access to the co-located VMs for long periods prior to deployment. Interference detection must be a quicker, online activity. Finally, the sheer volume of new VMs deployed daily at a large public provider may cause scalability issues.

Given these challenges, we propose DeepDive, a system for transparently and efficiently identifying and managing interference in IaaS providers. We contribute:

1. A method for transparently obtaining the ground truth about interference, including a black-box detection of application behavior and the ability to pinpoint the culprit resource for interference using only low-level metrics.

2. A *warning system* that reduces the overhead of detailed interference analysis by learning about normal, non-interfering behaviors.

3. A technique for leveraging global information to increase scalability that uses the behavior of VMs running the same workload on other PMs.

4. A mechanism for transparently and cheaply migrating the culprit VM, by using a simple synthetic benchmark to mimic the low-level behavior of a VM and its impact on other VMs before actual migration.

5. Results using realistic workloads that show: i) DeepDive transparently infers performance loss with high accuracy (less than 5% error on average), identifies inter-

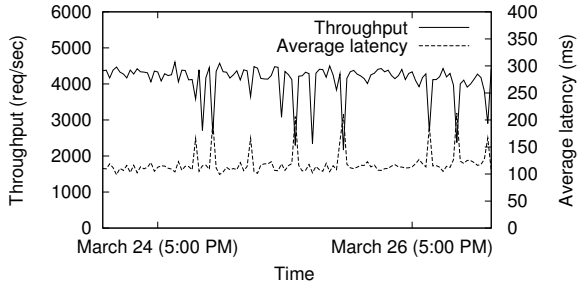


Figure 1: Measured performance of a service running on EC2 under a fixed workload and resource configuration. Performance is periodically affected by co-located VMs.

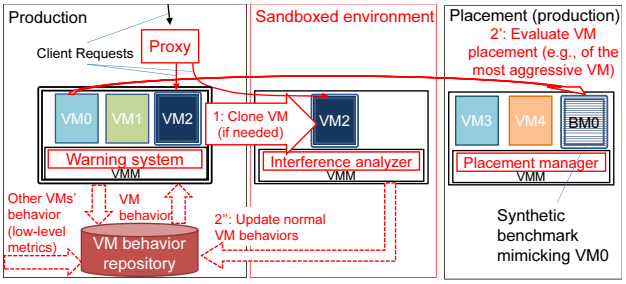


Figure 2: DeepDive overview, showing how it detects and mitigates the effect of interference on VM2.

ference, and pinpoints the culprit resource; ii) it is highly accurate (no false negatives) and has low overhead (few profiling machines); and iii) it makes quick (less than a minute) and accurate VM placement decisions.

To our knowledge, DeepDive is the first end-to-end system that transparently and efficiently handles interference on any major server resource, including I/O. Its deployment would have two key benefits. First, it would enable cloud providers to meet their service-level objectives using fewer resources, which would increase user satisfaction and reduce energy costs. Second, the smarter VM placement would enable cloud customers to purchase fewer resources from the provider.

2 Background and Motivation

Virtualization software chronically lacks effective performance isolation, especially in the context of hardware caches and I/O components. For instance, recent efforts [15] reveal that interference may cause same-type VMs (e.g., those offering the same amount of virtual resources) to exhibit significantly different performance over time. This impact can be seen in our experiment using Cassandra [8] (a key-value store) running on Amazon EC2. We deploy one Cassandra VM and monitor its performance under a fixed workload and resource allocation during a three-day period. As shown in Figure 1, although both the workload and virtual resources remain the same, Cassandra faces many periods of significantly degraded performance. We attribute the performance losses to interference as we tightly control the experiment, except of course for the virtualization platform and the PM, where interference can occur.

Faced with such losses, users might compensate by overprovisioning their VMs [26, 27, 33], which increases their costs. However, overprovisioning is not a panacea, especially for "scale out" applications that dynamically increase the number of running VMs while keeping the instances affected by interference in the active set. As a result, many (potential) customers still find interference as a barrier to migrating their loads to the cloud [6].

3 Approach

DeepDive operates in parallel with applications, seeking to provide application performance that is comparable to, or ideally the same as, that observed in an isolated environment. Figure 2 highlights DeepDive's main components and the way they interact. DeepDive transparently deals with interference by inspecting low-level metrics, including hardware performance counters and readily available hypervisor (VMM) statistics about each VM. To reduce the overhead of interference detection and mitigation, DeepDive introduces two interference analyses that differ in their accuracy and overhead.

DeepDive first relies on a **warning system** running in the VMM to conduct early interference analysis. This analysis is fast, and incurs negligible overhead as we can collect the required statistics without affecting the applications currently running on the PM[1]. DeepDive places these statistics in a multi-dimensional space, where the interference and non-interference cases cluster into easily separable regions.

Figure 3 depicts the decision-making process in the warning system by illustrating the important cases in the multi-dimensional space (shown here only using two dimensions for clarity). One option is for the current measurements to fall within a cluster of acceptable behaviors (Figure 3(a)). If that is not the case but other VMs running this workload are behaving similarly (e.g., due to a change in the client-induced workload), again there is no need to perform further interference analysis (Figure 3(b)). Further investigation is required only if the current measurement is substantially different (i.e., by more than an automatically-determined threshold) from both the existing behaviors as well as other VMs running the same workload (Figure 3(c)).

While the warning system reduces DeepDive's overhead, it is not perfectly accurate and cannot pinpoint the source of interference. DeepDive thus relies on an **interference analyzer** to perform a highly reliable but expensive analysis, when necessary. Only when the warning

[1] We use the terms "PM", "server", and "machine" interchangeably.

Name	Description	Name	Description
cpu_unhalted	Clock cycles when not halted	*resource_stalls*	Cycles during which resource stalls occur
inst_retired	Number of instructions retired	*bus_tran_any*	Number of completed bus transactions
l1d_repl	Cache lines allocated in the L1 data cache	*bus_trans_ifetch*	Number of instruction fetch transactions
l2_ifetch	L2 cacheable instruction fetches	*bus_tran_brd*	Burst read bus transactions
l2_lines_in	Number of allocated lines in L2	*bus_req_out*	Outstanding cacheable data read bus requests duration
mem_load	Retired loads	*br_miss_pred*	Number of mispredicted branches retired
iostat	T_{disk} presents all the idle CPU cycles while the system had an outstanding disk I/O request.		
netstat	T_{net} presents all the idle CPU cycles while the system had a packet in the Snd/Rcv queue.		

Table 1: Low-level metrics used to differentiate normal VM behaviors from interference. The $iostat$ and $netstat$ tools can be used to approximate I/O-related stalls associated with different VMs, using VM introspection tools like XenAccess.

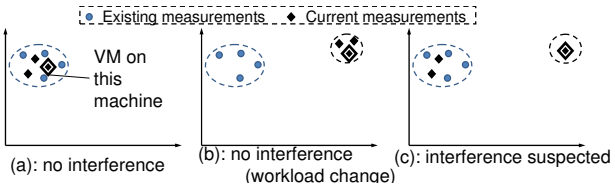


Figure 3: The warning system uses previously collected data and current global measurements, to decide whether DeepDive should further investigate interference.

system suspects that one or more VMs are subjected to interference, DeepDive invokes the analyzer to conduct the exhaustive interference analysis.

The analyzer clones the VM on-demand and executes it in a sandboxed environment. By using a proxy to duplicate client requests, the cloned VM is subjected to the same workload as the VM co-located with other tenants. The analyzer then uses the low-level measurements to estimate the performance of the original and cloned VMs. The estimates should be similar – different by less than an *operator-defined threshold* percentage – in the absence of interference. This VM cloning, workload duplication, and comparison approach has been studied extensively in [33, 36]. The approach provides the ground truth, and enables DeepDive to pinpoint the dominant sources (server components) of interference. The analyzer uses the classic cycles per instruction (CPI) model to transparently identify these sources. Researchers have used this model to detect performance issues other than interference, e.g. [9]. We augment it with system-level metrics that extend the CPI stack to include I/O.

In the absence of interference, the analyzer updates the repository of VM behaviors with this new information. If interference does exist, the analyzer forwards its findings to the **VM-placement manager** to determine a preferable (e.g., minimal) change in VM placement that will eliminate or at least reduce interference. The default behavior is to migrate the most aggressive VM, in terms of its use of the resource that is causing interference.

The VM-placement manager tries to find a PM that will be the best match (e.g., non-interference causing) for the VM at hand. It does so by running a synthetic benchmark that mimics the behavior of the VM for a short time on another PM (with other VMs present), and evaluates whether interference reappears. If it does not, DeepDive can migrate the VM to that PM. If it does, the VM-placement manager tries a different PM.

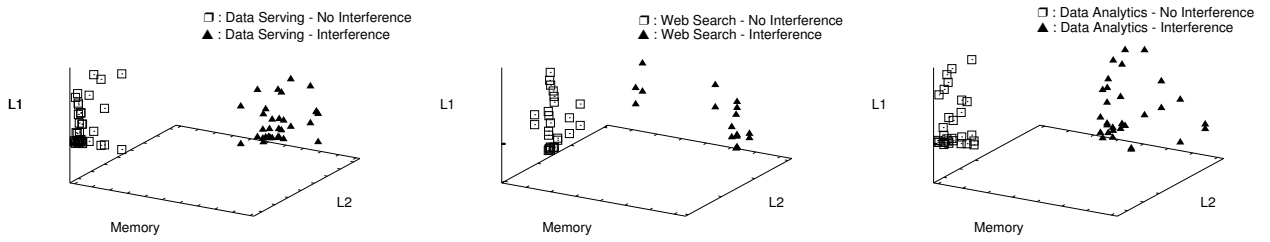
3.1 The warning system

The warning system prevents unnecessary interference analyzer invocations by differentiating workload changes from interference. It does so based on the metrics listed in Table 1, which represent the major PM resources (cores, memory, disk, and network interface), and have been enough for our experiments to date. Vasić *et al.* [33] considered a larger set of metrics, but found it to be overkill. Nevertheless, one can automatically determine whether a metric should be considered; Vasić *et al.* solved a similar feature selection problem [33].

The system uses both local and global information to infer if interference may be happening. It first locally tries to match the current values of the metrics against the previously learned set of normal behaviors. If it cannot find a match, it globally checks whether other VMs running the same code are experiencing similar behavior.

More precisely, when first faced with a VM, the warning system has no information about it and activates the interference analyzer. The analyzer then provides the warning system with: i) a set of normal VM behaviors S that are obtained in isolation, and form the ground truth, and ii) a vector of *metric classification thresholds* M_T used to filter out the workload noise from actual interference. Note that these classification thresholds are different from the operator-defined performance threshold for acceptable performance degradations (Section 3.2), and are set automatically by the clustering algorithm (described below). From this point on, the warning system continuously collects the metrics and tries to retrieve a match from the set of normal VM behaviors, respecting the acceptable metric deviations M_T.

Like any other statistical method, the warning system can only identify performance anomalies (interference)



(a) Data Serving (b) Web Search (c) Data Analytics

Figure 4: Metric values when running under different workload and interference scenarios.

if they are exceptional. Fortunately, our measurements performed on a real-world platform (Figure 1) suggest that anomalies are indeed exceptional in practice. Even if performance anomalies were common for an application, i.e. they cannot be used to detect that the application is undergoing interference, DeepDive would eventually learn so via invocations of the interference analyzer.

To prevent VM load changes unrelated to interference from causing analyzer invocations, we normalize the metrics with respect to the amount of work performed (the number of instructions retired). We find that the metrics' normalized values are persistent across a wide range of load intensities. This finding is critically valuable, since cloud loads frequently fluctuate over time.

Local information. To demonstrate experimentally that the warning system can differentiate normal from interference behaviors, we use typical cloud workloads under different quantitative and qualitative load changes, and interference conditions. Specifically, in Figure 4, we extensively experiment with the Data Serving, Web Search, and Data Analytics workloads from CloudSuite [20]. (More details about these workloads appear in Section 4.) Although we collect the dozen or so metrics listed in Table 1, the figure includes only three of them for clarity. The figure presents normalized metric values relating to the first-level cache (L1), the second-level cache (L2), and main memory. Each point in the graphs depicts a different experimental setting, including various load intensities, and different key and word popularities for Data Serving and Web Search, respectively. In the absence of interference, the data points cluster on one side of the space. Once we inject differently modulated interference effects, the normalized metric values experience significant deviation, which allows the warning system to detect new interference conditions. (We detail the interfering VM in Section 4.1.)

Global information. To further reduce the number of invocations of the analyzer, the warning system leverages the fact that cloud applications regularly execute the same code on many (perhaps dozens or even thousands of) VMs. This enables the warning system to diagnose if the observed deviations come from interference or application behavior changes. If the VMs executing the same code, spread across multiple PMs, observe similar metric value deviations at about the same time, it is highly likely that the application is subjected to workload changes and further interference analysis is not necessary. Furthermore, DeepDive considers several metrics, which further reduces the chance that multiple VMs reporting similar behavior is a consequence of interference.

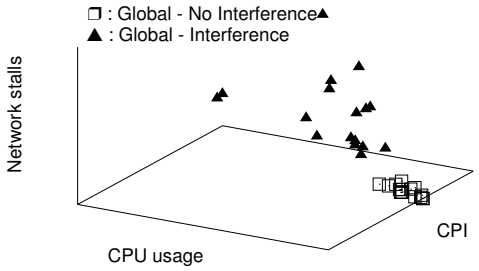


Figure 5: Metric values for Data Analytics. Observing multiple VMs prevents unneeded invocations of the analyzer.

To illustrate the use of global information, we perform a set of experiments with our Data Analytics workload running across nine PMs in our cluster. We inject varying amounts of network interference into the cluster by progressively co-locating more interfering VMs that run a network-intensive benchmark (*iperf*). This scenario stresses the warning system because interference manifests only when the mappers and reducers (from the Hadoop MapReduce-like framework) have to fetch data remotely. Figure 5 plots some of the normalized metrics (relating to network and core utilization) obtained from each of the PM's local warning systems. The metrics corresponding to the PMs where we run the interfering VMs clearly deviate from the remaining VMs' behaviors. The figure hence demonstrates that DeepDive: i) deals with I/O-related interference, and ii) can further minimize the profiling overhead by merely observing the behavior of VMs running the same workload on different PMs.

DeepDive's ability to use global information relies on the assumption that it knows which VMs are running the same application. This is a reasonable assumption, since VMs can be rented in a pre-configured state. Moreover, cloud providers often provide load balancing functional-

ity that tenants explicitly request from the cloud provider for groups of VMs that execute the same code.

False positives and false negatives. False positives occur when the warning system unnecessarily invokes the analyzer under non-interference conditions. For instance, changes in a VM's working set or qualitative workload changes (e.g., the request mix substantially shifts) may lead to substantial statistical variation. Although false positives may sporadically lead to unnecessary analyzer invocations, they are mostly benign and only marginally affect DeepDive's overhead. We have verified this empirically by running extensive experiments under realistic workload conditions.

On the other hand, if the warning system confuses interference with normal workload changes – a false negative – the impact is more severe. Fortunately, our sensitivity analysis demonstrates that the vector of metric thresholds M_T determined by a standard clustering technique (described below) prevents false negatives, while still maintaining high warning system efficiency. Moreover, cloud providers might periodically (e.g., at a frequency driven by VM priority) invoke the analyzer to reduce a potential non-zero false negative rate.

Clearly, the challenge here is to define metric thresholds $\mathbf{M_T}$ that properly separate representative VM behaviors from noise, while also properly identifying interference. If the thresholds are too strict, even minor deviation from prior VM behaviors would cause the warning system to fire. On the other hand, excessively loose thresholds might let interference proceed undetected. We leverage the *expectation-maximization* clustering algorithm [21] to produce interference-free clusters in N-dimensional space, where N is the number of metrics that DeepDive uses. In producing the clusters, the algorithm also defines the metric thresholds. DeepDive improves the clustering by providing a set of constraints [10, 11] along with the collected VM behaviors – when diagnosing a VM's behavior with interference, the analyzer also prevents the algorithm from assigning this behavior to an interference-free cluster. This has a positive effect on the detection rate, as we have verified empirically.

Shortly after a VM's deployment, the metric space is empty or sparsely populated. To create the interference-free clusters, the warning system operates in a conservative mode – every drop in VM performance above the performance threshold causes invocation of the analyzer. This is how DeepDive ensures that no interference goes undetected, and accelerates learning of the interference-detecting metric thresholds.

3.2 The interference analyzer

If the warning system suspects that one or more VMs may be facing interference, it invokes the analyzer to confirm. To do so, the analyzer uses VM cloning, workload duplication, and VM performance comparison. If interference is indeed present, the analyzer also determines which resource is the most likely to be causing the interference (e.g., shared cache, I/O).

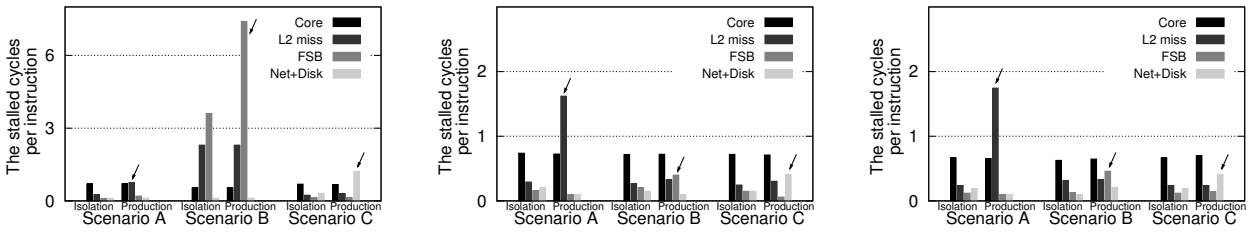
Identifying the ground truth. DeepDive uses the same approach to determine VM performance in the absence of interference as DejaVu [33]. Though we do not claim any novelty in this approach, we summarize it here for completeness. DeepDive clones the VM under test in a sandboxed environment that uses non-work-conserving schedulers to tightly control the resource allocation. The amount of time to complete VM cloning depends on the amount of state in the VM, but is typically small compared to the frequency of invocation of the analyzer. DeepDive relies on a proxy that intercepts the clients' traffic to: 1) duplicate and send copies of the requests to the sandboxed environment, and 2) forward the traffic to/from the production VM to avoid negatively impacting the applications running inside that VM. DeepDive can then compare the metrics in isolation and in production. Others [33, 36] have studied this approach and its challenges (including how to tackle non-determinism) extensively, so we do not repeat this study here.

Performance analysis. Given the statistics from the production and sandboxed environments, DeepDive uses the analyzer's performance model to transparently estimate the performance degradation that a VM is experiencing due to interference. Given this model, DeepDive can opt for VM migration if the degradation is substantial, or refrain from any action otherwise.

Since we do not expect the VMs to assess and communicate their performance levels, the key question here is knowing when the VM's performance is degraded by simply looking at low-level metrics. The analyzer contrasts the *instructions retired* rate in production with that in isolation (in the sandbox) to approximate how much the shared resources contribute to the overall degradation: $Degradation = Inst^{production}/Inst^{isolation}$.

Once the analyzer estimates the degradation, it may proceed in one of two ways. If the degradation is below the operator-defined performance threshold, the analyzer notifies the warning system about the false alarm. This extends the warning system's set of acceptable VM behaviors with the new metrics' values. If the degradation exceeds the threshold, the analyzer forwards the results of its analysis to the VM placement manager, which may migrate the VM to a more appropriate PM.

Importantly, [7, 19] have shown that the number of instructions retired is not always a reliable performance metric in multithreaded applications, since spin-based synchronization may cause timing and thread interleaving variations. This is not a serious problem for DeepDive for two reasons. First, the computed degradation need not be accurate with respect to absolute per-



(a) Data Serving (b) Web Search (c) Data Analytics

Figure 6: Breakdown of stalled cycles in production and isolation. Our analysis reveals the sources of interference.

formance; rather, it simply needs to properly identify anomalies. Second, if these inaccuracies become a problem in practice, we can leverage prior efforts that exclude spinning instructions, or augment the measurements to account only for the useful computation [19]. Multithreading has not been a problem for us so far.

Identifying dominant sources of interference. If the amount of performance loss requires invocation of the VM-placement manager, the analyzer pinpoints the resources that are likely the culprits using CPI analysis augmented with system-level metrics (to capture I/O). The augmented CPI "stack" captures the amount of work the VM is doing, while identifying where it is spending time. Intuitively, interference causes the VM to suffer more stall cycles, and perform less useful work.

Our root cause analysis hence estimates a breakdown of the various run-time stall components of the server:

$$T_{overall} = \underbrace{T_{core} + T_{off\_core}}_{\text{CPI analysis using hardware counters}} + \underbrace{+T_{disk} + T_{net}}_{\text{using system-level statistics}}$$

where T_{core} represents the time running instructions on the core (and hitting in private caches), T_{off_core} represents the stalled cycles due to memory accesses (including shared caches), T_{disk} represents the time waiting for disk, and T_{net} represents network-related stalls. We infer these values from the metrics in Table 1. The metrics are clearly architecture-dependent, but sufficiently generic for DeepDive not to be tied to any particular architecture, as shown in our longer technical report [28].

We estimate the resources' individual contributions to the performance degradation via the discrepancies in the metrics obtained in isolation and production:

$$Factor_{resource} = \frac{T_{resource}^{production} - T_{resource}^{isolation}}{T_{overall}^{production}}$$

To validate this performance model, we run a set of experiments with the Data Serving, Web Search, and Data Analytics workloads. Figure 6 contrasts the various resource stalls in the production environment (which is undergoing interference) and in isolation (in the sandbox). Each experiment carefully tunes the interference, so as to move it from the last level cache (Scenario A) to the front side bus (Scenario B) to the I/O subsystem (Scenario C). We then invoke the analyzer to estimate the amount of performance loss, and identify the resources that primarily contribute to it. We mark the resources identified by the analyzer with arrows in the figure. We observe that the analyzer correctly identifies the culprit resources as their growing (degrading) factors clearly dominate over the remaining resources.

3.3 The VM-placement manager

If the analyzer detects interference on a PM, DeepDive runs the VM-placement manager to determine a new VM placement. The manager can implement multiple policies for selecting which VM to migrate: it may select the VM that is suffering the most from interference, or it may select the VM using the culprit resource most aggressively. Although we view the placement policy as orthogonal to this work, we design a simple policy to evaluate our placement manager. Upon identifying a resource that is the source of interference, the placement manager selects the VM that is most aggressive in using the resource, and then migrates it if an appropriate destination PM exists. To ensure better performance isolation, DeepDive repeats this process until the interference is sufficiently reduced, or ideally eliminated altogether.

The remaining challenge is ensuring that a VM migration will not cause even worse interference on the destination PM. A naive placement manager might speculatively migrate the selected VMs in the hope that this will not cause further interference on the destination PMs. However, this could result in numerous and expensive VM migrations (especially for applications with large memory and/or persistent state), as well as prolonged periods of severe performance degradation. DeepDive therefore anticipates the resulting interference conditions on the destination PM prior to actual VM migration.

Toward this end, DeepDive uses a novel synthetic benchmark that can mimic the behavior of an arbitrary VM. The key goal is that an actual VM and its synthetic counterpart should exhibit similar interference characteristics, when co-located with other VMs running on a PM. The benchmark models the working set size, data

locality, instruction mix, level of parallelism, and disk and network throughput of a VM. In more detail, it is a collection of loops that exercise the different PM resources to match the metric values collected from an actual VM. The resources can be exercised locally to a PM, except for the network interface. For this resource, the benchmark spawns a thread that acts as a communication partner for a benchmark running on another PM. The loops execute numbers of iterations given as inputs to the benchmark. Thus, creating the benchmark involved learning the set of input values that best approximates any set of metric values. We used a standard regression algorithm for this training. Though the training phase may take a long time (a few days in our experiments), this training is done offline and only once for each server type. Choosing a particular configuration, after the training phase, takes only a few seconds. Although, one can use existing, more sophisticated workload synthesizers; we find this extra sophistication unnecessary.

The placement manager uses the benchmark to evaluate potential migrations. Specifically, given a set of metric values to reproduce, it runs the benchmark (with the proper learned inputs) in a VM on all candidate PMs concurrently. The runs take less than a minute in our experiments. With metric data collected from these runs, the manager picks the best destination PM for the migration.

3.4 Discussion

Can DeepDive tackle interference due to an oversubscribed network? Currently, DeepDive can tackle interference at the network interface, but requires a well-provisioned connection to the sandbox to determine the impact of network oversubscription. This is not a major constraint, since the number of PMs required for the sandbox is small, as we demonstrate in the next section.

Can DeepDive deal with non-determinism? DeepDive can tolerate deviations coming from different sources, such as OS-level non-determinism (e.g., periodic flushing of dirty pages). DeepDive views such non-deterministic events as noise, as they are typically too short and infrequent. Nevertheless, if they are persistent across multiple monitoring epochs, DeepDive is able to recognize this and label the behavior as normal.

Can DeepDive deal with oscillating interference conditions? While we have not focused on possible interference oscillations in this work, interference might vary over time. This would require us to repeat the interference analysis to ensure better guarantees on interference detection. In fact, we could install a simple controller that would react only upon detections that are persistent across multiple epochs.

Can DeepDive deal with heterogeneity? Our experience so far has been with homogeneous PMs. This is reasonable since cloud providers typically use disjoint sets of homogeneous PMs for simpler management. Nevertheless, DeepDive can deal with heterogeneity by grouping the low-level metrics by PM type, performing the CPI analysis according to PM type, and training a synthetic benchmark for each PM type.

Can DeepDive degrade performance while evaluating a placement scenario? We run our benchmark only for tens of seconds until we collect the necessary metrics. We think that this is acceptable compared to the impact of a full migration. Furthermore, the cloud operator can prioritize and explicitly avoid certain PMs.

Can DeepDive deal with false negatives? One might be able to design an adversarial workload that would resemble interference conditions. Section 3.1 discusses how DeepDive tackles false negatives.

Can DeepDive be ported to different architectures? One of the authors ported DeepDive to a NUMA (non-uniform memory access) server with two quad-core Core i7-based processors. The port took just a few days to complete – we provide more details in our report [28].

4 Evaluation

4.1 Experimental infrastructure

Servers and clients. We run our production and sandboxed environments on up to 10 servers with Intel Xeon X5472 processors. The servers have eight 3-GHz cores, with 12 MB of L2 cache shared across each pair of cores. The servers also feature 8 GB of DRAM, two 250-GB 7200rpm disks, and one 1-Gb network port.

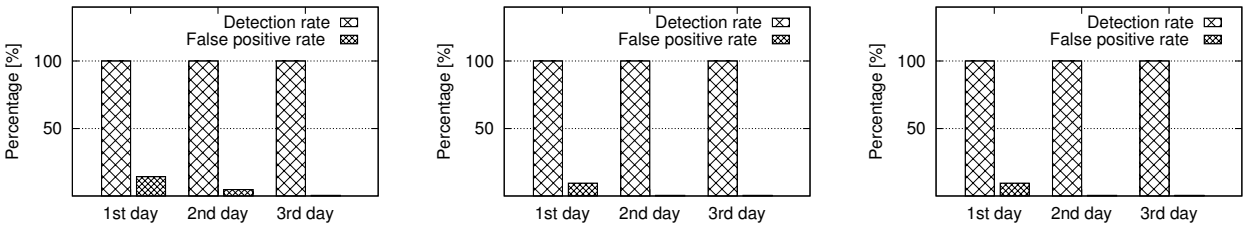
The servers run the Xen VMM. We configure the VMs to run on virtual CPUs that are pinned to separate cores (we assign two cores per VM). We allocate enough memory for each VM to avoid swapping to disk.

The clients run on a separate machine with four 12-core AMD Opteron 6234 processors running at 2.4 GHz, 132 GB of DRAM, and two 1-Gb network ports.

Cloud workloads. We use diverse, representative cloud workloads from CloudSuite [20]. Our *Data Serving* workload consists of one instance of Cassandra [8]. To experiment with different loads, we instrument clients from the Yahoo! Cloud Service Benchmark [14] to vary both the key popularities and the read/write ratio.

Our *Web Search* workload involves a single index serving node (available from the Nutch open-source project [2]) that holds a 2GB index. To experiment with different loads, we instrument the Faban client emulator [3] to vary word popularities and the number of client sessions (driven by the traces described below).

Our *Data Analytics* workload uses Hadoop [4] to run a modified Bayes classification example from the Mahout package [1] across 35 GB of Wikipedia data. The cluster consists of nine VMs configured with 2 GB of memory



(a) Data Serving (b) Web Search (c) Data Analytics

Figure 7: Detection and false positive rates while replaying the HotMail traces. DeepDive always detected the injected interference. The false positive rate quickly decreases as DeepDive learns more about normal behaviors.

and two dedicated cores, and the master which is provisioned with 8GB of RAM and four cores.

Real-world traces. To evaluate DeepDive under dynamic workloads, we use real load intensity traces to drive the execution of our cloud workloads. Specifically, we use traces from Microsoft's HotMail from September, 2009. The traces represent the aggregated load across thousands of servers, averaged over 1-hour periods. We ensure that the maximum number of active client sessions is within the servers' maximum capabilities.

In addition to load traces, we injected interference conditions mimicking a real cloud platform. Specifically, we rented four Amazon EC2 instances and let our Data Serving workload run for a three-day period. During this period, we continuously measured the performance reported by our client emulator. Whenever the client reported performance degradation of at least 20%, we labeled these performance crises as interference. We later use the time slots corresponding to the cloud's performance crises to drive our stress workloads (described below) on a co-located VM while replaying the traces. We further quantify the cloud's performance crises and use this information to drive the inputs of our stress workloads so as to cause similar performance degradation with respect to the particular VM we are stressing.

Using the clients' measured performance (e.g., response time), we evaluate DeepDive's ability to identify interference conditions. The clients label a certain performance loss as due to interference only if the amount of loss is larger than 20%. In Section 4.3, we demonstrate that DeepDive is capable of dealing with arbitrary interference conditions.

Interfering workloads. We evaluate DeepDive with three interfering workloads. Our *memory-stress* workload is inspired by the stress test from Mars *et al.* [26]. It aggressively exercises shared resources, like last-level caches and the memory controller. The workload takes the desired working set size as an input. We use *iperf* as our *network-stress* workload. It takes the desired network throughput as an input, and creates bi-directional UDP data streams to exercise network resources accordingly. Finally, we designed a simple *disk-stress* workload that copies files from one source to another, while respecting the maximum transfer rate defined as an input.

4.2 How accurate is the warning system?

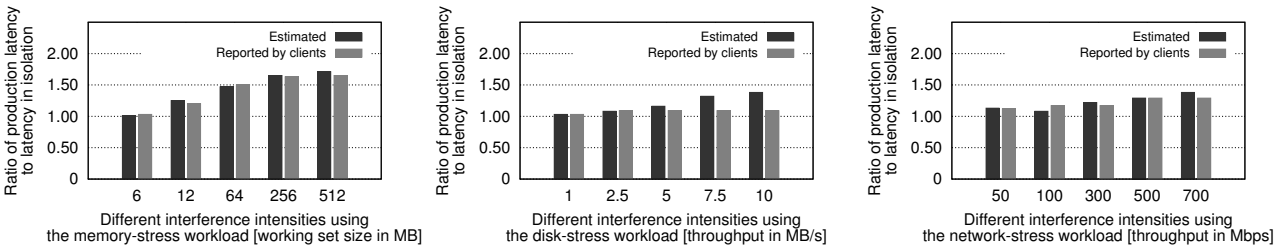
To demonstrate the effectiveness of the warning system, we clear the set of VM behaviors before each experiment. This forces the the warning system to rely solely on the information it obtained from the analyzer in the previous steps, as described in Section 3. Figures 7(a) to 7(c) plot the detection rate and the false positive rate of DeepDive while running our workloads. The detection rate measures DeepDive's consistency in identifying interference, whereas the false positive rate reflects scenarios where the warning system unnecessarily invoked the analyzer. In these experiments, we use memory-stress to generate interference, and vary the working set size to reproduce interference amounts that we obtained from our experiments on Amazon EC2. Because this workload primarily affects memory-related metrics that vary at a fine grain, this is the most challenging scenario for DeepDive to separate normal from interference conditions.

The figures show that DeepDive reliably identifies the interference, each time VM performance is substantially affected by the co-located VMs. Besides the detection rate, the number of analyzer invocations is important, as it determines DeepDive's overhead. On the first day after deployment, DeepDive shows a fairly high false positive rate, as it is still learning the normal behaviors. Starting from the second day, this rate drops to near-zero, as the warning system recognizes behaviors it has seen earlier. We did not observe false negatives in our experiments.

Importantly, recall that false positives do not result in unnecessary VM migrations, since the interference analyzer will realize that these metric deviations correspond to workload changes, rather than interference.

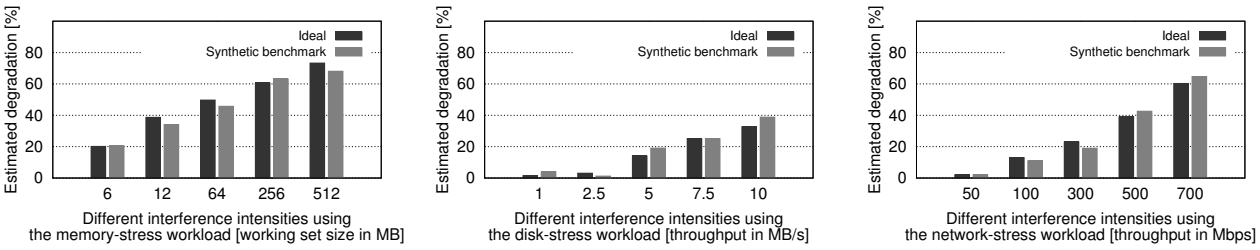
4.3 How accurate is the analyzer?

We now run experiments to demonstrate that DeepDive accurately estimates performance degradation under various interference conditions. We use client emulators for our workloads that continuously report average perfor-



(a) Data Serving (b) Web Search (c) Data Analytics

Figure 8: DeepDive accurately and transparently estimates performance loss from the metrics' values.



(a) Data Serving (b) Web Search (c) Data Analytics

Figure 9: The synthetic benchmark accurately reproduces the performance loss of its real counterpart.

mance, enabling us to compare the client-reported degradations with those estimated by the analyzer.

We run the experiments at the maximum-possible request rate. We allow the servers to warm up for several minutes and start reporting stable performance. At this point, we launch the stress workloads on a co-located VM to inject interference. Given our workloads, and the server components they primarily exercise, we co-locate: i) memory-stress with Data Serving, ii) network-stress with Data Analytics, and iii) disk-stress with Web Search. We vary the interference intensity by varying: i) the working set size of memory-stress from 6 MB to 512 MB, ii) the throughput of network-stress from 50 Mbps to 700 Mbps, and iii) the file transfer rate of disk-stress from 1 MB/s to 10 MB/s. Our goal is to select the stress workloads' inputs so as to replicate the cloud's performance losses seen in our experiments on Amazon EC2.

Figure 8 plots both the estimated and client-reported latency degradations for Data Serving and Web Search, and task completion time degradations for Data Analytics, reported by the interference-suffering VM. Each group of bars represents a different amount of interference, yielding performance degradation roughly from 5% to 50%. We observe that the analyzer's CPI analysis can faithfully approximate the degradation across the interference levels. In particular, we observe that the analyzer estimates the degradation within 10% accuracy in the worst case, and less than 5% on average.

4.4 How robust is DeepDive's placement?

Here we evaluate the ability of DeepDive's synthetic benchmark to mimic the behavior of a VM in two ways. First, we monitor the performance degradation that both the monitored VM and its synthetic representation experience when co-located with our stress test workloads. If they match, the synthetic benchmark can successfully be used to quickly test if a migrated VM would no longer suffer interference. To evaluate the synthetic clone's accuracy under different interference conditions, we leverage our three stress workloads to tune interference intensities. Figures 9(a) to 9(c) contrast the performance loss reported by the real VM and its synthetic representation, while the real VM runs different cloud applications. We see that the synthetic benchmark can closely approximate the performance loss of a real VM – the median and average estimation error of our synthetic benchmark across all our experiments were 8% and 10%, respectively. These results can be improved, especially if representative interference conditions are considered during the training of the synthetic benchmark.

Next, we show how the placement manager migrates an aggressive VM that is the culprit for interference to a destination PM so as to minimize the resulting interference. In response to detecting an interference-inducing VM (memory-stress), DeepDive runs the synthetic representation of this aggressive VM on three PM candidates, each of which is running one of our workloads. Based on these runs, the placement manager selects the destination PM on which the analyzer reports the least interference. Figure 10 plots the resulting performance loss at that PM relative to the best (but impractical) scenario where the placement manager learns the interference effects on the destination PM by actually performing VM migration. During the experiment, we also record the resulting performance loss for all the possible placements, allowing us to: i) compute the average performance loss,

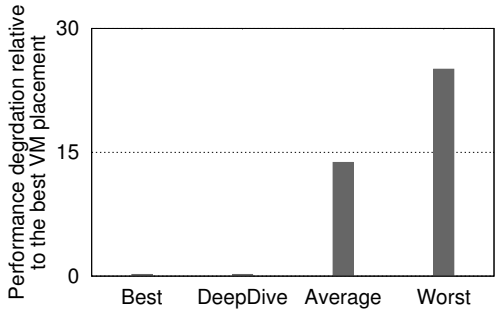


Figure 10: The placement manager properly predicts interference on the possible destination PMs.

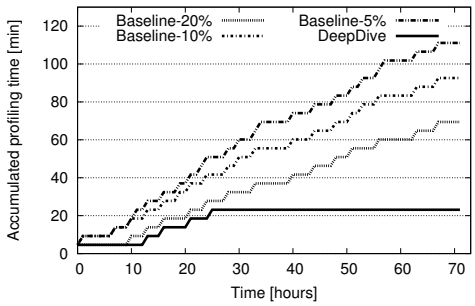


Figure 11: DeepDive's profiling overhead is low, and diminishes as it learns more about the VM behaviors.

and ii) label the placement with the highest performance loss as the worst. We observe from the figure that DeepDive finds the best destination PM relying on its synthetic benchmark to estimate the interference. This result is important, because it shows that we can entirely eliminate expensive and yet worthless (for placement) VM migration that could cause performance loss elsewhere.

4.5 What is the overhead of DeepDive?

DeepDive imposes a small per-VM memory overhead. For example, even when a VM is experiencing interference every hour, DeepDive requires less than 5KB to record the VM's behavior for the whole day. Storing this information into a repository is not an issue, as there are many works on high-performance NoSQL datastores.

We next explore DeepDive's profiling overhead, i.e. the amount of time and the number of machines required by the interference analyzer. We have conducted our evaluation using both live experiments with the Data Serving workload (it invokes the analyzer most frequently) and simulations. Running live experiments in our testbed helps us understand how often DeepDive triggers the analyzer in dynamic, realistic environments, and gives us an idea of the overall profiling overhead. Using this information, we drive simulations to analyze the scaling properties of DeepDive when applied to large-scale datacenters with high VM-arrival rates.

Using real experiments, Figure 11 plots the accumulated profiling time for a VM undergoing interference for both DeepDive and a baseline approach. The baseline triggers the analyzer every time performance varies more than a threshold (5%, 10%, and 20%). Triggering the analyzer too frequently renders the baseline unscalable and infeasible in practice. On the other hand, DeepDive relies on its warning system and its observed VM behaviors to prevent unnecessary VM profiling. The figure shows that DeepDive's overhead accumulates to only twenty minutes of profiling over 3 days. In fact, after the first day, no more profiling is needed.

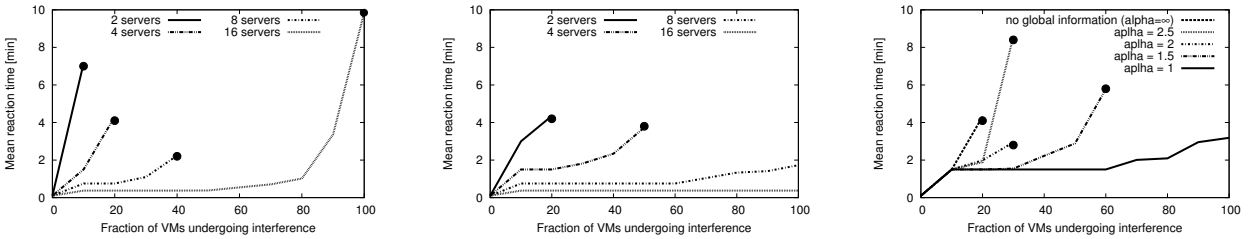
To extrapolate from these results, we next drive our simulator to trigger the analyzer exactly at the points in time that were previously recorded by our live experiment. We also used Matlab to model DeepDive's profiler as a simple queue: i) the VM arrival rate follows a Poisson process (we also experiment with a lognormal distribution of VM arrivals below), ii) the service time is replicated from the live experiments, and ii) the datacenter handles 1000 new (incoming) VMs every day.

Figure 12(a) presents DeepDive's reaction time as a function of the percentage of VMs undergoing interference. The figure plots the reaction time as long as the system is stable (mean service time < mean inter-arrival time), and the waiting time is acceptable (less than 10 minutes). As expected, the mean reaction time decreases as DeepDive uses more profiling servers. Most importantly, the figure demonstrates a desirable scaling behavior. For instance, only four profiling servers provide reaction time within four minutes, even under an aggressive rate of 20% of VMs undergoing interference.

These results assume that each VM runs a different workload, thus preventing DeepDive from being able to leverage global information. We design another set of experiments where VM reoccurrence follows a typical Zipf distribution – a few cloud tenants execute their workloads on a large number of VMs (available global information), and the remaining tenants run their deployments on a handful of VMs ("the long tail"). Figure 12(b) shows that leveraging global information significantly improves DeepDive's reaction time and allows it to reduce the number of profiling servers required (by 2x in these experiments).

To mimic various deployment scenarios, we vary the power-law tail index (from light- to heavy-tailed, using the α parameter) while using four profiling servers. Figure 12(c) plots the mean reaction time as a function of interference. While leveraging global information is most effective under the "light tail" conditions (α=1), it substantially improves DeepDive's reaction time for all the scenarios we considered.

To demonstrate DeepDive's scaling under more bursty workloads, we repeat the same set of experiments under a lognormal VM-arrival distribution, again assuming 1000 new VMs per day. The results (available in [28]) show



(a) Using only local information. (b) Using both local and global information. (c) Under a range of VM popularities [5].

Figure 12: Reaction time for 1000 new VMs per day. Curves stop where the system becomes unstable or excessively slow.

that fewer than 10 profiling machines are required, even under an extreme new-VM arrival scenario.

5 Related Work

Interference analysis. Most of the prior efforts on analyzing interference focus on on-chip contention and/or require application feedback. Recent efforts [12, 18, 25, 26, 37] demonstrate that an analysis of the sensitivity of workloads to co-located applications may accurately predict the degradation due to interference. In public clouds however, applications are not available prior to their deployment and often run for a long time, so cloud providers cannot easily perform this analysis. Thus, DeepDive does not rely on prior knowledge of applications or their interactions.

To speedup interference analysis, Paragon [16] uses a few stress experiments with each new application and a recommendation system to identify the best placement for the application with respect to interference. In contrast, DeepDive collects low-level metrics (the augmented CPI stack) from production VMs without stress tests. Moreover, because it was implemented in a virtualized environment, DeepDive can easily rely on VM migration for changing placements when workloads change and interference reoccurs.

Concurrently with our work, Zhang *et al.* [35] proposed CPI^2, a method for detecting and eliminating CPU interference on shared clusters. Our approach differs because: i) DeepDive uses CPI, not only to detect interference, but also to pinpoint its root cause, ii) DeepDive extends CPI analysis by including I/O, and iii) DeepDive leverages its synthetic benchmark to estimate the potential impact of a migrated VM on alternative PMs.

Focusing on IaaS clouds and long-running workloads, DejaVu [33] relies on comparing the performance of a production VM and a replica of it that runs in a sandbox to detect interference. If interference is present, DejaVu overprovisions virtual resources to mitigate its effects. Unfortunately, DejaVu relies on user/application assistance to identify interference and cannot pinpoint its cause. Moreover, overprovisioning is an inefficient approach for tackling interference.

Workload profiling and characterization. Sample-based profiling tools, like Magpie [23] and Pinpoint [13], produce workload models and automatically manage failures in distributed systems. Although these tools are useful for understanding workload (mis)behaviors, they are not useful in virtualized environments where cloud providers do not have access to the applications running inside VMs. Without requiring such access, DeepDive can pinpoint the main source of VM interference, and migrate VMs to reduce or even eliminate it.

Synthetic benchmarks. Given their easy development, synthetic benchmarks are often used to mimic behaviors of a specific application on different hardware platforms. Even more conveniently, tunable benchmarks can closely approximate a large portion of an arbitrary application's behavior by merely determining a suitable set of input parameters [32]. Several recent efforts [22, 29, 30, 31] have also demonstrated that one can reproduce any application's behavior using a limited number of the application's characteristics, such as the memory access pattern and instruction dependencies. These previous efforts inspired the design of our synthetic VM benchmark. Importantly, we are the first to use such a benchmark to manage interference.

Recently, Bubble-Up [26], Paragon [16], and Bobtail [34] proposed test benchmarks for placing VMs or applications. Bubble-Up uses a benchmark to exercise the memory system and characterize the effect it has on a co-located application. Similarly, Paragon uses multiple benchmarks to identify sources of interference and their impact on a co-located application. Bobtail employs a simple test program to determine whether the VMs already running on a PM are CPU-intensive. In contrast to these systems, our simple benchmark reproduces the behavior of each VM that DeepDive intends to migrate, and considers all resources that can cause interference, including disk and network I/O.

Performance modeling. Recent efforts have tried to predict performance by relying on regression models. For example, Lee *et al.* [24] combine processor, contention, and penalty models to estimate performance in multiprocessors. Similarly, Deng *et al.* [17] rely on hardware performance counters to model the perfor-

mance (and power consumption) of the memory subsystem. These works are orthogonal to DeepDive, since it does not try to predict performance per se, but rather to pinpoint the resource that is causing the interference. Furthermore, our framework is not tied to a specific architecture, and focuses on all key shared system resources.

6 Conclusion

Cloud services are becoming increasingly popular. A key challenge that cloud service providers face is how to identify and eliminate performance interference between VMs running on the same PM. This paper proposed and evaluated DeepDive, a system for transparently and efficiently identifying and managing interference. DeepDive quickly identifies that a VM may be suffering interference by monitoring and clustering low-level metrics, e.g. hardware performance counters. If interference is suspected, DeepDive compares the metrics produced by the VM running in production and in isolation. If interference is confirmed, DeepDive starts a low-overhead search for a PM to which the VM can be migrated.

Acknowledgments

This research was funded in part by the TRANSCEND Strategic Action Grant from Nano-Tera.ch and NSF grant CNS-0916878. Nedeljko Vasic is also supported by Swiss NSF grant FNS 200021-130265. We thank the anonymous reviewers and our shepherd Manuel Costa for their careful reading and valuable comments. We would also like to thank Babak Falsafi and his team for providing us with the latest version of CloudSuite.

References

[1] Apache mahout. `http://mahout.apache.org`.

[2] Apache nutch. `http://nutch.apache.org`.

[3] Faban framework. `http://java.net/projects/faban`.

[4] HDFS. `http://hadoop.apache.org/core`.

[5] Pareto distribution. `http://en.wikipedia.org/wiki/Pareto_distribution`.

[6] What are the barriers to cloud computing. `http://www.interxion.com/cloud-insight/`.

[7] A. R. Alameldeen et al. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 2006.

[8] Apache Foundation. The Apache Cassandra Project. `http://cassandra.apache.org/`.

[9] R. Azimi, et al. Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. In *ICS*, 2005.

[10] S. Basu, et al. Active semi-supervision for pairwise constrained clustering. In *SDM*, 2004.

[11] M. Bilenko, et al. Integrating constraints and metric learning in semi-supervised clustering. In *ICML*, 2004.

[12] D. Chandra, et al. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[13] M. Y. Chen, et al. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[14] B. F. Cooper, et al. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[15] J. Dejun, et al. EC2 performance analysis for resource provisioning of service-oriented applications. In *NFPSLAM-SOC*, 2009.

[16] C. Delimitrou et al. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[17] Q. Deng, et al. Memscale: active low-power modes for main memory. In *ASPLOS*, 2011.

[18] M. Dobrescu, et al. Toward predictable performance in software packet-processing platforms. In *NSDI*, 2012.

[19] L. Eeckhout. *Computer Architecture Performance Evaluation Methods*. 2010.

[20] M. Ferdman, et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, 2012.

[21] M. Hall, et al. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[22] A. Joshi, et al. The return of synthetic benchmarks. In *SPEC Benchmark Workshop*, 2008.

[23] T. Kielmann, et al. Magpie: Mpi's collective communication operations for clustered wide area systems. *SIGPLAN Not.*, 1999.

[24] B. C. Lee, et al. Cpr: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.

[25] J. Machina et al. Predicting cache needs and cache sensitivity for applications in cloud computing on cmp servers with configurable caches. In *IPDPS*, 2009.

[26] J. Mars, et al. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. *IEEE Micro*, 2012.

[27] R. Nathuji, et al. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys*, 2010.

[28] D. Novaković, et al. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. Technical Report 183449, EPFL, 2013.

[29] A. Phansalkar, et al. Measuring program similarity: Experiments with spec cpu benchmark suites. In *ISPASS*, 2005.

[30] T. Sherwood, et al. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[31] K. Skadron, et al. Challenges in computer architecture evaluation. *Computer*, 2003.

[32] E. Strohmaier et al. Architecture independent performance characterization and benchmarking for scientific applications. In *MASCOTS*, 2004.

[33] N. Vasić, et al. DejaVu: Accelerating Resource Allocation in Virtualized Environments . In *ASPLOS*, 2012.

[34] Y. Xu, et al. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, 2013.

[35] X. Zhang, et al. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[36] W. Zheng, et al. Justrunit: Experiment-based management of virtualized data centers. In *USENIX*, 2009.

[37] S. Zhuravlev, et al. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.

Efficient and Scalable Paravirtual I/O System

Nadav Har'El
nadav@harel.org.il

Abel Gordon
abelg@il.ibm.com

Alex Landau
landau.alex@gmail.com

Muli Ben-Yehuda*
muli@cs.technion.ac.il

Avishay Traeger
avishay@il.ibm.com

Razya Ladelsky
razya@il.ibm.com

IBM Research — Haifa

*Technion and Hypervisor Consulting

Abstract

The most popular I/O virtualization method today is paravirtual I/O. Its popularity stems from its reasonable performance levels while allowing the host to *interpose*, i.e., inspect or control, the guest's I/O activity.

We show that paravirtual I/O performance still significantly lags behind that of state-of-the-art non-interposing I/O virtualization, SRIOV. Moreover, we show that in the existing paravirtual I/O model, both latency and throughput significantly degrade with increasing number of guests. This scenario is becoming increasingly important, as the current trend of multi-core systems is towards an increasing number of guests per host.

We present an efficient and scalable virtual I/O system that provides all of the benefits of paravirtual I/O. Running host functionality on separate cores dedicated to serving multiple guest's I/O combined with a fine-grained I/O scheduling and exitless notifications our I/O virtualization system provides performance which is 1.2x–3x better than the baseline, approaching and in some cases exceeding non-interposing I/O virtualization performance.

1 Introduction

In recent years, hardware and software improvements for x86 machine virtualization made it possible to run virtualized workloads with performance approaching that of a physical machine (*bare-metal* performance). However, to achieve the desired bare-metal performance, I/O intensive virtual workloads require direct access to a hardware device [10]. For this purpose, modern hypervisors implement a technique called *device assignment* [7, 14, 30], "PCI passthrough" or "DirectPath I/O".

Device assignment achieves its performance by bypassing the host software on the I/O path, but this bypass also means giving up a lot of virtualization flexibility: With device assignment, the host software cannot offer a virtual device with no physical counterpart (e.g., a virtual disk stored as a file in the host's filesystem). Nor can it *interpose* on the guest's I/O, i.e., inspect or modify the guest's I/O, which is necessary for many virtualization features such virtual networking and security scanning. Device assignment also requires more expensive hardware (an IOMMU and SRIOV) and complicates VM live migration [31] and memory overcommitment [30]. For these and other reasons, most real-world applications of virtualization today—including most enterprise data centers and most cloud computing sites—do not use device assignment.

Instead, the most popular I/O virtualization technique today is *paravirtual I/O* [2], exemplified by KVM's virtio [23] and VMWare's VMXNET3 [28]. In paravirtual I/O, the host presents to its guests a software-based (virtual) I/O device. All I/O passes through the host software, retaining the ability to interpose on the guest's I/O and all the flexibility described above.

But paravirtual I/O's interposition comes with significant performance penalty for I/O-intensive guests, as already noted in previous work [5, 6, 13, 16, 29]. Traditional paravirtual I/O implementations suffer from two problems: The first is the slowdown of a single guest, mainly caused by *exits* [1] — switches back and forth between guest and host context. The second is lackluster scalability — when the host has multiple I/O-intensive guests, the competition between these guests cause significant reduction in throughput and increase in latency. These problems are becoming increasingly serious, as the current trend is towards multi-core systems with an increasing number of guests per host, and towards faster networks with expectation of lower latency and higher bandwidth.

We present ELVIS (**E**fficient and sca**L**able para-**V**irtual **I**/O **S**ystem). ELVIS solves the above two problems, and provides all the benefits of paravirtual I/O with performance approaching — and sometimes surpassing — that of device assignment. ELVIS's design is presented in Section 2: It is designed to be oblivious to the type of I/O activity (e.g., block or network), to maximize

throughput, to minimize latency, and to scale linearly in the number of I/O-intensive guests. ELVIS alleviates the overhead of paravirtual I/O by running host functionality on dedicated cores that are separate from guest cores, and by avoiding exits on the I/O path. ELVIS efficiently and fairly handles multiple guests by using a new fine-grained I/O scheduler that decides when and for how long to serve each guest.

We describe ELVIS's implementation in the KVM hypervisor in Section 3 and experimentally evaluate it in Section 4. We thoroughly evaluate ELVIS's performance using throughput-oriented and latency-oriented benchmarks on both network-intensive and block-intensive workloads. We evaluate its scalability by running different experiments with up to 14 I/O-intensive guests. In the majority of benchmarks, ELVIS improved performance when compared with paravirtual I/O by up to 3x and was within 90% of device assignment, and sometimes even exceeding it. In the worst case benchmark ELVIS was only within 70% of device assignment but still improved paravirtual performance by 1.4x.

The main contributions of this work are as follows:

1. We demonstrate and evaluate, for the first time, how a new feature announced for future x86 processors — *posted interrupts* — can be exploited to improve paravirtual I/O performance. We efficiently emulate posted interrupts on today's processors by extending our previous work on Exit-Less Interrupts [10].

2. However, we show that posted interrupts only solve part of the problem. We contribute a novel fine-grained I/O scheduler which together with exit-less notifications provides a complete, efficient and scalable, paravirtual I/O solution.

2 ELVIS Design

In this section we introduce the design of ELVIS, our proposed model for an Efficient and scaLable para-Virtual I/o System. ELVIS is based on the familiar paravirtual I/O model [2, 23], the state-of-the-art mechanism for I/O virtualization with interposition. We improve on traditional paravirtual I/O performance by avoiding the exits associated with I/O request and reply notifications. We improve scalability with a fine-grained I/O scheduling mechanism, allowing a single I/O thread to efficiently serve multiple VMs.

The design we present in this section can be applied to different paravirtual I/O implementations in different hypervisors. In Section 3, we present in more detail our implementation in the KVM hypervisor, and its in-kernel paravirtual I/O implementation, vhost.

2.1 The paravirtual I/O model

In paravirtual I/O the host *interposes* on the guest's I/O, i.e., each I/O request is handled by the host. The guest's driver (the *front-end*) sends each I/O request to the host (*back-end*), which handles it and later returns a reply.

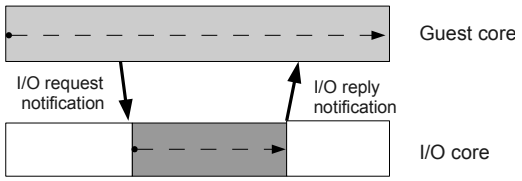


Figure 1: Ideal paravirtual model.

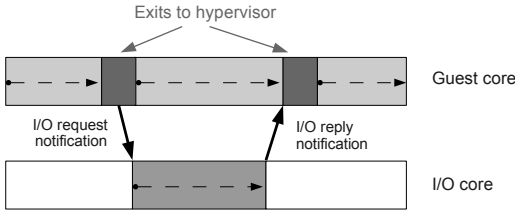


Figure 2: Slowdown when exits require notifications.

I/O requests are asynchronous: A guest does not block until getting the reply. In some cases, a long time might pass until a reply, e.g., disk reads or packet receive requests. So generally, the host does not fully handle the I/O request at the time of the request. Rather, the host has a separate *I/O thread* which handles the I/O requests.

On multi-core systems, it has been shown [12, 17, 15] that performance can be improved by dedicating a separate core (a sidecore) for the I/O thread, instead of time-sharing the same core for both the guest and its I/O thread. Moving the I/O thread to a separate core not only leaves the guest's core with more cycles (and therefore improves the guest's peak performance), it also improves overall system efficiency as context switches are avoided. SplitX [13] studied the costs associated with such context switches, and found that in addition to their direct cost, there is another indirect cost of cache pollution, as each of the two alternating contexts (guest and I/O thread) runs slower for some time after each context switch. Aiming at improved performance, ELVIS therefore runs the guest and the I/O thread on separate cores.

Figure 1 illustrates this ideal paravirtual I/O model: The guest and I/O thread run on separate cores. The two cores efficiently communicate using shared memory buffers, and additionally require some mechanism for *notifications*: the guest wants to notify the I/O core of new I/O requests, and the I/O core wants to notify the guest when previous requests have completed.

In non-virtual environments, there is a light-weight architectural mechanism, Inter-Processor Interrupts (IPI) to send notifications between cores. But unfortunately, there are no mechanisms in currently available x86 hardware to send notifications to or from a running guest, without first existing to the hypervisor. This can lead to two exits for each I/O request, as illustrated in Figure 2:

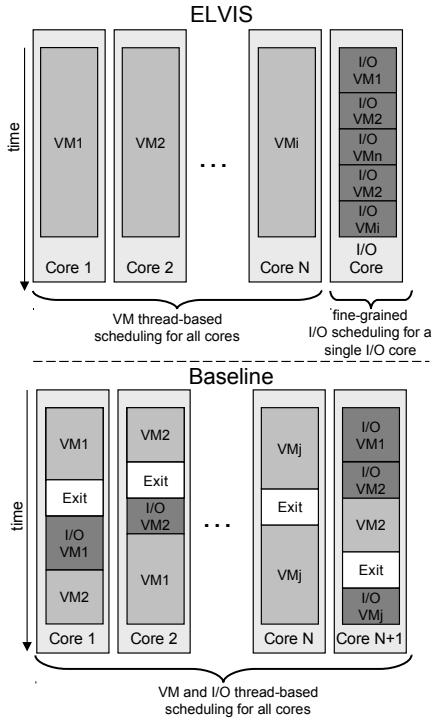


Figure 3: Comparing ELVIS's fine-grained I/O scheduling (top) to thread-based I/O scheduling (bottom).

When the guest wants to notify the I/O core of a new request in the shared buffer, it cannot directly send an IPI to the I/O core so it exits to have the hypervisor do this. Then, when the I/O core completes the operation and wants to notify the guest, it cannot remotely inject a virtual interrupt into the running guest, and needs to cause the guest to exit first (e.g., using an IPI) so that the hypervisor can inject the virtual interrupt. Some implementations even suffer a third exit, when the guest completes handling the virtual interrupt and writes to the End-of-Interrupt (EOI) register. ELVIS improves paravirtual I/O performance by replacing the two exit-causing notifications with new exit-less notification mechanisms, as we explain in Sections 2.3 and 2.4 below.

2.2 Fine-grained I/O scheduling

One I/O core is often capable of handling I/O from several I/O-intensive VMs, as we demonstrate in Section 4. However, the common approach to handle I/O is to create a separate I/O thread per VM and let the hypervisor's scheduler run these multiple threads on one or more cores.

ELVIS adopts a more *fine-grained* approach to I/O scheduling: A single I/O thread runs on an I/O core, and handles the I/O requests of multiple VMs. Figure 3 illustrates how fine-grained I/O scheduling differs from thread-based scheduling. We expect fine-grained I/O scheduling to achieve better throughputs and latencies than thread-based I/O scheduling: When several VMs have high I/O loads, thread-based I/O scheduling may service one VM for a long time, delaying I/O in other VMs until the OS decides to switch threads. Contrast this with fine-grained I/O scheduling, which can inspect the request queues it is serving, and can more fairly and promptly switch between them. The benefits of fine-grained scheduling are even more pronounced when the I/O thread uses polling, as it often does in ELVIS as explained below.

We show in Section 4.7 that indeed fine-grained I/O scheduling improves paravirtual I/O performance and scalability on multi-core machines. It allows an I/O core to handle more VMs with better throughput and latency.

2.3 Exitless I/O request notifications

In the paravirtual I/O model, the driver in the guest writes its I/O requests to a shared memory buffer. The driver then notifies the I/O thread that new work is pending. The x86 architecture provides no mechanism besides an exit for the guest to interrupt a host thread, so the request notification involves an exit, as shown in Figure 2.

In ELVIS, we avoid request notifications (and their associated exits) by *polling* in the host's I/O core [17, 5]. The guest writes its request to memory shared with the hypervisor, as usual, and does not employ any further exit-causing notification mechanism. The host polls this memory from the separate I/O core, handling requests as they are noticed.

Polling requires a dedicated I/O core, but as explained above, we generally want to share this core among several guests. With fine-grained I/O scheduling, ELVIS already has one I/O thread handling requests from several VMs, so now it needs to poll several VMs. In Section 3 we discuss how we efficiently and fairly poll several VMs without hurting the quality of service (namely, throughput and latency) to individual VMs.

For workloads which are not I/O-intensive, the waste inherent in excessive polling may outweigh the benefits of exitless notifications. It is therefore beneficial to dynamically switch between polling and traditional exit-based guest-to-host notifications. Such switching is often used in the context of interrupt mitigation [20, 24], and has also been used for paravirtual I/O by VMWare's VMXNET3 [28].

2.4 Exitless I/O reply notifications

In the paravirtual I/O model, when the I/O thread completes handling an I/O request it writes its reply to the shared memory area, and then notifies the guest.

Unfortunately, unlike the case of request notifications above, it is not practical to simply avoid using reply notifications. Avoiding these notifications means that each guest would need to poll for new replies [5], wasting a significant number of cycles that could otherwise be used

to run more useful work or just kept unused to reduce power consumption. Since we cannot avoid reply notifications, our goal is to make them as efficient as possible, and in particular exitless.

The architectural mechanism of notifying an OS of some event is via an interrupt. I.e., the I/O core wishes to cause an interrupt inside a guest running on a different core. On existing x86 processors, a hypervisor can only *inject* interrupts into the guest from the same core running it, and therefore the reply notification requires causing the guest to exit (e.g., by sending an Inter-Processor Interrupt (IPI) to the core running the guest), at which point the hypervisor injects the desired virtual interrupt.

Intel has recently announced that unspecified future processors will include a new feature called *posted interrupts*. Posted interrupts will allow one core to inject a virtual interrupt into a guest currently running on a different core — without the guest having to exit first. AMD also announced a similar future mechanism in their processors, and named it *doorbell interrupts*.

ELVIS avoids the reply-notification exits by emulating posted interrupts on existing x86 processors, using the Exit-Less Interrupts (ELI) technique [10]: When the I/O core wishes to inject a certain virtual interrupt into the guest running on a different core, it writes the interrupt vector (i.e., the interrupt number) into a memory location shared with the guest, and then sends a fixed IPI to the guest's core. Normally, receiving this IPI would cause the guest to exit, but we have this interrupt delivered in the running guest by asking the processor to deliver all interrupts to the running guest, with the guests interrupt descriptor table (IDT) shadowed so that only the fixed IPI is actually delivered to the guest and the rest cause an exit to the hypervisor. Once the fixed IPI is delivered to the guest, the handler for the vector number stored in the shared memory location is invoked. The ELI paper [10] focused on assigned devices and on the interrupts they generate but we extended this mechanism for delivering an IPI directly to the guest.

ELI works by asking the processor to deliver all interrupts to the running guest, with the guest's interrupt descriptor table (IDT) shadowed so that only the intended IPI is actually delivered to the guest and the rest cause an exit to the hypervisor. The ELI paper focused on assigned devices and on the interrupts they generate — but we can extend this mechanism for delivering an IPI directly to the guest.

3 ELVIS Implementation

To validate the ELVIS design, we implemented it in the KVM hypervisor. KVM [11] is implemented as a Linux kernel module that extends the kernel with hypervisor capabilities, driven by a QEMU [4] user process.

KVM offers two different implementations for paravirtual I/O devices: (1) a user-space implementation, part of QEMU; and (2) an in-kernel implementation, *vhost*. Both implement the same protocol, *virtio* [23], and share the same guest drivers. We based our implementation on vhost because it performs significantly better than the user-space alternative [27]. Vhost currently implements two paravirtual device types — network (*vhost-net*) and block device (*vhost-block*) and by modifying only their common base (vhost), we get ELVIS for both types of devices — as we show in Section 4.

We implemented ELVIS in KVM/vhost as follows:

3.1 Fine-grained I/O scheduling

Normally, vhost creates a separate I/O thread per paravirtual device, so that I/O handling can proceed in parallel to the guest running, boosting performance on multi-core systems. Each I/O thread potentially handles multiple *virtqueues* (queues of I/O requests and their replies [23]), e.g., a send queue and a receive queue in the paravirtual network device *vhost-net*.

With fine-grained I/O scheduling, we no longer create a separate I/O thread per device. Instead, we create only one I/O thread per dedicated I/O core, and each such thread now handles virtqueues from multiple virtual devices and multiple VMs. All these devices share a single *work queue*, to which vhost adds work when it is notified by the guest of a new I/O request, or when vhost discovers that a previous I/O request has completed (e.g., a packet has arrived, and can be returned to the guest). Note that this model does not affect the isolation and security properties of vhost.

Despite the fine-grained I/O scheduling, in some cases when one I/O thread handles many guests with very high throughput, a large number of I/O requests may arrive on a virtqueue before they can be handled, overflowing the virtqueue's ring buffers if they are not big enough. We found that in some of the network benchmarks presented in Section 4, the rings that Qemu allocates with a fixed default size 256 were occasionally overflowed. Increasing this default size to 512 was enough in all our experiments, and we used this new default in all baseline and ELVIS configurations in Section 4.

3.2 Mixing latency- and throughput-sensitive workloads

One of the challenges of implementing fine-grained I/O scheduling is deciding when to switch between virtqueues. Latency-sensitive workloads perform best when we only handle a virtqueue for a very short duration, and quickly move on to the next. High-throughput workloads, on the other hand, benefit from allowing more processing on each virtqueue before switching to the next. When guests are mixed — some care about latency and some about throughput — we need to carefully consider the needs of both.

Our implementation uses several heuristics to decide when to leave a virtqueue and proceed to the next: We always leave a queue after doing a certain maximum amount of work on it, even if it is not yet empty. We may leave a queue earlier (though not before we did some minimum amount of work on it), if we recognize that another non-empty queue is *stuck* and therefore likely to be latency-sensitive. We call a queue *stuck* if a certain time has passed since it last received new work. A latency-sensitive workload which waits for replies before sending further requests will get "stuck" in this sense, while a high-throughput workload which continuously creates new requests will not be found stuck. In Section 4.6, we show how these heuristics are indeed effective when high-throughput and low-latency workloads are served by the same I/O thread.

3.3 Placement of threads, memory and interrupts

Modern large multi-core systems have a multi-socket, or NUMA, design where part of the memory is closer to some of the cores. On such systems, performance is best if a guest running on a particular socket is serviced by an I/O thread running on the same socket, and if and the virtqueues shared between them are allocated from memory closest to this socket.

Our implementation therefore dedicates one I/O core (or more) per socket, and pins an I/O thread to each I/O core. Each I/O thread is configured to handle only virtqueues belonging to guests running on cores on the same socket. We ensure that an SMP guest does not spread across multiple sockets by limiting its vcpu threads to only run on cores on one socket.

Finally, when the virtual device is based on a physical device (e.g., vhost-net uses the host network), performance can be improved with IRQ affinity: We direct interrupts from the physical device to the I/O core, avoiding interrupts (and their exits) on guest cores and improving cache hit rates.

3.4 Exitless I/O request notifications

In KVM, the guest notifies the host of new I/O requests by executing a programmable I/O (PIO) instruction, causing an exit. We avoid these exits by replacing these notifications with polling in the I/O thread:

The virtio protocol allows the host backend to tell the guest driver not to send notifications for a certain virtqueue. This flag is normally used for short durations, to avoid further notifications while the host is servicing a particular virtqueue. But in ELVIS, we permanently disable notifications for virtqueues which we intend to poll. Instead of waiting for notifications, we supplement vhost's work queue with a new *poll queue*, which lists the virtqueues that are being polled. In a round-robin fashion, considering the heuristics we previously described, we poll each virtqueue. If we discover new requests, we handle them, just like a notification would be handled.

Even with polling enabled, the work queue continues to be relevant: E.g., the network backend adds an item to it when a packet arrives, so an outstanding receive request would be completed. So we interleave looking for new work in both work and poll queues. For fairness, any time work is done on a virtqueue for any reason, this virtqueue is moved to the end of the poll queue.

It is important that polling be as efficient as possible, to ensure that unsuccessful polling of relatively-idle virtqueues does not significantly hurt performance of other virtqueues. By having the I/O thread map the memory of all polled virtqueues in its own memory space, polling a virtqueue for newly available work becomes nothing more than a simple memory read. To further reduce the impact of unsuccessful polling, our implementation enables polling on a virtqueue only after exceeding a predefined notification rate, and later disables polling (re-enabling exit-causing notifications) when activity on this virtqueue subsides. These optimizations allow us not to waste precious cycles on polling virtqueues which only infrequently see requests and exits.

3.5 Exitless I/O reply notifications

Vhost notifies the guest of a reply for a previous I/O request by injecting the guest with a virtual interrupt, coming from the virtual device. When the guest is currently running, KVM first forces it to exit by sending an inter-processor interrupt (IPI) to the core running the guest, and only then KVM on that core can inject the desired virtual interrupt.

We replaced this exit-causing mechanism with our exit-less mechanism, allowing the I/O core to send a virtual interrupt to a guest running on another core without causing the guest to exit first. Current x86 processors do not yet support such exitless cross-core interrupt injection, known as *posted interrupts*. We emulated it extending an efficient software-only technique known as *Exit-Less Interrupts* (ELI) [10], as explained in Section 2.4.

4 Evaluation

In this section, we experimentally evaluate and analyze the performance of our implementation. We look at both network-intensive and block-intensive workloads, and consider both throughput and latency. We evaluate scalability with experiments going up to 16 cores. We analyze how fine-grained I/O scheduling and exitless notifications contributed to the performance improvement.

The results show that ELVIS improved I/O interposition performance by 1.2x–3x compared to traditional paravirtual I/O, and that ELVIS scaled linearly to more guests. For most of the workloads, ELVIS interposition overhead — how far it was from state-of-the-art non-interposing I/O virtualization — was less than 10% when

enough cores were dedicated to handling the I/O of multiple VMs. In the worst case, the overhead was 30%.

4.1 Experimental Setup

Our test machine is an IBM System x3550 M4, equipped with a dual-socket, 8-cores-per-socket Intel Xeon E2660 CPU running at 2.2 GHz with hyper-threading disabled. The system includes 56GB of memory and an Intel x520 dual port 10Gbps SRIOV NIC. We used a second identical server connected directly by two 10Gbps fibers as the remote end of the benchmarks. We set the Maximum Transmission Unit (MTU) to its default size of 1500 bytes; we did not use jumbo Ethernet frames. The host ran Linux 3.1.0 and QEMU 0.14.0. The guests used only one VCPU and ran Linux 3.1.0. The guests' memory was backed by huge (2 MB) pages in the host. For setups using paravirtual I/O, we bridged the guests' virtual NICs with the host's physical NICs using macvtap.

4.2 Experimental Methodology

Performance-minded applications would typically dedicate whole cores to guests (a single VCPU per core), thus we limited our benchmarks to this case. We compared ELVIS against traditional paravirtual I/O to show our performance and scalability improvements, and against state-of-the-art non-interposing I/O virtualization to analyze the interposition overhead. Altogether, we measured and compared four configurations:

ELVIS: This configuration measured ELVIS performance. For it, we ran each VM with a paravirtual NIC or block device using our ELVIS-enabled KVM. To analyze ELVIS scalability and avoid performance interference caused by NUMA, we partitioned the physical resources symmetrically across the two CPU sockets. For benchmarks with $N \leq 7$ VMs we only used cores from the first CPU socket and one 10Gb port. We dedicated a core for each of the VMs (VCPU threads) and one core (the I/O core) to run a single ELVIS I/O thread. We also set the IRQ affinity to deliver the NIC's interrupts only to the I/O core. For benchmarks running $N > 7$ VMs, we enabled a second ELVIS I/O thread, used the second 10Gb port, and configured the second socket exactly in the same way we configured the first socket. The memory of each VM was pinned to the CPU socket running the VCPU thread. The Intel NICs were initialized without SRIOV support.

Baseline: This configuration represented traditional paravirtual I/O. We ran each VM using the unmodified KVM. To use the same amount of physical resources as the ELVIS configuration, for benchmarks running $N \leq 7$ VMs, we limited the Linux host to use only $N+1$ cores and one 10Gb port; The Linux scheduler decided on which core to run each of the VCPU and I/O threads. Similarly, for benchmarks running $N > 7$ VMs, we gave the host $N+2$ cores and the two 10Gb ports. The Intel NICs were initialized without SRIOV support and the NICs' interrupts were balanced across the cores in use. The NUMA node used to back the memory of each VM was decided by the Linux kernel.

Baseline with Affinity: We used this configuration to analyze how the Linux scheduler, IRQ balancer and NUMA memory allocation affect traditional paravirtual I/O. This setup is similar to Baseline except we explicitly partitioned the physical resources. For benchmarks using $N \leq 7$ virtual machines we only used cores from the first CPU socket and one 10Gb port. We dedicated a core to run each VCPU thread and one core (the I/O core) to run all the KVM I/O threads. We also set the IRQ affinity to deliver the NIC's interrupts only to the I/O core. For benchmarks running $N > 7$ VMs, we configured the second socket and used the second 10Gb port exactly in the same way we configured the first socket. The memory of each VM was pinned to the NUMA node (CPU socket) responsible for running the VCPU thread.

No Interposition: We used this configuration to analyze ELVIS's interposition overhead. For this setup we allocated the physical resources in a similar way we did for ELVIS: one dedicated core per VCPU and up to two 10Gb ports. We multiplex each 10Gb port across the VMs using device assignment (SRIOV and ELI [10]) so the hypervisor didn't interpose on the I/O. ELVIS uses additional dedicated cores to run the I/O threads, thus, to make a fair comparison, we kept one core per socket unused for No Interposition. We count this as the ELVIS inherent resource overhead ($1/7$ in the case of eight cores).

4.3 Network throughput

We used three different and well-known benchmarks to show ELVIS can virtualize and interpose I/O efficiently for network intensive workloads. For these benchmarks, ELVIS improved Baseline throughput up to 3x. Compared to No Interposition, ELVIS I/O interposition overhead was less than 10% in most of the cases and less than 30% in the worst case when we allocated sufficient cores to handle the I/O of multiple virtual machines. In addition, ELVIS always scaled much faster than Baseline up to 14 VMs. Baseline with Affinity didn't scale at all.

We considered the following three network benchmarks for the evaluation:

Netperf TCP stream opens a single TCP connection to the remote machine, and makes repeated `write()` calls of 64 bytes.

Apache is an HTTP server. We used *ApacheBench* to load the server. ApacheBench ran on the remote machine and repeatedly requested a static 4KB page from 2 concurrent threads per VM.

Memcached is an in-memory key-value storage server. It is used by many high-profile Web sites for caching

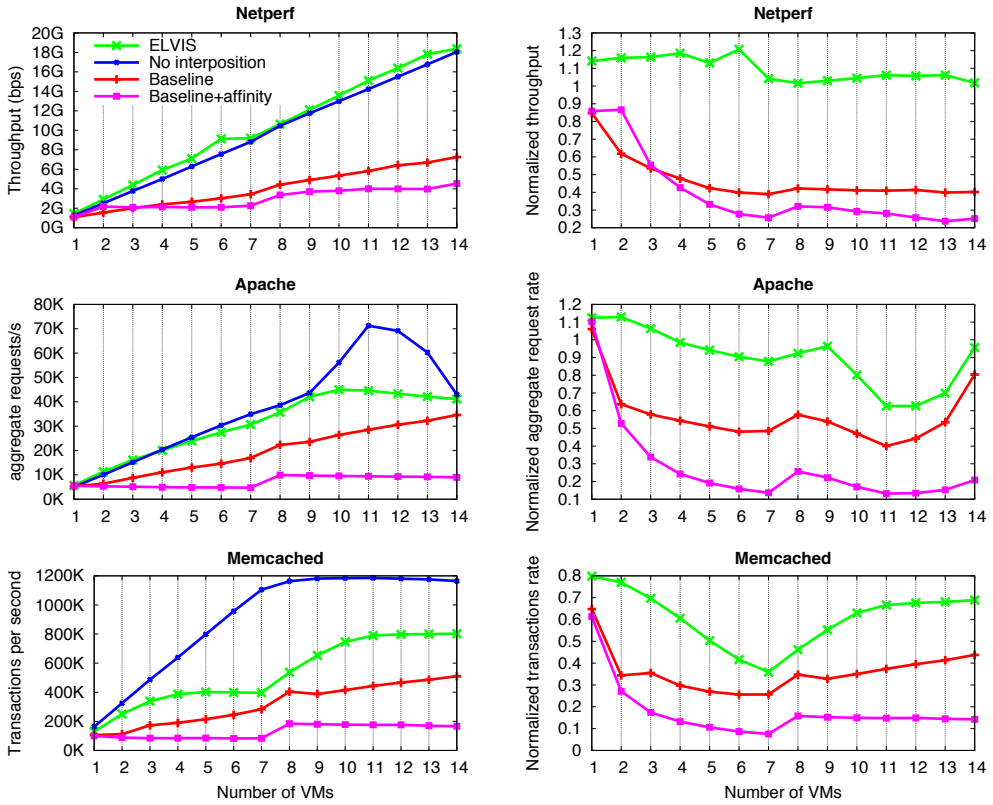


Figure 4: Comparing network throughput with ELVIS to that of the baseline and no-interposition configurations. Graphs on the left show for each of the three benchmarks, running on 1 up-to 14 VMs, the total throughput from all VMs. Graphs on the right are the same measurements shown normalized as a fraction of no-interposition performance.

results of slow database queries, thereby significantly improving the site's overall performance and scalability. We used the *Memslap* benchmark, part of the *libmemcached* client library, to load the server and measure its performance. Memslap runs on the remote machine, sends a random sequence of memcached `get` (90%) and `set` (10%) requests to the server and measures the request completion rate. We configured memslap to perform 32 concurrent requests per VM.

To verify ELVIS can scale using a single I/O core per socket, we ran the experiments using 1 through 14 VMs. Figure 4 compares the results for each of the three benchmarks using the four configurations previously described. We show on the left the aggregated throughput for all the VMs and on the right the same measurements normalized as a fraction of No Interposition. ELVIS improved Baseline throughput by 6%-200%. Baseline with Affinity as well as Baseline suffered from performance degradation due to the costly exit-based notifications and inefficient I/O scheduling. In these two configurations the Linux kernel couldn't make good scheduling decisions because it has no information about the content of the virtio queues. As evident from Figure 4, Baseline with Affinity didn't scale because one CPU core was used to run all the I/O threads which were competing for CPU cycles and starving each other. In contrast, in the Baseline case, the Linux kernel had more flexibility because threads could run on any core. The I/O threads could be scheduled instead of VCPU threads, unintentionally throttling the system. When a VCPU thread doesn't run, the VM doesn't perform I/O and releases CPU cycles to process pending I/O.

Baseline did better than Baseline with Affinity but scaled very slowly compared to ELVIS for all the benchmarks. ELVIS managed to scale almost perfectly for Netperf and Apache. The reason Apache stopped scaling after 10 VMs is because our remote machine was saturated. For Memcached, ELVIS scaled up to 3 VMs. At this point, the I/O core was saturated and ELVIS could not scale any more with a single I/O core. With more than 7 VMs, ELVIS used an additional dedicated core and Memcached continued scaling up to 11 VMs.

So far we demonstrated ELVIS performed and scaled better than traditional paravirtual I/O running network intensive workloads. However, we didn't show ELVIS I/O interposition overhead. For this purpose, we compare

ELVIS against No Interposition. We can see in Figure 4 that ELVIS results were pretty close to No Interposition. The performance degradation caused by ELVIS I/O interposition was less than 1%, 10% and 30% for Netperf TCP stream, Apache and Memcached respectively when the I/O cores and the remote machine were not saturated. In the case of Memcached, ELVIS required an additional I/O core to continue scaling after 3 VMs.

For some cases, ELVIS was even better than No Interposition. As we discussed in Section 3 and analyzed later in Section 4.6, ELVIS balances between throughput and latency by batching queued requests and coalescing the reply notifications. In the case of Netperf TCP stream and Apache, this mechanism improved throughput, making ELVIS up to 15% better than No Interposition. For example, when running Netperf TCP stream, ELVIS reduced the interrupt rate of each guest from 30K to 10K compared to No Interposition. However, in the case of Memslap, the same mechanism degraded the performance of the guests, and ELVIS performed up to 30% worse than No Interposition when we allocated sufficient cores to handle I/O.

Baseline suffered 142K, 109K and 146K exits/second for Netperf, Apache and Memcached respectively when we used only a single VM. As expected, ELVIS reduced the exits rate to less than 800 exits/second for all the benchmarks. Most of these remaining exits are not related to I/O — e.g., 500 of them are related to timer interrupts. The number of exits per VM for Baseline and Baseline with Affinity decreased as the number of VMs increased. That's because also in these setups the I/O threads batched more requests and coalesced more notifications, reducing the total number of exits/second. In addition, in the case of Baseline, the I/O threads were sometimes scheduled instead of VCPU threads, unintentionally throttling the system and further reducing the number of exits. For example, Baseline with 7 VMs handled 53K, 39K, 60K exits/sec per VM for Netperf, Apache and Memcached respectively.

4.4 Network latency

We measured ELVIS's latency using Netperf UDP-RR (request-response), which sends a UDP packet and waits for a reply before sending the next. Baseline with Affinity did not scale and latency increased to 400μsec because the I/O threads starved each other. Baseline managed to scale because the I/O threads could run on any core. Figure 5 presents the results, omitting Baseline with Affinity for clarity. With a single VM, ELVIS reduced Baseline's latency by 8μsec. With multiple VMs ELVIS reduced the average latency per VM up to 28μsec. This improvement was possible because ELVIS's fine-grained I/O scheduling, as opposed to thread-based scheduling, combined with exitless notifications managed to keep latency less than 1% worse than No Interposition. In Section 4.7 we analyze the performance contributions of fine-grained I/O scheduling and exitless notifications separately.

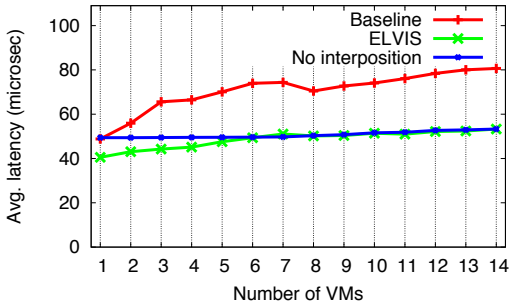


Figure 5: Average latency measured by UDP-RR, on 1 to 14 VMs. With ELVIS, latency is lower, and remains low even when each I/O core serves multiple VMs.

We notice in Figure 5 that when running less than 6 VMs, ELVIS again out-performs No Interposition. Here, batching and interrupt coalescing cannot explain this curious phenomenon, as it did for the network throughput workloads. There was a different reason: the NIC's latency increased when we enabled SRIOV. We ran a single instance of Netperf UDP-RR on bare-metal Linux with SRIOV disabled and we did the same test with SRIOV enabled. When SRIOV was disabled, the bare-metal Linux used a Physical Function in the same way KVM used it for ELVIS. But when we enabled SRIOV in bare-metal Linux, we intentionally used a Virtual Function in th same way the guests used them for No Interposition. We noticed that Netperf UDP-RR latency when running on bare-metal Linux increased by 22% when we used the Virtual Function (SRIOV was enabled).

4.5 Block workloads

We next analyzed the performance of ELVIS under an I/O-intensive block workload. To avoid physical disk bottlenecks and allow the VMs to achieve their maximum throughput, we assigned a single 1GB ramdisk on the host to the virtual machines using virtio. Each VM ran four write threads and four read threads, each of which performed 4KB random I/O on the paravirtual device using filebench. We bypassed the guest's buffer cache by opening the virtio device with the O_DIRECT flag, so that all I/O requests would pass the guest-host boundary. In this environment, we tested the ELVIS, Baseline, and Baseline with Affinity. We did not test No Interposition because in the case of a ramdisk, there is no physical device to assign to the guests.

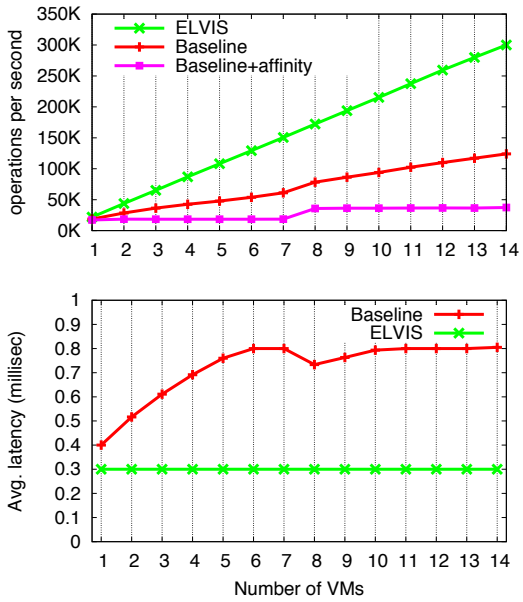


Figure 6: Filebench random read/write performance. No Interposition was not evaluated because the virtual disks exposed to the guests are backed by a ramdisk in the host (no physical device to assign).

As we can see in Figure 6, ELVIS scales perfectly up to 14 VMs. The bottleneck for each VM in this case is guest CPU saturation. Each VM performs about 21,613 operations/second on average, regardless of the number of running VMs, with a standard deviation of only 128. In addition, each VM experiences low latencies for its block accesses (0.3ms), regardless of the number of VMs running. For Baseline with Affinity, the I/O core is saturated immediately, and so aggregate throughput does not increase. Latency increases linearly to 3.0s and is omitted from the graph to improve clarity (as in Section 4.4). Baseline without affinity does scale, but not well. With one VM running, it achieves 18,715 ops/s with 0.4ms latency. When we reach 14 VMs, each achieves only 8,862 ops/s on average and the average latency doubles to 0.8ms. With one VM, ELVIS has 17% better throughput with 25% better latency, and with 14 VMs, ELVIS has 2.4x better throughput with less than half the latency.

4.6 Mixed latency- and throughput-sensitive guests

Sections 4.3 and 4.4 showed ELVIS was able to run throughput intensive and latency sensitive workloads efficiently in separated runs. However, ELVIS efficiency handling latency sensitive workloads may be influenced by throughput intensive workloads and vice versa when they run concurrently. While ELVIS handles I/O for a throughput-intensive VM, a latency-sensitive VM may be delayed. To decrease latency, ELVIS can scan I/O requests more often and serve latency-sensitive VMs immediately, but the cycles spent for scanning pending requests and switching between VMs degrades the performance of throughput intensive VMs.

To evaluate how ELVIS deals with this situation, we ran multiple instances of Netperf TCP stream representing throughput intensive workloads, simultaneously with multiple instances of Netperf UDP-RR representing latency sensitive workloads. We repeated the experiment running different combinations: M VMs ran TCP stream while $N-M$ VMs ran UDP-RR (for $N=7$, $M=1$ to 6). Figure 7 shows the TCP stream average throughput per VM and UDP-RR average latency per VM.

We compared the average performance per VM we obtained in this setup with the single VM results we obtained in sections 4.3 and 4.4. As shown in Figure 4, TCP stream achieved 1.45Gbps and 1.08Gbps when it ran in a single VM using ELVIS and Baseline respectively. Figure 7 shows that when 1 or 2 TCP stream were competing with 5 or 6 UDP-RR, ELVIS did not degrade TCP stream throughput. The latency of UDP-RR increased by 28μsec and 38μsec. In contrast, Baseline degraded TCP stream throughput by 17% and 26% when 1 or 2 TCP stream competed with 6 or 5 UDP-RR. Latency increased by 28μsec and 38μsec, but still 13μsec and 7μsec higher than ELVIS.

In general, for all the combinations, ELVIS degraded TCP stream by 0%-32% and increased UDP-RR latency by 28μsec-45μsec. In contrast, Baseline, degraded TCP stream by 16%-52% and increased UDP-RR latency by 29μsec-129μsec.

In all the configurations, except No Interposition, TCP stream throughput per VM degraded as we added more instances of UDP-RR. And UDP-RR latency increased when we added more TCP stream instances. As depicted in Figure 7, ELVIS managed to balance between throughput and latency sensitive workloads efficiently while Baseline and Baseline with Affinity didn't.

4.7 Impact of fine-grained I/O scheduling and Exitless notifications

To analyze how fine-grained I/O scheduling and exitless I/O notifications contributed to the performance improvements, we measured Netperf TCP stream using only fine-grained I/O scheduling and compared the results with ELVIS, Baseline and Baseline with Affinity.

Figure 8 shows the results for all the configurations using 1 to 7 VMs. The graph on the left shows the aggregated throughput while the graph on the right shows the throughput normalized as a fraction of Baseline. Fine-grained I/O scheduling and ELVIS improved Baseline throughput by 2.7x and 3x respectively when running more than one VM. As expected, with a single VM, Baseline with Affinity performance was similar to fine-grained I/O scheduling. That's because in both cases we

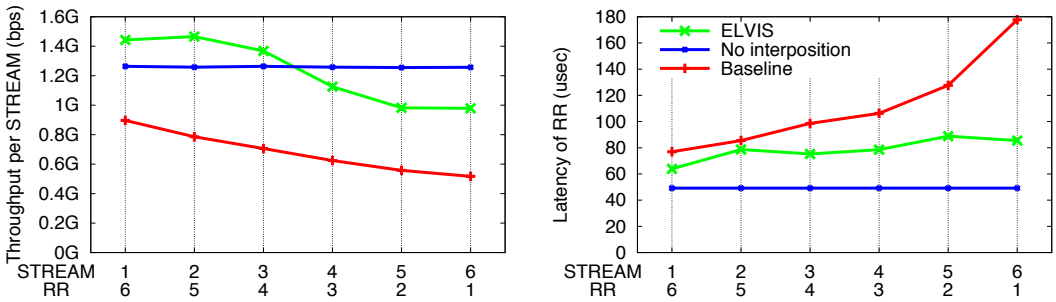


Figure 7: Comparing ELVIS to No Interposition and Baseline configurations when run a mix of Netperf TCP stream and UDP-RR with 7 VMs. The graph on the left shows the average TCP stream throughput per VM while the graph on the right shows the average UDP-RR latency per VM. The X axis shows how many VMs where running TCP stream and how many VMs where running UDP-RR.

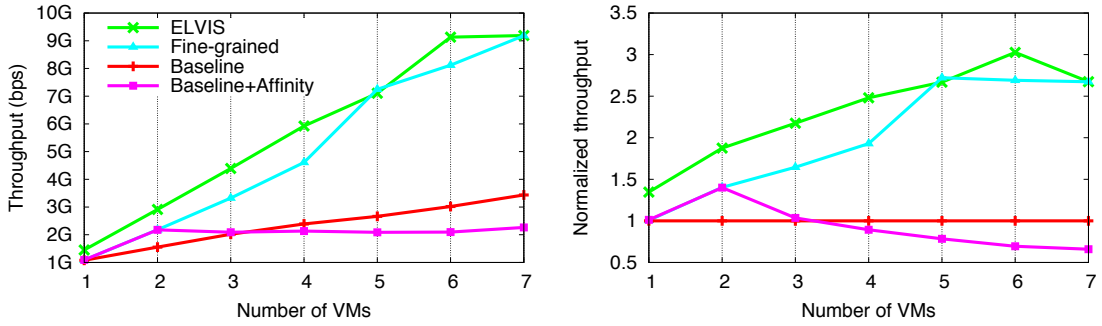


Figure 8: Comparing TCP stream throughput with ELVIS to that of Fine-grained I/O scheduling, Baseline and Baseline with Affinity configurations. The graph on the left shows the total throughput running 1 to 7 VMs. The graph on the right shows the same measurements normalized as a fraction of Baseline performance

used one dedicated core to process I/O and one dedicated core to run the VM. There was no need to schedule I/O for multiple VMs, thus fine-grained I/O scheduling did not have any impact on the performance. In this single VM case, the performance benefits of ELVIS come from exitless I/O request and replies. As depicted in the graph, ELVIS over-performed Baseline with Affinity by 33%.

Baseline achieved lower performance than Baseline with Affinity up to 3 VMs due to the overhead caused by balancing interrupts, I/O threads and the VCPU threads across the cores. As we discussed in Section 4.3, Baseline performed and scaled better than Baseline with Affinity when running more than 3 VMs because the I/O core used by Baseline with Affinity became saturated after 2 VMs. While Baseline scaled slowly up-to 7 VMs, it suffered from inefficient I/O scheduling. Fine-grained I/O scheduling contributed the most to ELVIS, giving a performance boost of 1.4x–2.7x over Baseline. The improvement of ELVIS against fine-grained I/O scheduling started decreasing after 5 VMs because ELVIS was approaching line rate while fine-grained I/O scheduling had free bandwidth to continue scaling. These results shows that fine-grained I/O scheduling is extremely important to scale linearly while exitless notifications are required to improve performance.

4.8 ELVIS and NUMA

As described in Section 2, ELVIS uses dedicated cores to handle I/O. The number of cycles consumed by the I/O cores reading and writing to the in-memory virtio queues is critical because the fewer cycles ELVIS spends to read/write from/to the queues, the more cycles the I/O core has to serve more requests. Thus, to reduce cycles consumed by memory operations, its preferable to handle the I/O of a given guest in the same socket (NUMA node) in which the guest runs. Otherwise, the I/O core will waste additional cycles to access memory managed by a remote NUMA node. To exemplify this phenomenon, we ran 1 to 7 instances of Netperf TCP stream in separate guests served by a single ELVIS I/O core. We compared two configurations: NUMA aligned and unaligned. In the NUMA aligned configuration we ran all the guests and handled I/O in the same socket. In the NUMA unaligned configuration, we ran all the guests in the one socket but handled I/O in the other socket. Fig-

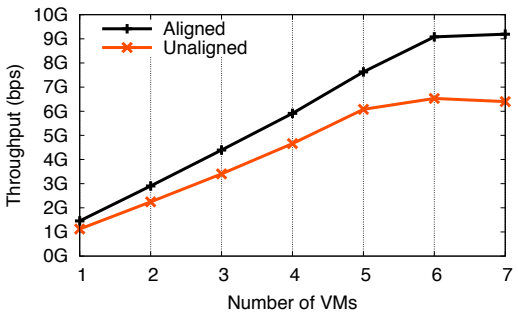


Figure 9: Comparing Netperf TCP stream performance when the I/O thread run in the same socket (Aligned) or different socket (Unaligned) where the VMs run.

ure 9 shows the results. NUMA aligned scaled perfectly and achieved line rate with 6 VMs. In contrast, NUMA unaligned was saturated after 5 VMs and achieved only 70% of NUMA aligned performance with 7 VMs. In addition, even for a single VM, NUMA aligned performed 26% better than NUMA unaligned. These results confirm ELVIS is extremely sensitive to NUMA. To maximize ELVIS performance and scalability, at least one core per socket should be dedicated to handle I/O. We believe this constraint is fairly weak because as the number of cores per socket increases year by year the resource overhead for having a dedicated core per socket will decrease. In addition, as network links continue to be improved, we will need more cores to virtualize more bandwidth at lower latencies.

5 Related Work

Many researchers have addressed different aspects of I/O virtualization over the years. In this section, we focus primarily on those works that are most closely related to ELVIS: those that seek to improve the performance of software-based I/O virtualization.

Menon et al. [19] and Santos et al. [25, 22] optimized Xen's paravirtual I/O model, focusing specifically on networking. Their goal was to achieve 10Gb/s for a single virtual machine, using Xen [2], which (at the time) did not use the architectural support for virtualization, leading to different decision considerations. Ongaro et al. [21] looked at the impact of guest scheduling on guest I/O performance, also in the context of Xen. VAMOS [8] reduces the overhead of paravirtual I/O by replacing many low-level I/O requests with fewer application-level requests. In contrast, ELVIS does not require modifying guests' applications. Mansley et al. [18] proposed a hybrid of paravirtual I/O and device assignment where the slow path goes through the hypervisor and the fast path goes directly to the device. Apart from the inherent problems with device assignment (e.g., difficult live migration), this approach does not work when there is no physical device, e.g., a virtual disk backed by a file. ELI [10] explored exitless interrupts in the context of device assignment but did not consider paravirtual I/O.

Dedicating cores to specific functions is well known to increase performance under certain conditions (e.g., [12, 15, 3, 26]). However, when using this approach in virtualized systems, special care has to be taken to ensure that inter-core communication does not cause exits and is both fast and scalable, so as not to become a bottleneck itself. We elaborate on these issues in Section 2.1.

Several works dedicated one core to the hypervisor and left the rest of the cores for (mostly) running guest functionality. VPE [17] did not remove costly exits for host-to-guest notifications and was networking specific; SplitX [13] relied on new hardware which is not available; another [5] was block-specific and used polling for both guest-to-host and host-to-guest notifications. In contrast to all of the above, ELVIS uses exitless notifications for host-to-guest notifications, is agnostic to the type of I/O protocol being used, achieves excellent performance on existing hardware, does not require any new hardware, and scales linearly in the number of guests.

In our short paper [9] we reported preliminary work on improving paravirtual I/O performance. In this paper, we present a complete system with new techniques and ideas, and provide a comprehensive evaluation of both performance and scalability for multiple workloads.

6 Conclusions and Future Work

Paravirtual I/O is the most popular I/O virtualization model used in virtualized data centers as well as cloud computing sites because it enables useful features such as live migration and software-defined-networks. These features, and many more, require from the hypervisor to interpose on the I/O activity of its guests. The performance and scalability of this interposition are extremely important for cloud providers and enterprise data centers. A guest running an I/O intensive workload should not affect the performance of other guests. The I/O resources must be shared fairly among guests depending on SLAs. The way we share the I/O resources should not affect the performance of the guests and should not have scalability limitations. For all these reasons we designed ELVIS, a low-overhead and scalable I/O interposition mechanism. Using two dedicated cores ELVIS can interpose on the I/O activity of up to 14 I/O-intensive guests and achieve performance that is 1.2x–3x better than the baseline while still scaling linearly.

We show that exitless requests and replies are required to improve performance and fine-grained I/O scheduling is required to improve scalability. Intel and AMD have recently announced that unspecified future processors will support exitless replies. This hardware capability alone will not solve the whole problem — exitless

requests and fine-grained I/O are also required.

In the future, we plan to improve our fine-grained I/O scheduling to consider guests' SLAs. In addition, we also plan to improve ELVIS to dynamically allocate or release I/O cores depending on the system load and guests' workloads.

Acknowledgments

The authors wish to thank Mike Day, Anthony Liguori, Shirley Ma, Badari Pulavarty from IBM's Linux Technology Center and our shepherd Paul Leach for the insightful comments and discussions. The research leading to the results presented in this paper is partially supported by the European Community's Seventh Framework Programme under grant agreement #248615 (IOLanes).

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *Architectural Support for Programming Languages & Operating Systems* (2006).

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[3] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 29–44.

[4] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference* (2005), pp. 41–46.

[5] BEN-YEHUDA, M., BOROVIK, E., FACTOR, M., ROM, E., TRAEGER, A., AND YASSOUR, B.-A. Adding advanced storage controller functionality via low-overhead virtualization. In *Conference on File & Storage Technologies (FAST)* (2012).

[6] DONG, Y., YANG, X., LI, X., LI, J., TIAN, K., AND GUAN, H. High performance network virtualization with SR-IOV. In *International Symposium on High Performance Computer Architecture* (2010).

[7] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).

[8] GORDON, A., BEN-YEHUDA, M., FILIMONOV, D., AND DAHAN, M. VAMOS: Virtualization aware middleware. In *USENIX Workshop on I/O Virtualization (WIOV)* (2011).

[9] GORDON, A., HAR'EL, N., LANDAU, A., BEN-YEHUDA, M., AND TRAEGER, A. Towards exitless and efficient paravirtual I/O. In *SYSTOR* (2012).

[10] GORDON, A., NADAV, A., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: Bare-metal performance for I/O virtualization. In *Architectural Support for Programming Languages & Operating Systems* (2012).

[11] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)* (2007).

[12] KUMAR, S., RAJ, H., SCHWAN, K., AND GANEV, I. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Operating Systems & Computer Architecture (WIOSCA)* (2007).

[13] LANDAU, A., BEN-YEHUDA, M., AND GORDON, A. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)* (2011).

[14] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2004).

[15] LIAO, G., GUO, D., BHUYAN, L., AND KING, S. R. Software techniques to improve virtualized I/O performance on multi-core systems. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2008).

[16] LIU, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *International Parallel & Distributed Processing Symposium (IPDPS)* (2010).

[17] LIU, J., AND ABALI, B. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM International Conference on Supercomputing (ICS)* (2009).

[18] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: safe and fast device access from unprivileged domains. In *Conference on Parallel Processing (Euro-Par)* (2007).

[19] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference* (2006).

[20] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS) 15* (1997), 217–252.

[21] ONGARO, D., COX, A. L., AND RIXNER, S. Scheduling i/o in virtual machine monitors. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)* (2008).

[22] RAM, K. K., SANTOS, J. R., TURNER, Y., COX, A. L., AND RIXNER, S. Achieving 10Gbps using safe and transparent network interface virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2009).

[23] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR) 42*, 5 (2008), 95–103.

[24] SALIM, J. H., OLSSON, R., AND KUZNETSOV, A. Beyond Softnet. In *Anual Linux Showcase & Conference* (2001).

[25] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference* (2008).

[26] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conference* (2010), p. 5.

[27] TSIRKIN, M. S. virtio- and vhost- net: need for speed. KVM Forum, 2010.

[28] VMWARE INC. Esx server 2 - architecture and performance implications. Tech. rep., VMWare, 2005.

[29] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *International Symposium on High Performance Computer Architecture* (2007).

[30] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct device assignment for untrusted fully-virtualized virtual machines. Tech. Rep. H-0263, IBM Research, 2008.

[31] ZHAI, E., CUMMINGS, G. D., AND DONG, Y. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium (OLS)* (2008), pp. 261–268.

vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core

Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, Dongyan Xu
Department of Computer Science, Purdue University

Abstract

In a virtual machine (VM) consolidation environment, it has been observed that CPU sharing among multiple VMs will lead to I/O processing latency because of the CPU access latency experienced by each VM. In this paper, we present vTurbo, a system that accelerates I/O processing for VMs by offloading I/O processing to a designated core. More specifically, the designated core – called turbo core – runs with a much smaller time slice (e.g., 0.1ms) than the cores shared by production VMs. Most of the I/O IRQs for the production VMs will be delegated to the turbo core for more timely processing, hence accelerating the I/O processing for the production VMs. Our experiments show that vTurbo significantly improves the VMs' network and disk I/O throughput, which consequently translates into application-level performance improvement.

1 Introduction

Cloud computing is arguably one of the most transformative trends in recent times. Many enterprises and businesses are increasingly migrating their applications to public cloud offerings such as Amazon EC2 [1] and Microsoft Azure [8]. By purchasing or leasing cloud servers with a pay-as-you-go charging model, enterprises benefit from significant cost savings in running their applications, both in terms of capital expenditure (*e.g.*, reduced server costs) as well as operational expenditure (*e.g.*, management staff). On the other hand, cloud providers generate revenue by achieving good performance for their "tenants" while maintaining reasonable cost of operation.

One of the key factors influencing the cost of cloud platforms is *server consolidation*—the ability to host multiple virtual machines (VM) in the same physical server. If the cloud providers can increase the level of server consolidation, *i.e.*, pack more VMs in each physical machine, they can generate more revenue from their infrastructure investment and possibly pass cost savings on to their customers. Two main resources that typically dictate the level of server consolidation, memory and CPU. Memory is strictly partitioned across VMs, although there are techniques (*e.g.*, memory ballooning [29]) for dynamically adjusting the amount of memory available to each VM. CPU can also be strictly partitioned across VMs, with the trend of ever increasing number of cores per physical host. However, given that each core is quite powerful, another major scaling factor comes by allocating multiple VMs per core. While the ever increasing core count in modern systems may suggest the possibility of a dedicated core per VM, it is not likely to happen any time soon, as evidenced in current cloud computing environments such as Amazon EC2, where a 3GHz CPU may be shared by three small instances.

In practice, CPU sharing among VMs can be quite complicated. Each VM is typically assigned one or more virtual CPUs (vCPUs) which are scheduled by the hypervisor on to physical CPUs (pCPUs) ensuring proportional fairness. The number of vCPUs is usually larger than the number of pCPUs, which means that, even if a vCPU is ready for execution, it may not find a free pCPU immediately and thus needs to wait for its turn, causing *CPU access latency*. If a VM is running I/O-intensive applications, this latency can have a significant negative impact on application performance. While several efforts [30, 12, 17, 28, 19, 25] have made this observation in the past, and have in fact provided solutions to improve VMs' I/O performance, the improvements are still moderate compared to the available I/O capacity because, they do not explicitly focus on reducing the most important and common component of I/O processing workflow—namely *IRQ processing latency*.

To explain this more clearly, let us look at I/O processing in modern OSes today. There are two basic stages involved typically. (1) Device interrupts are processed *synchronously* in an IRQ context in the kernel and the data (*e.g.*, network data, disk reads) is buffered in kernel buffers; (2) The application eventually copies the data from kernel buffer to its user-level buffer in an *asynchronous* fashion whenever it gets scheduled by the process scheduler. If the OS were running directly on a physical machine, or if there were a dedicated CPU for a given VM, the IRQ processing component gets scheduled almost instantaneously by preempting the currently running process. However, for a VM that shares CPU with other VMs, the IRQ processing may be significantly delayed because the VM may not be running when the I/O event (*e.g.*, network packet arrival) occurs.

IRQ processing delay can affect both network and disk I/O performance significantly. For example, in the case of TCP, incoming packets are staged in the shared memory between the hypervisor (or privileged domain) and the guest OS, which delays the ACK generation and can result in significant throughput degradation. For UDP

flows, there is no such time-sensitive ACK generation that governs the throughput. However, since there is limited buffer space in the shared memory (ring buffer) between the guest OS and the hypervisor, it may fill up quickly leading to packet loss. IRQ processing delay can also impact disk write performance. Applications often just write to memory buffers and return. The kernel threads handling disk I/O will flush the data in memory to the disk in the background. As soon as one block write is done, the IRQ handler will schedule the next write and so on. If the IRQ processing is delayed, write throughput will be significantly reduced.

Unfortunately, none of the existing efforts explicitly tackle this problem. Instead, they propose indirect approaches that moderately shorten IRQ processing latency hence achieving only modest improvement. Further, because of the specific design choices made in those approaches, the IRQ processing latency cannot be fundamentally eliminated (*e.g.*, made negligible) by any of the designs, meaning that they cannot achieve close to optimal performance. For instance, the vSlicer approach [30] schedules I/O-intensive VMs more frequently using smaller micro time slices, which implicitly lowers the IRQ processing latency, but not significantly. Also it does not work under all scenarios. For example, if two I/O latency-sensitive VMs and two non-latency-sensitive VMs share the same pCPU, the worst-case IRQ processing latency will be about 30ms, which is still non-trivial, even though it is better than without vSlicer (which would be 90ms). Similarly, another approach called vBalance [10] proposes routing the IRQ to the vCPU that is scheduled for the corresponding VM. This may work well for SMP VMs that have more than one vCPU, but will not improve performance for single vCPU VMs. Even in the SMP case, it improves the chances that at least one vCPU is scheduled; but fundamentally it does not eliminate IRQ processing latency because each vCPU is contending for the physical CPU independently.

To solve this problem more fundamentally, we aim to make the IRQ processing latency for a CPU-sharing VM almost similar to the scenario where the VM is given a dedicated core. To achieve this, we propose a new solution called vTurbo, that involves two basic ideas. First, we leverage the existence of multiple cores in modern processors to designate a specialized turbo-sliced core (or *turbo core* for short), for synchronous processing threads in the guest OS. In terms of actual hardware, the turbo core is no different from a regular core, except that the hypervisor-level scheduler schedules VMs on this core with *extremely small quantum* (*e.g.*, 0.1ms). Second, we expose this turbo-sliced core to each VM as a "co-processor" just dedicated to kernel threads that require synchronous processing, such as IRQ handling. The other regular kernel threads are scheduled on a regular core with regular slicing just like what exists today. Since the IRQ handlers are executed by the turbo core, they are handled almost synchronously with a magnitude smaller latency. For example, assuming 5 VMs and 0.1ms quantum for the turbo core, an IRQ request is processed within 0.4ms compared to 120 ms (assuming 30ms time slice for regular cores).

The turbo core is accessible to all VMs in the system. If a VM runs only CPU-bound processes, it may choose not to use this core since its performance is not likely to be good due to frequent context switches. Even if a VM chooses to schedule a CPU-bound process on the turbo core, it has virtually no impact on other VMs' turbo core access latency thus providing good isolation between VMs. We ensure fair-sharing among VMs with differential requirement between regular/turbo cores because, otherwise, it would motivate VMs to push more processing to the turbo core. Thus, for example, if there are two VMs—VM1 requesting 100% of the regular core, and VM2 requesting 50% regular and 50% turbo cores, the regular core will be split 75-25% while VM2 obtains the full 50% of the turbo core, thus equalizing the total CPU usage for both VMs. We also note that, while we mention one turbo core in the system, our design seamlessly allows multiple turbo cores in the system driven by I/O processing load of all VMs in the host. This makes our design extensible to higher bandwidth networks (10Gbps and beyond) and higher disk I/O bandwidths that require significant IRQ processing beyond what a single core can provide.

To summarize, the main contributions of this paper are as follows:

(1) We propose a new class of high-frequency scheduling CPU core named turbo core and a new class of co-vCPU called turbo vCPU. The turbo vCPU pinned on turbo core(s) is used for timely processing of the I/O IRQs thus accelerating I/O processing for VMs in the same physical host.

(2) We develop a simple but effective VM scheduling policy giving general CPU cores and turbo core magnitudes different time-slice. The very small CPU time-slice on turbo cores grants VM low scheduling delay and low I/O IRQ processing latency.

(3) We have implemented a prototype of vTurbo based on Xen. Various evaluations prove the effectiveness of vTurbo. Our micro-benchmark results show that vTurbo can significantly improve the TCP throughput (up to 3×), UDP throughput (up to 4×), and disk write throughput (up to 2×). Our evaluation with application-level benchmarks shows that vTurbo improves application-specific performance as well. For example, Olio's throughput is increased by 38.7%. NFS' throughput is improved by up to 2×.

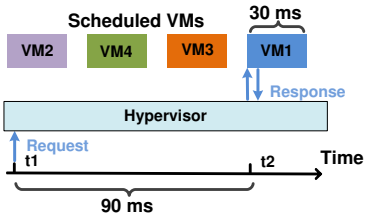


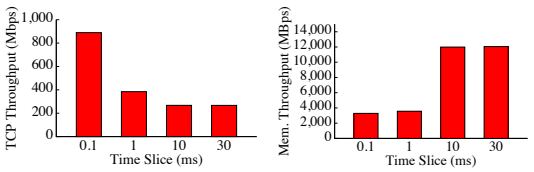
Figure 1: Impact of VM CPU sharing on I/O processing.

The rest of the paper is organized as follows. We motivate the problem in detail in Section 2 followed by the vTurbo design in Section 3. In Section 4 we describe the implementation of vTurbo prototype based on Xen. Section 5 presents evaluation results, followed by related work and conclusions.

2 Motivation

Let us first focus on receive-side I/O processing. In a non-virtualized system, all receive-side I/O events (*e.g.*, network packet arrival) are typically handled by specific IRQ routines corresponding to each device (*i.e.*, disk controller or NIC) in the OS kernel. The data is stored in a kernel buffer first, and once the user process is scheduled, it copies the data from the kernel buffer to the user buffer. Since I/O-bound processes usually have higher priority, they get scheduled relatively quickly and the data is subsequently processed by the application thus achieving high I/O throughput. However, in a virtualized system with several VMs sharing a physical CPU, each VM gets only a slice of the physical CPU, which means the incoming I/O event will need to wait until the VM gets access to the CPU. Such a CPU access latency will significantly affect the timeliness of IRQ processing, resulting in low I/O throughput.

We illustrate this negative effect using an example shown in Figure 1. In this example, 4 VMs share a physical CPU. VM1 runs a mixed workload that includes both CPU-bound tasks and I/O-bound applications, while VM2 to VM4 run only CPU-bound applications. Assuming a proportional-share VM scheduling policy (adopted by Xen and VMware ESX), VM1 gets only 25% of CPU when all VMs are busy, which means that roughly 75% of time, VM1 has to wait in *runqueue* and cannot process I/O events immediately. When an I/O request for VM1 reaches the hypervisor at t_1, VM1 cannot process this request and respond until t_2. If the I/O-bound application in VM1 is a TCP server, for instance, the client will stop sending data to the server once the client's TCP window is full, due to lack of acknowledgments from the server while VM1 is in *runqueue*. If VM1 runs a UDP server, even though the client can continue to send data to the server without getting responses, the packets will be dropped by the hypervisor once the shared buffers (between the hypervisor and guest OS) are full. As a result, throughput of either TCP or UDP for



(a) TCP throughput (b) Memory throughput

Figure 2: Impact of micro-timeslice on TCP throughput and memory throughput

VM1 would be much lower than the available capacity.

In the reverse direction (*i.e.*, when a process sends packets or writes to the disk), the user process first copies data to the kernel buffer associated with the particular output (*e.g.*, socket, file descriptor). For some I/O mechanisms such as asynchronous network packet sends and disk writes, the call to output the data will return to the user process immediately after the data is copied to the kernel buffer. The kernel components associated with the corresponding device will asynchronously write the data to the device. However, this task cannot be continued efficiently if the hypervisor schedules the vCPU out while the kernel component is waiting for the completion of the write to the device, resulting in low throughput.

There are other sources of delay for interrupt processing even after the I/O event reaches the VM. These include long periods in which, the VM runs with interrupts disabled, locking conflicts for shared data structures (such as TCP accept queue [26]) and overhead of dispatching interrupts in virtualized environments [13]. However, most of these latencies lie within sub-millisecond range in the average case [20, 18], while the scheduling delay causes the interrupt processing to be delayed for tens of milliseconds (in our example, the average scheduling delay is about 35ms for Xen VMM).

Symmetric multi-processing (SMP) VMs can take advantage of a multi-core architecture to execute many different applications in parallel and improve the overall system throughput. In an SMP-VM, two or more allocated vCPUs are scheduled by the hypervisor scheduler on any available pCPUs and thus, each vCPU has a higher chance to get scheduled. However, the SMP-VM may still suffer from scheduling delays, if none of the vCPUs can be scheduled in because the pCPUs are all busy executing other vCPUs. Thus, we cannot guarantee that the vCPU running an IRQ gets scheduled in time when a target VM receives an I/O request.

2.1 Existing Approaches

Now we discuss several existing approaches addressing the problem of CPU sharing impacting I/O performance of VMs and discuss why they do not work well.

Reducing CPU time-slice. One intuitive approach to solve the scheduling latency problem is to uniformly re-

duce the VM scheduling time-slice [15]. In proportional-share scheduling, the worst-case scheduling delay of each VM is *(Number of sharing VMs -1)* $\times$ *time-slice*. A small scheduling time-slice enables VMs to get scheduled more frequently thus improving the I/O throughput of VMs. However, the short time-slice results in more frequent context switches which may hurt the performance of memory-intensive or CPU-bound applications. We conduct a simple experiment to demonstrate this problem.

In our experiment, 4 single vCPU VMs share one physical CPU. One VM hosts a TCP server, the client is running in another physical machine in the same LAN. Iperf [5] is used to measure the server's TCP throughput. We vary the scheduling time-slice from 0.1ms to 30ms, which is the default time-slice of Xen. From Figure 2(a) we can find that, smaller time-slice leads to higher TCP throughput. Especially, with a 0.1ms time-slice, the average TCP throughput is up to 900 Mbps which is close to the bandwidth of 1Gbps network card used in our experiment. However, the performance of memory/CPU bound applications degrades under smaller time-slice as shown in Figure 2(b). Here, we run STREAM [6] benchmark in one of the 4 VMs[1]. So, simply reducing the CPU time-slice cannot simultaneously benefit both I/O-intensive applications and CPU-intensive applications. Hence this approach is not suitable for cloud environments where mixed workloads are common.

Sending I/O interrupts to active vCPU To reduce the IRQ processing delay and improve I/O throughput for SMP-VMs, a recent approach called vBalance [10] sends I/O interrupts to the active vCPU of the target VM. In this way, I/O interrupts can be processed in a more timely fashion and I/O throughput may be improved. However, there are still several issues with this method. As discussed before, an SMP-VM may have increased chances to get scheduled because of the multiple vCPUs assigned to it. But there is no fundamental guarantee that the SMP-VM have at least one vCPU running at any time. If none of the vCPUs is running, an I/O interrupt still cannot be processed in time. Besides, even if the I/O interrupt is sent to an active vCPU successfully, the I/O cannot be finished if the vCPU executing the I/O application is not running simultaneously. This specifically impacts TCP, where the application vCPU may be in the *runqueue* holding the ownership of the lock structure, hence the kernel-level TCP processing cannot generate an ACK in time for incoming TCP packets. We suspect this is the main reason [10] only reports 400Mbps TCP throughput in a 1Gbps LAN environment.

Differentiated VM scheduling Tuning VM scheduling policy is another method to speed up I/O processing. vSlicer [30] schedules each latency-sensitive VM (LSVM) more frequently with a smaller micro time-slice, which enables more timely processing of I/O events by LSVMs. There are two caveats of this approach. First, we need to know which VMs are LSVMs running latency-sensitive applications in advance and adjust the VM scheduler configuration accordingly. Second, vSlicer reduces the scheduling delay but does not completely eliminate it, as discussed earlier. It, therefore, does not improve the TCP/UDP throughput significantly, although it does reduce application-perceived I/O latency.

[1] We conducted a similar experiment in [30]. But here we set even smaller time-slice (0.1ms) and contrast TCP and memory throughput under such a time-slice.

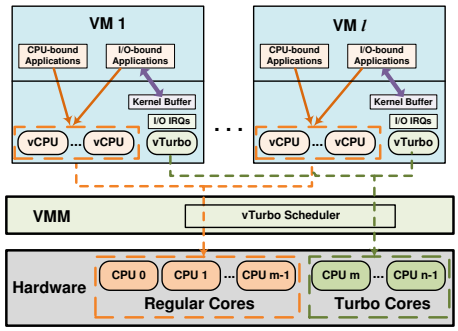


Figure 3: Architecture of vTurbo

3 Design

The discussion in the previous section suggests that if we use a very small value as the CPU time-slice, I/O performance of CPU-sharing VMs can be significantly improved. However, we also showed that such an approach may hurt the performance of CPU-bound VMs, for which larger time-slice is desirable. To address this dilemma, we leverage one *key degree of freedom* that has not been exploited hitherto: The CPU time-slice for each core may *not* be the same for a multi-cores system.

Thus, in our approach called vTurbo, we designate one (or more) core(s) in the system as what we call a *turbo core*, which is just any regular physical core, except that we set a very small (*e.g.*, 0.1ms) CPU time-slice for it. We expose the turbo core to each VM *in addition to* the regular cores, and allow the guest OS to schedule I/O-bound threads (*e.g.*, IRQ handling) in the turbo core thus speeding up I/O processing significantly. The guest OS still schedules CPU-bound workloads on cores with the regular time-slice. As such vTurbo achieves I/O processing speedup without impacting CPU-bound workloads.

In effect, vTurbo focuses on re-factoring the interface between the hypervisor and guest OS, with the new abstraction of turbo core. This approach is completely transparent to applications running in VMs, a key advantage of practicality. Another benefit of vTurbo is that it does not require classification of VMs into I/O- or CPU-intensive VMs, as required by some solutions such as vSlicer [30]. Such classification is difficult as most VMs

in practice run a combination of I/O and CPU workloads. Of course, the guest OS now needs to identify I/O-bound threads such as IRQ processing and schedule them on the turbo core. But that is not hard as there are only a handful of such threads. Our approach also guarantees CPU *fairness* among all VMs. Any VM using the turbo core will essentially not obtain any "extra" CPU beyond its fair share—an important property in multi-tenancy clouds.

The architecture of vTurbo requires changes to both the hypervisor and the guest OS. At the hypervisor level, the VM scheduler needs to accommodate the new turbo core abstraction. At the guest kernel-level, we need to modify the VM process scheduler to pin certain threads to the turbo core in addition to a few changes to the TCP protocol stack. In the following subsections, we discuss these in more detail.

3.1 Modifications to Hypervisor

We mainly need to modify the VM scheduler in the hypervisor to support the turbo core abstraction. Upon host initialization, we designate a set of cores in the host as turbo cores. The number of turbo cores is configurable, and our current version statically assign turbo cores based on user configuration. However, we believe that our system can be improved by having a dynamic method to assign turbo cores based on the available machine capacity (*i.e.*, total number of cores), number of VMs, demand for the turbo core, and overall I/O intensity (*e.g.*, a host with multiple active NICs or 10GB/s NICs may require more turbo cores). One can also dynamically change the number of turbo cores via administrative tools (such as *xm* tools in Xen). While the current implementation of vTurbo randomly selects the turbo cores, we can incorporate parameters such as cache affinity to further improve their performance.

In vTurbo, each VM is assigned a *turbo vCPU* in addition to its regular vCPUs. The turbo vCPU is assigned to one of the turbo cores in the host. This step is performed during VM initialization. For instance, if a user launches an SMP-VM configured with 2 vCPUs, the VM will have 3 vCPUs after initialization. Among these, the 0^{th} vCPU is the turbo vCPU, whereas the 1^{st} and 2^{nd} vCPUs are regular vCPUs.

Based on our empirical study (discussed in Section 2), we set 0.1ms as the CPU scheduling time-slice for turbo cores (as it enables the VM to reach up to 900Mbps TCP throughput for a 4 CPU-sharing VMs scenario). Since only interrupt processing runs on turbo cores, frequent context switches caused by the small turbo core time-slice does not affect the performance of interrupt processing much because of the very short duration of the processing. According to our measurements (Figure 4), when 4 VMs each running an iperf server share one turbo core, the order of magnitude of cache miss per second on

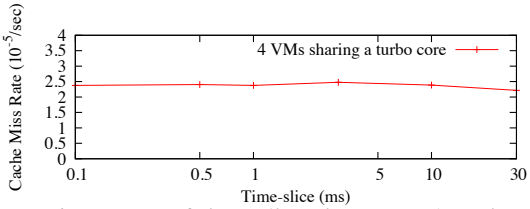


Figure 4: Impact of time-slice size on cache misses on turbo cores.

the turbo core is only 10^{-5}, which is negligible. The CPU time-slice for regular cores is set be a much larger value — 30ms in the current implementation which is the default time-slice of Xen. The vTurbo VM scheduler uses per-core scheduling timer to trigger scheduling code to select the next vCPU from the *runqueue*. We achieve CPU time-slice differentiation by setting these timers to 0.1ms for turbo cores and 30ms for regular cores.

Once the vCPUs are assigned to turbo cores and regular cores, our next concern is to correctly handle vCPU migration in the presence of turbo cores. vCPU migration allows to balance the CPU load among the available cores in the system. However, if we let the vCPUs to migrate freely among available cores, there is a possibility that a regular vCPU be migrated to a turbo core making undesirable effects. To solve this, we restrict migration of regular vCPUs to only among all regular cores and migration of turbo vCPUs only among turbo cores. We do not allow a turbo vCPU to migrate to a regular core or vice versa. This is done by changing each vCPU's affinity to the corresponding set of cores. Hence vTurbo scheduler not only determines the appropriate mapping between vCPUs and physical cores, but also ensures fair CPU sharing among all VMs.

VM scheduling policy Since we intend to use the turbo core only for I/O activity, we cannot treat it as a regular core and apply the existing scheduling policy to guarantee fair sharing among VMs. The challenge is to determine the CPU share of VMs for turbo and regular cores in the presence of heterogeneous workloads (*i.e.*, when a VM is CPU intensive, I/O-intensive, or both).

Current schedulers (*e.g.*, Xen's) use simple credit-based scheduling algorithm for achieving global load balancing and work-conservation. For instance, in Xen's credit scheduler, a VM is assigned some amount of credits periodically based on the priority of the VM. As the vCPUs belonging to a particular VM run on physical CPUs, credits are deducted from that VM. When the scheduler needs to make a decision, it uses the amount of available credits for each VM to decide which vCPU will run on the physical CPU. To accommodate turbo cores in our system, we mainly need to modify the credit assignment portion of the credit scheduling algorithm to account for the turbo vCPU execution time.

Specifically, assume l VMs are sharing an n-core host with m regular cores and n-m turbo cores. Let rd_i denote

the percentage demand for regular cores (CPU-bound component) and let td_i denote the percentage demand for turbo cores (I/O-bound component) for VM_i. We assume the demand for regular core and turbo core in two consecutive scheduling periods does not change much (if it does, we account for and adjust it in future rounds). So both rd_i and td_i are calculated based on the consumed CPU cycles by the VM in the previous scheduling period. Since our scheduler is work-conserving, the division of the total capacity among the regular and turbo cores is determined by the following:

$$C^R_{tot} = \sum_{i=1}^{l} rd_i \;\; and \;\; C^T_{tot} = \sum_{i=1}^{l} td_i$$

The total capacity demand of the system is:

$$C_{tot} = C^R_{tot} + C^T_{tot}$$

The fraction of CPU allocated for a VM out of this total capacity is determined by its assigned weight wt_i. Hence each VM's fair share (FS_i) of CPU is given by:

$$FS_i = (C_{tot} \times wt_i)/(\sum_{j=1}^{l} wt_j)$$

In vTurbo, we first allocate turbo core capacity fairly among VMs, as all of the VMs' IRQ processing is performed by the turbo vCPUs and starvation of turbo vCPUs (even for CPU-bound VMs) will result in application performance hit. So VM_i's fair share of the turbo core (FS^T_i) is calculated as:

$$FS^T_i = (C^T_{tot} \times wt_i)/(\sum_{j=1}^{l} wt_j)$$

Once VM_i's turbo core share is determined, we allocate the rest of its CPU share from the regular cores. The fraction of the allocation is given by:

$$FS^R_i = FS_i - \hat{FS^T_i}$$

where $\hat{FS^T_i}$ denotes the *actual* usage of the turbo core by VM_i in the previous scheduling period. We use FS^T_i and FS^R_i to determine the proportion of credits given to VMs out of total credits in the turbo core pool and regular core pool, for the next scheduling period. Table 1 shows the CPU allocation results from experiments with our prototype, where two VMs—with equal weight—share one regular core and one turbo core, under various workload demands. Columns 2 and 3 of the table indicate the CPU demand of each VM (*i.e.*, CPU utilization if they were run without CPU sharing); Columns 4 and 5 indicate measured consumption in the previous scheduling period; Columns 6 and 7 indicate the allocated shares of regular and turbo cores based on our policy; and Columns 8 and 9 show the *measured* consumption of regular ($\hat{FS^R_i}$) and turbo ($\hat{FS^T_i}$) core capacity in the next scheduling period. The results confirm that our policy allocates CPU with proportional fairness.

	Demand		Measured		Allocated		Consumed	
	Reg.	Turbo	rd_i	td_i	FS^R_i	FS^T_i	$\hat{FS^R_i}$	$\hat{FS^T_i}$
VM1	100	0	50	0	50	0	50	0
VM2	100	0	50	0	50	0	50	0
VM1	100	0	50	0	100	0	100	0
VM2	100	100	50	100	0	100	0	100
VM1	100	100	50	50	50	50	50	50
VM2	100	100	50	50	50	50	50	50
VM1	100	15	50	15	70	35	70	15
VM2	100	55	50	55	30	35	30	55

Table 1: VMs' CPU demand and allocated CPU shares under different scenarios

3.2 Modifications to Guest OS

Process scheduler As noted before, if CPU-bound workload were scheduled on the turbo cores, its performance would degrade due to frequent context switches. Since process scheduling inside the VM is transparent to the hypervisor's VM scheduler, we should make the guest OS's process scheduler aware of the turbo core to prevent user processes and non-I/O-related kernel threads from being scheduled on the turbo core. This can be achieved by setting scheduler affinity rule which sets the affinity of the non-I/O related threads to regular vCPUs. In Linux, this can be easily done by a scheduling mechanism known as Linux CPU isolation [3] (by setting a kernel parameter).

I/O buffers in guest OS With the above change, we can reduce IRQ processing delay to extremely small values. However, low IRQ processing delay by itself does not automatically translate into high I/O throughput, because of a critical *locking behavior* between the kernel and application threads as we explain below. The network receive path in typical OSes (*e.g.*, Linux) consists of two main steps: (1) Processing IRQ in kernel and buffering data in kernel buffer; (2) Application reading the data from kernel buffer and clearing it. Since the CPU time-slice of regular cores is still 30ms in vTurbo, the CPU access delay on the regular core will make the kernel buffer full very quickly and stop the IRQ threads from buffering more data, which would lead to poor I/O performance.

To address this problem and to keep the turbo vCPU busy processing IRQs, we need to tune the kernel buffer to store more received data while the application running on regular vCPU is blocked. As an example when 4 single-vCPU (excluding turbo vCPU) VMs are sharing one regular core, the CPU access delay is up to 90ms (*(4 - 1) × 30ms*). To keep the IRQ threads on turbo vCPU busy, all data received during this period need to be buffered. So if the bandwidth of NIC is B_N, the minimum kernel buffer required (B_{min}) is: $B_{min} = B_N \times Scheduling\_Delay$ (*i.e.*, the required kernel buffer is proportional to the number of VMs sharing the same CPU core). In fact, the real kernel buffer we need is

almost always much larger than B_{min}. For example, in our experimental environment with 1Gbps NICs, if 4 VMs share one CPU, the kernel buffer for UDP should be around 11.25MB. However, we did not obtain high throughput (more than 900Mbps) until we set the UDP kernel buffer (*net.core.rmem_max*) to about 40MB.

Algorithm 1 Generating ACK for Backlog Queue

```
1: rcv.nxt is the seq. number of expected packet for receive queue
2: bl.nxt is the seq. number of expected packet for backlog queue
3: seq is the seq. number of received packet
4: if backlog_queue is empty then
5:     if rcv.nxt ≥ bl.nxt then
6:         /* initial status or packets in backlog queue are all acked by
           process context */
7:         bl.nxt = rcv.nxt;
8:         bl.online = 1; /* enable ACK generation */
9: else
10:    if bl.online == 0 and bl.nxt ≤ rcv_nxt then
11:        /* packets in backlog queue are acked by process context */
12:        bl.online = 1; /* enable ACK generation */
13:        bl.nxt = rcv.nxt;
14: if bl.online == 1 then
15:    if bl.nxt == seq then
16:        /* packet to be added to backlog queue is in order */
17:        update(bl.nxt);
18:    else
19:        /* stop ACK generation due to out-of-order packet */
20:        bl.online = 0;
21: if add_backlog() is successful and bl.online == 1 then
22:    tcp_ack_backlog(); /* generate and send ACK */
```

Modifications to VM's TCP stack While simply setting the guest kernel buffer to a high value ensured good UDP performance, it did not improve TCP throughput at all. Upon a deeper investigation, we found the following problem: In TCP, when a data segment is received, the receiver generates an ACK to inform the reception of the segment. The sender uses this ACK to confirm the reception of data as well as for congestion control. Now, using the turbo core, we eliminate the long delay for processing incoming data segments. With our additional I/O buffering enabled, the IRQ context now buffers all these data packets. However, the locking behavior in the VM's TCP stack still prevents the ACK generation in a timely manner, hence reducing TCP throughput significantly.

Specifically, when the user process is calling function *recv()*, it locks the socket to prevent the IRQ threads from modifying the socket structure while it is reading from the socket buffers. If a new data segment arrives during this period, the IRQ process will queue it in the *backlog* queue without generating an ACK. When the receiving process engages in a tight receiving loop, the socket gets locked frequently by the process context. Moreover, the process can get scheduled out of the regular core while it is holding the lock. When this happens, ACKs will not be generated for a long period (until the process gets scheduled and releases the lock), even though the turbo core can accept and buffer TCP segments from the network. As a result, the sender will throttle down the sending rate leading to sub-par TCP throughput.

We make a simple modification to the VM's TCP stack to enable ACK generation from the IRQ context running in the turbo core, even when the socket structure is locked by the user process. The high-level steps performed by our modification are shown in Algorithm 1 which runs in the *softIRQ* context just before queuing the packet in the TCP backlog queue. Here, when the IRQ thread discovers that the socket is locked by the user process, it checks whether the new data segment is in-order. If so, an ACK is generated for the data packet, which will then be marked as acknowledged and queued. Note that we are not modifying the socket structure as it is currently owned by the process context. This is somewhat similar to vSnoop [17], although vSnoop is implemented purely in the driver domain whereas the ACK generation here is from within the guest VM. Thus we have access to VM's TCP information and can afford much larger buffers (compared to the limited ring buffer space in vSnoop). If a flow encounters an out-of-order packet, we disable this ACK generation until the missing segments are recovered by the usual slow path of TCP processing. This small modification helps achieve TCP throughput close to the line rate.

4 Implementation

We have implemented a prototype of vTurbo based on Xen 4.1.2. vTurbo only requires small modifications to the VM scheduler in hypervisor (about 400 lines of code) and guest OS kernel (less than 200 lines of code).

Hypervisor To differentiate between regular cores and turbo cores, we added a field to the per-core data structure *schedule_data*, to indicate the CPU time-slice for the specific core–30ms for regular cores and 0.1ms for turbo cores. Our implementation allows the flexibility of changing these values dynamically via *xm* tools.

Our vTurbo scheduler inherits most of its functionality from Xen's credit scheduler which provides the proportional fairness and work-conserving properties. We added and modified functionality of the main scheduler code of the credit scheduler to accommodate turbo cores and turbo vCPUs. Specifically, we modified function *csched_schedule()*, which is responsible for selecting vCPUs from the *runqueue* to run on physical cores and setting the scheduling timer of turbo cores to 0.1ms.

We assign each VM a turbo vCPU by modifying the VM's configuration so that an extra vCPU is added during the configuration parsing step of VM initialization performed by the Xen tools. Also during this step, the turbo vCPUs are pinned to the set of turbo cores and regular vCPUs are pinned to the regular cores by modifying the loaded VM's configuration. By doing this, we do not have to modify the scheduler code to prevent undesirable vCPU migrations (discussed in Section 3), because the credit scheduler will adhere to the CPU affinity rules set

in the configuration.

Algorithm 2 vTurbo accounting algorithm

Require: $num\_tcore \geq 1$
Require: $num\_rcore \geq 1$
Require: $num\_vm \geq 1$

Regular_accounting triggered every 30ms:
$tcore\_usage = get\_rcore\_usage()$; /* C^R_{tot} */
$rcore\_usage = get\_tcore\_usage()$; /* C^T_{tot} */
for vm in vm_list **do**
  $vm.credits = vm.weight \times$
  $(tcore\_usage + rcore\_usage)/vm\_weight\_sum$;
  $vm.tcredits = get\_turbo\_core\_usage(vm)$;
  $vm.rcredits = vm.credits - vm.tcredits$;
  $ratio$ = 300; /* = 30/0.1 */
  $vm.vturbo\_slice = vm\_vcredits/ratio$;
  $update\_rcredits(vm.rcredits)$;

vTurbo_accounting triggered every 0.1ms:
for vm in vm_list **do**
  $update\_tcredits(vm.vturbo\_slice)$;

CPU accounting is conducted by function *csched_acct()* in the credit scheduler. We extended this function by implementing two accounting routines for regular and turbo vCPUs individually as shown in Algorithm 2. They run at different frequencies in accordance with CPU scheduling frequencies (e.g., 30ms for regular vCPUs and 0.1ms for turbo vCPUs), because updating credits faster or slower than the scheduling frequency would cause inaccurate state of vCPUs in terms of *OVER* and *UNDER* priorities in Xen. The vTurbo accounting routines are simple, incurring very low overhead considering the high frequency of their execution. Functions *get_rcore_usage()* and *get_tcore_usage()* retrieve the consumed clock cycles by regular vCPUs and turbo vCPUs of all VMs respectively; while functions *update_rcredits()* and *update_tcredits()* set the calculated credits for regular cores and turbo cores for the next scheduling period. Function *get_turbo_core_usage()* retrieves the the clock cycle usage by the turbo vCPU of a given VM. We do not change method *burning_credits()* in the credit scheduler, which deducts credits from the VMs based on their running time on the cores. Instead we implement a new method for vTurbo credit deduction.

Guest OS Our modification to the TCP stack, to generate early ACKs for packets buffered in backlog queue, is mainly in function *tcp_v4_rcv()*. There are 3 kernel buffers to buffer received TCP packets: (1) receive queue, (2) prequeue, and (3) backlog queue. When a socket is not locked, received packets are buffered in receive queue. However, if the application process locks the socket while fetching data from the kernel, packets received during that period will be buffered in backlog queue. We modified the backlog queuing path of function *tcp_v4_rcv()* to verify a received packet is "expected" and if so, call function *tcp_ack()* to generate an ACK for the received packet. Since very few packets (less than 0.1%) go to prequeue in CPU sharing VMs, we disable prequeue in vTurbo to simplify our implementation.

5 Evaluation

We first evaluate the effectiveness of vTurbo for different types of I/O operations via a series of micro-benchmarks. We then use NFS, SCP, and Apache Olio [2] to evaluate the application-level performance improvement by vTurbo.

Experimental setup Our testbed consists of servers with quad-core 3.2GHz Intel Xeon CPUs and 16GB of RAM. They are connected via Gigabit Ethernet, except for the experiments with 10Gbps Ethernet. These servers run Xen 4.1.2 as hypervisor and Linux 3.2 in both domain0 and guest VMs. We pin domain0 to one of the cores in all our experiments.

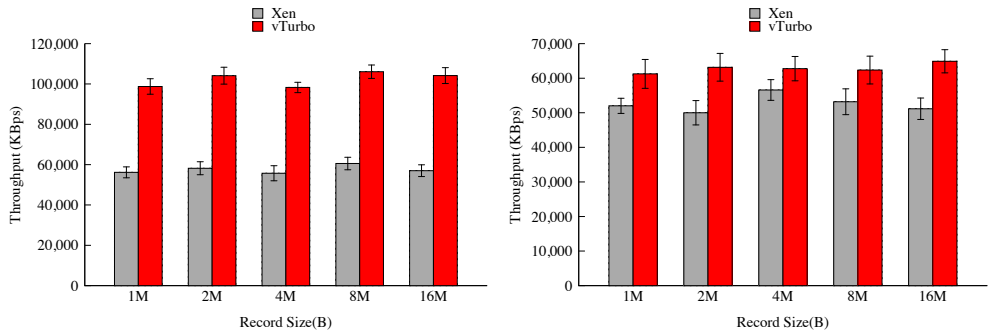
5.1 Micro-Benchmark Results

In this section we evaluate the performance of vTurbo for various types of I/O. We use *lookbusy* [7] to keep the CPU utilization at determined levels during experiments.

File read and write We use IOzone benchmark [4] to read/write a 1GB file from/to disk and measure the read/write throughput. Figure 5 shows the read and write throughput—in comparison with the vanilla Xen–when we vary the record size from 1MB to 16MB.

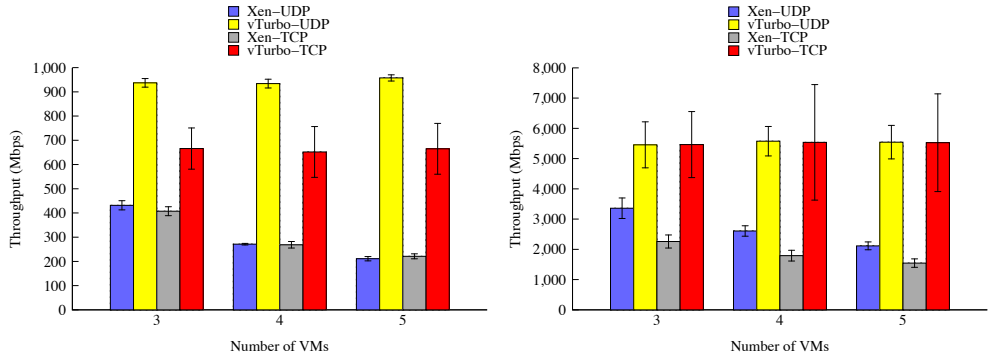
From Figure 5(a) we see that the disk write throughput is improved significantly (by 75% to 82%); whereas the disk read throughput (Figure 5(b)) sees less improvement (only up to 26%). The main reason is that, when the process performs a write, the data is immediately written to the file system cache and the *write()* call returns. So the process can keep writing while the regular vCPU is scheduled. The dirty pages of the disk cache are flushed to the disk by a kernel thread executed by the turbo vCPU. Therefore with vTurbo, disk write throughput is greatly improved. However, when the process performs a read for a fresh block from the disk, it gets blocked until the actual data blocks are read from the disk. Meanwhile the hypervisor may schedule other vCPUs on the regular core. The turbo vCPU will be able to handle the disk read completion interrupt and place the data in the process' buffer while the regular vCPU is scheduled out. But the process will not be able to make further read requests until it is scheduled again. Hence in this case, vTurbo achieves less throughput improvement than in the case of disk write.

UDP throughput To measure the benefit of vTurbo to network I/O we first measure the UDP throughput improvement achieved by vTurbo. In these experiments, we use iperf to send a stream of UDP packets for 10 seconds to a VM sharing a core with 2, 3, or 4 other VMs. The average throughput (averaged over 10 runs) observed at the



(a) Write (b) Read

Figure 5: File read/write throughput.



(a) 1Gbps Network (b) 10Gbps Network

Figure 6: TCP and UDP throughput.

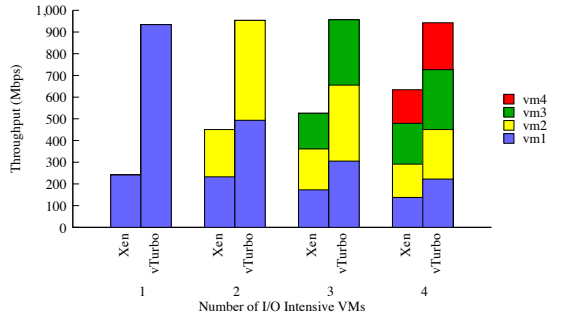


Figure 7: UDP throughput: multiple I/O-intensive VMs.

VM on vanilla Xen and vTurbo is shown in Figure 6(a) by blue and yellow bars, respectively. With vanilla Xen, the UDP throughput starts to decrease when the number of VMs sharing the core increases. This is because, when UDP packets arrive at domain0, the target VM may not be scheduled and the packets have to be buffered in domain0. But the space in domain0 is limited and hence once this buffer fills up, packets will be dropped causing the throughput to go down. With vTurbo, the target VM's network IRQ processing threads get scheduled frequently and hence the buffer in domain0 can be drained frequently. This leads to much less packet drops thereby achieving close-to full network bandwidth (1Gbps).

Next, we evaluate the impact of sharing the turbo core among multiple I/O-intensive VMs. We reuse the setup in the previous experiment. But instead of 1 VM receiving a UDP packet stream, we increase the number of VMs receiving UDP streams from 1 to 4. Figure 7 shows the aggregate throughput achieved as well as the throughput seen by individual VMs. In both vanilla Xen and Xen with vTurbo configurations, we see that the I/O bandwidth is fairly shared among VMs. However, vTurbo achieves (close to) wire speed and outperforms vanilla Xen irrespective of the number of I/O-intensive VMs.

TCP throughput We use a setup similar to the UDP experiments to measure the TCP throughput improvement achieved by vTurbo. In this experiment, we send a 200MB file using iperf to a VM from another server and we vary the number of VMs sharing the same core with the receiving VM. Figure 6(a) shows the TCP throughput on vanilla Xen and Xen with vTurbo by grey and red bars, respectively. Recall that with vTurbo, the TCP stack is modified to generate ACKs when the regular vCPU is holding the socket ownership and scheduled out. As the figure shows, vTurbo improves TCP throughput significantly (by 63% - 200%). However the TCP throughput achieved by vTurbo still does not reach the full available network bandwidth. The reason is, even with our modification, if a packet loss happens, we have to resort to the (usual) slow code path where packet loss recovery is subject to regular vCPU scheduling delay,

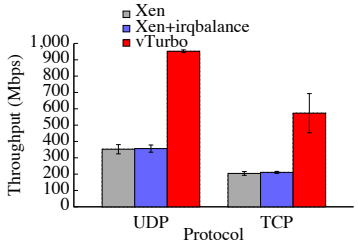
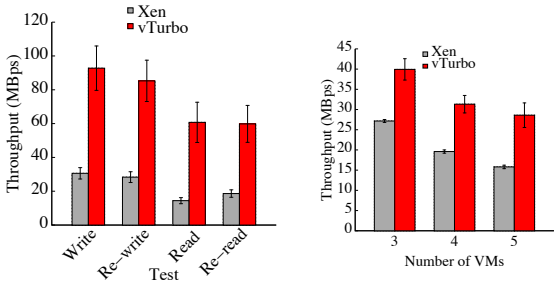


Figure 8: UDP and TCP throughput for VSMP VMs.

which negatively affects the TCP throughput.

10Gbps Ethernet To evaluate the benefit of vTurbo with 10Gbps Ethernet, we repeat the UDP and TCP experiments. In our setup, two physical servers are connected via 10Gbps Ethernet. In the UDP experiment, we use netperf [9] to send a 10-second UDP stream to the target VM sharing a core with 2 to 4 other VMs. In the TCP experiment, we send a 500MB file using iperf from one physical server to a VM running in the other server, varying the number of VMs sharing the same core with the receiving VM. The results in Figure 6(b) indicate that, in a 10Gbps network, vTurbo achieves a pattern of improvement for both UDP and TCP throughput similar to that in the 1Gbps network. However, since the regular core is shared by multiple VMs, the application does not get enough CPU cycles to copy the buffered data from kernel space to user space, hence we can not achieve line speed.

Benefit of vTurbo to VSMP VMs To show the benefit of vTurbo to SMP VMs, we use iperf to send TCP and UDP traffic (in different runs) to a VM which is assigned 2 vCPUs. In this experiment, we run 4 VMs each with 2 vCPUs. These vCPUs are restricted to run in the first 2 cores of the quad-core processor, but are allowed to migrate between the two cores. Similar to previous experiments, we pin domain0 to the 3rd core and, for vTurbo, we use the 4th core as the turbo core. In the vanilla Xen configuration, we first disable *irqbalance* in VM and allow the interrupts to be directed only to vCPU0 of the VM. Next we enable *irqbalance* so that the interrupts can be balanced between the two vCPUs. In the vTurbo configuration, interrupts are routed to the turbo vCPU. Figure 8 shows the TCP and UDP throughput when transferring 200MB of data to the VSMP VM. vTurbo vastly outperforms both *irqbalance*-on and *irqbalance*-off configurations. However, the TCP throughput is lower than that under the "4 single-vCPU VMs" configuration (for both vanilla Xen and vTurbo configurations – see Figure 6(a)) . We conjecture that this is due to the vCPU migrations between the two physical cores and the iperf receiver process migrations between the two vCPUs of the VSMP VM.



(a) NFS read/write throughput. (b) SCP throughput.

Figure 9: SCP and NFS throughput.

5.2 Application-Level Results

NFS server throughput NFS uses TCP to transport commands and data blocks between the NFS client and server. We use the NFS tests in IOzone to evaluate the benefit of vTurbo to the NFS server. We export a directory of a VM using NFS and run IOzone in another server which mounts this exported directory. We pin the NFS server VM's vCPU to a single core shared by three other VMs with 30% CPU utilization. Figure 9(a) shows the file read and write throughput (file size: 1GB). vTurbo significantly outperforms vanilla Xen for all types of operations. The results for "Read" and "Re-read" operations are especially interesting (and somewhat surprising). Recall that, for file read/write micro-benchmarks, vTurbo does *not* improve disk read throughput much. Yet we observe significant improvement in NFS read and re-read throughput. After some investigation, we figure out the reasons for the improvements here: First, NFS utilizes *pre-fetching* for sequential read operations where multiple read operations are issued in advance. Second, Linux NFS implementation uses *in-kernel* data transfer from files to sockets. As such, the server process is able to process many read requests while the regular vCPU is scheduled and to delegate the actual file block transfer operations to the kernel threads run by the turbo vCPU, hence achieving much higher throughput.

Secure copy (SCP) throughput SCP involves both CPU activity (for encryption and decryption of data) and I/O activity. We copy a 1GB file using SCP from a client to a VM sharing a core with 2, 3, or 4 other VMs. In this experiment the *sshd* process which is receiving the file is scheduled at the regular vCPU while *both* TCP processing and disk I/O handling threads are scheduled at the turbo vCPU. Figure 9(b) shows that vTurbo improves SCP throughput by 53% to 66%.

Apache Olio To assess the benefit of vTurbo to a cloud application, we use Apache Olio, an event calender developed using Web 2.0 technologies. The Apache Olio benchmark consists of 3 components: (1) a web server to process user requests, (2) a MySQL database server to

	Single Instance		Two Simultaneous Instances			
			Instance 1		Instance 2	
Operation	Count Vanilla Xen	Count vTurbo	Count Vanilla Xen	Count vTurbo	Count Vanilla Xen	Count vTurbo
HomePage	4028	5602	3918	5334	3839	5311
Login	1629	2190	1524	2121	1540	2109
TagSearch	5183	7198	4888	6822	4892	6778
EventDetail	3856	5274	3701	5075	3630	5013
PersonDetail	405	562	379	550	381	508
AddPerson	127	178	131	177	120	167
AddEvent	300	402	280	416	279	413
Total	15528	21406	14821	20495	14681	20299
Rate(ops/sec)	51.8	71.3	49.4	68.3	48.9	67.7
Improvement (%)	-	**37.6%**	**-**	**38.2%**	**-**	**38.4%**

Table 2: Results from Apache Olio experiment (single- and two-instance)

store user profiles and event information, and (3) an NFS server to store images and documents specific to events. We use the PHP version of the benchmark.

In our setup, we host the 3 Olio components in 3 different VMs each in a separate physical host. In each host we pin the Olio VM's vCPU to a single core, which is shared by 3 other VMs having 20% of CPU load. We stress the Olio service with 400 client threads generating requests using the Faban client simulator for 6 minutes.

In Table 2, the "Single Instance" (2^{nd} and 3^{rd}) columns show the breakdown of total operations (averaged over 3 runs) performed by Olio on vanilla Xen and on vTurbo, respectively. vTurbo achieves higher operation counts than vanilla Xen for all types of operations during the same period. This is because vTurbo improves communication performance among the three Olio components as well as file write performance of MySQL and NFS servers. With vTurbo the overall throughput of the Olio service is improved from 51.8 ops/second to 71.3 ops/second – a 37.6% improvement.

Next, we evaluate the performance of *two* simultaneous instances of Olio, with the same set of components hosted by the same physical servers. In this experiment, of the 4 CPU cores of each server, we dedicate one core to domain0 and one core as the turbo core shared by all VMs. In our replicated Olio configuration, we pin the two copies of each Olio component to the 2 remaining cores respectively, with each core shared by 3 other VMs. Columns 4, 5, 6, 7 of Table 2 show the breakdown of total operations performed by the two Olio instances, which are started at the same time and run for the same 6-minute period. Compared with the "Single Instance" results, most rows see a slight reduction of operation throughput for both vanilla Xen and vTurbo configurations. We believe this is due to the sharing of resources such as the disk and network. However, we observe that with vTurbo, the overall Olio throughput is increased by 38.2% and 38.4% for instances 1 and 2, respectively.

6 Related Work

We have discussed some of the recent and most related efforts in optimizing I/O processing for virtualized systems in Section 1 and Section 2. In this section, we discuss other related work in the same problem space. These efforts can be categorized into two categories: I/O path tuning and VM scheduling optimization.

I/O path improvements vSnoop [17] and vFlood [12] are two related efforts to improve TCP throughput of VMs. vSnoop offloads ACK generation from a VM to its driver domain to hide the VM scheduling delay from the TCP sender. We adopt a similar idea in vTurbo*inside* the guest OS to generate ACKs while the receiving process on the regular core has locked the socket. vSnoop only benefits TCP receiving, it does not benefit UDP or disk I/O. Moreover, due to the limited shared buffer space in the driver domain, vSnoop can only accelerate *small* TCP flows. On the other hand vTurbo can improve throughput of TCP/UDP receive—regardless of flow size—and disk write. It can also benefit disk read if data pre-fetching is used by applications (as shown by the NFS throughput results in Section 5.2). One can consider vTurbo as an alternative to vSnoop with extra features. vFlood offloads TCP congestion control to driver domain to hide VM scheduling delay from receiver thus improving the performance of TCP send. vFlood has the same problem as vSnoop: It only works for small TCP flows. IsoStack [27] offloads the entire TCP processing engine to a dedicated core. The main advantage of IsoStack is that, it can reduce cache misses and reduce synchronized accesses to shared state of the TCP stack by multiple cores (*e.g.*, socket structures). vTurbo in spirit offloads IRQ processing (only) to a separate core, with the goal of mitigating the impact of VMs' regular core access latency. In [24, 22, 23], Menon *et al.*propose optimizations for device virtualization using techniques such as checksum offload, segmentation offload, packet coalescing, scatter/gather I/O, and offloading device driver functionality. SR-IOV [11] devices and IOMMUs such as Intel VT-d [14] enable the hypervisor to directly assign devices to guests. This allows the guest to directly interact with the device eliminating the virtulization overhead. However, even in this case, scheduling delays still impact the interrupt processing delay. We believe that vTurbo is complementary to both SR-IOV and VT-d, since it enables the

processing of the interrupt and data associated with the interrupt as soon as the interrupt is delivered to the VM.

VM scheduling optimization Adapting the scheduling policy in the hypervisor-level VM scheduler can also improve I/O performance perceived by applications running in VMs. vSlicer [30] reduces CPU scheduling delay and hence the application-perceived latency—to a certain degree by setting smaller time-slice for latency-sensitive VMs. Still, that time-slice is not small enough to improve TCP/UDP throughput in LAN/datacenter environments. With vTurbo, IRQ processing delay is reduced to sub-millisecond. And the concurrent I/O processing on regular and turbo cores brings significant I/O throughput improvement. We note that vTurbo and vSlicer can be *integrated* to achieve both low latency and high throughput for VM I/O. A soft-realtime VM scheduler is proposed in [21] to reduce response time of I/O requests thus improving the performance of soft-realtime applications such as media servers. But its preemption-based scheduling policy may violate CPU share fairness when a VM is performing heavy I/O activities. MRG [16] is a VM scheduler to improve I/O performance for MapReduce jobs. This scheduler facilities MapReduce job fairness by introducing a two-level group credit-based scheduling policy. Through batching of I/O requests within a group the efficiency of map and reduce tasks is improved and superfluous context switches are eliminated. However MRG is a MapReduce-specific scheduler; and it works well only when the VMs and the driver domain share the same CPU core.

7 Conclusion

We have presented vTurbo, a system that aims at accelerating I/O processing for VMs sharing the same core in a multi-core host. More specifically, vTurbo significantly reduces IRQ processing latency by dedicating one or more turbo core(s) to IRQ processing for all hosted VMs. The time-slice of a turbo core is magnitudes smaller than that of a regular core hence achieving negligible IRQ processing latency. vTurbo involves a VM scheduling policy that enforces fair sharing of both regular and turbo cores among VMs. Our evaluation of a vTurbo prototype shows that it vastly improves network and disk I/O throughput and consequently application-level performance for hosted VMs.

8 Acknowledgments

We thank our shepherd, Jonathan Walpole and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF grants 0855141, 1054788, and 1219004. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Apache Olio. `http://http://incubator.apache.org/olio/`.

[3] CPU isolation extensions. `http://lwn.net/Articles/270623/`.

[4] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[5] The Iperf Benchmark. `http://www.noc.ucf.edu/Tools/Iperf/`.

[6] J. McCalpin. The STREAM benchmark. `http://www.cs.virginia.edu/stream/`.

[7] Lookbusy-a synthetic load generator. `http://www.devin.com/lookbusy/`.

[8] Microsoft Cloud Platform (Microsoft Azure). `http://www.windowsazure.com/`.

[9] The Netperf Benchmark. `http://www.netperf.org`.

[10] CHENG, L., AND WANG, C.-L. vbalance: Using interrupt load balance to improve i/o performance for smp virtual machines. In *ACM SoCC* (2012).

[11] DONG, Y., YU, Z., AND ROSE, G. SR-IOV networking in Xen: architecture, design and implementation. In *WIOV* (2008).

[12] GAMAGE, S., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *ACM SoCC* (2011).

[13] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS* (2012).

[14] HIREMANE, R. Intel virtualization technology for directed I/O (Intel VT-d). *Technology@ Intel Magazine 4*, 10 (2007).

[15] HU, Y., LONG, X., ZHANG, J., HE, J., AND XIA, L. I/o scheduling model of virtual machine based on multi-core dynamical partitioning. In *ACM HPDC* (2010).

[16] KANG, H., CHEN, Y., WONG, J. L., SION, R., AND WU, J. Enhancement of Xen's scheduler for MapReduce workloads. In *ACM HPDC'11* (2011).

[17] KANGARLOU, A., GAMAGE, S., KOMPELLA, R. R., AND XU, D. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE SC* (2010).

[18] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: predictable low latency for data center applications. In *ACM SoCC* (2012).

[19] KESAVAN, M., GAVRILOVSKA, A., AND SCHWAN, K. Differential Virtual Time (DVT): Rethinking I/O service differentiation for virtual machines. In *ACM SoCC* (2010).

[20] LARSEN, S., SARANGAM, P., HUGGAHALLI, R., AND KULKARNI, S. Architectural breakdown of end-to-end latency in a tcp/ip network. *International Journal of Parallel Programming 37*, 6 (2009), 556–571.

[21] LEE, M., KRISHNAKUMAR, A. S., KRISHNAN, P., SINGH, N., AND YAJNIK, S. Supporting soft real-time tasks in the Xen hypervisor. In *ACM VEE* (2010).

[22] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *USENIX ATC* (2006).

[23] MENON, A., SCHUBERT, S., AND ZWAENEPOEL, W. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ACM ASPLOS* (2009).

[24] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *USENIX ATC* (2008).

[25] PATNAIK, D., KRISHNAKUMAR, A., KRISHNAN, P., SINGH, N., AND YAJNIK, S. Performance implications of hosting enterprise telephony applications on virtualized multi-core platforms. Tech. rep., IPTComm, 2009.

[26] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *ACM EuroSys* (2012).

[27] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly efficient network processing on dedicated cores. In *USENIX ATC* (2010).

[28] WALDSPURGER, C., AND ROSENBLUM, M. I/O virtulization. In *Communications of the ACM* (2012).

[29] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *USENIX OSDI* (2002).

[30] XU, C., GAMAGE, S., RAO, P. N., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. vslicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *ACM HPDC* (2012).

When Slower is Faster: On Heterogeneous Multicores for Reliable Systems

Tomas Hruby Herbert Bos Andrew S. Tanenbaum
The Network Institute, VU University Amsterdam
{*thruby,herbertb,ast*}*@few.vu.nl*

Abstract

Breaking up the OS in many small components is attractive from a dependability point of view. If one of the components crashes or needs an update, we can replace it on the fly without taking down the system. The question is how to achieve this without sacrificing performance and without wasting resources unnecessarily. In this paper, we show that heterogeneous multicore architectures allow us to run OS code efficiently by executing each of the OS components on the most suitable core. Thus, components that require high single-thread performance run on (expensive) high-performance cores, while components that are less performance critical run on wimpy cores. Moreover, as current trends suggest that there will be no shortage of cores, we can give each component its own dedicated core when performance is of the essence, and consolidate multiple functions on a single core (saving power and resources) when performance is less critical for these components. Using frequency scaling to emulate different x86 cores, we evaluate our design on the most demanding subsystem of our operating system—the network stack. We show that *less is sometimes more* and that we can deliver better throughput with slower and, likely, less power hungry cores. For instance, we support network processing at close to 10 Gbps (the maximum speed of our NIC), while using an average of just 60% of the core speeds. Moreover, even if we scale all the cores of the network stack down to as little as 200 MHz, we still achieve 1.8 Gbps, which may be enough for many applications.

1 Introduction

More and more hardware vendors are developing heterogeneous multicore architectures. Well known examples include the so-called *big.LITTLE* [1] ARM, the NVIDIA Tegra-3 [2], its recently announced successor Tegra 4, and the x86-compatible Xeon Phi [4]. The *big.LITTLE* ARM combines two big Cortex-A15 cores with two little Cortex-A7 on the same die, and Samsung recently announced a 4 + 4 version [5]. The Tegra-3 is a Cortex-A9-based quad-core CPU that includes a fifth "companion" Cortex-A9 that is slower (capped at 500MHz) and less power hungry. For sheer number of cores, the 50+ core x86-compatible Intel Xeon Phi processor is especially impressive. It serves as an extension of many little cores to accompany the host's big cores and lives on a separate PCIe card.

In all three cases, the different cores share a large subset of the instruction set architecture (ISA), so that the same code can easily run on any of the cores in the system. The main difference of the cores is their microarchitecture which is designed for different optimal operation points. This means that the *LITTLE* slower, simpler, and in-order cores (designed for power efficiency at low frequencies) cannot deliver performance equal to the *big* ones which are out-of-order and operate at higher frequencies. The same is true for the Tegra and Xeon Phi. For instance, the host x86 processors feature out-of-order cores with a deep pipeline while the Xeon Phi cores are much simpler, in-order Pentium cores with shallow pipelines to allow for efficient 4-way hyper-threading. In addition, they feature new vector instructions to support scientific workloads.

The research community has advocated such heterogeneity for many years [15] to make processing more efficient, in terms of both performance and power. However, the focus was primarily on applications, leaving the operating system by the wayside. This is remarkable, because the operating system performs a significant amount of work on behalf of the applications [23, 21].

Moreover, the changes to the system remain mostly limited to making execution on different cores possible and to finding the best schedule. Exceptions include the proposal by Strong et al. [28] to migrate long-running system calls to *system cores*—cores more suitable for running OS workloads. FlexSC[26], meanwhile, aims to remove the overhead of switching between applications

and the system by running each on different cores. As a side effect, the system can run on core(s) that differ from those that host applications.

The NewtOS operating system described in this paper is a UNIX-like multiserver system that offers these major benefits:

High reliability For instance, our operating system supports dynamic updates without any downtime and survives crashes of core OS components. We described these aspects in [11, 7], and [10], and will not discuss them further in this paper.

High performance Building on a design described in [11], we show that we support network processing at 10 Gbps on COTS hardware (this paper).

The unique features of multiserver systems, composed of independent user space servers, typically trade reliability for performance. With many processes involved, the communication and context switching to share the processor constitute a significant overhead. NewtOS [11], a high-performance derivative of MINIX 3, shows that it is possible to mitigate this overhead by dedicating cores to system servers, which communicate through asynchronous channels. With their own cores, the system servers can run whenever needed from a warm cache, and without having to compete with other processes or wait for the kernel to schedule them. Moreover, the system's asynchronous communication channels allow the system servers to work independently and thus increase the parallelism within the system and streamline the processing. As a result, we were able to support TCP-based network processing at up to 5 Gbps [11]. Since then, we have further improved our design. We will show in Section 4 that we now support TCP at close to 10 Gbps—the maximum speed of our network card.

The cores of common platforms are designed for generic usage and over-provisioned [20] for running OS code. Dedicating cores results in a very coarse grained resource assignment, which leads to inefficient use of the available hardware. Looking at current trends, we anticipate more designs in the *big.LITTLE* fashion, which will have plenty of smaller, slower, in-order cores with a higher number of threads, accompanied by big, fast cores that can efficiently use the instruction level parallelism of application code. However, the big cores will become a minority.

In this paper, we explore how such architectures can help to balance performance and resource consumption. Specifically, we show that we can run the OS components on multiple slower cores, while still achieving high performance. Alternatively, the system can consolidate components on a few cores (saving power and resources) and still achieve reasonable performance.

Our contributions are:

1. We explore the hardware design space by emulating the future platforms on current hardware using frequency scaling to find out how fast the processors should be and what type of cores would suit systems the best.
2. We show that our system can deliver the same or better performance with smaller, simpler and slower cores—without compromising reliability. Our case study shows it is suitable for high-speed networking.
3. The system has a potential to dynamically reconfigure itself to use the most appropriate resources and free resources it does not need for a particular workload.

In the rest of the paper we discuss our motivations and the background in Section 2. We present details of the NewtOS design in Section 3. We explore the design space and evaluate various setups of our system in Section 4 and we put it in perspective of related work in Section 5. Finally, we conclude in Section 6.

2 Big cores, little cores and combinations

Heterogeneous processor architectures are rapidly becoming popular. In this section, we focus on Intel products and sketch some of the properties of the architectures and analyze some trends in this field.

2.1 BOCs and SICs

We start our discussion with a comparison of fast cores and slow cores. Specifically, the first two columns of Table 1 compares the Intel Core i7 "Bloomfield" with the Knights Ferry processor. The Core i7 is a prime example of a big out-of-order core (BOC) with a design that is geared for high single threaded throughput and produced by 45nm technology. In contrast, the cores on the Knights Ferry (45nm) are much simpler in-order cores (SICs) that provide only a fraction of the i7's performance.

Given the estimated die size of the Knights Ferry, the table shows the space reduction of the simple cores compared to the big i7 cores. Compared to the i7, the Knights Ferry die hosts 8× the number of cores and 16× the number of threads. It is worth noting that the difference in die size per core is 3× (and 6× per thread). While the cache size per core is obviously smaller, threaded cores can compensate for this [22]. Finally, the difference in the peak clock speed is equally remarkable.

The last column of Table 1 shows data for the successor of Knights Ferry, a recently released product called Xeon Phi. Its core count is even larger, but its die size is not public. Intel markets it as a "50+ core beast" and released up to 62 cores on a single die. With each core hosting

	Core i7 Bloomfield (45nm)	Knights Ferry (45nm)	Xeon Phi (22nm)
Die size	$263mm^2$	est. $700mm^2$	not released
Cores / threads	4 / 8	32 / 128	50+ / 200+
Die area per core / thread	$65.7/32.9mm^2$	$21/5.5mm^2$	not released
Clock speed	max 3.33 GHz	1.2 GHz	1 GHz
LL cache size	8 MB	8 MB	25+ MB
	out-of-order	in-order	in-order

Table 1: Comparison of Core i7, Knights Ferry and Xeon Phi

	Transistor count	Die size
4-core i7 + GPU	1.4 bil.	$160mm^2$
62-core Xeon Phi	5 bil.	not released

(a) Transistor count

# i7 cores	# i7 threads	# Phi cores	# Phi threads
4	8	44	176
8	16	27	108
12	24	10	40

(b) Core i7 and Xeon Phi configurations

Table 2: Given the known transistor counts shown in (a) and a 22nm production process, we can roughly project the options for different configurations that merge Core i7 and Xeon Phi cores (b)

4 threads of execution, this amounts to 200+ threads on a single chip. Interestingly, Xeon Phi is still a cache coherent design, unlike one of its research predecessors—the single chip cloud (SCC).

2.2 Configurations of BOCs and SICs

It is likely that future designs will see interesting new combinations of BOCs and SICs. For instance, rather than keep it as a separate co-processor, Intel may well merge its Xeon Phi with other Intel cores on a single die [17], in the same way that GPUs and general purpose cores have merged on a single die. What sort of processor should we expect? Clearly, there are many options. In this section, we explore possible combinations in an approximate manner.

Table 2 shows different combinations of Core i7 "Sandy Bridge" and Xeon Phi cores, taking into account the number of transistors for a quad-core Core i7 in 22nm technology as well as the number of transistors of a 62-core Xeon Phi produced by the same technology. The Core i7 die may also contain an integrated GPU. Given these transistor counts, Table 2b shows different configurations that would fit on a die with mixed cores. The simple division also accounts for each core's cache share as caches take up a big portion of the die size. Each line represents a configuration with the 5 billion transistor budget of Xeon Phi die where some of the 62 cores are replaced by i7 cores.

Finally, for the sake of completeness, Table 2b also shows the resulting number of threads. The number of threads matters, because we will see that for OS functionality it is often not needed to dedicate a full core to each component. Instead, a simple container for a process' context is good enough, as long as we can let the hardware do the context switching (and not the software) and we can suspend and resume efficiently with instructions like MWAIT and MONITOR. A hardware thread is well-suited to serve as a container. It has a set of replicated registers and, depending on the architectures, hardware switches the threads automatically when the active one stalls or it tries to schedule a different thread every cycle.

As consolidating multiple components on a single core saves resources, more threads are attractive. Moreover, many platforms today still do not offer enough cores for us to be able to dedicate one to each component—although the number of cores per die is growing steadily. By using threads instead, we can implement our design even on today's platforms.

For instance, NewtOS has about 30 system processes in its default installation out of which about 10 are important for performance. These include the process manager, the memory manager, the storage stack, the network stack, the file systems, the disk and the network drivers. The calculation in Table 2b shows that even with twelve i7 cores, there would be enough threads to dedicate one to each of our system's processes. Note that based on previous research [20, 18], the Xeon Phi cores are most likely a better match for the system processes than the i7 cores. The platform would therefore still offer plenty of big cores for applications, while the small multithreaded cores would optimize the resources for the system.

An alternative to chips preconfigured with a fixed num-

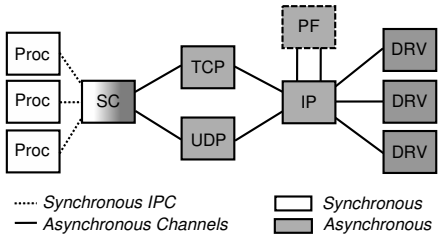


Figure 1: Design of the NewtOS network stack

ber of BOCs and SICs, are architectures that can merge smaller cores into bigger ones [12]. Another example is MorphCore [13], an architecture which can switch a core type between a i7-like BOC and a heavily threaded Phi-like SIC in run time.

3 NewtOS

The crux of this paper is the following: while evaluating the performance of the stack, we realized that it actually delivered higher throughput when we scaled the frequency of some cores down. In addition, we found that the performance of fairly slow cores is good enough for many use cases. We present an OS that explicitly exploits these properties.

Specifically, NewtOS is a high-performance clone of MINIX 3 that provides the same reliability with much reduced overhead. For instance, we completely redesigned the network stack based on new communication principles that allow different components to execute independent request simultaneously. This distinguishes NewtOS from other multiserver operating systems and increases the network throughput from hundreds of megabits per second to gigabits.

3.1 The NewtOS network stack

The heart of the NewtOS network stack is LwIP [8], a simple and portable network stack for embedded systems used by many research projects. Note that this stack is not designed for high performance but rather for its small memory footprint. As a result, its performance is not directly comparable to highly tuned stacks of commodity systems. Nevertheless, we support network processing at 10 Gbps even though we use slow cores.

One of the main design goals of NewtOS is reliability. Thus, we allow even core components of the operating systems to be replaced on the fly, without taking the system down (and often with no noticeable disruption at all) [9]. For instance, we can replace our implementation of IP or the network driver while keeping all existing network connections. Similarly, if one of its components crashes, the OS recovers automatically and often transparently [11].

To make this possible, we split the stack into several components (TCP, UDP, IP, drivers and packet filters) to reduce the chance that an error in the stack may lead to a crash of the entire stack. Likewise, we isolated functions that are easy to restart from those which are not due to large dynamic state. Besides IP, TCP, and UDP, the network stack supports an optional BSD packet filter (PF). The syscall server is the component that provides a POSIX interface to user processes. Figure 1 shows individual parts of the stack.

All shaded components in Figure 1 are fully asynchronous, while the syscall server translates synchronous system calls from user processes to asynchronous messages within the stack. The syscall server is the only process of the stack which frequently uses traditional rendez-vous based communication provided by the kernel. All other components communicate using point-to-point channels, which are shared user space memory queues accompanied by fast signaling. This mechanism is located almost purely in user space to take the kernel out of the loop (removing all overhead due to context switches, and pollution of TLBs, caches, and branch predictors).

We take advantage of the x86-specific MWAIT instruction to suspend execution of cores. Thus, we need not send high-overhead interprocessor interrupts, but wake up a waiting core by a mere memory write. Unfortunately, MWAIT is a privileged instruction in Intel chips[1]. If it were not, there would be no need for the kernel for normal mode of operation. We see it as a hardware deficiency.

Our most efficient communication model runs each component on its own dedicated core, so scheduling is not needed and the component can run at anytime out of a warm cache. However, we also allow components to share a core with other processes. In such a case, the scheduler informs the components and they transparently fallback to *notifications*, a standard method provided by microkernels. It delivers special void messages in a similar way to how devices send interrupts to the processor.

It is useful to emphasize that the key performance problems that plagued multiserver systems in the past have been the high overheads due to context switching and scheduling. While the research community heavily optimized the interprocess communication on microkernels like L4 [19] to achieve much better performance, neither of these bottlenecks could ever be eliminated on a uniprocessor. However, dedicating a core to each component fixes both. Further details of the design of the network stack and the fast communication are discussed in [11].

3.2 Dynamic reconfiguration

In contrast to monolithic systems, NewtOS resembles a distributed system. Such systems can embrace diversity

and accommodate to a changing environment. This is also true for NewtOS. Each system component can run on a dedicated core or share it with other process and the core can be either big or little. Although we dedicate cores for peak performance, we can consolidate processes on a single core or even a thread if they are not used heavily.

For instance, most of today's traffic uses the TCP protocol, and dedicating a core to the UDP component is probably overkill. On the other hand, when UDP is used heavily (e.g., for video streaming), NewtOS can migrate UDP to its own core. Similarly, the network stack is not used at all times in many deployments, or at least not at its peak throughput. Thus, we can pin all its components to a single dedicated core, or even have it share a core with other processes most of the time. When the workload changes and the system detects an overload of some cores, it can redistribute itself to find the best configuration.

We argue that in many cases SICs are best for the operating system, while BOCs are better suited for applications. However, this is not true always and depends on the situation. For instance, for a web server-like workload, running the server processes on threaded SICs probably delivers higher throughput than using a small number of threads on BOCs.

Even if it allows the stack to run at high speeds, dedicating the BOCs to the network stack is probably not a good idea and the resources can be used more efficiently, unless we run, for example, a complicated intrusion detection algorithm in the packet filter. Likewise, it seems unwise to sacrifice the BOCs to the storage stack. Storage needs to do many unpredictable lookups with little instruction level parallelism while briskly delivering data to the applications and writing them back to a disk. It is likely that we can do so on slower cores, saving the BOCs for the applications. Phrased differently, the components of the system should get the resources they need and no more.

Besides good performance, power consumption is also important. Here also, we should provision a system for its peak performance, while using no more resources than needed during quieter times. The system on a heterogeneous platform can find its sweet spot using only a handful of cores. On platforms with fine-grained power gating, the system can turn off the unused cores and thus save power. Likewise, picking the right type of cores is crucial to balance the performance per Watt ratio. As we show in the evaluation in Section 4, slower cores frequently result in only small drop in performance whereas the potential for power savings is significant.

We do not consider the scheduling in this paper. As there is a lot of work on scheduling in such environments (discussed in Section 5), we are solely interested in the performance and efficiency of different configurations and designing the scheduler for NewtOS is future work.

3.3 Non-overlapping ISA

At this moment, we limit ourselves to heterogeneous architectures with an overlapping ISA [15]. In this section, we argue that by virtue of its design our system has the potential to embrace architectures with different ISAs too. We do not currently have a machine with non-overlapping ISAs on the same processor to evaluate our solution, but we briefly sketch how we can use existing features to support such platforms.

Specifically, we can use NewtOS' *live update* functionality to change the version of a component to run on a different architecture. We originally developed live update to allow us to fix buggy components with new, patched versions without the need of shutting the system down. Doing so greatly reduces maintenance of the system, disruption of its operation, and the time between diagnosing a bug and application of the fix. However, we can also replace a component with the same component compiled for a different ISA.

The update is fairly straightforward since both versions are based on the same code. Mere recompilation with different compiler settings produces the desired version. Moreover, the transition from one ISA type to another is simple because it is done only when the state is stable and the memory layout of data structures on both architectures is likely the same. Finally, we initiate the transition only at the top of the component's main loop, so that we can mostly forget about different layouts of the stack. In case of a discrepancy between the memory layout for each of the ISA versions, we provide an automatically generated transition function [10]. In practice, changing a component to a new architecture is simpler than updating a component to a new version. In contrast to a proper update, both versions for the different ISAs use identical data structures which may differ by offsets and alignment, but not by different items in structures. The system may provide a version for different ISAs when installed or use just-in-time compilation to generate one when need. If migration between ISAs is frequent, the system can cache a version for each to speed up the migration.

We can use the same mechanism as an optimization for overlapping ISAs too. Some cores may have a feature which allows the system code to run faster. For instance, file systems can take advantage of checksum instructions to verify data read from a disk or advanced instructions to encrypt the data. In such cases, the system component does not rely on the extra instructions for its correct operation, but can benefit from them if available.

4 Evaluation on a high-performance stack

We now evaluate the network stack of NewtOS on a dual socket quad-core Intel Xeon E5520 with hyper-threading.

The peak clock speed of the chips is 2267 MHz and it is possible to scale it down to 1600 MHz in steps of 133 MHz. According to ACPI, power consumption of each chip at its maximal frequency may be as much as 80W and at the lowest frequency 34W. Unfortunately, it is not possible to scale frequencies of the cores independently and all cores of each chip run at the same speed.

However, modern Intel processors like the E5520 can still scale each core independently using thermal throttling[2] to allow further scaling in steps of 12.5% of the clock speed. Thermal throttling means that by setting the chip to 1600 MHz, it is possible to scale down to 200 MHz in the same-sized steps. Although the core still runs at the base frequency (1600 MHz), some cycles are "thrown away" and the execution slows down proportionally. We can do this for each core individually, however both threads on the same core are throttled equally. Thus, the Xeon E5520 allows us to explore both threading, and high/low frequency trade-offs. While we cannot compare in-order versus out-of-order microarchitectures, we believe that a 200 MHz core is slow enough to match the performance of wimpy cores.

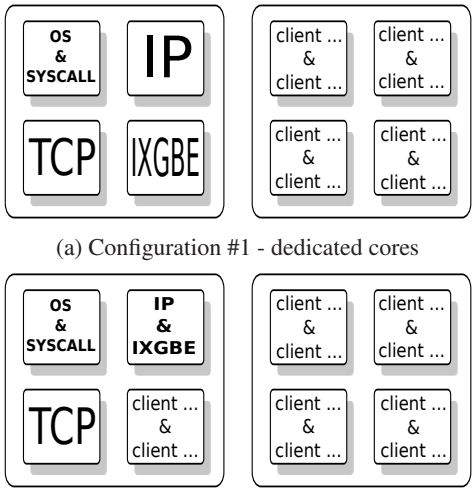
To remove bandwidth limitations, and to show that a multiserver system can scale to multigigabit range, we implemented a driver for the i82599 10G Intel network chip. The driver is fairly simplistic but has standard offloading features for the outgoing traffic. We connect our machine to a Linux 3.7.1 system running on a 12-core AMD Opteron 6168 at 1.9 GHz.

Our test case is the same as in [11] which we used to stress the system when demonstrating its reliability. We run an `iperf` server on the Linux machine and connect from NewtOS. `Iperf` is a standard tool for measuring and tuning network performance. The clients send data as fast as possible, trying to saturate the network hardware, the buses, the memory, or the CPU. We verified that the Linux machine is able to receive at 10G by connecting from Linux running on the same machine as we use to run NewtOS. We use multiple streams to get the best performance. LwIP does not support TCP window scaling, and is therefore not able to have enough data in transition to saturate the 10G link on a single stream.

4.1 Test configurations

We experimentally evaluated several configurations of the network stack to determine the most demanding components of the stack. Not surprisingly, TCP ranked highly. Based on these experiments, in performance-critical scenarios, the OS must choose between the two basic setups shown in Figure 2. We will evaluate them across a range of clock settings.

In both cases we place all processes of the core system (OS) on the first CPU and the network stack components



(a) Configuration #1 - dedicated cores

(b) Configuration #2 - hyper-threading

Figure 2: Test configurations - large squares represent the quad-core chips, small squares individual cores and '&' separates processes running on different hyper-treads.

involved in processing TCP traffic on the remaining 3 cores of the same chip. These components are TCP, IP and the 10G ethernet driver (IXGBE). Communication between components on different chips is slower as they do not share cache.

The syscall server shares its core with the rest of the system (OS), but runs in a different thread (denoted by the '**&**' symbol). It extensively uses kernel communication, but uses the CPU lightly. Nevertheless, it needs its own thread to use the fast signaling when translating synchronous messages from the clients to the TCP component and back. Otherwise, TCP would need to use *notifications* for correct operation when replying to the syscall server, resulting in a serious performance hit.

In both configurations, TCP has its own dedicated core, the spare hyper-thread is idle. The two configurations differ in only one thing. The first configuration (Figure 2a) also dedicates a full core to IP and the driver while the respective other threads are idle. The second configuration (Figure 2b) places both IP and the driver on the same core, but runs them in different threads. The rationale behind this choice is that TCP is the most demanding component while IP and the driver have similar CPU utilization as we demonstrate in the remainder of this section. We denote the second configuration with HT for employment of hyper-threading.

The test clients run on the remaining cores and threads. In other words, in configuration #1 (Sections 4.3 and 4.4), they all run on the other quad-core chip, while in configuration #2 (Section 4.5), they also run on one of the cores of the first chip. The scheduler distributes them equally.

MHz	drop	Mbps	drop	TDP(W)	drop
2267	–	8641	–	80	–
1867	18%	8152	6%	48	40%
1600	30%	7840	9%	34	57%

Table 3: Performance loss versus potentially saved power

4.2 Methodology of measurements

Throughout our experiments, we measure two basic values: (1) CPU utilization for each component, and (2) bitrate. We use time stamp counters to time the events. Intel guarantees that they are synchronized across all cores and tick at a constant rate regardless of frequency scaling. Each component has its log for events which we process after a test run finishes.

To measure the CPU utilization, we mark an event right before the kernel call which suspends the core and right after the call returns. In fact, this measures time actively spent in each component, so the actual CPU utilization is higher. Measuring the time this way is closer to using a single long latency instruction instead of a kernel call.

4.3 Frequency scaling 2267–1600 MHz

The first experiment is to explore how configuration #1 behaves when we change the frequency of the chip. We present the measurements in Table 3. The first line represents the baseline: all the cores run at the peak clock speed and the chip draws maximum power. As expected, we see that the bitrate drops when the clock speed goes down. As the drop is fairly small, we show only one intermediate value. The last line stands for the lowest frequency and power consumption.

Observe that scaling the cores down to the lowest frequency can save up to 57% of power, but the drop in throughput is not nearly as significant, a mere 9%. There are many cases in which 7.8 Gbps is more than enough while the opportunity to save 46 Watts is important. Also note that at maximum power the throughput is 8.6 Gbps. Later in this section we show that it is possible to throttle the cores even more and deliver better throughput than at the peak clock speed.

The CPU utilization measurements show that running the stack on high frequencies is probably suboptimal. The TCP component uses the core at approximately 70% while IP and the driver use their cores below 40%. The components spend much of their time polling the communication channels. If there is no work to do, they poll for a little while longer and eventually call the kernel to block them on MWAIT.

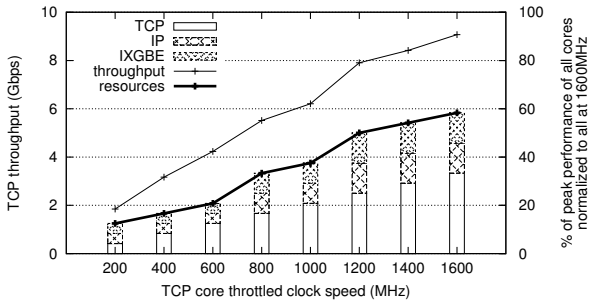


Figure 3: Throughput compared to the combined performance of the resources in use — the best combinations

4.4 Throttling below 1600 MHz

To demonstrate how our system behaves on much slower cores we use thermal throttling to artificially slow the cores down. We start our measurements by scaling all 3 cores to the minimum. Since TCP is the component which uses its core the most, we scale it up by one step for each new measurement and we try to match it with the best setting for the other 2 cores. Our experience is that if we increase the speed of the TCP core and the bitrate does not improve proportionally, we must speed up the other cores by one step too. Adding more does not help and may even lead to throughput degradation. We present our results for the best configurations in Figure 3, which compares the bitrate (the thin line) and the performance of the cores we need (the bars). In this case, 100% is the combined performance of all 3 cores running unthrottled at 1600 MHz. The thick line connects tops of the bars to highlight how the throughput scales with the added resources.

The important observations in Figure 3 are :

- Scaling the 3 cores to 12.5% of their total performance (200 MHz) delivers 1.8 Gbps which is enough for many applications like video streaming, web browsing or online gaming.
- The stack achieves approximately the same or higher throughput (7.9 Gbps) at 50% of resource utilization (bar 1200 MHz) than when all cores run unthrottled (7.8 Gbps as we reported in Table 3).
- Using TCP core clocked at 1600 MHz and the other two at 600 MHz is just 60% of performance of all of them running at 1600 MHz and only 40% of all running at 2267 MHz. This “low-power” configuration exceeds the performance of all cores at 1600 MHz as well as at 2267 MHz.

We emphasize that results are *average* bitrates of each test run. The average throughput at 60% of the combined

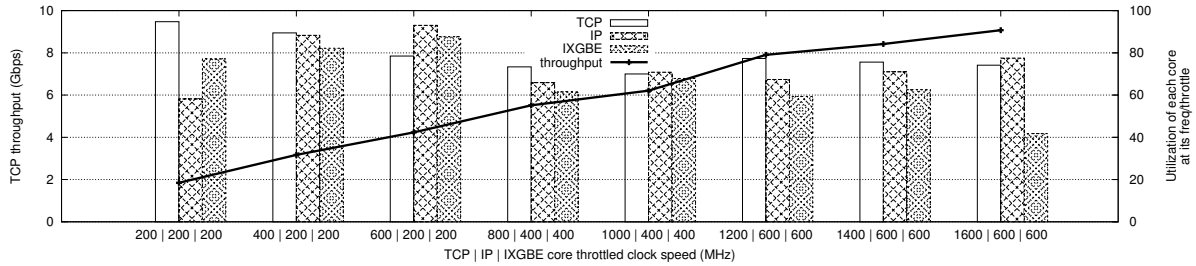


Figure 4: Configuration #1 – CPU utilization of each core throttled to % of 1600 MHz.

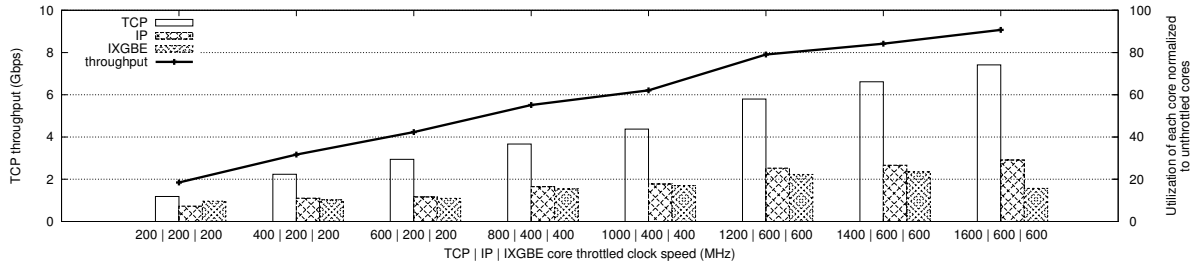


Figure 5: Configuration #1 – CPU utilization normalized to cores at 1600 MHz unthrottled.

resources (the rightmost result in Figure 3) reaches up to 9.1 Gbps with peaks approximating 10 Gbps. Slower sometimes really *is* faster!

Figure 4 presents CPU utilization of each core. The utilization is with respect to each core's throttling and each set of bars stands for one configuration. The sets form three clusters determined by the speed of the slower cores. The three sets in the first cluster show how the utilization of the IP and driver cores increases as the higher frequency permits TCP to process more data. In the third set the utilization of the slower cores approaches 100% and exceeds utilization of the TCP core. Therefore we must match speeding up TCP by speeding up the others too, if the current throughput is not enough. The same pattern repeats in each of the clusters.

Although the relative utilization of the TCP core drops slightly, Figure 5 clearly shows that the CPU time obtained by TCP directly determines the final throughput. Figure 5 presents the same data as Figure 4, but normalized to a core running unthrottled. The important observation is that the utilization of the other two cores does not grow equally fast. The main reason is that unlike TCP, IP and the driver do not touch the TCP payload. TCP must copy all the data from the client applications to the address space of the stack. It is a lengthy operation which thrashes caches and makes the core stall while the time is reported as used. The copy overhead can be significant, between 60 and 70%.

Interestingly, the faster the core, the higher the overhead. The explanation is that the difference between CPU speed and memory speed grows leading to more stalls. Without the copy overhead, CPU utilization would be comparable to the other two components. For completeness, we mention that some of this overhead can be reduced by letting the network devices transfer data directly from and to the user space buffers.

We did not measure throttling for higher clock speeds than 1600 MHz, because the results show that increasing the speed does not yield significant benefits, and because we want to make the point that lower clock speeds are good enough for even the most demanding OS components. Although we can only guess how much energy would our emulated low power cores use, for example, Intel reports that the thermal design power is less or equal to 3.5W for its Atom N2600 processors at 1.6GHz.

4.5 Hyper-threading

The same set of experiments for configuration #2 evaluates the effect of threaded cores. Threads are not equal to full cores as they share the same pipeline. Their advantage is that they allow the core to use cycles which would be otherwise wasted when the pipeline stalls due to slow memory. If the code running on one of the threads has a high cache hit rate and good branch prediction, execution of additional threads has diminishing returns. However, we do not expect system code to behave optimally. Mes-

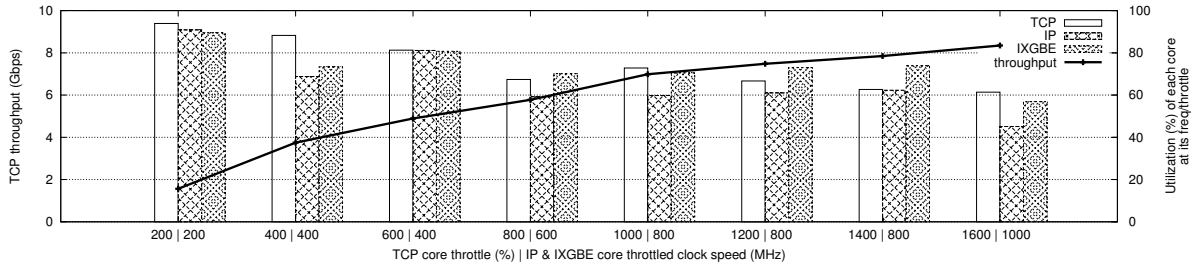


Figure 6: Configuration #2 (HT) – CPU utilization of each core throttled to % of 1600 MHz.

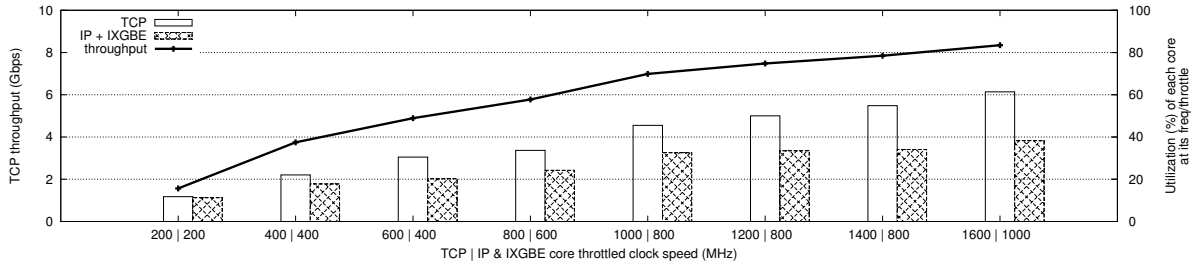


Figure 7: Configuration #2 (HT) – CPU utilization of each core normalized to 1600 MHz.

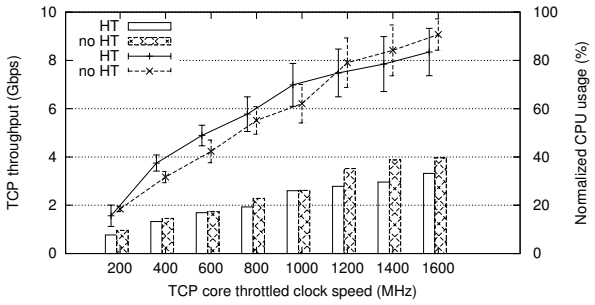


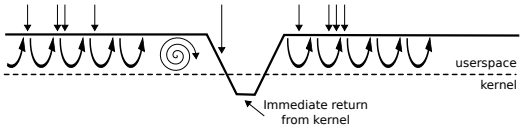
Figure 8: Comparison of configuration #1 (no HT) and #2 (HT). Lines represent bitrate, bars represent CPU utilization normalized to 3 cores at 1600 MHz

sages created on different cores are not in the remote cache until read for the first time and the CPU can hardly guess which execution path the code takes in the next loop. Therefore system code should benefit from threading.

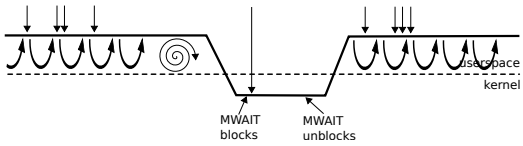
Based on the previous measurements of configuration #1 and the fact that a thread is not a full-blown core, we expected that the core which hosts IP and the driver should run at least at double the speed of a core hosting either IP or the driver to deliver the same throughput. Figure 6 shows that the actual clock speed we require is sometimes equal to, but mostly less than what we expected. Figure 7 shows the normalized values. The crosshatched bar represents the utilization of IP and the driver running on the same core. Since each component also accounts the time when their threads are not active the utilization of a single core could exceed 100%. Therefore, the bar represents mean value of both threads.

The main reason why we could run at lower clock speeds than we originally expected, is that running more threads on the same core, uses the cycles of the core more efficiently and reduces the amount of sleep time. Since the execution of both processes is interleaved, there is a higher probability that while a processes' thread is inactive, the other processes of the stack create some new jobs. Thus, when the thread activates again, instead of finding the work queues empty, the process can carry on. The benefits of sharing a core between IP and the driver is the easiest to observe when comparing the experiments with the slowest cores. Although using two cores at 200 MHz is just 66% of the resources of 3 dedicated cores at the same speed, the throughput is 77% or 1.4 Gbps.

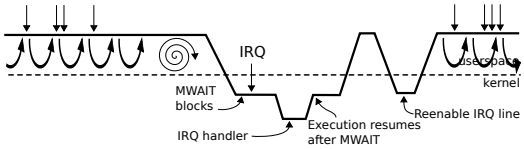
Figure 8 compares the performance of configurations #1 and #2. As long as the variance in the bitrate is low, using the threaded core outperforms configuration #1 with an extra core. The bars present the combined CPU utilization of both configurations normalized to all 3 cores running unthrottled at 1600 MHz. In all cases the normalized utilization is lower for configuration #2 while the performance is higher when TCP core runs at or below 1000 MHz. As the transmission of data gets more bursty, the ability to use more cycles per time unit on the dedicated cores to get the work done quicker, outweighs the



(a) Job arrives between deciding to sleep and halting the core



(b) Job arrives when execution is suspended on MWAIT



(c) Driver - an IRQ arrives when execution is suspended

Figure 9: Sleep overhead in different situations. The thick line represents transition between execution in user space and kernel in time. Arrows mark job arrivals, loops denote execution of the main loop and spirals denote polling.

benefits of threading. Higher bitrate leads then to higher CPU utilization.

4.6 Why is slower faster?

When a job queue is empty, we can either keep on polling which is fine only when jobs arrive frequently. Otherwise the cores use energy while not doing any useful work. The common (and probably right) thing to do is to put the cores to sleep when there is no work to do. Idly waiting for new jobs to arrive is costly when we sleep at the wrong moments, because we reduce the time we could have used for processing until the overhead due to sleeping becomes so high that we observe the "slower is faster" effect.

As we cannot predict the future, we may put a core to sleep just before a new job arrives (Figure 9a). As we still use a kernel call for sleeping, the call introduces some latency, even though the execution does not block and returns to user space immediately. The way back through the kernel is not free. It is even more expensive when a job arrives just after suspending the core (Figure 9b) as MWAIT has a relatively long latency, in the order of microseconds. Nevertheless, blocking on MWAIT is much faster than using traditional kernel IPC. Especially, since such IPC would slow down the *sending* core too.

The worst case is when an interrupt wakes up a driver core. The interrupt handling routine adds up to the MWAIT latency. In addition, the interrupt line should stay disabled until the driver masks the interrupt in the device. At that point, it has to reenable the interrupt line, which incurs another trip to the kernel (Figure 9c). Thus, drivers are the most sensitive to frequent sleeping. The solution may be to run the driver in the privileged mode of a *virtual machine*.

Polling harder eliminates some of the "slower is faster" effect. However, designing a polling algorithm which adapts to unpredictable conditions is complicated. The ideal solution is to use an efficient *sleep* instruction in user space. On the other hand, sleeping will always have some latency. The take-away message is the following: to avoid the *expensive* idle time, the scheduler should pick the cores and hyper-threads on which it places the components carefully and scale them so that they are always highly used—with little opportunity to sleep.

Clock speed (MHz)	Bitrate (Gbps)
2267	4.3
1600	3.4
200	0.4

Table 4: Bitrate vs. CPU clock speed on a single core

4.7 Stack on a single core

In case of shortage of cores due to high demand from applications, or when cores are turned off to save power, the entire network stack of NewtOS can keep operating on a single core. Table 4 presents measurements of the stack's performance as a function of the core speed. The stack has a throughput of up to 4.3 Gbps on a big fast core and 400 Mbps on a 200 MHz wimpy core. The throughput of the slow core is good enough for many common activities, but the fast core cannot scale further. More importantly, a network stack running on a single core has a much higher latency. If a process has work to do, it hogs the core until it exhausts its time quantum while others are on hold. Then the scheduler is free to pick any runnable process of the stack which increases non-determinism in the execution. Running the stack on dedicated cores removes these deficiencies and the throughput of a single fast core is similar to the configuration with a TCP core at 600MHz and IP and driver cores at 200MHz.

5 Related work

Kumar et al. proposed single-ISA heterogeneous multicores for power reduction [15] and to improve performance of multithreaded workloads [16]. They demonstrated that applications need a good mix of single-threaded performance and high throughput. Due to the

diversity in application code, heterogeneous platforms outperform homogeneous ones with the same die size. This makes the heterogeneity promising for the future. Although we do not focus primarily on applications but on the operating system code, the similarity is that we also exploit the fact that each system component has its own requirements for optimal execution. Moreover, the system components are user space processes like applications. However, the system code differs from applications, thus its optimal requirements are different too.

Operating systems are key to leveraging the heterogeneous platforms as only the scheduler can make decisions where each application runs, therefore the schedulers got the most attention. Kumar et al. [16] proposed a whole range of sampling heuristics that permute threads on cores to find the best assignment. As the execution of applications changes during different phases, Becchi et al. [6] proposed a dynamic algorithm which constantly measures the IPC ratio of threads and tries to run on the big cores those threads that would benefit the most.

Permuting the threads and sampling them on all types of cores is an overhead. Therefore Koufaty et al. [14] designed a scheduler which monitors execution of each thread on its current core only. It uses existing low overhead performance monitoring counters to collect performance related data which the system can use to estimate what type of a core is the most suitable for the given thread. This algorithm relies on a model which translates the performance statistics to the bias of each thread to a certain type of a core.

Most heterogeneous scheduling algorithms use the speed up factor, the ratio between how fast an application runs on a small and a big core. Saez et al. [25] suggest a more comprehensive utility factor of how effectively the whole mix of running application uses the machine.

Instead of using available performance counters to feed data into the models which predict the performance of the threads on different types of cores, hardware monitoring and prediction engines [27] and performance impact estimators [29] were proposed as hardware extensions. The hardware estimates the possible speed-up on its own and the scheduler can use this direct feedback to decide which applications would benefit from running on the big cores and which can run on the small ones.

In contrast to this work, we do not focus on the performance of applications, instead our main focus is on the system. First of all, our system can use all the different heuristics or hardware estimations to schedule application on the cores which are not dedicated to the system. Second, our system is a collection of user space processes and the scheduler can use the same or similar techniques to find their optimal placement. On the other hand, execution of the system differs in several aspects. Each component is responsible for a small subset of all the system tasks, therefore they have little variance during their execution as the requests they serve are similar. The system code follows the same patterns which differ from application code. The scheduler's goal is not to let the system finish as quickly as possible, but to deliver optimal service to the changing mix of applications and workloads using the available resources. In contrast to the applications, which are opaque for the system, the system designer knows more about the system components and the components themselves can help the scheduler by providing various hints. For instance, a component can detect and signal when the recipient of its messages cannot keep up and thus may benefit from a faster core. Similarly, applications can give hints to the system, for instance, when the estimated time of downloading a file is in minutes, the application can tell the system, that it is not a sudden spike in the load and reconfiguring is worthwhile.

Mogul et al. proposed operating system friendly cores in [20], primarily to save power. They argue that many features which the operating systems do not use draw a lot of power while not contributing to performance of the operating system. They propose that the system should run on the *optimized* cores and the execution should transfer from the application cores to the system cores when necessary. The migration is a bottleneck which they address in [28]. Migrating the execution means that the cache locality is poor. In contrast to their experiments with Linux, we have a system which is more suitable for heterogeneous platforms. NewtOS moves execution only by sending a message to another core and benefits from cache locality of the code and data of the component running on the core. Of course, locality of the user data passed between the cores is poor, however, in many cases the components do not need to touch the data until the DMA of a device transfers them. Cache locality of the messages is also poor, but this data should not be cached after the message is sent in the first place. Unfortunately, the current hardware does not allow us such a fine-grained control over cache and data transfers. Strong et al. [28] also use networking for evaluation. They model the power usage of the hypothetical cores while we use frequency scaling to approximate performance of such a hardware.

Netmap [24] and OpenOnload [3] projects demonstrated high bandwidth networking in user space. In contrast to NewtOS, both need a driver in a monolithic kernel. Although most of the faults crash only the application, there is still a chance that a bug in the driver can bring the whole system to a halt. Netmap shows that a 900 MHz core is good enough to transfer 10 Gbps of small packets between the device and the user space application, however, netmap only deals with routing and does not offer a generic networking support to applications. On the other hand, OpenOnload transparently intercepts any application requests and uses a library with custom made

hardware to transfer data directly between applications and devices. We endorse this approach as it would remove the copying overhead between applications and our stack.

6 Conclusions

We have demonstrated that a processor's fast cores may not be ideal for system workloads and that less can be more in some situations. We presented a network stack evaluation of a reliable and dependable system. The results support our claim that it is possible for such a system to perform well, using much more constrained resources than usually available. We use current hardware to approximate future processors and we show the potential benefits. Unlike many other researchers, we do not focus on the applications. The operating system plays a key role in the execution of applications and we should give it equal attention. However, performance should not be the only criterion, the system is also responsible for security, reliable execution and easy maintenance. NewtOS design recovers from crashes and allows administrators to update its components while it is running. Although our case study covers only one part of a generic operating system, we are confident that the findings apply to other parts and to other systems as well.

Acknowledgments

This work has been supported by the ERC Advanced Grant 227874 and EU FP7 SysSec project. We would like to thank Valentin Priescu for implementing the frequency scaling driver. Likewise, we would like to thank Dirk Vogt for implementing the IXGBE driver for MINIX 3.

References

[1] ARM - big.LITTLE Processing. `http://www.arm.com/products/processors/technologies/biglittleprocessing.php`.

[2] NVIDIA - Variable SMP architecture. `http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf`.

[3] OpenOnload. `http://www.openonload.org/`.

[4] The Intel Xeon Phi Coprocessor. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[5] Samsung to outline 8-core big.LITTLE ARM processor in February. `http://www.engadget.com/2012/11/20/samsung-to-outline-8-core-big-little-arm-processor-in-february/`, Nov. 2012.

[6] BECCHI, M., AND CROWLEY, P. Dynamic Thread Assignment on Meterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd conference on Computing frontiers* (2006), CF '06.

[7] CRISTIANO GIUFFRIDA, L. C., AND TANENBAUM, A. S. We Crashed, Now What? In *Proceedings of the 6th International Workshop on Hot Topics in System Dependability* (2010).

[8] DUNKELS, A. Full TCP/IP for 8-bit architectures. In *International Conference on Mobile Systems, Applications, and Services* (2003).

[9] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and Automatic Live Update for Operating Systems. In *Proceedings of ASPLOS-XVIII* (2013).

[10] GIUFFRIDA, C., AND TANENBAUM, A. S. Safe and Automated State Transfer for Secure and Reliable Live Update. In *Proceedings of the Fourth International Workshop on Hot Topics in Software Upgrades* (2012).

[11] HRUBY, T., VOGT, D., BOS, H., AND TANENBAUM, A. S. Keep Net Working - On a Dependable and Fast Networking Stack. In *Proceedings of Dependable Systems and Networks (DSN 2012)* (Boston, MA, June 2012).

[12] IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. F. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture* (2007).

[13] KHUBAIB, SULEMAN, M. A., HASHEMI, M., WILKERSON, C., AND PATT, Y. N. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.

[14] KOUFATY, D., REDDY, D., AND HAHN, S. Bias Scheduling in Heterogeneous Multi-Core Architectures. In *Proceedings of the 5th European conference on Computer systems* (2010), EuroSys '10.

[15] KUMAR, R., FARKAS, K. I., JOUPPI, N. P., RANGANATHAN, P., AND TULLSEN, D. M. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (2003).

[16] KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st annual international symposium on Computer architecture* (2004), ISCA '04.

[17] LATIF, L. IDF: Intel is looking at ARM's Big Little architecture. `http://www.theinquirer.net/inquirer/news/2205764/idf-intel-is-looking-at-arms-big-little-architecture`.

[18] LI, T., AND JOHN, L. K. Operating system power minimization through run-time processor resource adaptation. *Microprocessors and Microsystems 30*, 4 (2006).

[19] LIEDTKE, J., ELPHINSTONE, K., SCHÖNBERG, S., HRTIG, H., HEISER, G., ISLAM, N., AND JAEGER, T. Achieved IPC Performance (Still The Foundation For Extensibility), 1997.

[20] MOGUL, J. C., MUDIGONDA, J., BINKERT, N., RANGANATHAN, P., AND TALWAR, V. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro 28*, 3 (May 2008).

[21] NELLANS, D., BALASUBRAMONIAN, R., AND BRUNV, E. A Case for Increased Operating System Support in Chip Multiprocessors. In *In Proc. of 2nd IBM Watson P=ac 2* (2005).

[22] OLUKOTUN, K., HAMMOND, L., AND LAUDON, J. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. 2007.

[23] REDSTONE, J. A., EGGERS, S. J., AND LEVY, H. M. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proceedings of ASPLOS-IX* (New York, NY, USA, 2000).

[24] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (2012), USENIX ATC'12.

[25] SAEZ, J. C., FEDOROVA, A., KOUFATY, D., AND PRIETO, M. Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems. *ACM Trans. Comput. Syst. 30* (Apr. 2012).

[26] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of Symp. on Oper. Sys. Des. and Impl.* (2010).

[27] SRINIVASAN, S., ZHAO, L., ILLIKKAL, R., AND IYER, R. Efficient Interaction Between OS and Architecture in Heterogeneous Platforms. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 62–72.

[28] STRONG, R., MUDIGONDA, J., MOGUL, J. C., BINKERT, N., AND TULLSEN, D. Fast Switching of Threads Between Cores. *SIGOPS Oper. Syst. Rev. 43* (April 2009).

[29] VAN CRAEYNEST, K., JALEEL, A., EECKHOUT, L., NARVAEZ, P., AND EMER, J. Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA '12.

Notes

[1] MWAIT is optionally unprivileged in AMD chips starting with family 10h, but we use Intel due to hyper-threading and better scaling.

[2] It is usually possible to scale AMD chips to lower speeds than Intel ones (e.g., from 1.9 GHz to 800 MHz). However, the Intel-specific mechanism of thermal throttling allows us to emulate much lower speeds

IAMEM: Interaction-Aware Memory Energy Management

Mingsong Bi
Intel Corporation
mingsong.bi@intel.com

Srinivasan Chandrasekharan
University of Arizona
schandra@cs.arizona.edu

Chris Gniady
University of Arizona
gniady@cs.arizona.edu

Abstract

Energy efficiency has become one of the most important factors in the development of computer systems. As applications become more data centric and put more pressure on the memory subsystem, managing energy consumption of main memory is becoming critical. Therefore, it is critical to take advantage of all memory idle times by placing memory in low power modes even during the active process execution. However, current solutions only offer energy optimizations on a per-process basis and are unable to take advantage of memory idle times when the process is executing. To allow accurate and fine-grained memory management during the process execution, we propose Interaction-Aware Memory Energy Management (IAMEM). IAMEM relies on accurate correlation of user-initiated tasks with the demand placed on the memory subsystem to accurately predict power state transitions for maximal energy savings while minimizing the impact on performance. Through detailed trace-driven simulation, we show that IAMEM reduces the memory energy consumption by as much as 16% as compared to the state-of-the-art approaches, while maintaining the user-perceivable performance comparable to the system without any energy optimizations.

1 Introduction

Modern computer systems ranging from netbooks to server clusters are relying on large system memory to provide high performance for data intensive applications. While a computer system contains many energy hungry components, the energy consumption of main memory is becoming more significant and can surpass energy consumption of other components. For example, as much as 40% of the total system energy is consumed by the memory subsystem in a mid-range IBM eServer machine [14]. The demand for higher memory capacity is not limited to data servers. Even portable computers are experiencing a rapid growth in memory capacity to accommodate user demand for higher processing capability and richer multimedia experiences. As a result, current portable systems, e.g. notebooks, are commonly sold with 8GB of main memory or more, and ultraportable systems such as netbooks with 4GB.

Energy optimization of the memory subsystem is being addressed at both hardware and software levels. At the hardware level, energy efficiency is primarily gained through advances in manufacturing processes to create denser modules and lower per-bit energy consumption. In addition, low-power states are also added to the modern SDRAM and are exposed to the system software, enabling OS-driven energy management. While energy management that utilizes multiple power states can be implemented in hardware, it is usually delegated to the operating system. The operating system has a detailed view of the running applications and the demand they place on the system, and therefore, allows for more sophisticated energy management. While the additional context available at the OS level provides better energy management possibilities, the task of designing an efficient energy management technique is not easy due to the long power-state transition delays that can be exposed to the application execution.

Performance and the overall energy efficiency may suffer if the overheads of power state transitions are not addressed properly, since the entire system has to stay on longer and consume additional energy. In addition, the user may even be irritated to the extent that he or she completely disables the energy management mechanisms. Fortunately, maximal memory performance is usually not necessary to meet the user's performance expectations. For example, CPU or I/O bound tasks in interactive applications may not be noticeably degraded when the memory is operating in a low-power state. Furthermore, the perceived performance of real-time applications such as video players, games, or teleconferencing may not be affected by the power state transitions, as long as the system maintains perceptual continuity for the user. Therefore, it is critical to distinguish the memory-intensive tasks that may expose transition delays to the users from the tasks with low memory activity. Subsequently, the former tasks must be executed

with high memory performance, while the latter can be executed with lower performance to improve energy efficiency.

To take advantage of these opportunities, we propose Interaction-Aware Memory Energy Management (IAMEM), a highly accurate and transparent mechanism for memory energy management in interactive systems. Compared to the existing mechanisms: (1) IAMEM provides energy optimizations within the running process that the user is interacting with as well as all other processes in the system, as compared to previous approaches that only improved energy efficiency of memory occupied by processes waiting for execution [8, 13]; (2) IAMEM is a unified approach that addresses energy efficiency of both the buffer cache and the virtual memory, while previous approaches only proposed individual solutions for either the buffer cache or the virtual memory [2, 8, 13]. Subsequently, we make the following contributions in this paper: (1) identify and quantify the memory behavior of common interactive applications and show large opportunity for improvement; (2) utilize high-resolution context of user interactions to accurately predict memory demand for tasks initiated by the user; (3) propose IAMEM, a unified energy management mechanism for the entire memory subsystem; (4) compare IAMEM with the existing state-of-the-art mechanisms through a detailed study.

2 Motivation

Current trends in providing larger on chip CPU caches result in main memory seeing fewer accesses from the CPU, which creates longer memory idle times. Subsequently, the majority of memory energy is consumed in the idle state. Energy consumption of main memory can be significantly reduced by transitioning memory devices to a low-power state during the idle periods. However, accessing memory in a low-power state incurs high transition latency, and as a result, degrades the system performance and may increase the overall energy consumption. Therefore, it is crucial to ensure that the associated performance degradation can be hidden behind the application execution and not exposed to users.

A simple way to provide memory energy management is to keep the memory devices occupied by currently running process in a high-power state and power down all other memory devices. This per-process energy management is employed by Power-Aware Virtual Memory (PAVM) [8]. In this approach, the memory devices used by the newly scheduled process are powered up to provide high performance for the running process, during the context switch, while the other memory devices occupied by non-executing processes are kept in a low-power state to save energy. While this per-process approach provides significant benefits for a multitasking environment, it fails to address the energy consumption within a single process. Furthermore, current multicore CPUs with hyper-threading support can have tens of processes concurrently executing and accessing a wide range of memory addresses, rendering PAVM ineffective. To improve energy efficiency in such scenarios, we need a finer-granularity and more aggressive energy management that is able to reduce energy consumption within a single process during the process execution.

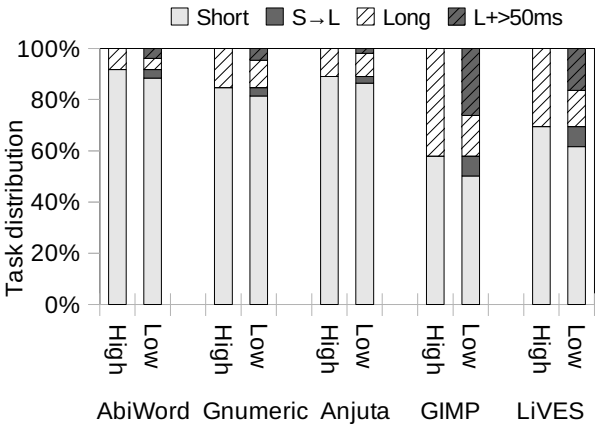


Figure 1: Distribution of tasks with memory in high and low power states. Shaded bar represents the fraction of short tasks extended over 50ms, while shaded-crossed bar represents the fraction of long tasks extended by more than 50ms.

2.1 Per-Task Energy Management

Fortunately, the full performance is usually not needed in interactive applications, since users are unable to perceive certain amount of short delays. We can exploit that observation in designing more aggressive energy management mechanisms. Prior studies in human-computer interaction have established the human perception threshold to be between 50-100ms, indicating that events with durations falling within this threshold are not perceived by the user [23]. Completing task execution earlier than the perception threshold is meaningless since the user will not notice this amount of time and cannot initiate tasks any faster. Therefore, any task shorter than the perception threshold can be potentially executed at a lower performance level, so that its execution time can be stretched up to, but not beyond the perception threshold. The resulting lower power consumption can improve energy efficiency while the user's behavior is unaffected.

To account for all possible users and prevent any potential performance degradation, we assume the lower bound of 50ms as the perception threshold for all users.

In the remainder of this paper, we refer to any task finishing within 50ms as a short task and any task running longer than 50ms as a long task. A short task appears instantaneous to the user even if extended up to 50ms. A long task, however, is perceivable to the user even at the highest performance level. Since the user would not be able to perceive the time difference within 50ms, a long task can be safely prolonged by up to 50ms, assuring that the user's think flow is not interrupted and the subsequent behavior is not affected [3, 20]. To take advantage of the allowed delays in interactive applications, we can potentially put the entire memory subsystem into a low-power state between memory operations and power it up upon a memory access request. The transition latency due to this on-demand power up may slow down a single memory access; nevertheless, from the task perspective, the user may not notice the aggregated delays, as long as the scaled task does not exceed the user's perception threshold as discussed above.

Figure 1 examines the scenario of keeping memory in a low-power state (Low) between accesses for several interactive applications and compares it to the standard system that keeps memory in a high-power state (High). These interactive applications are described in detail in Section 4 of this paper. The tasks from each application are further classified into short tasks and long tasks. Keeping memory in the low-power state can extend task execution beyond the user's perception threshold, as shown in Figure 1 by shaded area in each category. We observe that 93% of short tasks and 58% of long tasks stay within the user's perception tolerance. The longer tasks suffer more since they perform more memory operations and as a result, expose more transition delays. At this point, we can draw two significant observations: (1) there is a tremendous opportunity to save energy within a running application by keeping memory in a low-power state; and (2) some tasks have to be executed with memory in the high performance state, otherwise the degradation would be noticed by the user. The observations justify the need for an intelligent mechanism that is able to accurately identify memory intensive tasks from the majority of low-demand tasks that can be executed at low memory performance.

The majority of tasks in interactive environments are initiated directly by users and the performance demand within an application exhibits a strong correlation to User Interactions (UIs) with the application [1, 4]. In this paper, we will leverage the high-resolution context of user interactions to categorize UI-triggered tasks and correlate their memory behavior with the user interactions. By utilizing this correlation, the proposed mechanisms will select the best memory power states to match the tasks' performance demand.

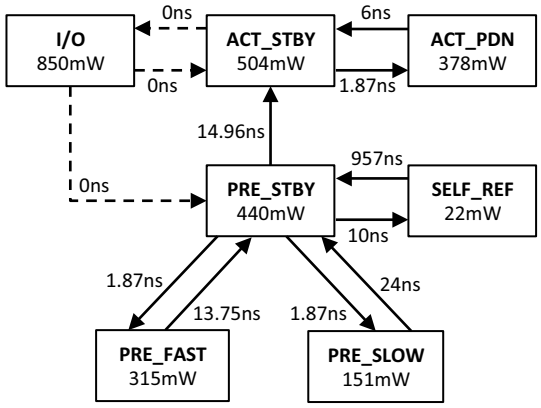


Figure 2: Power specifications for a rank consisting of 8 Micron 1Gbit DDR3-1066 devices.

2.2 Memory Power States

Synchronous Dynamic RAM (SDRAM) is widely used in computers as main memory in the form of Double-Data-Rate (DDR), followed by DDR2 and DDR3. We focus on DDR3 DRAM in this paper since it is the mainstream DRAM architecture in today's computer systems. DDR3 SDRAM is packaged into a DRAM module that commonly consists of two DRAM ranks. The smallest power management unit in DDR3 is the rank and all devices in one rank are operating at the same power state [18]. Each rank in a DRAM module can operate in several different power states: (1) Active Standby state (ACT_STBY): the state where memory can read or write data without any delay; (2) Active Power Down state (ACT_PDN): the power down state that offers some energy savings while minimizing transition delays; (3) Precharge Standby state (PRE_STBY): the intermediate state for transitioning to a much lower energy states; (4) Precharge Power Down Fast (PRE_FAST): the fast power down state where DLL's are still locked; (5) Precharge Power Down Slow (PRE_SLOW): the slow power down state where DLL's are not locked anymore; In both PRE_FAST and PRE_SLOW states several subcomponents of a rank are disabled to reduce power, such as I/O buffers, sense amplifier, row/column decoder, etc.; And finally (6) Self Refresh state (SELF_REF): in addition to previous states, the external clock and on-die termination are disabled to reduce power consumption even further.

Figure 2 illustrates the power specifications for a DDR3-1066 rank [19, 17], including the power consumption, the power state transition, and the associated resynchronization latency. Memory I/Os can only be performed with memory in a high-power state (ACT_STBY); therefore, the rank in low-power states (ACT_PDN, PRE_STBY, PRE_FAST, PRE_SLOW or

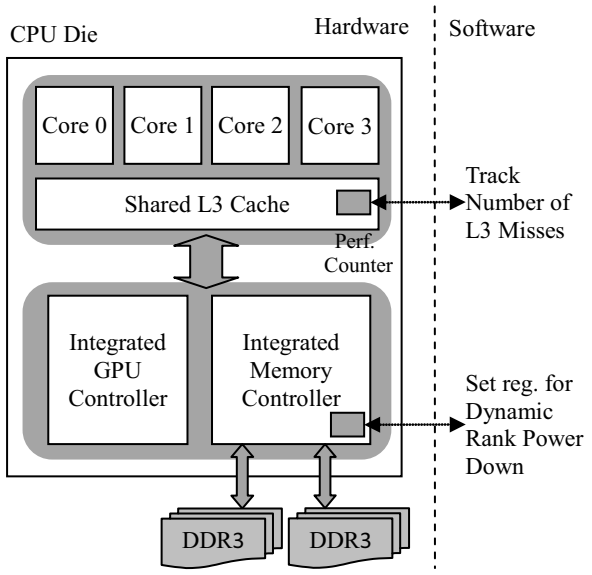


Figure 3: Architecture of Core i CPU.

SELF_REF) has to be transitioned to ACT_STBY state before performing any I/O, potentially exposing a high resynchronization delay to the application. Large number of resynchronizations may cause overall delays long enough to become perceptible to the user and degrade the performance. Therefore, it is critical to control the amount of power state transitions per UI-triggered task.

2.3 Hardware Support

Energy management at the operating system level requires support from hardware to control memory power states and monitor system behavior in detail. Figure 3 shows Intel's most recent Core i CPU with integrated memory controller for low-overhead power state management and monitoring. In addition, the Core i CPU provides a mechanism called Dynamic Memory Rank Power Down to power down memory ranks automatically after a specified memory idle time has elapsed [11]. Subsequently, the operating system only needs to set up the associated register with the desired timeout value to enable this feature. The integrated memory controller will then perform the power state transitions automatically, after the preset timeout expires.

Detailed system monitoring is further provided by the Core i CPU through a set of registers called performance counters, which are crucial for monitoring memory accesses. Under normal operations, virtual memory accesses are invisible to the operating system, while only occasional page faults results in OS being invoked. Performance counters, however, enable the OS to monitor memory activity when applications are performing memory I/Os, such as the number of CPU cache misses that result in main memory accesses. However, exact timing of each memory request, that would allow us to determine memory access burstiness, is not available.

3 IAMEM Design

Observing that there exists a strong correlation between user interactions and the required performance, we propose Interaction-Aware Memory Energy Management for entire memory space in interactive systems. IAMEM will transparently exploit UI events to speculate about the desired performance, and dynamically manage the memory power states to meet the task demand. Subsequently, we will discuss the following components in this section: (1) Unified energy management mechanisms that address all types of accesses to physical memory; (2) High-detail and low-overhead monitoring and detection of tasks triggered by user interactions; (3) Accurate classification and correlation of tasks and the associated processing demand; (4) Online training and prediction for determining the desired memory power for upcoming tasks; and (5) Optimizations to prevent perceivable performance degradation.

3.1 Memory Space

Physical memory in modern operating systems is divided into three categories: (1) kernel space that is strictly reserved for the OS kernel, its data structures, device drivers, etc.; (2) the buffer cache for caching previously accessed disk blocks to improve the file system performance; and (3) user space that is allocated as Virtual Memory (VM) for user processes. The majority of memory space is dynamically allocated to the buffer cache and virtual memory of running processes, based on the current demand for each type of memory.

Memory ranks used for the buffer cache can be efficiently managed in server environments by hiding power-state transition overheads behind the kernel processing time [2]. While the mechanism worked well in server workloads where the buffer cache occupies large space spanning several ranks, it has limited applicability for interactive applications where the buffer cache occupies smaller space and usually shares the rank with the kernel data structures. Subsequently, upon a first kernel memory access, the rank is powered up making large portion of the buffer cache accessible without further delays. Even if the buffer cache occupies several ranks, interactive applications, in general, put lesser pressure on the buffer cache than server applications, such that only a small fraction of overall accesses may require powering up additional ranks. Therefore, we consider memory space occupied by the kernel data structures and the buffer cache as a single kernel memory space and

do not distinguish them further. Subsequently, we propose unified energy management mechanisms that manage all memory spaces based on user interaction patterns to guarantee user-perceivable performance while maximizing energy savings.

3.2 UI and Task Monitoring

X Window Server manages interactions with client applications in Linux GUI environment. UI elements are contained in windows organized in a window tree associated with the application. IAMEM relies on a monitoring layer (X Monitor) between the X Server and client applications to uniquely identify individual UI elements [4], as shown in Figure 4. Both keystrokes and mouse interactions with the application are monitored. The unique ID of a given UI element is generated from the element's position within its containing window and that window's position within the window tree. Subsequent interactions with a particular UI element generate the same interaction ID, and the same categorization for the task to follow. The operation of the UI capture mechanism is entirely transparent to the user and does not require any modification to the applications.

Every user interaction results in a task that requires a certain amount of processing to accomplish the goal. The task can be short, such as keystroke capture and display on the screen, or long that involves large amounts of CPU activity, memory I/Os, and even other device I/Os. To reduce the overhead of task detection, IAMEM assumes that a task completes as soon as the OS idle process (swapper process in Linux) begins running and the task is not blocked by I/O, or when the application receives a new UI event [1, 16]. IAMEM uses the Time Stamp Counter to accurately measure the CPU cycles taken to process the task in user and kernel mode, which also include cycles for memory accesses. In addition, IAMEM uses the performance counter to measure the number of accesses to main memory, during task execution, by counting the number of misses in the last level CPU cache.

3.3 UI and Task Correlation

IAMEM classifies tasks by the individual interaction IDs of the triggering UI events. To predict the memory power demand for each task category, IAMEM utilizes a prediction table implemented as a hash table indexed by the interaction IDs. Once the completion of a task is detected, an α aged average method is used to update and record the task processing demand described by the execution time and the number of memory references. The α aged average method captures past behavior of the interaction and also allows quicker adaptation to new

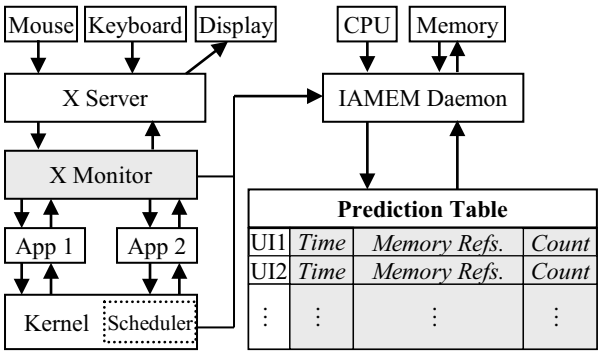


Figure 4: IAMEM design architecture.

behavior patterns, if the memory behavior of the interaction changes. Figure 4 illustrates the prediction table with the following variables for each interaction ID: (1) a weighted sum of all previous tasks' computation time *Time*; (2) a weighted sum of all previous tasks' memory references $MemoryRefs$; and (3) a weighted count of all observed task instances *Count*. For the most recent task with computation time T and memory references M, those three table variables are updated as follows with a predefined weight α ($\alpha <= 1$):

$$Time = \alpha * Time + T \quad (1)$$

$$MemoryRefs = \alpha * MemoryRefs + M \quad (2)$$

$$Count = \alpha * Count + 1 \quad (3)$$

Note that T is recorded as the actual computation time with memory in the active state. When memory transitions occur, the time spent in power state transitions should be deducted from the actually monitored time.

3.4 Power State Prediction

Each time a UI event occurs, IAMEM performs a table lookup using the captured interaction ID. It first predicts the incoming task's processing demand as the average of the retrieved demand history. To maintain the user-perceivable performance, an appropriate deadline D is selected, which is either 50ms for short tasks, or the task's computation time, with memory in the active state, plus 50ms for long tasks. Based on this deadline, IAMEM calculates *PS*, the lowest possible power state of a memory rank, to fit the task execution within the deadline when all predicted memory accesses encounter power state transitions. The prediction algorithm is shown in Figure 5. The resulting *PS* gives us the needed power state to accomplish the task before the deadline and can be any of the five states: Active Standby (ACT_STBY), Active Powerdown (ACT_PDN), Precharge Powerdown Fast (PRE_FAST),

$T_{avg} = Time/Count$;
$M_{avg} = MemoryRefs/Count$;

$if \ \ (T_{avg} <= 50ms)$
    $D = 50ms$;
$else$
    $D = T_{avg} + 50ms$;

$if \ \ (T_{avg} + M_{avg} * L_{SELF\_REF} < D)$
    $PS = SELF\_REF$;
$else \ \ if \ \ (T_{avg} + M_{avg} * L_{PRE\_SLOW} < D)$
    $PS = PRE\_SLOW$;
$else \ \ if \ \ (T_{avg} + M_{avg} * L_{PRE\_FAST} < D)$
    $PS = PRE\_FAST$;
$else \ \ if \ \ (T_{avg} + M_{avg} * L_{ACT\_PDN} < D)$
    $PS = ACT\_PDN$;
$else$
    $PS = ACT\_STBY$;

Figure 5: The power-state prediction algorithm for an upcoming task. L_{SELF_REF}, L_{PRE_SLOW}, L_{PRE_FAST} and L_{ACT_PDN} are the transition latencies from SELF_REF, PRE_SLOW, PRE_FAST and ACT_PDN state to ACT_STBY state, respectively.

Precharge Powerdown Slow (PRE_SLOW), or Self Refresh (SELF_REF). PRE_STBY is not considered as a viable state for the predictor as ACT_PDN offer better energy efficiency and lower delays than PRE_STBY. PRE_STBY should only be used as an intermediate connecting state when a lower power state (PRE_SLOW, PRE_FAST and SELF_REF) is selected, but not as a steady end state to run the task. Finally, *PS* is set in the memory controller which transitions a rank to the predicted power state after any access to that rank finishes. Therefore, a rank is transitioned to ACT_STBY state upon the first access request and then transitioned to *PS* after that access completes. Once a task completes and the CPU enters an idle state, all ranks are set to SELF_REF state and will remain in that state until a new memory request arrives.

Selection of ACT_STBY state indicates that the running task cannot tolerate any transition delays to finish before the deadline and thus the task must be executed at the highest performance. Selection of low-power states, on the other hand, indicates that the running task can tolerate power-state transition delays associated with the selected power state while still being able to meet the deadline. We should note that the calculations in Figure 5 assume the worst case scenario where memory I/Os are not clustered but arrive one at a time, since we are unable to capture the exact access patterns but only the total number of accesses during the task execution. Subsequently, the calculated delays are the maximum predicted delays the task may encounter, minimizing the possibility that the tasks would continue past the deadline. If memory accesses are bursty, arriving together, the actual exposed delays will be lower.

To avoid exposing potentially large delays to the users while still providing some energy savings, we utilize the ACT_PDN state during the training of the predictor, when the entry in the prediction table is not found. The ACT_PDN state significantly reduces energy consumption as compared to ACT_STBY while keeping the delays low. Finally, interaction IDs are unique across applications and thus can be maintained in a single table in the kernel across executions for all applications, further minimizing the impact of training.

3.5 Improving Prediction Granularity

IAMEM prediction mechanism described earlier utilizes only a single number of memory references to all memory spaces. Once the prediction is made, both user and kernel memory are maintained in the same predicted power state. If this state turns out to be ACT_STBY state, user memory occupied by the given task and the entire kernel memory will be fully powered during the task execution. This behavior may be detrimental to energy efficiency, if for example, the task is computationally intensive with low kernel activity. Subsequently, we extend the design of IAMEM by using two performance counters to monitor references to user and kernel memory individually. We further split MemoryRefs variable in Figure 4 into two fields UserRefs and KernelRefs. We note that user memory and kernel memory including the buffer cache are allocated into separate memory ranks to maximize management efficiency.

Similar to the previous algorithm, a dual prediction algorithm is proposed. In the dual prediction algorithm IAMEM first calculates the average task length T_{avg}, the average number of user memory references M_u and kernel memory references M_k. Then based on the allowed task extension E, IAMEM calculates the lowest combination of power states for user memory PS_u and for kernel memory PS_k, to keep the overall transition delays from both memory spaces below E. Finally, in the case of multiple concurrent threads the thread with the highest demand for memory will dictate the power state for memory.

3.6 Improving Monitoring Accuracy

So far, we have relied on monitoring memory accesses to estimate the worst-case transition overheads for a given task, by assuming that every memory access will require a power state transition. However, some of memory ref-

erences may arrive in a cluster and encounter one power state transition. To address this issue, we can use another performance counter to count the actual number of power state transitions that a given task encountered. This will allow IAMEM to update the variables of kernel and user memory reference in the prediction table with the actual number of memory references that encountered power state transitions. Subsequently, the algorithms in single and dual prediction remain unchanged except the *M* variables now correspond to the numbers of power state transitions that occurred in user and kernel memory. Utilization of the actual power state transitions accounts for traffic burstiness and as a result, eliminates the inaccuracy resulting from monitoring only memory accesses in the original design.

3.7 Preserving Performance

When a low-power state is predicted for a given task, the accumulated transition delays could become exposed to the user, resulting in performance degradation for longer tasks. The delays can be significant when the long task with high memory activity is mispredicted and the memory is kept in a low power state. To prevent excessive task extension, we propose an early-detect optimization to detect the possible user-perceivable degradation before the task completion. Observing that the scheduler in the kernel will interrupt task execution every 100ms to check if other processes should be scheduled, we add a monitoring module into the scheduler to monitor the given task execution for potentially missed deadlines. Every time the scheduler is invoked, the monitoring module reads the performance counter that counts the number of power state transitions from a low-power state to ACT_STBY state and calculates the actual delay that the current running task has encountered so far. Once the delay exceeds the allowed 50ms extension, the monitoring module will raise the memory power state to the next higher level. If the next higher selection is ACT_PDN state, the monitoring will continue. Seeing an additional delay of 50ms, due to the power state transitions, the memory power state is switched to ACT_STBY for the remaining task execution. This optimization will minimize the potential delays that are exposed to the user.

4 Methodology

We use trace-driven simulation to evaluate the proposed IAMEM and compare it with the following mechanisms:

- **PAVM**. Power-Aware Virtual Memory: The existing state-of-the-art per-process mechanism which keeps the ranks occupied by the currently running process and the ranks occupied by kernel memory in the ACT_STBY state during the process execution, while keeping all other ranks in SELF_REF.
- **ODPD**. On-Demand Powerdown: An existing mechanism that keeps all ranks in the system in the ACT_PDN state during execution and makes transitions to ACT_STBY on memory request arrival. This is the special case of the Dynamic Rank Power Down technique implemented in Intel Core i CPUs, with the idle timeout value set to zero to minimize energy consumption.
- **ODSR**. On-Demand Self Refresh: We propose a complementary mechanism to ODPD that keeps all ranks in the system in SELF_REF and transitions to ACT_STBY upon a memory request arrival.
- **ORACLE**. A per-task mechanism that utilizes the future knowledge to select optimal power states for ranks occupied by user and kernel memory for each incoming task.

Each of the evaluated mechanisms will put all ranks in the system to SELF_REF state when a task completes and the system begins idling. The simulator includes a task scheduler as well as a memory simulator. The memory simulator includes a memory controller and two DDR3-1066 DIMMs, each consisting of two 1GB ranks. The ranks are allocated to minimize fragmentation of memory for running processes across multiple ranks [13]. Finally, the energy management mechanism makes memory power decisions upon each UI event and the memory simulator executes the corresponding power-state transitions for the accessed ranks and calculates energy consumption according to Figure 2.

The application traces used in the simulation were collected using a modified Linux kernel 2.6.30 running on Intel Core i7-920 CPU with 4GB DDR3-1066. All traces contain data of UI events and process activity from a large number of usage sessions in the GNOME environment. UI events were collected with the modified X-Monitor, including the timestamp of each event and the interaction ID uniquely identifying the GUI component. Process activity traces were collected with *Linux Trace Toolkit* that logs program execution details from a patched Linux kernel. Based on the ordered event timestamps, we are able to simulate the dynamic execution progress of the traced applications, so that the computation time for each UI-triggered task can be calculated accurately.

We set up two performance counters: one for counting the event MEM_LOAD_RETIRED.L3_MISS that occurred in user mode, and the other for counting the same event in kernel mode [11] to measure last level (L3) cache misses. Each of the counters was read as soon as an UI event was captured, and was read again upon the completion of the triggered task. The difference of

Traced applications	Trace length	Num.of interaction IDs	Num.of short tasks	Num.of long tasks	Average task length	Average user memory ref.	Average kernel memory ref.
AbiWord	3.1 hrs	179	22840	2080	0.02 sec	7794	2378
Gnumeric	2.9 hrs	342	7804	1416	0.05 sec	19979	5013
Anjuta	3.9 hrs	458	14300	1768	0.03 sec	4127	2457
GIMP	3.3 hrs	228	3356	2440	0.14 sec	39409	13660
LiVES	2.6 hrs	166	2728	1208	0.51 sec	67319	381907
Memtester	0.1 hrs	1	0	20	167.78 sec	264112456	10389729

Table 1: Statistics of application traces.

the two counter values is considered as the number of references in user or kernel memory during this task execution.

We use five commonly executed interactive applications and one benchmark, shown in Table 1: *AbiWord* – a word processing application; *Gnumeric* – a spreadsheet application; *Anjuta* – an Integrated Development Environment for C/C++ and Java development; *GIMP* – an image processing application; *LiVES* – an integrated video editing and playback application; *Memtester* – a memory benchmark that intensively tests the performance of main memory. The traces were collected over a period of several hours and the trace length is shown in the second column. The number of interaction IDs presents the total number of unique user interactions in each application and serves as an indicator of the GUI complexity for the application. In case of *Memtester*, there is only one interaction to start the benchmark. Table 1 also lists the numbers of short tasks (shorter than 50ms), long tasks (longer than 50ms), and the average task length for each application. Finally, the average numbers of memory references in user and kernel memory specifically indicate the per-task demand on the memory subsystem.

4.1 Performance Demand of Applications

Figure 6 shows the distribution of task processing demand for each application based on ORACLE's optimal power selection for user and kernel memory that maximizes energy savings while eliminating delays exposed to the user. *AbiWord*, *Gnumeric*, *Anjuta* and *GIMP* have generally lower performance demand because most of their tasks require users to think to complete interactive operations such as editing text. Subsequently, user memory can stay in either SELF_REF or PRE_SLOW state for a majority of the time during the task execution.

On the other hand, *LiVES* work on large video clips, resulting in significantly higher demand for memory performance. As a result, user memory has to spend more of its time in the higher performance states (PRE_FAST, ACT_PDN and ACT_STBY) to prevent delays from being exposed. Finally, *Memtester* is a memory intensive

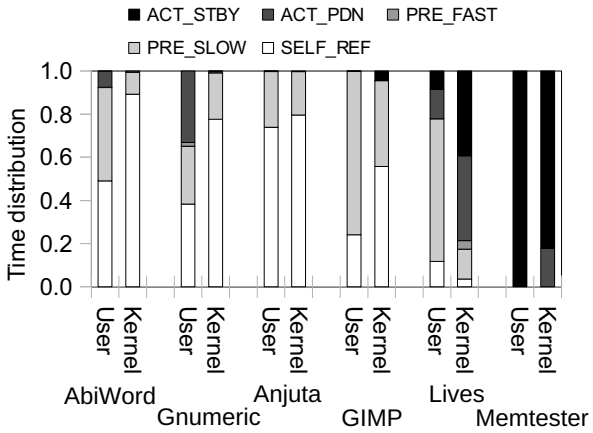


Figure 6: Task performance demand based on the ORACLE's optimal power.

benchmark that constantly reads and writes the memory and requires the maximum performance. Therefore, user memory must stay in the ACT_STBY state all the time to prevent user perceived delays.

Figure 6 also shows that the performance requirements from kernel memory is lower than user memory. This is because the applications usually invoke few system calls and perform most processing in their own virtual address space, creating lower kernel memory activity as shown in Table 1. Subsequently, kernel memory can stay for 77% of the time in SELF_REF state for *AbiWord*, *Gnumeric*, *Anjuta*, and *GIMP*. However, *LiVES* and *Memtester* demand higher performance from kernel memory and kernel memory must stay in the high power state for the majority of the time to prevent performance degradation.

5 Evaluation

5.1 Energy

Figure 7 shows the average per-task memory energy consumption for each mechanism and application. The energy bars are divided into energy consumed in five power states: ACT_STBY, ACT_PDN, PRE_FAST, PRE_SLOW, and SELF_REF. The energy consumption

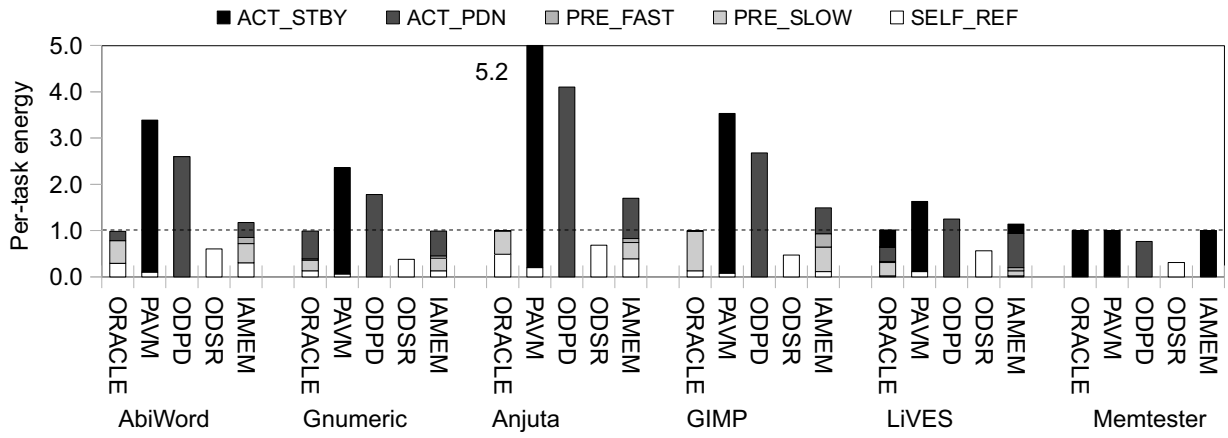


Figure 7: Average per-task memory energy consumption for processing each task normalized to ORACLE.

Applications	ORACLE	PAVM	ODPD	ODSR	IAMEM
AbiWord	1093.05	1503.50	1364.87	1028.45	1105.24
Gnumeric	1119.14	1444.48	1309.74	979.28	1118.74
Anjuta	1276.92	1698.20	1573.95	1245.92	1342.80
GIMP	1262.46	2029.42	1769.18	1100.90	1411.61
LiVES	1648.19	2313.06	1921.93	1202.21	1787.75
Memtester	5022.28	5025.04	3803.16	1547.72	4998.94

Table 2: Total memory energy consumption (in Joules) during the entire application runtime.

of each mechanism accounts for all ranks in the system during the task execution, and is normalized to ORACLE.

PAVM always keeps user and kernel memory in ACT_STBY state during process execution, therefore consuming the most energy, 187% more than ORACLE. ODPD follows PAVM with 119% more energy consumption than ORACLE, because it keeps all memory in ACT_PDN state during the task execution and it is clearly more than necessary for most tasks to eliminate user perceived delays, as show in Figure 6. Additionally, ODPD does not attempt to optimize energy as other mechanisms, which only transition the accessed ranks to the appropriate state. IAMEM closely matches the energy consumption profile of ORACLE through sophisticated demand matching prediction, yielding close to optimal energy efficiency. Subsequently, IAMEM shows less than 14% difference in energy consumption from ORACLE, reducing the energy consumption of PAVM and ODPD by 59% and 47% respectively. In the best case occurring in *AbiWord* and *Anjuta*, where PRE_SLOW and SELF_REF states combined can fit 94% of the time for user memory and 98% of the time for kernel memory, as shown in Figure 6, IAMEM yields as much as 69% improvement in energy efficiency as compared to PAVM and ODPD.

ODSR consumes the least amount of energy when we only consider main memory, yielding 26% less energy consumption than ORACLE, since all ranks are kept in SELF_REF state that has the lowest power demand. However, executing tasks with lower energy consumption than ORACLE is inefficient as it will expose delays to the user and may further increase the energy consumption of the entire system due to the longer runtime. Furthermore, ODSR misses the goal of this paper for transparent energy optimizations that do not expose delays to the user. This scenario also occurs for ODPD in *Memtester*. As shown in Figure 6, *Memtester* requires ACT_STBY state for most of the execution time to avoid performance degradation. However, ODPD utilizes ACT_PDN, and exposes delays to the user. IAMEM still performs almost the same as ORACLE in this case, since it keeps using ACT_STBY state for user memory as required, while recognizing the relatively less demand for kernel memory performance and using the lower ACT_PDN state when necessary.

While Figure 7 shows the energy consumption for processing tasks, it does not reflect the memory energy consumption over the entire execution time since the system idle time is not included. Each of the mechanisms puts all memory ranks in SELF_REF state when the system becomes idle, consuming the same amount of idle energy. Therefore, the overall improvements in energy efficiency originate from the energy savings ob-

tained during the task execution. Table 2 shows the total memory energy consumption during the whole program execution, including the idle time, for each application and each mechanism. As we can see, even considering the total execution time of several hours long (Table 1), IAMEM still gains significant energy savings over PAVM and ODPD, and matches energy consumption of ORACLE with less than 3% difference. For the first five applications, IAMEM consumes 25% and 15% less energy as compared to PAVM and ODPD, respectively. IAMEM energy reduction drops in *Memtester* due to the full power demand from the extremely memory-intensive tasks. Nevertheless, IAMEM still offers slight energy savings as compared to PAVM in this case.

5.2 Performance

While reducing energy consumption is important, preservation of the performance is the goal of this research. Performance degradation can negate any energy savings and negatively affect the user experience. Figure 8 shows the average task length for each application and each mechanism normalized to ORACLE. The task runtime is divided into: 1) task processing time; and 2) the power-state transition time due to the transitions from the lower power state to ACT_STBY state.

We notice that ORACLE introduces some amount of transition time into the task processing time as compared to PAVM that maintains the original task runtime. However, the shorter runtime in PAVM does not translate into better performance since users are not able to notice the shorter task completion and initiate any subsequent interactions. ORACLE always selects the best power state to fit the task execution within the user's perception threshold. The included power-state transition time in ORACLE is not exposed to the user, keeping the user behavior unchanged just like in PAVM. Similarly, ODPD executes most tasks at the higher performance level than desired, resulting in excess energy consumption as shown in Figure 7. Therefore, a task that is executing longer than its execution in ORACLE exposes noticeable delay to the user, while running the task faster is not energy efficient.

Due to the large transition latency (971.96ns) from SELF_REF state, ODSR incurs the most performance degradation, prolonging task execution by 54% on average as compared to ORACLE. *Memtester* exposes the worst case for ODSR with 160% more delay exposed to the user. Memory intensive tasks in *Memtester* result in noticeable delays even in case of ODPD, which only encounters 6ns transition latency for each memory access. IAMEM dynamically recognizes memory intensive tasks and provides appropriate power state similarly to ORACLE, only exposing slightly more than 1% delay to the user for each application. Combining the results

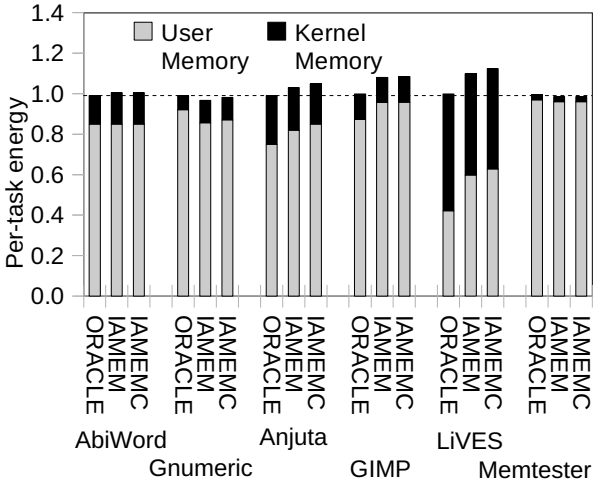


Figure 9: Energy consumption of IAMEM with single and dual prediction.

from Figure 7 and Figure 8, we observe that IAMEM is the most energy efficient mechanism and almost perfectly matches the behavior of ORACLE. This indicates that utilization of the interaction context allows IAMEM to accurately predict the demand placed on the system and achieve near-optimal energy efficiency without performance degradation.

5.3 The Need for Dual Prediction

Figure 7 showed IAMEM with dual prediction for user memory and kernel memory separately, as described in Section 3.5. Alternatively, IAMEM may also view user and kernel memory as a whole, predicting only a single power state for both memory spaces. Figure 9 compares the average per-task energy of IAMEM with single power prediction (IAMEMC), normalized to ORACLE. This separation allows us to study the contribution of energy consumed by user and kernel memory to the total per-task energy consumption.

As shown in Figure 6, difference in demand for kernel and user performance allows IAMEM to reduce energy consumption in both kernel and user spaces. Therefore, IAMEM reduces the combined energy consumption of user and kernel memory by 3%, on average, as compared to IAMEMC. This benefit of the dual prediction justifies the need for predicting power individually for user and kernel memory.

5.4 Delay Reduction with Early-Detect

Preserving performance and bounding delays exposed to the user is critical for overall energy efficiency and the user's satisfaction. Therefore, IAMEM adopts the early-

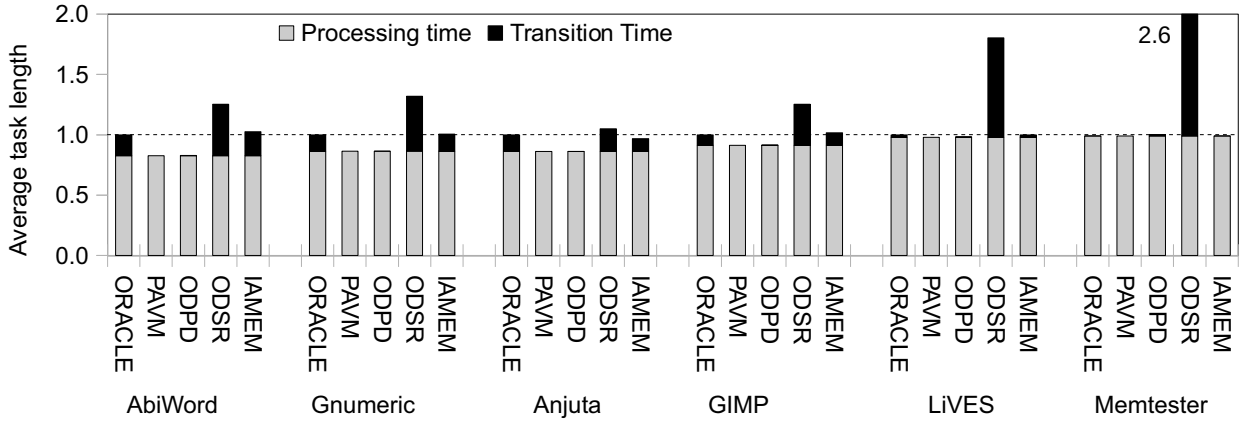


Figure 8: Average per-task length normalized to ORACLE.

detect optimization which uses the scheduler to check every 100ms as discussed in Section 3.7. This optimization will eliminate significant performance degradation for the long running tasks. Table 3 compares the average exposed delays of the long running tasks (longer than the 100ms scheduler interval) for IAMEM with and without this optimization. The delays shown only count the extra times that exceed the allowed 50ms extension and may be noticed by the user. We can see that IAMEM with this optimization significantly reduces the amount of delays exposed to the user for long running tasks. The final small extension above 50ms is at the lower end of the perception threshold range (50-100ms), and will be unnoticeable to the user.

Alternatively, we can remove the optimization since IAMEM without early-detect still manages to keep the delays reasonable that may not be noticed by most users. In addition, IAMEM without early-detect would reduce the per-task energy consumption in Figure 7 by additional 2%, on average. However, the goal of this research is to maximize energy savings without affecting the user experience. The early-detect optimization is critical in achieving this goal, and we subsequently included it in IAMEM implementation and all previous results reflect the inclusion of this optimization.

Application	Num.of long running tasks	Num.of delayed long running	Delay w/ early-detect	Delay w/o early-detect
AbiWord	1808	380	4.5 ms	5.0 ms
Gnumeric	800	220	7.6 ms	9.8 ms
Anjuta	984	120	3.5 ms	13 ms
GIMP	1536	408	13 ms	15 ms
LiVES	760	188	10.8 ms	11.7 ms
Memtester	14	2	51 ms	53 ms

Table 3: Average delays of the delayed long running tasks in standard IAMEM with the early-detect optimization and alternative design without the optimization.

6 Related Work

Energy management for main memory can be implemented in hardware, software, or the combination of both. Hardware level approaches generally utilize the memory controller to monitor the memory traffic as well as the access pattern, and make the power state transitions for specific memory devices based on the observed energy-saving opportunities. Lebeck et al. [12] studied the interaction of page placement with static and dynamic hardware policies to reduce memory power dissipation. The cooperation between the hardware and the OS was also studied in [12]. Subsequently, Pisharah et al. [22] proposed another approach to save memory energy by introducing a hardware called Energy-Saver Buffers to hide the resynchronization costs when reactivating memory modules. Fan et al. [7] further investigated memory controller policies for manipulating DRAM power states in cache-based systems and developed an analytic model that approximates the idle time of DRAM chips using an exponential distribution. Furthermore, observing that significant energy is consumed when memory is actively idle during DMA transfers, Pandy et al. [21] proposed several energy management mechanisms to improve the concurrency level between multiple I/Os to maximize the memory utilization.

Hardware-level energy management may suffer from inaccuracy and may cause unexpected performance degradation. Software-level energy management, on the other hand, can provide more detailed context of execution to make timely power state transitions. Delaluz et al. [5] proposed a compiler-directed approach to cluster the data across memory banks and insert power-state

transition instructions into programs by profiling. However, compiler-directed schemes can only work on a single application at a time and demand sophisticated program analysis support. To address the issue, Delaluz et al. [6] also proposed an operating system based solution where the OS scheduler directs the power state transitions by keeping track of accesses for each process in the system. Subsequently, Huang et al. [8] proposed Power-Aware Virtual Memory that manages the power states of memory devices on a per-process basis. A cooperative software-hardware mechanism [10] was further proposed to combine PAVM and the underlying hardware. Huang et al. [9] also proposed memory-reshaping mechanisms that coalesce short idle periods into longer ones through page migration to maximize energy savings. Targeting the buffer cache, Bi et al. [2] utilized the OS I/O handling routines to hide the delays due to memory power state transitions to minimize the impact of aggressive energy management. Finally, Li et al. [15] proposed a mechanism to guarantee the performance by temporarily disabling memory energy management.

7 Conclusion

As current applications are becoming more data-centric, computer systems are equipped with larger capacity and higher performance main memory. As a result, energy consumption of main memory is significantly increasing. In this paper, we addressed interactive systems where most tasks are initiated by the user, and presented the design of IAMEM, a unified approach to manage the energy consumption of the entire memory space. By correlating the memory performance demand to the user interactions, IAMEM is able to accurately select suitable memory power states for UI-triggered tasks, saving energy while preserving the performance of the system. We have shown that compared to the state-of-the-art mechanisms, IAMEM saves 28%-68% of the memory energy consumed for task processing, resulting in up to 16% reduction of the total memory energy consumption during the entire program execution. In addition to the significant energy savings, IAMEM also successfully maintains the user-perceivable performance by hiding delays associated with energy management.

8 Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 0844569.

References

[1] BI, M., CRK, I., AND GNIADY, C. Iadvs: On-demand performance for interactive applications. In *HPCA* (2010).

[2] BI, M., DUAN, R., AND GNIADY, C. Delay-hiding energy management mechanisms for dram. In *HPCA* (2010).

[3] CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. The information visualizer, an information workspace. In *CHI '91* (New York, NY, USA, 1991), ACM, pp. 181–186.

[4] CRK, I., BI, M., AND GNIADY, C. Interaction-aware energy management for wireless network cards. In *SIGMETRICS* (2008).

[5] DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., SIVASUBRAMANIAM, A., AND IRWIN, M. J. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers 50*, 11 (2001), 1154–1173.

[6] DELALUZ, V., SIVASUBRAMANIAM, A., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. Scheduler-based dram energy management. In *DAC* (2002), ACM, pp. 697–702.

[7] FAN, X., ELLIS, C., AND LEBECK, A. Memory controller policies for dram power management. In *ISLPED '01* (New York, NY, USA, 2001), ACM, pp. 129–134.

[8] HUANG, H., PILLAI, P., AND SHIN, K. G. Design and implementation of power-aware virtual memory. In *ATEC '03* (Berkeley, CA, USA, 2003), USENIX Association, pp. 5–5.

[9] HUANG, H., SHIN, K. G., LEFURGY, C., AND KELLER, T. Improving energy efficiency by making dram less randomly accessed. In *ISLPED* (New York, NY, USA, 2005), ACM, pp. 393–398.

[10] HUANG, H., SHIN, K. G., LEFURGY, C., RAJAMANI, K., KELLER, T. W., HENSBERGEN, E. V., AND III, F. L. R. Software-hardware cooperative power management for main memory. In *PACS* (2004), vol. 3471, Springer, pp. 61–77.

[11] INTEL. Intel core i7-800 and i5-700 desktop processor series, 2009. Intel Documentation.

[12] LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. Power aware page allocation. *SIGPLAN Not. 35*, 11 (2000), 105–116.

[13] LEE, M., SEO, E., LEE, J., AND KIM, J.-S. Pabc: Power-aware buffer cache management for low power consumption. *IEEE Transactions on Computers 56*, 4 (2007), 488–501.

[14] LEFURGY, C., RAJAMANI, K., RAWSON, F., FELTER, W., KISTLER, M., AND KELLER, T. W. Energy management for commercial servers. *Computer 36*, 12 (2003), 39–48.

[15] LI, X., LI, Z., ZHOU, Y., AND ADVE, S. Performance directed energy management for main memory and disks. *Transactions on Storage 1*, 3 (2005), 346–380.

[16] LORCH, J. R. Using user interface event information in dynamic voltage scaling algorithms. In *MASCOTS* (2003), pp. 46–55.

[17] MICRON. Ddr3 memory power calculator. Micron Documentation and Support.

[18] MICRON. 1gb: x4, x8, x16 ddr3 sdram features, 2009. Micron Documentation and Support.

[19] MICRON. Calculating memory system power for ddr3, 2009. Micron Tech Notes.

[20] MILLER, R. B. Response time in man-computer conversational transactions. In *AFIPS '68 (Fall, part I)* (New York, NY, USA, 1968), ACM, pp. 267–277.

[21] PANDEY, V., JIANG, W., ZHOU, Y., AND BIANCHINI, R. Dma-aware memory energy management. In *HPCA* (2006), IEEE Computer Society, pp. 133–144.

[22] PISHARATH, J., AND CHOUDHARY, A. An integrated approach to reducing power dissipation in memory hierarchies. In *CASES* (New York, NY, USA, 2002), ACM, pp. 88–97.

[23] SCHNEIDERMAN, B. *Designing the user interface: strategies for effective human-computer interaction.* Addison-Wesley, 1998.

XLH: More effective memory deduplication scanners through cross-layer hints

Konrad Miller Fabian Franz
Marc Rittinghaus Marius Hillenbrand Frank Bellosa

Karlsruhe Institute of Technology (KIT)

Abstract

Limited main memory size is the primary bottleneck for consolidating virtual machines (VMs) on hosting servers. Memory deduplication scanners reduce the memory footprint of VMs by eliminating redundancy. Our approach extends main memory deduplication scanners through Cross Layer I/O-based Hints (XLH) to find and exploit sharing opportunities earlier without raising the deduplication overhead.

Prior work on memory scanners has shown great opportunity for memory deduplication. In our analyses, we have confirmed these results; however, we have found memory scanners to work well only for deduplicating fairly static memory pages. Current scanners need a considerable amount of time to detect new sharing opportunities (e.g., 5 min) and therefore do not exploit the full sharing potential. XLH's early detection of sharing opportunities saves more memory by deduplicating otherwise missed short-lived pages and by increasing the time long-lived duplicates remain shared.

Compared to I/O-agnostic scanners such as KSM, our benchmarks show that XLH can merge equal pages that stem from the virtual disk image earlier by minutes and is capable of saving up to four times as much memory; e.g., XLH saves 290 MiB vs. 75 MiB of main memory for two VMs with 512 MiB assigned memory each.

1 Introduction

In cloud computing, virtual machines (VMs) permit the flexible allocation and migration of services as well as the consolidation of systems onto fewer physical machines, while preserving strong service isolation. However, in that scenario the available main memory size limits the number of VMs that can be colocated on a single machine.

There may be plenty of redundant data between VMs (inter-vm sharing), e.g., if similar operating systems (OSes) or applications are used in different VM instances. Moreover, previous studies have shown that the memory footprint of VMs often contains a significant amount of pages with equal content within a single instance (self-sharing) [3]. In both cases, memory can be freed by collapsing redundant pages to a single page and sharing it in a copy-on-write fashion. However, such pages cannot be identified using traditional sharing mechanisms (see § 6.1) as the isolation of VMs leads to the so-called semantic gap [8]; that is lost semantic information between abstraction layers. The host, for example, does not know which ones of the guests' memory pages represent file contents.

Prior work has made deduplication of redundant pages possible and thereby lowered the memory footprint of guests. In the following, we use *host* interchangeably with virtual machine monitor (VMM), hypervisor, or host OS to describe the system layer underneath the guest OS.

Paravirtualization closes the semantic gap through establishing an appropriate interface between host and guest [6, 18] to communicate semantic information. This implies modifying both host and guest.

Such an interface has previously been used to help deduplicating *named* memory pages—memory pages backed by files: Satori [18] successfully merges named pages in guests employing sharing-aware virtual block devices in Xen [2]. Paravirtualization-based approaches have only been used selectively and rudimentarily to make sharing of *anonymous* memory (e.g., heap/stack memory) possible, through hooking calls such as `bcopy` [6].

Applying these modifications to all guests and keeping them compatible with the latest developments at the kernel and hypervisor level is at least a great burden. It might not even be possible at all to modify commercial or legacy guests due to license restrictions or the lack of source code. Moreover, the lack of semantic information that the host has about guest activities is actually one of the key features of virtualization: The host does not know nor needs to know the OS, file system, etc. inside the VM.

Memory scanners mitigate the semantic gap by scanning for duplicate content in guest pages [1, 25]. They index the contents of memory pages at a certain rate, regardless of the pages' usage semantics.

Scanners have their downside when it comes to efficiency. Especially the merge latency, the time between establishing certain content in a page and merging it with a duplicate, is higher in systems based on content scanning compared to paravirtualization-based systems that merge pages synchronously when they are established.

Memory scanners trade computational overhead and memory bandwidth with deduplication success and latency. Although the scan rate (pages per time interval) is often variable and may be fine-tuned [10, 23], it is generally set to scan very slowly to keep the scanner's CPU and memory bus resource usage low. The default scan rate for Linux/KSM is 1000 pages per second, which results in a scan time of almost 5 minutes per 1 GiB of main memory.

XLH is our contribution that combines the key benefits of both previous approaches. We have observed that:

- All types of memory contents (named and anonymous) contribute to memory redundancy.
- Many shareable pages in the host's main memory originate from accesses to background storage: when multiple VMs create or use the same programs, shared libraries, configuration files, and data from their respective virtual disk images (VDIs).

The main contribution of this paper is to observe guest I/O in the host and to use it as a trigger for memory scanners in order to speed up the identification of new sharing opportunities. For this purpose, XLH generates page hints in the host's virtual file system (VFS) layer, whenever guests access their background store. XLH then indexes these hinted pages soon after their content has been established and thus moves them earlier into the merging stage. In consequence, XLH can find short-lived sharing opportunities and shares redundant data longer than regular, linear memory scanners without raising the overall scan rate.

We have implemented our approach in Linux' Kernel Samepage Merging (KSM) and evaluated its properties. Measurements of kernel build and web server scenarios show that XLH deduplicates equal pages that stem from the VDI earlier by minutes and is capable of merging between 2x and 5x as many sharing opportunities than the baseline system. For the kernel build benchmark, XLH performs constantly better than KSM even if the scan rate is set 5x lower. Our evaluation shows that XLH is able to reach its effectiveness with little to no additional CPU overhead or loss in I/O throughput compared with KSM.

We only modify the *host* in our approach—XLH would not benefit from and thus does not make use of paravirtualization. In fact, due to the generality of our approach, XLH also works for deduplicating native processes when no virtualization is involved. Note that XLH does not solely target disk accesses but issues hints for all I/O that goes through the VFS interface, including network file systems such as NFS. Overall, I/O-advised scanning makes more effective detection of sharing opportunities possible without the need to modify guests.

The remainder of this paper is structured as follows: We analyze semantic and temporal memory duplication properties in the following Section 2 to back up and motivate our approach. We then review Kernel Samepage Merging (KSM)—the memory scanning basis for our implementation—in Section 3 before we describe our approach and the implementation of our prototype thoroughly in Section 4. In Section 5, we present the results of our evaluation. We give an overview of related work on memory deduplication in Section 6. Finally, we conclude and depict future research directions in Section 7.

2 Analysis of Memory Duplication

Whether the use of deduplication techniques is effective or not depends mainly on the target workload. Using memory deduplication does improve a system's memory density if the memory footprint of the hosted applications is sufficiently similar. This is generally the case if the same OS, similar programs/libraries and/or data are used.

Duplication quantity An empirical study on memory sharing of VMs for server consolidation performed by Chang et al. found that the amount of redundant pages can be as low as 11% but also as high as 86% depending on the OS and workload [7]. Gupta et al. measured the amount of duplicated memory across three VMs and found that almost 50% of the allocated memory could be saved through memory deduplication [12]. We have performed a study ourselves and found 110 MiB of redundant memory in typical desktop workloads (LibreOffice, Firefox). We moreover measured 400 MiB (39%) of redundant data in one of our benchmarks (§ 5.2).

Duplication sources The sources of duplicated pages and their distribution vary greatly between workloads. Barker et al. measured the number of identical page frames in Ubuntu Linux 10.10 while running a typical set of desktop applications. In their study, over 50% of identical page frames stem from process heaps. They furthermore identified shared library based pages to be the second largest source of duplication (43%) [3]. In a study performed by Kloster et al., between 64% and 94%

of redundant data were located in page caches [14]. In our own aforementioned analysis of desktop workloads 70% of the duplicate pages were part of the page caches while 14% of the duplicates were not backed by files (anonymous). The last 15% were either free or reserved (e.g., driver pages). Although it seems to be favorable to focus on named pages, as many sharing opportunities can be found in page caches, a significant amount of duplicates may stem from anonymous memory regions. In consequence, for a deduplication system to be effective, it needs to exploit sharing opportunities from all sources.

Temporal characteristics of duplicates In our kernel build benchmarks, 80% of all encountered sharing opportunities lived between 30 seconds and 5 minutes. In this scenario, using brute-force scanning to detect short-lived sharing opportunities is not effective. Additionally, it wastes sharing potential by identifying longer-lived sharing opportunities late in the scan process. Figure 1 depicts the significance of the merge latency on how many pages are shared at any given point in time: The later memory is indexed by the scanner, the later a shared page can be established. Indexing sharing opportunities earlier adds to the sharing potential and to longer sharing time-frames.

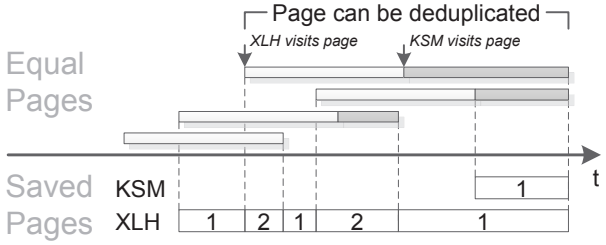


Figure 1: Memory scanners index pages after an expected value of half a scan cycle. XLH visits I/O pages immediately after they are established. If a duplicate is not found until a large proportion of the mean sharing time is over, the deduplication effectiveness is lowered significantly.

3 Memory Scanning with KSM

Our prototype is based on Kernel Samepage Merging (KSM) [1] which is a popular memory deduplication approach in the Linux kernel. KSM single-threadedly scans for and merges equal main memory pages. It is not bound to VMs but works on anonymous memory regions of any process. However, KSM only regards specifically advised pages (`madvise`) as mergeable. QEMU [4] invokes the appropriate call for the memory of each VM.

Page States Every advised page in the host is in one of three states: (1) frequently fluctuating, (2) sharing candidate yet potentially unstable, and (3) shared.

Data Structures KSM allocates a tree-node containing information such as a checksum and sequence number linked to every advised virtual page in the host. Pages that have changed between scan rounds (1) are not recorded or regarded in the scan process until their modification frequency decreases. The tree-nodes of all other pages are linked together into two red-black trees using their pages' full content as the key/index. The *unstable tree* (2) records pages that do not change frequently and are in consequence suitable sharing candidates. They are neither shared yet, nor protected from being written to—their content is thus not stable and may be modified after insertion. The *stable tree* (3), in contrast, stores pages that have already been merged and marked copy-on-write.

Scan Process KSM searches for pages that do not change frequently by gradually calculating a hash value for every page. If the calculated hash differs from the one recorded in the previous scan round, the record is updated but the page is not inserted into either of the trees (1). If the hash value has not changed between scan rounds, the associated page is inserted into the unstable tree (2), employing its content as key. If the unstable tree already contains a page with the same content, the pages are merged, marked read-only, and inserted into the stable tree (3). For any subsequently scanned page, KSM first checks if its content matches a page in the stable tree, in which case the pages are merged immediately.

When all advised pages have been scanned, the unstable tree is dropped and the process is repeated. Only the hash values and the stable tree remain.

4 I/O-Advised Deduplication with XLH

In the following paragraphs we discuss how XLH generates (§ 4.1), stores (§ 4.2), and processes (§ 4.3) deduplication hints interleaved with the periodic memory scan. KSM uses the full page content as the index into its trees. Writing to pages in the unstable tree is not prohibited; such writes, however, may break the reachability in the subtree of that page, thereby lowering the deduplication effectiveness. We present two solutions in § 4.4.

4.1 Generating Deduplication Hints

When a VM reads data from a virtual disk image (VDI), the virtual DMA controller in the host handles the request and reads the physical disk on behalf of the guest (Figure 2). Our assumption is that the target of that DMA transaction is a page in the guest's page cache and thus a good sharing candidate. We assume the same for writes: When a page cache page in the guest is flushed to disk—a new file is created or an old file is written— the host trans-

lates this into a write to the VDI. XLH detects these operations and generates deduplication hints for the source and target guest pages. In contrast to read operations on non-cached files, writes in the guest may be not immediately visible to the host as they can be delayed by the guest's page cache. Generally, the delay is much shorter than the time to the next visit of a traditional memory scanner, however.

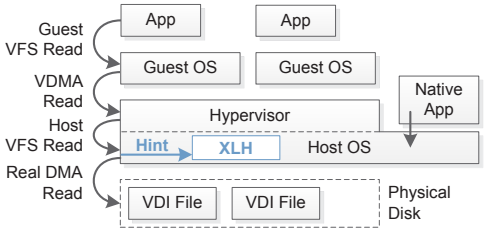


Figure 2: Host-VFS read and write operations are used to trigger hints for the main memory scanner.

As opposed to previous I/O-based approaches, we did not modify the guests in any way. Our mechanism even works for regular, non-VM processes, thereby enabling XLH to also deduplicate main memory efficiently in native environments (e.g., when using Zero-Install applications). Note that our approach is generic and may be applied to other environments as well, e.g., the Win32 file API layer. Moreover, the hint generation is fully decoupled from the deduplication process; there might as well be more than one hint source and other triggers for hints, e.g., from a page fault handler.

4.2 Storing Hints and Coping with Bursts

Hints need to be stored until they are asynchronously processed by the memory scanner. Memory scanners adhere to a certain scan rate, which is generally set to a low value (e.g., 1000 pages per second) to keep the overall impact of the scanner on the system performance within reasonable bounds. This way, however, memory scanners cannot always keep up with processing the number of incoming hints. The hint rate may be constantly higher than the scan rate, leading to an ever-growing buffer and suggesting the use of a pruning mechanism. Moreover, I/O is bursty; in consequence, I/O-based hints are also issued in bursts. Some million hints can be generated in a matter of seconds leading to a long backlog of hints and thus to outdated hints by the time the scanner gets to them. This effect calls for an aging mechanism.

We have at first stored our hints in an unbounded queue. When running our benchmarks, the system eventually fell behind to a state in which it could not find *any* sharing candidates through the hinting mechanism at all, as the hinted pages had already changed their content before they were processed.

A *bounded circular hint-stack* (Figure 3), however, proved to be an appropriate data structure to store hints with low overhead. The hint-stack keeps the history of the last unprocessed `stack_size` disk accesses.

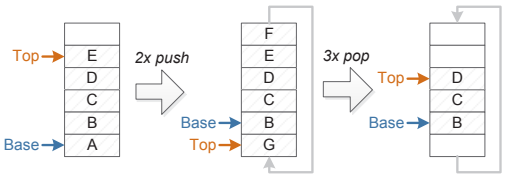


Figure 3: Storage of hints in the bounded circular stack.

Due to the nature of a bounded circular stack, XLH always processes the newest hints first while old hints are overwritten when the stack is full—an automatic pruning and aging mechanism which turned out to be fast and robust. Periodic maintenance is not required.

The stack size is configurable through procfs. In our benchmarks we found that XLH shares most pages if a full stack can be processed by the memory scanner within about 15 to 30 seconds. At KSM's default scan rate this results in a stack size of about 8k to 16k entries.

4.3 Processing Deduplication Hints

Our hint processing loop, depicted in Figure 4, runs interleaved with the full system scan spurts (wake-ups) that KSM already implements. XLH shares the global rate limit set for KSM and produces roughly the same CPU-load as an unmodified KSM with the same settings.

The interleaving ratio is configurable; `hint_runs` hint-processing spurts are interleaved with `scan_runs` scan spurts. A ratio of 0:1 corresponds to the original KSM implementation. Our default ratio is 1:1. Using this policy, XLH can guarantee that the linear scan, which can catch non-I/O sharing opportunities, does not starve due to a flood of hints.

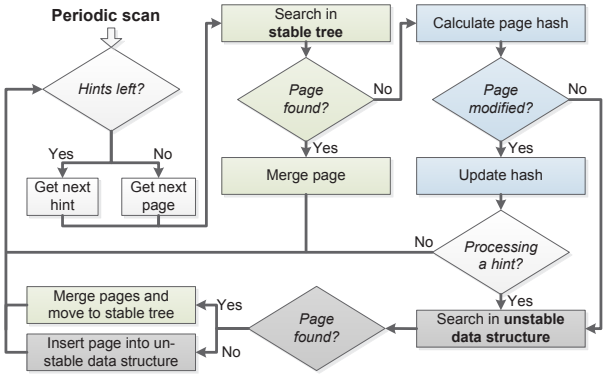


Figure 4: The high-level workflow of our new hint processing loop. When all hints have been processed before the scan rate is exceeded, XLH continues with the linear KSM scan to keep the rate constant.

When XLH is in a scan spurt, it runs the traditional linear scanning policy, only. In a hint-processing spurt, however, XLH processes hints as long as it has hints left and it has not exceeded its scan allowance. The remaining scan slots are then used for the linear scan.

Our mechanism first checks whether the hinted page's content is already in the stable tree. In this case, XLH remaps the page to the one in the stable tree and frees the hinted page. If the hinted page is not in the stable tree, XLH calculates the checksum of the page's content and checks the unstable data structure. If a sharing partner is found, XLH merges the pages and moves the resulting page into the stable tree. If XLH cannot find a sharing candidate it adds the page to the unstable data structure.

4.4 Degeneration of the Unstable Tree

The original KSM implementation has a heuristic that keeps frequently written pages from being inserted into the unstable tree. Only pages that keep the same hash value between consecutive scan rounds are considered (see Section 3). XLH however adds pages to the unstable data structure when processing hints that do not have a sharing partner at that point in time.

As pages are *not* marked read-only on insertion into the unstable tree but remain writable for the VM, the location in the tree is purely based on the content the page had at the time of its insertion. If pages in the unstable tree are subsequently modified, the tree may degenerate and entire branches may become unreachable (see Figure 5).

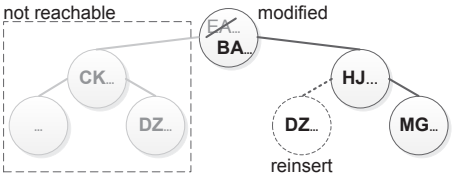


Figure 5: Nodes in the unstable tree may become unreachable due to modification of the contained pages. In this example, the first byte of the content of a page has changed from 'EA...' to 'BA...' leading to a second page with the content starting with 'DZ...' to be inserted a second time. The page duplication is not detected.

Although, the pages associated with virtual DMA operations are generally part of the guest's page cache, and are thus modified infrequently (see Section 2), the effect of the degeneration is not negligible and reduces the effectiveness of the merging stage. Even running Linux with an *unmodified* KSM reveals that almost 70% of the nodes cannot be reached in the unstable tree after a full scan, due to page modifications after insertion (kernel benchmark).

In KSM, a very radical approach is chosen to clean up broken branches of the unstable tree: When a full scan has been performed, the entire unstable tree is dropped and a new one is built from scratch. KSM's repair mechanism is slowed down by our modifications as the number of full memory scan cycles per time decreases: The scan rate stays constant, but multiple hints can and will be issued on the same pages during a scan cycle leading to multiple visits of pages within a scan round. As XLH worsens the unstable tree's degeneration we provide two possible solutions in our implementation and compare their characteristics in § 5.4:

Read-Only Unstable Tree Nodes One way to counter the more likely degeneration of the unstable tree is to mark hinted pages that are inserted into the unstable tree as read-only. This way XLH can use write faults on hinted pages as a signal to remove these pages from the unstable tree and thereby prevent the tree from degenerating when hinted pages are modified. That is not the same mechanism as breaking COW pages, which happens when writing to a page in the stable tree. The page does not need to be copied but is only marked read-write and removed from the unstable tree.

Unstable Hash Table An alternative option is to replace the unstable tree with a hash table. When a page in the table is modified, the reachability of other pages in the hash table is not affected as there are no inner nodes that can be broken. However, we have to pay attention to the runtime effects of a hash table. Traditional hash tables work well for a fixed working set size.

5 Evaluation

We were particularly keen to see whether XLH can merge more pages with an overhead that is comparable to KSM, our baseline system. Consequently, we chose the amount of main memory that is saved when deduplicating different workloads with fixed computational and memory overhead settings as our prime metric in the evaluation. After describing our benchmark setup in § 5.1, we explore the deduplication effectiveness for several different workloads in § 5.2. As we wanted to get results that are relatable to prior publications, we have chosen two of the benchmarks that were used to evaluate Satori [18]: Compiling the Linux kernel and the Apache web server performance when serving static files to httperf [19]. We have also mixed both benchmarks. Additionally, we have measured how long it takes for the baseline system as well as XLH to deduplicate the almost static memory footprint when solely booting many VMs.

We have confirmed that the overhead stays in the area of the baseline system using three metrics: Time spent in the deduplication stage, total time the benchmarks require from start to completion, and the CPU usage (via

top) during the execution. To push our hint generation and storage implementation to its limit, we have complemented our kernel build and Apache benchmarks with the file system benchmark bonnie++ [9]. Details can be found in § 5.3.

Finally, we have explored the runtime impact of our two solutions to the degenerating unstable tree problem. In § 5.4, we show that both solutions lead to comparable deduplication performances.

5.1 Benchmark Setup

We integrated XLH in Linux 3.4 and use QEMU [4] with KVM—a popular virtualization environment—in our benchmarks. KSM already provides data structures, mechanisms and the linear scanning policy for memory deduplication. We extended the Linux kernel by only around 600 SLOC.

All benchmarks have been conducted on a PC with an Intel i7 quad-core processor, 24 GiB RAM, and an SSD. Ubuntu 11.04 served as the host and also as the guest OS. Guests were assigned one VCPU each.

Unless specifically stated otherwise, we use the parameters in Table 1 for our benchmarks. The mapping of the sleep-time between spurts and the number of slots in the hint buffer are listed in Table 2. Intuitively, one would need a larger hint buffer for longer wake-up intervals as more hints aggregate between runs. Recall that XLH uses the fixed size of the hint buffer for pruning outdated hints.

Parameter	Value	Description
scan_run	1	Interleave each scan spurt...
hint_runs	1	... with one hint spurt
pages_to_scan	100	# of pages to scan on wake-up
hash_table_size	256 K	# of unstable hash slots
RAM size	512 MiB	Size of virtual main memory

Table 1: Default settings in our various benchmarks.

sleep_time	20 ms	100 ms	200 ms
stack_size	40960	8192	4096
full scan time	44 s	220 s	440 s

Table 2: The mapping of sleep time and hint buffer slots in our benchmarks. The time of a full scan cycle for two VMs with 512 MiB each is also shown.

In our experiments, we first determine the maximum available sharing opportunities without merging pages to show how far from the optimum the different approaches are. That is achieved with a kernel module comparable to Exmap [5], which once per second dumps page table information and page content digests. Then we re-run the experiments with different configurations of KSM and XLH. Internal information and statistics such as the number of exploited sharing opportunities are directly dumped from the deduplication code through sysfs.

5.2 Deduplication Effectiveness

The goal of XLH is to increase the memory density of virtualized environments by identifying sharing opportunities more quickly. When equal pages are identified earlier, the time that those pages are shared is extended. Furthermore, new sharing opportunities can be detected and shared which were previously not exploited due to slow scan cycles.

The metric we use to compare XLH with the baseline system KSM is the merge effectiveness at equal scan rates and thus at equal load settings. We define the merge effectiveness as the number of merged pages at a certain point in time after starting the benchmark. A steeper rise in those graphs indicates that more pages are merged in a given time interval, which is a consequence of fewer pages being checked before merging a page. A higher level in those graphs indicates a greater amount of saved memory and thus a better approach in terms of effectiveness.

In the following benchmarks, we compare XLH in both implementation flavors, read-only tree (**XLH RO**) and hash table (**XLH HT**), which have been described in § 4.4, with the KSM scanner in its vanilla implementation (**KSM**) as well as with an improved KSM version that marks the unstable tree read-only to mitigate unstable tree degeneration (**KSM RO**).

Booting many VMs One of the main reasons to do memory deduplication in a virtualization scenario is to be able to quickly consolidate many VMs on a single physical machine. TravisCI [22] spawns a new VM for every compute job they run for customers. Jobs often run shorter than 5 minutes, however. XLH performs particularly well when it comes to booting VMs as most of the boot process consists of loading secondary storage contents (programs, libraries) into main memory.

We have booted 25 VMs in parallel, starting 10 seconds apart. When using *XLH* all VMs were fully booted after 530 seconds using approximately 5 GiB of physical memory. With KSM, the total boot time has been almost exactly the same. However, up to this point KSM had only merged 53% of the sharing opportunities that XLH had had merged.

Kernel Build We have compiled the Linux kernel in two VMs on the same host. Before performing the benchmark, we have fully booted the VMs and waited until the static sharing opportunities were shared. The resulting deduplication effectiveness for the kernel build at different scan rates is depicted in Figure 6.

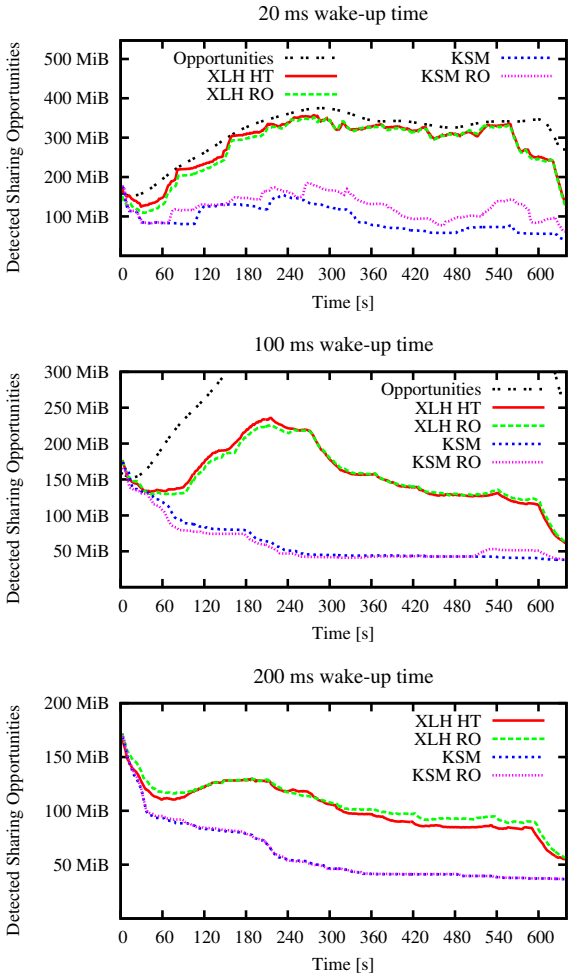


Figure 6: Kernel build merge performance with varying wake-up times. The first two graphs also show the maximum sharing opportunities.

In this benchmark, our extension always performs significantly better than KSM. For a short wake-up interval of 20 ms XLH shares almost all available sharing opportunities. Even with those very aggressive settings, where KSM occupies about 70% of a CPU core for its scan process, XLH can merge 2x to 5x as many pages as KSM. Both KSM and XLH are currently not multi-threaded and thus limited by the speed of a single CPU core. For longer wake-up intervals XLH also deduplicates 2x to 3x more effectively. In this benchmark, XLH even deduplicates more effectively than KSM if it scans 5 times slower (Figure 7).

The following numbers are taken from the 20 ms benchmark: XLH detects and shares almost 10 times as many new sharing opportunities in total (172000 vs. 17500). When considering the sharing opportunities that both systems detect, XLH detects those opportunities 243 seconds earlier (median) than KSM. The histogram of the time

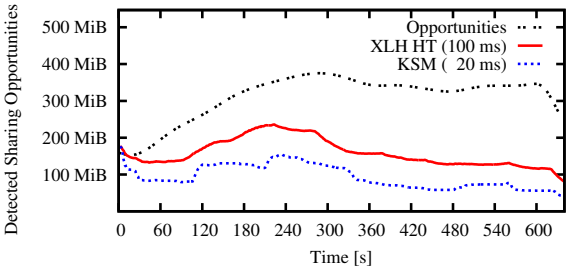


Figure 7: XLH performs constantly better than KSM in the kernel build even if the scan rate is 5x lower.

that pages remain shared while running the benchmark (Figure 8) shows that we can find many additional, short-lived sharing opportunities that KSM is not capable of detecting.

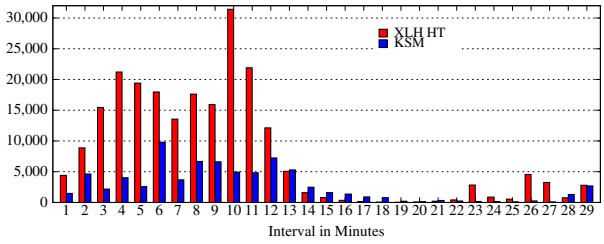


Figure 8: Histogram depicting the time. Sharing opportunities stay shared throughout the kernel-build benchmark.

Apache We have set up an Apache web server to serve random files in each one of two VMs. One httperf instance per server requests files in a predefined order.

The request order cannot be randomized directly in httperf. To make this benchmark less deterministic than the previous kernel build, we emulate random access patterns by statically shuffling the *file names* of the generated, served files in the servers. This way, when httperf accesses the same file name on both instances, different files will be returned; files with the same content will in consequence be returned at different times in the benchmark.

The total size of the served files exceeds the size of the page cache in each VM. Yet, parts of the guests' page caches overlap and therefore sharing opportunities exist even though file accesses are random.

We have configured httperf to establish 24000 connections per VM and to request 20 objects per second through each one of the connections from the Apache web servers. The merge performance of different scan rate configurations can be found in Figure 9.

When XLH cannot keep up with processing the stream of requests and constantly drops hints, our effectiveness is lowered significantly. XLH needs to process matching hints from both virtual machines in order to merge the

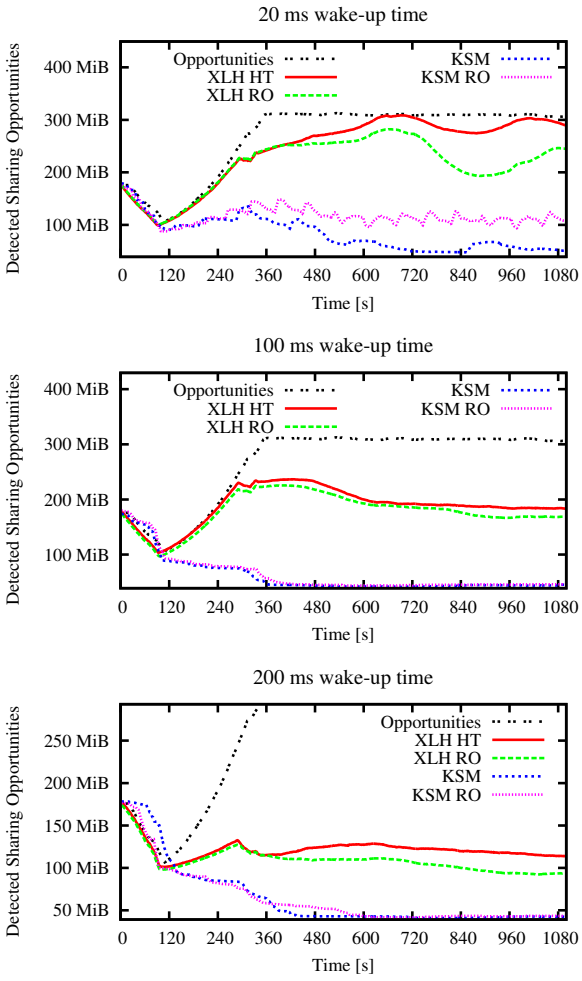


Figure 9: Apache/httperf merge performance.

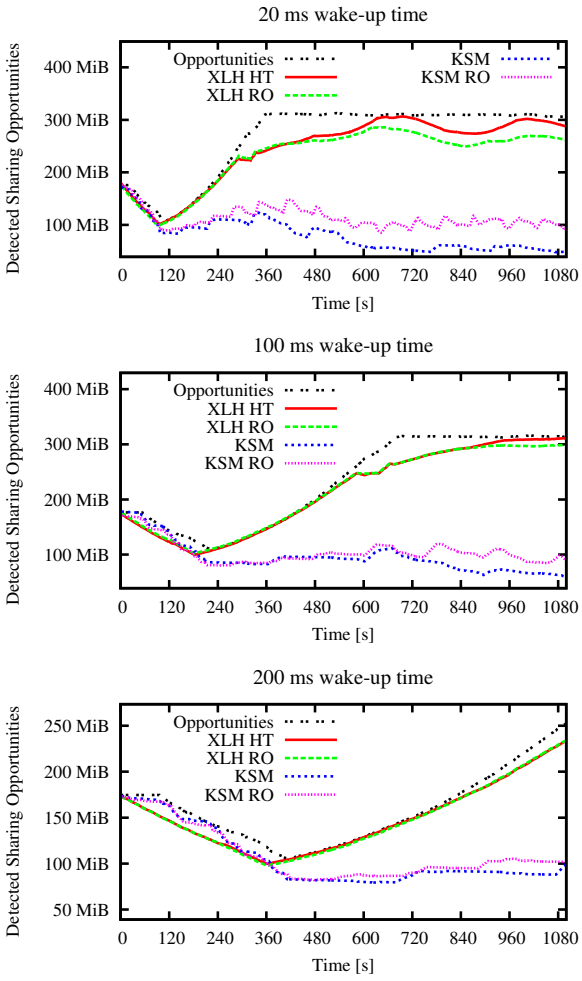


Figure 10: Apache/httperf merge performance with scaled request rates: No hints are dropped in these benchmarks.

two pages. Dropping one of the potential merge partners is enough to make the deduplication dependent on the linear scanner to find the other page in time. The effect of dropped hints can be easily seen when reducing the request rate. If XLH can process most of the hints, we get almost perfect results for all three scan rates (Figure 10). In this benchmark we have scaled the number of requests per second down to 1/4 of the original setting for 100 ms wakeup time and to 1/8 for 200 ms wakeup time.

Just like in the kernel benchmark, almost 10 times as many new sharing opportunities are detected and shared in the full-speed 20 ms benchmark (242233 vs. 22012). In this scenario, XLH detects those sharing opportunities 215 seconds earlier than KSM in the median. The histogram of the time that pages remain shared while running the benchmark is depicted in Figure 11.

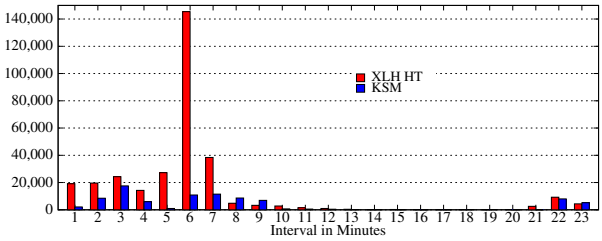


Figure 11: Histogram depicting the time, sharing opportunities stay shared throughout the apache benchmark.

Mixing Scenarios We have also run a benchmark in which both previously described benchmarks were executed at the same time. One VM was compiling the kernel while the other one was serving files with Apache.

In our benchmarks we can see a draw between vanilla KSM and XLH when mixing benchmarks (Figure 12).

That also reflects in the total number of sharing opportunities detected. XLH merges only 11% more sharing opportunities than KSM in the 20 ms benchmark (19483 vs. 17573). Theoretically however, KSM may be as much as twice as good as XLH with an interleaving ratio of 1:1 in case of completely useless hints. We have not encountered such results, though. In such a case, the user of the system may fine tune XLH through the interleaving ratio to mitigate this effect. Moreover, in cloud computing environments such as Amazon EC2, the provider may colocate VMs with similar memory footprints. Although the total number of shared pages is comparable, XLH merges pages 110 seconds earlier in the median. The histograms of XLH's and KSM's sharing time look similar in this benchmark (Figure 13).

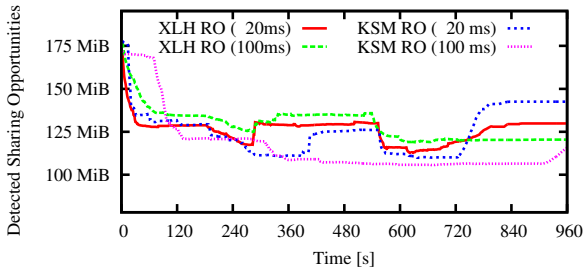


Figure 12: Merge performance with mixed workloads.

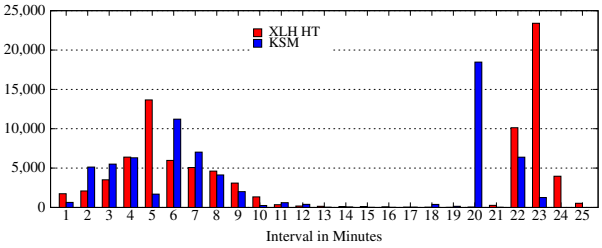


Figure 13: Histogram depicting the time, sharing opportunities stay shared throughout the mixed benchmark.

5.3 Deduplication Efficiency

XLH does not trade memory bandwidth and CPU cycles for the gained effectiveness; KSM can already do this within limits if set to scan aggressively.

Total Runtime Overhead The runtime variation between XLH and the default KSM was below 1% in our kernel build and Apache experiments. That is also true for the throughput that we have measured with httperf. To support the claim that XLH does not increase the system's load further than KSM with equal scan rates, we have measured the CPU consumption of the scanner thread when building the Linux kernel (Table 3).

Approach	20 ms	100 ms	200 ms
XLH HT	67.05%	33.61%	16.94%
XLH RO	66.19%	34.13%	16.17%
KSM	68.75%	27.47%	16.32%
KSM RO	68.92%	28.12%	17.52%
Average	67.72%	30.83%	16.74%

Table 3: CPU consumption: mean calculated from `top` measurements taken every second.

We do not raise the effectiveness by doing more work, but by making smarter choices for when and where to invest duty cycles. That can also be clearly seen when we compare the number of pages that XLH needs to check until it finds a sharing candidate. In the kernel build scenario XLH needs to visit 2-5 pages until it finds a sharing candidate while the linear scan needs to visit 18-260 pages. In the Apache scenario XLH visits between 4-8 pages to find a sharing opportunity while KSM visits 16-30 pages.

Memory-Scanning Overhead XLH needs additional memory for the hint buffer, which contains an 8-byte pointer for each slot and locks to serialize accesses. Most of XLH's work is amortized by the fact that it does it in the place of an equally costly operation of KSM. A lookup in the stable or unstable tree costs the same whether it was triggered by a hint or by a periodic scan. Additional CPU cycles are needed by our hinting mechanism for storing and retrieving hints and for marking hinted pages read-only. Storing and retrieving hints is very cheap ($O(1)$).

We have confirmed that neither the VFS-based hint trigger nor the hint buffer is a bottleneck by stress-testing this particular subsystem via the bonnie++ [9] file system benchmark. Figure 14 shows that the disk throughput of our enterprise class SSD does not vary significantly when choosing XLH over KSM.

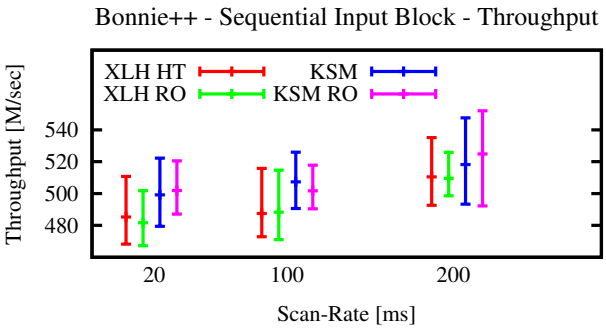


Figure 14: Disk throughput in 30 bonnie++ runs. The error-bars show the .05 and .95 percentiles.

5.4 The Unstable Tree's Stability

KSM only inserts pages into the unstable tree if their hash value has not changed since the last pass, whereas XLH inserts *hinted* pages immediately. We have examined how much this affects the stability of the unstable tree and the overall memory deduplication performance.

A good metric for the stability of the tree is the ratio between the number of nodes in the tree and the number of nodes that can be found when searching for each node in the tree. If a page cannot be found, it cannot be merged. Instead, it is inserted again—this time in another place (Figure 5). The percentage of reachable pages in the unstable tree for the kernel build benchmark after a full scan cycle is shown in Table 4.

Vanilla KSM	Hints (all RW)	Hints (hints RO)
33.6% - 53.0%	10.0%	98.9%

Table 4: Percentage of the unstable tree that is reachable at the end of a scan cycle.

We have found that pages that are part of the page cache are the ones that are most likely to change among all pages in the unstable tree. That happens when a page cache pages is written back, evicted, and replaced by another file's page in the guest.

Using I/O-based hints, pages from the page cache are inserted into the unstable tree earlier than other pages. That way, they have more time to degenerate the unstable tree, an unwelcome side effect. To mitigate this effect, we implemented two strategies:

Marking the Unstable Tree Read-Only One possible strategy to keep the unstable tree from degenerating is to mark pages that are inserted through a hint to be read-only. To show that XLH marks the "right" pages read-only and leaves the ones that do not degenerate the unstable tree read-write, we have run a benchmark where our hinting mechanism is active and *all* pages are unconditionally marked read-only when they are inserted into the unstable tree. Furthermore, we have added a modified KSM version without hinting that also marks all pages that are inserted into the unstable tree read-only. This effectively keeps the tree from degenerating altogether—all nodes are always reachable. The resulting deduplication performances are depicted in Figure 15.

In the long run, deduplication ratios depend on the quality of the unstable tree. If it degenerates, the effectiveness of KSM drops drastically. We get much higher deduplication ratios when XLH marks all items in the unstable tree read-only. However, there is not much benefit in also trapping updates of pages that were added to the unstable tree by scanning. Not marking pages read-only at all does damage the tree.

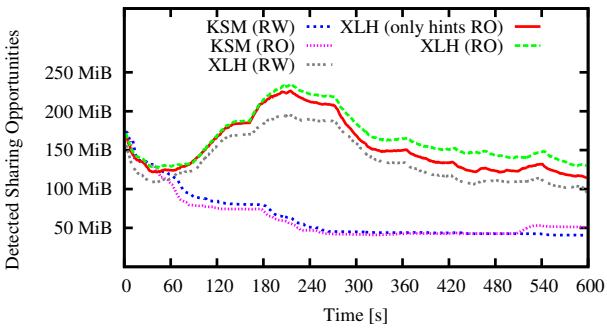


Figure 15: The merge performance depends heavily on the stability of the unstable tree and the temporal locality of unstable tree accesses.

Replacing Unstable Tree with a Hash Table Hash tables, as used in ESX [25], are a suitable choice to deal with unstable pages without the need for marking pages read-only or pruning degenerated data.

As KSM already calculates page hashes when scanning for duplicates, using a page hash as index into the hash table does not incur an extra cost. However, in contrast to the unstable tree, the hash table cannot easily be resized and thus does not scale well when the number of pages to monitor for redundant data changes frequently and to a large extent.

The performance depends highly on tuning the parameters to the workload at hand: If the hash table has many fewer entries than the number of pages in the system, then lookups become expensive due to chaining. That is also the case in workloads that generate many pages with colliding hash values. Yet, we observed in our benchmarks that the hash table approach performs as well or even better than using a read-only unstable tree when tuned to good values. We have used a hash table size of 256K entries in our various benchmarks throughout the paper.

5.5 Concluding Remarks

We have shown that XLH is able to quickly deduplicate the memory of newly booted VMs, which is especially beneficial when sandboxing short-running jobs or migrating many VMs at once. We have further demonstrated XLH's superior merging effectiveness compared to conventional linear memory scanners. XLH is capable of freeing up to 5x more memory than KSM by exploiting short-lived sharing opportunities, thereby finding 10x as many pages with equal contents. Moreover, XHL merges sharing opportunities 2-4 minutes earlier and thus leverages existing sharing potential. We have also evaluated XHL in an unfavorable scenario and found that it did not worsen the sharing performance compared to KSM. We also found XLH's influence on workload run-time and I/O throughput to be negligible.

6 Related Work

Virtual memory allows mapping different address space regions to the same region in physical memory. The underlying mapping mechanism can be used to establish communication and to allow coordination. Mapping can also reduce the memory footprint of processes and VMs by sharing memory regions of identical content. XLH is a novel approach to identify pages of same content.

6.1 Sharing of Cloned Content

In traditional systems, memory is shared between processes on two occasions: first, when the user explicitly requests shared memory through system calls and second, implicitly through copy-on-write (COW) semantics, when using process forking or memory-mapped files. In the latter case, memory pages are shared that point to the same file control block (i.e., an inode). When a file is copied, a new control block is created, which points to a copy of the same content. Due to referencing a different control block, accessing this copy via memory-mapped files will lead to redundant data in main memory even if the duplicated disk blocks are later merged via block-layer deduplication. Thus, using memory-mapped files to deduplicate memory among different VMs is not possible, as VMs generally do not share the same file system, but run from separate virtual disk image (VDI) files. Our approach, in contrast, is capable of deduplicating equal memory pages originating from different files and even different memory sources.

When a process is duplicated via forking, the parent's and child's entire address spaces are shared using COW. If either process writes to a page afterwards, the sharing of the target page is broken up. Android's Cygote uses this property to share the Dalvik VM and the core libraries among all processes [20]. This initial cloning has also been used to share whole guest operating systems [15, 24]. The COW semantic only allows sharing pages that already existed *before* a process or VM has been forked. Our approach exploits the full sharing potential because it also deduplicates equal pages that are created at run time, *after* forking.

6.2 Paravirtualization

An established approach to find duplicate main memory pages that stem from background storage is the instrumentation of guest operating systems with the goal to explicitly track changes. (Cellular) Disco's transparent page sharing uses a deduplicating COW-disk to identify file blocks that can be mapped to the same page in main memory due to equal content. It also hooks calls such as `bcopy` to keep track of shared content [6, 11].

The Xen [2] based Satori [18] seizes this suggestion and uses paravirtualized smart virtual disks to infer the sharing opportunities that stem from background storage.

Collaborative memory management (CMM) [21] uses paravirtualized Linux guests to share usage semantics of the guests' virtual memory system with the hypervisor. Its focus lies on determining the working set size of the guests, especially by telling the hypervisor which guest pages are unused and can thus be dropped. CMM was implemented for the IBM System zSeries.

XenFS [26] is a prototype for a file system that is shared between VMs and makes it possible to share caches and COW named page mappings across VMs. Two different approaches to shared page caches are Transcendent Memory [16, 17] and XHive [13]. Transcendent Memory provides a key-value store that can be used by guests to cache I/O requests in the hypervisor. XHive practically implements swapping to the hypervisor (i.e., move pages from the guest to the host). It gives pages that are used by multiple VMs a better chance to reside in memory, but outside of the VM's quota.

All techniques in this paragraph use paravirtualization techniques. They need to modify the guest to work. Our approach in turn works without such modifications and even works with non-VM processes.

6.3 Memory Scanning

The technique of periodically scanning main memory pages for equal content and then transparently merging those pages to share them in a COW manner was first introduced in VMware's ESX Server [25]. Linux also uses this technique under the name Kernel Samepage Merging (KSM) to increase the memory density of VMs [1]. ESX is dedicated to running VMs and thus may use memory scanning on all memory pages while KSM only scans pages that have been advised to be good sharing candidates through the `madvise` system call.

The KSM and ESX content-based page sharing approaches differ mainly in the way they catalog scanned pages: ESX calculates a hash value for every page when scanning and stores these values in a hint table. When a match is found in the hint table, ESX first re-calculates the hash value of the previously inserted page to check whether the content has changed since the last calculation. If not, the pages are compared bit-by-bit to rule out a hash-collision. Then, equal pages are merged, and their hash value is inserted into another table, the shared table.

KSM also calculates hash values, but only to check whether a page has changed between scan rounds. It does not use those hash values to infer equality between pages. All pages that have not changed between rounds are inserted into a tree (the full page, not the hash value); duplicates are found on insertion (see Section 3).

The general trade-off that is involved when using memory scanners is CPU utilization and memory bandwidth versus the time in which deduplication targets are identified. KSM and ESX both have a variable scan rate which is configured through setting sleep times and a number of pages that are scanned on every wake-up. Both KSM and ESX suggest scan rates that are fast enough to merge long-lived sharing opportunities with little overhead. However, the current implementations are not well suited to find short-lived sharing opportunities [7].

ESX scans pages in random order, while KSM scans linearly in rounds. Although the original ESX paper [25] states that it could be beneficial to define a heuristic for the scan order, neither KSM nor ESX propose a well suited policy to find sharing candidates more quickly. XLH is such a suggestion.

Pages with similar content can be shared to a great extend through storing compressed patches which are applied on access page faults. Such approaches like Difference Engine [12] could be combined with XLH to identify good candidates for sub-page sharing.

7 Conclusion

When it comes to consolidating many virtual machines on a single physical machine, the primary bottleneck is the main memory capacity. Previous work has shown that the memory footprint of virtual machines can be reduced significantly by merging equal pages. Identifying those pages can be achieved through scanning for equal contents in the host.

We have demonstrated that memory deduplication scanners can be improved significantly when informing the scanner of recently modified memory pages. XLH implements this idea by telling KSM about I/O operations. KSM then processes these pages preferably to deliver superior performance compared to linear scanning.

We have discussed various challenges, such as I/O bursts and degenerating data structures in KSM, and described design alternatives. Our evaluation shows that I/O-based hints can increase the effectiveness of memory scanners significantly without raising the overhead imposed by the scanner. XLH finds more sharing opportunities than KSM and detects them earlier by minutes. Thereby XLH exploits sharing opportunities within and across virtual machines that were not detectable by linear scanners before.

We believe that XLH is already beneficial for a variety of use cases as it is. Therefore, we intend to release our Linux kernel extension soon. We plan to closely analyze memory duplication properties of NUMA architectures to identify good deduplication policies for such systems in the future.

References

[1] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using KSM. In *Linux Symposium 2009*.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., ET AL. Xen and the art of virtualization. SOSP 2003.

[3] BARKER, S., ET AL. An empirical study of memory sharing in virtual machines. In *USENIX ATC 2012*.

[4] BELLARD, F. Qemu, a fast and portable dynamic translator. ATEC 2005.

[5] BERTHELS, J. Exmap memory analysis tool. `http://www.berthels.co.uk/exmap/`, 2006.

[6] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: running commodity operating systems on scalable multiprocessors. *Transactions on Computer Systems 1997*.

[7] CHANG, C.-R., ET AL. An empirical study on memory sharing of virtual machines for server consolidation. ISPA 2011.

[8] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *HotOS 2001*.

[9] COKER, R. The bonnie++ benchmark. `http://www.coker.com.au/bonnie++/`, 1999.

[10] EIDUS, I. How to use the kernel samepage merging feature, 2009. Documentation/vm/ksm.txt in Linux Kernel v3.0.

[11] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. SOSP 1999.

[12] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., ET AL. Difference engine: harnessing memory redundancy in virtual machines. *Communications of the ACM 2010 Volume 53*.

[13] KIM, H., JO, H., AND LEE, J. Xhive: Efficient cooperative caching for virtual machines. *Trans. on Computer Science 2011*.

[14] KLOSTER, J. F., KRISTENSEN, J., AND MEJLHOLM, A. Determining the use of Interdomain Shareable Pages using Kernel Introspection. Tech. rep., Aalborg University, 2007.

[15] LAGAR-CAVILLA, H. A., ET AL. Snowflock: rapid virtual machine cloning for cloud computing. EuroSys 2009.

[16] MAGENHEIMER, D., MASON, C., ET AL. Transcendent Memory and Linux. In *Linux Symposium 2009*.

[17] MAGENHEIMER, D., MASON, C., MCCRACKEN, D., AND HACKEL, K. Paravirtualized paging. WIOV 2008.

[18] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: Enlightened page sharing. In *USENIX ATC 2009*.

[19] MOSBERGER, D., AND JIN, T. httperf - a tool for measuring web server performance. *SIGMETRICS Perf. Eval. Review 1998*.

[20] PATRICK BRADY. Anatomy & Physiology of an Android. In *Google I/O Developer Conference 2008*.

[21] SCWIDEFSKY, M., ET AL. Collaborative memory management in hosted linux environments. In *Linux Symposium 2006*.

[22] TRAVISCI COMMUNITY. TravisCI: continuous integration service. `https://travis-ci.org/`, 2012.

[23] VMWARE, INC. ESX Server 3.0.1 Resource Management Guide, 2011.

[24] VRABLE, M., MA, J., CHEN, J., ET AL. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. SOSP 2005.

[25] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *SIGOPS Operating System Review 2002*.

[26] WILLIAMSON, MARK. Xen Wiki: XenFS. `http://wiki.xensource.com/xenwiki/XenFS`, 2007.

Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack*

Konstantinos Menychtas Kai Shen Michael L. Scott
Department of Computer Science, University of Rochester

Abstract

General-purpose GPUs now account for substantial computing power on many platforms, but the management of GPU resources—cycles, memory, bandwidth—is frequently hidden in black-box libraries, drivers, and devices, outside the control of mainstream OS kernels. We believe that this situation is untenable, and that vendors will eventually expose sufficient information about cross-black-box interactions to enable whole-system resource management. In the meantime, we want to enable research into what that management should look like.

We systematize, in this paper, a methodology to uncover the interactions within black-box GPU stacks. The product of this methodology is a state machine that captures interactions as transitions among semantically meaningful states. The uncovered semantics can be of significant help in understanding and tuning application performance. More importantly, they allow the OS kernel to intercept—and act upon—the initiation and completion of arbitrary GPU requests, affording it full control over scheduling and other resource management. While insufficiently robust for production use, our tools open whole new fields of exploration to researchers outside the GPU vendor labs.

1 Introduction

With hardware advances and the spread of programming systems like CUDA and OpenCL, GPUs have become a precious system resource, with a major impact on the power and performance of modern systems. In today's typical GPU architecture (Figure 1), the GPU device, driver, and user-level library are all vendor-provided black boxes. All that is open and documented is the high-level programming model, the library interface to programs, and some architectural characteristics useful for high-level programming and performance tuning.

For the sake of minimal overhead on very low latency GPU requests, the user-level library frequently communicates directly with the device (in both directions) through memory-mapped buffers and registers, bypassing the OS kernel entirely. A buggy or malicious application can easily obtain an unfair share of GPU resources (cycles, memory, and bandwidth). With no control over such basic functions as GPU scheduling, the kernel has no way to coordinate GPU activity with other aspects of resource management in pursuit of system-wide objectives. Application programmers, likewise, are seldom able to reason about—much less correct—performance anomalies due to contention or other interactions among the GPU requests of concurrently running applications.[1]

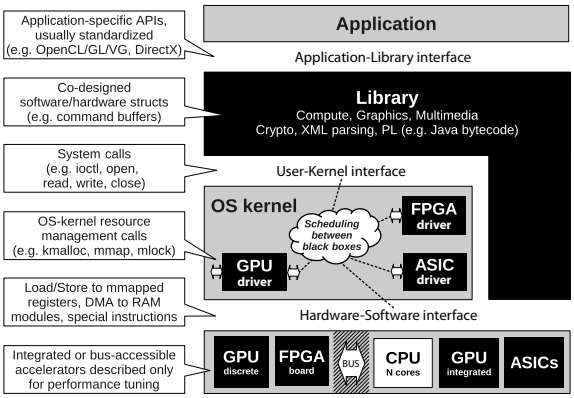


Figure 1: The GPU software / hardware architecture, with notes on interfaces and components. Gray areas indicate open system / application components while black areas indicate black-box components without published specifications or behaviors.

When GPUs were of limited programmability, and every request completed in a small, bounded amount of time, kernel control and performance transparency were much less important. As GPUs and other accelerators become more and more central to general-purpose computing, affecting thus whole-system resource management objectives, protected OS-level resource management will inevitably become a pressing need. To satisfy this need, the kernel must be able to identify and delay GPU request submissions, and tell when requests complete. A clean interface to expose this information need not compromise either proprietary technology or backward compatibility, and will hopefully be provided by vendors in the near future.

*This work was supported in part by the National Science Foundation under grants CCF-0937571, CCR-0963759, CCF-1116055, CNS-1116109, CNS-1217372, and CNS-1239423, as well as a Google Research Award.

[1] We use the term *request* to refer to a set of operations that run without interruption on the GPU — typically a GPU-accelerated compute or shader function, or a DMA request.

In anticipation of a future in which production systems manage GPUs and other accelerators as first-class computational resources, we wish to enable research with existing black-box stacks. Toward that end, we present a systematic methodology to uncover the same (limited) information about black-box interactions that vendors will eventually need to reveal. This includes the `ioctl` and `mmap` calls used to allocate and map buffers, the layout of command queues in memory shared between the library and the device, and, within that memory, the locations of device registers used to trigger GPU activity and of flags that announce request completion. While our inference is unlikely to be robust enough for production use, it provides a powerful tool for OS researchers exploring the field of GPU management.

Our work has clear parallels in white and gray-box projects, where vendors have supported certain free and open-source software (FOSS) (e.g., Intel contributes to the FOSS graphics stack used in GNU/Linux), or the FOSS community has exhaustively uncovered and rebuilt the entire stack. Projects such as Nouveau for Nvidia devices have proven successful and stable enough to become part of the mainline Linux kernel, and their developed expertise has proven invaluable to our initial efforts at reverse engineering. Our goal, however, is different: rather than develop a completely open stack for production use—which could require running a generation or two behind—we aim to model the black-box stack as a state machine that captures only as much as we need to know to manage interactions with the rest of the system. This machine can then be used with either FOSS or proprietary libraries and drivers. Compatibility with the latter allows us, in a research setting, to track the cutting edge of advancing technologies.

Previous work on GPU scheduling, including GERM [5], TimeGraph [11], and Gdev [12], has worked primarily with FOSS libraries and drivers. Our goal is to enable comparable OS-kernel management of black-box GPU stacks. PTask [13] proposes a dataflow programming model that re-envisions the entire stack, eliminating direct communication between the library and device, and building new functionality above the binary driver. Pegasus [9] and Elliott and Anderson [6] introduce new (non-standard) system interfaces that depend on application adoption. By contrast, we keep the standard system architecture, but uncover the information necessary for OS control.

2 Learning Black-Box Interactions

Our inference has three steps: (a) collect detailed traces of events as they occur across all interfaces; (b) automatically infer a state machine that describes these traces, and that focuses our attention on key structure (loops in particular); (c) manually post-process the machine to merge states where appropriate, and to identify transitions of semantic importance. We describe the first two steps below; the third is discussed in Section 3.

Trace Collection We collect traces at all relevant black-box interfaces (Figure 1). The application / library interface is defined by library calls with standardized APIs (e.g., OpenCL). The library / driver interface comprises a set of system calls, including `open`, `read`, `write`, `ioctl`, and `mmap`. The driver / kernel interface is also visible from the hosting operating system, with calls to allocate, track, release, memory-map and lock in kernel (pin) virtual memory areas. For the driver / hardware interface, we must intercept reads and writes of memory-mapped bus addresses, as well as GPU-raised interrupts.

We collect user- and kernel-level events, together with their arguments, and merge them, synchronously and in (near) order, into a unified kernel trace. Using DLL redirection, we insert a system call to enable kernel logging of each library API call. To capture memory accesses, we invalidate the pages of all virtual memory areas mapped during a tracked application's lifetime so that any access to an address in their range will trigger a page fault. Custom page fault handlers then log and reissue accesses. We employ the Linux Trace Toolkit (LTTng) [4] to record, buffer, and output the collected event traces.

For the purposes of OS-level resource management, it is sufficient to capture black-box interactions that stem from a GPU library call. Events inside the black boxes (e.g., loads and stores of GPU memory by GPU cores, or driver-initiated DMA during device initialization) can safely be ignored.

Automatic State Machine Inference If we consider the events that constitute each trace as letters from a fixed vocabulary, then each trace can be considered as a word of an unknown language. Thus, uncovering the GPU state machine is equivalent to inferring the language that produced these words—samples of event sequences from realistic GPU executions. We assume that the language is regular, and that the desired machine is a finite automaton (DFA). This assumption works well in our context for at least three reasons:

1. Automatic DFA inference requires no prior knowledge of the semantics of the traced events. For instance, it distinguishes `ioctl` calls with different identifiers as unique symbols but it does not need to know the semantic meaning of each identifier.

2. The GPU black-box interaction patterns that we are trying to uncover are part of an artificial "machine," created by GPU vendors using standard programming tools. We expect the emerging interactions to be well described by a finite-state machine.

3. The state machine provides a precise, succinct abstraction of the black-box interactions that can be used to

Event type	Meaning
`ioctl:0x??`	ioctl request : unique hex id
`map:[pin|reg|fb|sys]`	mmap : address space
`R:[pin|reg|fb|sys]`	read : address space
`W:[pin|reg|fb|sys]`	write : address space
`pin`	locked (pinned) pages
`reg`	GPU register area
`fb`	GPU frame buffer
`sys`	kernel (system) pages

Table 1: Event types and (for `map`, `R`, and `W`) associated address spaces constitute the alphabet of the regular language / GPU state machine we are trying to infer.

drive OS-level resource management. It provides a natural framework in which vendors might describe inter-black-box interactions without necessarily disclosing internal black-box semantics.

In the GPU state machine we aim to uncover, each transition (edge between two adjacent states) is labeled with a single event (or event type) drawn from an event set (or alphabet of the corresponding language). In practice, we have discovered that bigger alphabets for state transition events lead to larger and harder to comprehend state machines. We therefore pre-filter our traces to remove any detail that we expect, a priori, is unlikely to be needed. Specifically, we (a) elide the user-level API calls, many of which are handled entirely within the library; (b) replace memory addresses with the areas to which they are known to belong (e.g., registers, GPU frame buffer, system memory); and (c) elide `ioctl` parameters other than a unique hex id. This leaves us with the four basic event types shown in Table 1.

Given a set of pre-filtered traces, each of which represents the execution of the target GPU system on a given program and input, a trivial ("canonical") machine would have a single start state and a separate long chain of states for each trace, with a transition for every event. This machine, of course, is no easier to understand than the traces themselves. We wish to merge semantically equivalent states so that the resulting machine is amenable to human understanding and abstraction of interaction patterns. Note that our goal is not to identify the smallest machine that accepts the input event samples—the single-state machine that accepts everything fits this goal but it does not illustrate anything useful. So we must also be careful to avoid over-merging.

State machine reduction is a classic problem that dates from the 1950s [8], and has been studied in various forms over the years [2, 3, 7]. The problem is also related to state reduction for Hidden Markov Models [14]. Several past approaches provide heuristics that can be used to merge the states of the canonical machine. State merging in Hidden Markov Models, however, is more applicable to the modeling of (imprecise, probabilistic) natural phenomena than to capturing the DFA of a (precise) artificial system. Some reduction techniques target restricted domains (like the reversible languages of Angluin [2]); the applicability of these is hard to assess.

After reviewing the alternatives, we found Biermann and Feldman's k-tail state merging method [3] to be intuitive and easy to realize. Under this method, we merge states from which the sets of acceptable length-k-or-shorter event strings are the same. That is, when looking "downstream" from two particular states, if the sets of possible event strings (up to length k) in their upcoming transitions are exactly the same, then these two states are considered semantically equivalent and are merged. We make the following adaptations in our work:

• In theory, Biermann and Feldman's original approach can capture the precise machine if one exists. However, this guarantee [3, Theorem 5] requires an infeasible amount of trace data—exponential in the minimum string length that can distinguish two arbitrary states. We apply k-tail state merging on the canonical machine produced from a limited number of event samples, which can be easily collected. Our goal is not to fully automate precise inference, but rather to guide human understanding—to identify the transitions of importance for scheduling and other resource management.

• Our limited-sample k-tail state merging typically leaves us with a non-deterministic machine (multiple transitions on a given event from a given state). We perform additional state merging to produce a deterministic machine. Specifically, we repeatedly merge states that are reached from a common preceding state with the same transition event.

• As shown in Figure 2, larger ks yield larger machines. Intuitively, this happens because of more cautious state merging: the farther we look ahead, the harder it is to find states with exactly the same sets of upcoming event strings. If k is too large, the resulting machine will have too many states to guide human discovery of interaction patterns. If k is too small, we will merge too many states, leading to a loss of useful semantic information. To balance these factors, we choose a k that is in both a size plateau (the number of nodes in the graph does not change with small changes in k) and a node-complexity plateau (the number of nodes with in-degree or out-degree larger than 1 does not change). We avoid extreme values of k, favoring machines that are neither trivial nor unnecessarily complex. We also exploit the assumption that repetitive API-level commands should manifest as small cycles; we pick the smallest nontrivial machine in which such cycles appear.

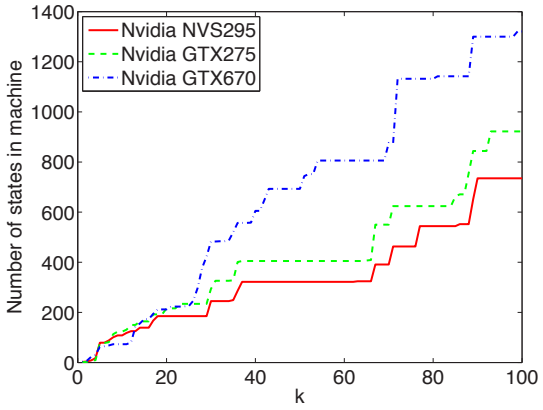


Figure 2: Sizes of state machines inferred by k-tail state merging for different values of k on three GPU devices.

3 State Machine Case Study

We have applied the techniques described in Section 2 on a system with an Intel Xeon E5520 "Nehalem" CPU and three Nvidia GPUs: (a) the NVS295, with a G98 chip core (CUDA 1.1 capability), (b) the GTX275, with a G200b chip core (CUDA 1.3 capability, "Telsa" micro-architecture), and (c) the GTX670, with a GK104 chip core (CUDA 3.0 capability, "Kepler" micro-architecture). We used a stock 3.4.7 Linux kernel and the Nvidia 310.14 driver and related libraries (CUDA 5.0, OpenCL 1.1 and OpenGL 4.2).

As input to our k-tail state merging, we used a collection of 5 traces of 4 to 9 thousand events, captured from running parameterized microbenchmarks. The microbenchmarks allow us to vary the numbers of contexts (GPU-accessible address spaces, analogous to CPU processes) and command queues (in-memory buffers through which requests are passed to the GPU); the numbers and types of requests; and the complexity of those requests (i.e., how long they occupy the GPU). The inferred state machines varied not only with k (Figure 2), but also with the model of GPU, even with the same software stack. Guided by the process described in Section 2, we ended up choosing k=35 for the NVS295, k=37 for the GTX275 and k=20 for the GTX670. As the earliest technology, the NVS295 is easiest to describe; we present it below.

Understanding Inferred DFAs Figure 3 presents the automatically inferred DFA for the NVS295 GPU for k=35; for k=34 short loops (e.g. nodes 159, 177, 178) disappear in favor of longer straight paths, while for smaller/larger ks, the graph becomes either too trivial or too complex to carry useful information. The semantics of each state are not yet clear; standard reverse-engineering techniques can help us attach meaning to both the states and the transitions by utilizing previously elided trace details. By guiding our attention to a hand-

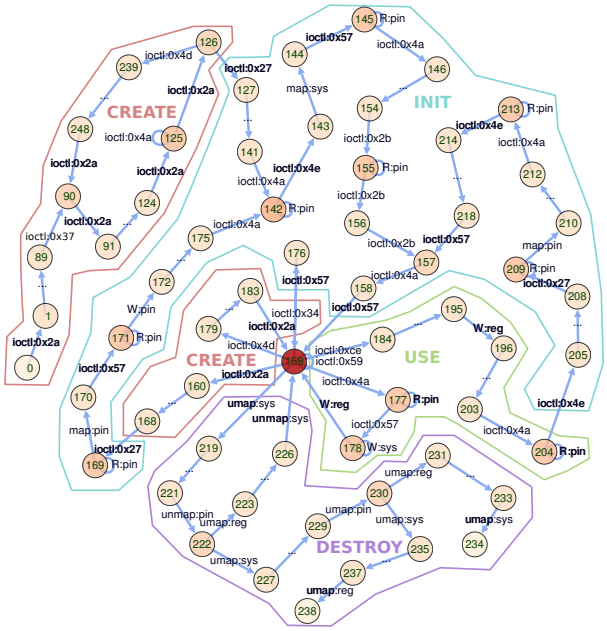


Figure 3: Inferred state machine of Nvidia NVS295 for k=35. Node "temperature" indicates frequency of occurrence. For clarity, some straight-line paths have been collapsed to a single edge (event id "...").

ful of states, and thus events in the trace, the inferred state-machine simplifies significantly the manual effort involved in this step. For example, we discovered that interrupts appear, in the general case, to occur at unpredictable times with respect to other events, and can be discarded.[2]

Identifying the start (0) and end nodes (234, 238) of the DFA in Figure 3 is trivial. The first `ioctl` event after node 0, with id `0x2a`, should be associated with the context setup process (CREATE), but is not unique to this phase. We employ previously suppressed `ioctl` arguments to uncover unique identifiers that act as invariants appearing only at or after setup. Unmapping events appearing in the graph's exit path (DESTROY) can be associated with previously established mapping calls through their arguments (`map(0xDEAD)` $\neq$ `unmap(0xBEEF)`). Realizing that important mappings have been setup or destroyed, allows us to expect (or stop waiting for) new GPU access requests.

The "epicenter" of the graph of Figure 3 is clearly node 159, so we focus on cycles around it. Since our traces can create and setup multiple contexts, command queues, etc, `ioctl` events with id `0x2a` and un-

[2]Options such as the `cudaDeviceBlockingSync` flag of the CUDA API can direct the GPU to raise an interrupt at request completion. This option may allow completion to be detected slightly earlier than it is with polling, but with either mechanism the state machine realizes completion as a (post-polling or post-interrupt) read, by the user library, of a reference counter updated by the GPU.

map events appear naturally around node 159 in cycles. Longer paths, composed primarily of `ioctl` and mapping calls (e.g., including nodes 158, 169 and 176) initialize buffers, establishing appropriate memory mappings for structures like the command queue. We identify and understand their form by storing and tracking carefully encoded bit patterns as they appear in buffers trough the trace. We also compare elided `ioctl` arguments with addresses appearing as map / unmap arguments so as to correlate events; we can thus know and expect particular buffers to be mapped and ready for use (INIT). For example, we understand that `ioctl` id `0x57` seems to associate the bus address and the system address of a memory-mapped area, which is a necessary command buffer initialization step.

Next, we focus on read / write patterns in the graph (e.g. cycle including nodes 177, 178). DMA and compute requests have a clear cause-effect relationship with the `W:reg` event on edge 178→159: a write to a mapped register must initiate a GPU request (USE). Similarly, the spin-like `R:pin` loop (e.g., at node 177) follows many GPU requests, and its frequency appears affected by the complexity of the requests; spinning on some pinned memory address must be a mechanism to check for completion of previously submitted requests. Last, we observe repetitive `W:sys` events (node 178) on the (request) path to `W:reg`, implying a causal relationship between the two. By manually observing the patterns exhibited by regularly changing values, we discover that GPU commands are saved in system memory buffers and indexed via `W:reg`.

The DFA inferred for the GTX275 (Tesla) GPU exhibits patterns very similar to the NVS295; the previous description applies almost verbatim. However, the newer GTX670 (Kepler) has brought changes to the software / hardware interface: it is `W:fb` events that capture the making of new DMA or compute / rendering requests. This means that the memory areas that seem to be causally related to GPU request submission are now accessible through the same region as the the frame buffer. Subtle differences in the use of `W:reg` can be noticed in the indexing pattern demonstrated by the `W:fb` arguments. In all other aspects, the Kepler GPU state machine remains the same as in previous generations, at least at the level of observable cross-black-box interactions.

The GPU Driver State Machine Figure 4 presents a distilled and simplified state machine (left) that captures the behavior common to our three example GPUs, and the OpenCL API calls that push the DFA into various states (right). For clarity of presentation, we have omitted certain global state and transition identifiers. Correspondences for other libraries (OpenGL, CUDA) are

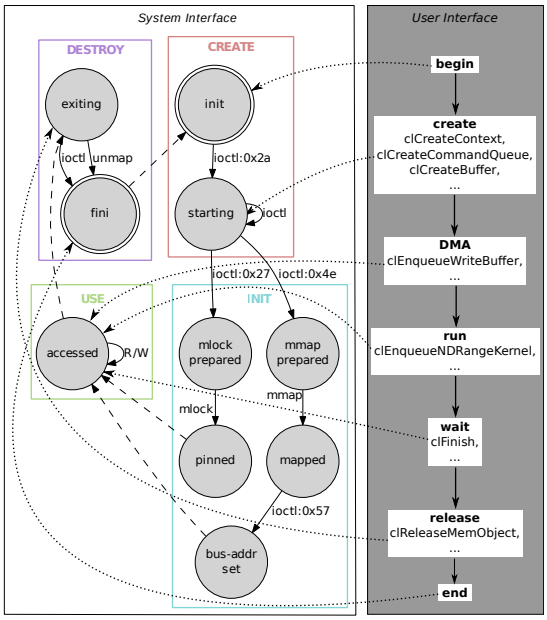


Figure 4: Semantically rich user-level API events can be mapped to state transitions at the system level.

similar. We have confirmed the validity of this state machine using realistic OpenCL / GL and CUDA applications. There exist only quantitative differences in the traces collected from 3D and compute libraries, such as the number and size of buffers mapped or variability in some elided `ioctl` parameters. Such differences do not alter the higher-level GPU driver model produced.

A GPU accelerated application typically begins with a sequence of `ioctl`, memory mapping and locking calls to build a new context (CREATE) and associate with it a set of memory areas (INIT). A small set of those, typically a command buffer and a circular queue of pointers to commands (the ring buffer), comprise the GPU command queue—a standard producer consumer communication mechanism [11]. Once initialization is complete, stores to memory-mapped GPU registers (USE) are used to point the GPU to new DMA and compute / rendering requests. Spins on associated system addresses (USE) are used to notice request completion. Cross references among these areas and elided tracing information make it possible to identify them uniquely. Unmapping operations (DESTROY) mark the ends of lifetimes of the command queue's buffers, and eventually context.

Given an abstract GPU state machine, one can build kernel-level mechanisms to intercept and intercede on edges / events (e.g. as appearing in Figure 4) that indicate preparation and utilization of the GPU ring buffer. Intercession in turn enables the construction of GPU resource managers that control the software / hardware interface, independently of the driver, yet in the protected setting of the OS kernel.

4 Conclusions and Future Work

We have outlined, in this paper, a systematic methodology to generate, analyze, and understand traces of cross-black-box interactions relevant to resource management for systems such as those of the GPU software / hardware stack. We used classic state machine inferencing to distill the numerous interactions to just a handful, and with the help of common reverse engineering techniques, revealed and assigned semantics to events and states that characterize an abstract black-box GPU state machine. In the process, we uncovered details about how OpenCL API requests (e.g., compute kernels) are transformed into commands executed on the GPU.

The suggested methodology is not fully automated, but significantly simplifies the clean-room reverse engineering task by focusing attention on important events. While we do not claim completeness in the inferred state machine description, we have defined and tested almost all combinations of run-time–affecting parameters that the library APIs allow to be set (e.g., multiple contexts, command queues, etc), and we have used a variety of graphics and compute applications as input to the inference process. No qualitative differences arise among different APIs: the inferred results remain the same. These experiments give us substantial confidence that the abstract, distilled machine captures all aspects of black-box interaction needed to drive research in OS-level GPU resource management. Validation through comparison to FOSS stacks is among our future research plans.

While our case study considered GPUs from only a single vendor (Nvidia), our methodology should apply equally well to discrete GPUs from other vendors (e.g., AMD and Intel) and to chip architectures with integrated CPU and GPU. As long as the GPU remains a co-processor, fenced behind a driver, library, and run-time stack, we expect that the command producer / consumer model of CPU / GPU interactions will require a similar, high-performance, memory-mapped ring-buffer mechanism. Available information in the form of previously released developer manuals from vendors like AMD [1] and Intel [10] supports this expectation.

The developed state machine provides application programmers with insight into how their requests are handled by the underlying system, giving hints about possible performance anomalies (e.g., request interleaving) that were previously hard to detect. More important, the machine identifies and defines an interface that can allow more direct involvement of the operating system in the management of GPUs. To effect this involvement, one would have to intercept and intercede on request-making and request-completion events, allowing an appropriate kernel module to make a scheduling decision that reflects its own priorities and policy. We consider the opportunity to build such OS-kernel level schedulers today, for cutting edge GPU software / hardware stacks, to be an exciting opportunity for the research community.

Acknowledgment

We are grateful to Daniel Gildea for helpful conversations on language inference. We also thank the anonymous reviewers and our shepherd Rama Ramasubramanian for comments that helped improve this paper.

References

[1] AMD. Radeon R5xx Acceleration: version 1.2, 2008.

[2] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, July 1982.

[3] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, 21(6):592–597, June 1972.

[4] M. Desnoyers and M. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Ottawa Linux Symposium*, pages 209–224, Ottawa, Canada, July 2006.

[5] A. Dwarakinath. A fair-share scheduler for the graphics processing unit. Master's thesis, Stony Brook University, Aug. 2008.

[6] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, Jan. 2012.

[7] K.-S. Fu and T. L. Booth. Grammatical inference: Introduction and survey. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(3):343–375, May 1986.

[8] S. Ginsburg. A technique for the reduction of a given machine to a minimal-state machine. *IRE Trans. on Electronic Computers*, EC-8(3):346–355, Sept. 1959.

[9] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[10] Intel. OpenSource HD Graphics Programmers Reference Manual: volume 1, part 2, 2012.

[11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[12] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[13] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *23th ACM Symp. on Operating Systems Principles*, pages 233–248, Cascais, Portugal, Oct. 2011.

[14] A. Stolcke and S. M. Omohundro. Hidden Markov model induction by Bayesian model merging. In *Advances in Neural Information Processing Systems 5*, pages 11–18, San Mateo, CA, 1993.

Mantis: Automatic Performance Prediction for Smartphone Applications

Yongin Kwon[1], Sangmin Lee[2], Hayoon Yi[1], Donghyun Kwon[1], Seungjun Yang[1], Byung-Gon Chun[3], Ling Huang[4], Petros Maniatis[4], Mayur Naik[5], Yunheung Paek[1]

[1]*Seoul National University,* [2]*University of Texas at Austin,* [3]*Microsoft,* [4]*Intel,* [5]*Georgia Tech*

Abstract

We present Mantis, a framework for predicting the performance of Android applications on given inputs automatically, accurately, and efficiently. A key insight underlying Mantis is that program execution runs often contain features that correlate with performance and are automatically computable efficiently. Mantis synergistically combines techniques from program analysis and machine learning. It constructs concise performance models by choosing from many program execution features only a handful that are most correlated with the program's execution time yet can be evaluated efficiently from the program's input. We apply program slicing to accurately estimate the evaluation cost of a feature and automatically generate executable code snippets for efficiently evaluating features. Our evaluation shows that Mantis predicts the execution time of six Android apps with estimation error in the range of 2.2-11.9% by executing predictor code costing at most 1.3% of their execution time on Galaxy Nexus.

1 Introduction

Predicting the performance of programs on smartphones has many applications ranging from notifying estimated completion time to users, to better scheduling and resource management, to computation offloading [13, 14, 18]. The importance of these applications—and of program performance prediction—will only grow as smartphone systems become increasingly complex and flexible.

Many techniques have been proposed for predicting program performance. A key aspect of such techniques is what *features*, which characterize the program's input and environment, are used to model the program's performance. Most existing performance prediction techniques can be classified into two broad categories with regard to this aspect: automatic but domain-specific [7, 16, 21] or general-purpose but requiring user guidance [10, 17].

For techniques in the first category, features are chosen once and for all by experts, limiting the applicability of these techniques to programs in a specific domain. For example, to predict the performance of SQL query plans, a feature chosen once and for all could be the count of database operators occurring in the plan [16]. Techniques in the second category are general-purpose but require users to specify what program-specific features to use for each given program in order to predict its performance on different inputs. For instance, to predict the performance of a sorting program, such a technique may require users to specify the feature that denotes the number of input elements to be sorted. For techniques in either category, it is not sufficient merely to specify the relevant features: one must also manually provide a way to compute the value of each such feature from a given input and environment, e.g., by parsing an input file to sort and counting the number of items therein.

In this paper, we present Mantis, a new framework to predict online the performance of general-purpose bytecode programs on given inputs automatically, accurately, and efficiently. By being simultaneously general-purpose and automatic, our framework gains the benefits of both categories of existing performance prediction techniques without suffering the drawbacks of either. Since it uses neither domain nor expert knowledge to obtain relevant features, our framework casts a wide net and extracts a broad set of features from the given program itself to select relevant features using machine learning as done in our prior work [25]. During an offline stage, we execute an instrumented version of the program on a set of training inputs to compute values for those features; we use the training data set to construct a prediction model for online evaluation as new inputs arrive. The instrumented program tracks various features including the decisions made by each conditional in the program (*branch counts*), the number of times each loop in the program iterates (*loop counts*), the number of times each method is called (*method call counts*), and the values that are assumed by each program variable (*variable values*).

It is tempting to exploit features that are evaluated at late stages of program execution as such features may be strongly correlated with execution time. A drawback of naïvely using such features for predicting program performance, however, is that it takes as long to evaluate them as to execute almost the entire program. Our efficiency goal requires our framework to not only find features that are strongly correlated with execution time, but to also evalu-

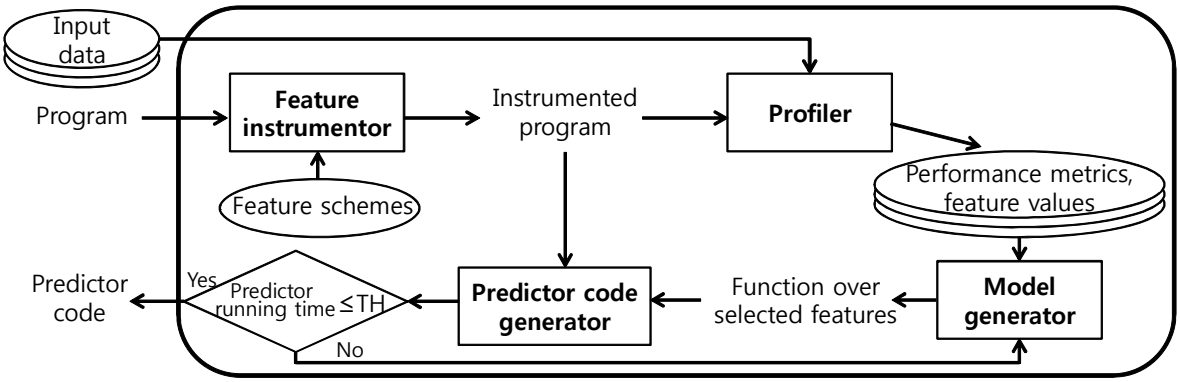


Figure 1: The Mantis offline stage.

ate those features significantly faster than running the program to completion.

To exploit such late-evaluated features, we use a program analysis technique called *program slicing* [44, 46]. Given a feature, slicing computes the set of all statements in the program that may affect the value of the feature. Precise slicing could prune large portions of the program that are irrelevant to the evaluation of features. Our slices are stand-alone executable programs; thus, executing them on program inputs provides both the evaluation cost and the value of the corresponding feature. Our application of slicing is novel; in the past, slicing has primarily been applied to program debugging and understanding.

We have implemented Mantis for Android applications and applied it to six CPU-intensive applications (Encryptor, Path Routing, Spam Filter, Chess Engine, Ringtone Maker, and Face Detection) on three smartphone hardware platforms (Galaxy Nexus, Galaxy S2, and Galaxy S3). We demonstrate experimentally that, with Galaxy Nexus, Mantis can predict the execution time of these programs with estimation error in the range of 2.2-11.9%, by executing slices that cost at most 1.3% of the total execution time of these programs. The results for Galaxy S2 and Galaxy S3 are similar. We also show that the predictors are accurate thanks to Android's scheduling policy even when the ambient CPU load on the smartphones increases.

We summarize the key contributions of our work:

- We propose a novel framework that automatically generates performance predictors using program-execution features with program slicing and machine learning.
- We have implemented our framework for Android-smartphone applications and show empirically that it can predict the execution time of various applications accurately and efficiently.

The rest of the paper is organized as follows. We present the architecture of our framework in Section 2. Sections 3 and 4 describe our feature instrumentation and performance-model generation, respectively. Section 5 describes predictor code generation using program slicing. In Section 6 we present our system implementation and evaluation results. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Architecture

In Mantis, we take a new white-box approach to automatically generate system performance predictors. Unlike traditional approaches, we extract information from the execution of the program, which is likely to contain key features for performance prediction. This approach poses the following two key challenges:

- What are good program features for performance prediction? Among many features, which ones are relevant to performance metrics? How do we model performance with relevant features?
- How do we compute features cheaply? How do we automatically generate code to compute feature values for prediction?

Mantis addresses the above challenges by synergistically combining techniques from program analysis and machine learning.

Mantis has an offline stage and an online stage. The offline stage, depicted in Figure 1, consists of four components: a feature instrumentor, a profiler, a performance-model generator, and a predictor code generator.

The feature instrumentor (Section 3), takes as input the program whose performance is to be predicted, and a set of *feature instrumentation schemes*. A scheme specifies a broad class of program features that are potentially correlated with the program's execution time. Examples of schemes include a feature for counting the number of times each conditional in the program evaluates to true, a

feature for the average of all values taken by each integer-typed variable in the program, etc. The feature instrumentor instruments the program to collect the values of features ($f_1, ..., f_M$) as per the schemes.

Next, the profiler takes the instrumented program and a set of user-supplied program inputs ($I_1, ..., I_N$). It runs the instrumented program on each of these inputs and produces, for each input I_i, a vector of feature values ($v_{i1}, ..., v_{iM}$). It also runs the original program on the given inputs and measures the performance metric (e.g., execution time (t_i)) of the program on that input.

The performance-model generator (Section 4) performs sparse nonlinear regression on the feature values and execution times obtained by the profiler, and produces a function (λ) that approximates the program's execution time using a subset of features ($f_{i1}, ..., f_{iK}$). In practice, only a tiny fraction of all M available features is chosen ($K \ll M$) since most features exhibit little variability on different program inputs, are not correlated or only weakly correlated with execution time, or are equivalent in value to the chosen features and therefore redundant.

As a final step, the predictor code generator (Section 5) produces for each of the chosen features a code snippet from the instrumented program. Since our requirement is to efficiently predict the program's execution time on given inputs, we need a way to efficiently evaluate each of the chosen features ($f_{i1}, ..., f_{iK}$) from program inputs.

We apply program slicing to extract a small code snippet that computes the value of each chosen feature. A precise slicer would prune large portions of the original program that are irrelevant to evaluating a given feature and thereby provide an efficient way to evaluate the feature. In practice, however, our framework must be able to tolerate imprecision. Besides, independent of the slicer's precision, certain features will be inherently expensive to evaluate: e.g., features whose value is computed upon program termination, rather than derived from the program's input. We define a feature as *expensive to evaluate* if the execution time of its slice exceeds a threshold (TH) expressed as a fraction of program execution time. If any of the chosen features ($f_{i1}, ..., f_{iK}$) is expensive, then via the *feedback loop* in Figure 1 (at the bottom), our framework re-runs the model generator, this time without providing it with the rejected features. The process is repeated until the model generator produces a set of features, all of which are deemed inexpensive by the slicer. In summary, the output of the offline stage of our framework is a predictor, which consists of a function (λ) over the final chosen features that approximates the program's execution time, along with a feature evaluator for the chosen features.

The online stage is straightforward: it takes a program input from which the program's performance must be predicted and runs the predictor module, which executes the feature evaluator on that input to compute feature values, and uses those values to compute λ as the estimated execution time of the program on that input.

3 Feature Instrumentation

We now present details on the four instrumentation schemes we consider: *branch counts*, *loop counts*, *method-call counts*, and *variable values*. Our overall framework, however, generalizes to all schemes that can be implemented by the insertion of simple tracking-statements into binaries or source.

Branch Counts: This scheme generates, for each conditional occurring in the program, two features: one counting the number of times the branch evaluates to true in an execution, and the other counting the number of times it evaluates to false. Consider the following simple example:

```
if (b == true) {
  /* heavy computation */ }
```

The execution time of this example would be strongly correlated with each of the two features generated by this scheme for condition (`b == true`). In this case, the two features are mutually-redundant and our performance-model generator could use either feature for the same cost. But the following example illustrates the need for having both features:

```
for (int i = 0; i < n; i++) {
  if (a[i] == 2) {
    /* light computation */ }
  else {
    /* heavy computation */ }
}
```

Picking the wrong branch of a conditional to count could result in a weakly correlated feature, penalizing prediction accuracy. The false-branch count is highly correlated with execution time, but the true-branch count is not.

Loop Counts: This scheme generates, for each loop occurring in the program, a feature counting the number of times it iterates in an execution. Clearly, each such feature is potentially correlated with execution time.

Method Call Counts: This scheme generates a feature counting the number of calls to each procedure. In case of recursive calls of methods, this feature is likely to correlate with execution time.

Variable Values: This scheme generates, for each statement that writes to a variable of primitive type in the program, two features tracking the sum and average of all values written to the variable in an execution. One can also instrument versions of variable values in program execution to capture which variables are static and what value changes each variable has. However, this creates too many feature values and we resort to the simpler scheme.

We instrument variable values for a few reasons. First, often the variable values obtained from input parameters

and configurations are changing infrequently, and these values tend to affect program execution by changing control flow. Second, since we cannot instrument all functions (e.g., system call handlers), the values of parameters to such functions may be correlated with their execution-time contribution; in a sense, variable values enable us to perform black-box prediction for the components of a program's execution trace that we cannot analyze. The following example illustrates this case:

```
int time = setfromargs(args);
Thread.sleep(time);
```

Similarly, variable value features can be equivalent to other types of features but significantly cheaper to compute. For example, consider the following Java program snippet:

```
void main(String[] args) {
  int n = Integer.parseInt(args[0]);
  for (int i = 0; i < n; i++) {...}
}
```

This program's execution time depends on the number of times the loop iterates, but the value of `n` can be used to estimate that number without executing the loop in the feature evaluator.

Other Features: We also considered the first k values (versions) of each variable. Our intuition is that often the variable values obtained from input parameters and configurations are changing infrequently, and these values tend to affect program execution by changing control flow. We rejected this feature since the sum and average metric captures infrequently-changing variable values well, and tracking k versions incurs higher instrumentation overheads. There might be other features that are helpful to prediction; exploring such features is future work.

4 Performance Modeling

Our feature instrumentation schemes generate a large number of features (albeit linear in the size of the program for the schemes we consider). Most of these features, however, are not expected to be useful for the performance prediction. In practice we expect a small number of these features to suffice in explaining the program's execution time well, and thereby seek a compact performance model, that is, a function of (nonlinear combinations of) just a few features that accurately approximates execution time. Unfortunately, we do not know a priori this handful of features and their nonlinear combinations that predict execution time well.

For a given program, our feature instrumentation profiler outputs a data set with N samples as tuples of $\{t_i, \mathbf{v}_i\}_{i=1}^{N}$, where $t_i \in \mathbb{R}$ denotes the i^{th} observation of execution time, and $\mathbf{v}_i$ denotes the i^{th} observation of the vector of M features.

Least square regression is widely used for finding the best-fitting $\lambda(\mathbf{v}, \beta)$ to a given set of responses t_i by minimizing the sum of the squares of the residuals [23]. However, least square regression tends to overfit the data and create complex models with poor interpretability. This does not serve our purpose since we have a lot of features but desire only a small subset of them to contribute to the model.

Another challenge we faced was that linear regression with feature selection would not capture all interesting behaviors by practical programs. Many such programs have non-linear, e.g., polynomial, logarithmic, or polylogarithmic complexity. So we were interested in non-linear models, which can be inefficient for the large number of features we had to contend with.

Regression with best subset selection finds for each $K \in \{1, 2, \ldots, M\}$ the subset of size K that gives the smallest Residual Sum of Squares (RSS). However, it is a discrete optimization problem and is known to be NP-hard [23]. In recent years a number of approximate algorithms have been proposed as efficient alternatives for simultaneous feature selection and model fitting. Widely used among them are LASSO (Least Absolute Shrinkage and Selection Operator) [43] and FoBa [48], an adaptive forward-backward greedy algorithm. The former, LASSO, is based on model regularization, penalizing low-selectivity, high-complexity models. It is a convex optimization problem, so efficiently solvable [15, 27]. The latter, FoBa, is an iterative greedy pursuit algorithm: during each iteration, only a small number of features are actually involved in model fitting, adding or removing the chosen features at each iteration to reduce the RSS. As shown FoBa has nice theoretical properties and efficient inference algorithms [48].

For our system, we chose the SPORE-FoBa algorithm, which we proposed [25], to build a predictive model from collected features. In our work, we showed that SPORE-FoBa outperforms LASSO and FoBa. The FoBa component of the algorithm helps cut down the number of interesting features first, and the SPORE component builds a fixed-degree (d) polynomial of all selected features, on which it then applies sparse, polynomial regression to build the model. For example, using a degree-2 polynomial with feature vector $\mathbf{v} = [x_1\ x_2]$, we expand out $(1 + x_1 + x_2)^2$ to get terms $1,\ x_1,\ x_2,\ x_1^2,\ x_1x_2,\ x_2^2$, and use them as basis functions to construct the following function for regression:

$$f(\mathbf{v}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_1 x_2 + \beta_5 x_2^2.$$

The resulting model can capture polynomial or sub-polynomial program complexities well thanks to Taylor expansion, which characterizes the vast majority of practical programs.

For a program whose execution time may dynamically

change over time as the workload changes, our performance model should evolve accordingly. The model can evolve in two ways: 1) the set of (non-linear) feature terms used in the model change; 2) with a fixed set of feature terms, their coefficients $\beta'_j s$ change. For a relatively stable program, we expect the former changes much less frequently than the latter. Using methods based on Stochastic Gradient Descent [9], it is feasible to update the set of feature terms and their coefficients $\beta'_j s$ online upon every execution time being collected.

5 Predictor Code Generation

The function output by the performance model generator is intended to efficiently predict the program's execution time on given program inputs. This requires a way to efficiently evaluate the features that appear in the function on those inputs. Many existing techniques rely on users to provide feature evaluators. A key contribution of our approach is the use of *static program slicing* [44, 46] to automatically extract from the (instrumented) program efficient feature evaluators in the form of *executable slices*—stand-alone executable programs whose sole goal is to evaluate the features. This section explains the rationale underlying our feature slicing (Section 5.1), describes the challenges of slicing and our approach to addressing them (Section 5.2), and provides the design of our slicer (Section 5.3).

5.1 Rationale

Given a program and a *slicing criterion* (p, v), where v is a program variable in scope at program point p, a *slice* is an executable sub-program of the given program that yields the same value of v at p as the given program, on all inputs. The goal of static slicing is to yield as small a sub-program as possible. It involves computing data and control dependencies for the slicing criterion, and excluding parts of the program upon which the slicing criterion is neither data- nor control-dependent.

In the absence of user intervention or slicing, a naïve approach to evaluate features would be to simply execute the (instrumented) program until all features of interest have been evaluated. This approach, however, can be grossly inefficient. Besides, our framework relies on feature evaluators to obtain the cost of each feature, so that it can iteratively reject costly features from the performance model. Thus, the naïve approach to evaluate features could grossly overestimate the cost of cheap features. We illustrate these problems with the naïve approach using two examples.

Example 1: A Java program may read a system property lazily, late in its execution, and depending upon its value decide whether or not to perform an expensive computation:

```
...;    // expensive computation S1
String s = System.getProperty(...);
if (s.equals(...)) {
  f_true++; // feature instrumentation
  ...; // expensive computation S2
}
```

In this case, feature `f_true` generated by our framework to track the number of times the above branch evaluates to true will be highly predictive of the execution time. However, the naïve approach for evaluating this feature will always perform the expensive computation denoted by `S1`. In contrast, slicing this program with slicing criterion (`p_exit`, `f_true`), where `p_exit` is the exit point of the program, will produce a feature evaluator that excludes `S1` (and `S2`), assuming the value of `f_true` is truly independent of computation `S1` and the slicer is precise enough.

Example 2: This example illustrates a case in which the computation relevant to evaluating a feature is interleaved with computation that is expensive but irrelevant to evaluating the feature. The following program opens an input text file, reads each line in the file, and performs an expensive computation on it (denoted by the call to the `process` method):

```
Reader r = new Reader(new File(name));
String s;
while ((s = r.readLine()) != null) {
  f_loop++;    // feature inst.
  process(s); // expensive computation
}
```

Assuming the number of lines in the input file is strongly correlated with the program's execution time, the only highly predictive feature available to our framework is `f_loop`, which tracks the number of iterations of the loop. The naïve approach to evaluate this feature will perform the expensive computation denoted by the `process` method in each iteration, even if the number of times the loop iterates is independent of it. Slicing this program with slicing criterion (`p_exit`, `f_loop`), on the other hand, can yield a slice that excludes the calls to `process(s)`.

The above two examples illustrate cases where the feature is fundamentally cheap to evaluate but slicing is required because the program is written in a manner that intertwines its evaluation with unrelated expensive computation.

5.2 Slicer Challenges

There are several key challenges to effective static slicing. Next we discuss these challenges and the approaches we take to address them. Three of these are posed by program artifacts—procedures, the heap, and concurrency—

and the fourth is posed by our requirement that the slices be executable.

Inter-procedural Analysis: The slicer must compute data and control dependencies efficiently and precisely. In particular, it must propagate these dependencies *context-sensitively*, that is, only along inter-procedurally realizable program paths—doing otherwise could result in inferring false dependencies and, ultimately, grossly imprecise slices. Our slicer uses existing precise and efficient inter-procedural algorithms from the literature [24, 33].

Alias Analysis: False data dependencies (and thereby false control dependencies as well) can also arise due to *aliasing*, i.e., two or more expressions pointing to the same memory location. Alias analysis is expensive. The use of an imprecise alias analysis by the slicer can lead to false dependencies. Static slicing needs may-alias information—analysis identifying expressions that may be aliases in at least some executions—to conservatively compute all data dependencies. In particular, it must generate a data dependency from an instance field write `u.f` (or an array element write `u[i]`) to a read `v.f` (or `v[i]`) in the program if `u` and `v` may-alias. Additionally, static slicing can also use must-alias information if available (expressions that are always aliases in all executions), to kill dependencies that no longer hold as a result of instance field and array element writes in the program. Our slicer uses a flow- and context-insensitive may-alias analysis with object allocation site heap abstraction [29].

Concurrency Analysis: Multi-threaded programs pose an additional challenge to static slicing due to the possibility of inter-thread data dependencies: reads of instance fields, array elements, and static fields (i.e., global variables) are not just data-dependent on writes in the same thread, but also on writes in other threads. Precise static slicing requires a precise static race detector to compute such data dependencies. Our may-alias analysis, however, suffices for our purpose (a race detector would perform additional analyses like thread-escape analysis, may-happen-in-parallel analysis, etc.)

Executable Slices: We require slices to be executable. In contrast, most of the literature on program slicing focuses on its application to program debugging, with the goal of highlighting a small set of statements to help the programmer debug a particular problem (e.g., Sirdharan et al. [40]). As a result, their slices do not need to be executable. Ensuring that the generated slices are executable requires extensive engineering so that the run-time does not complain about malformed slices, e.g., the first statement of each constructor must be a call to the super constructor even though the body of that super constructor is sliced away, method signatures must not be altered, etc.

5.3 Slicer Design

Our slicer combines several existing algorithms to produce executable slices. The slicer operates on a three-address-like intermediate representation of the bytecode of the given program.

Computing System Dependence Graph: For each method reachable from the program's root method (e.g., `main`) by our call-graph analysis, we build a Program Dependence Graph (PDG) [24], whose nodes are statements in the body of the method and whose edges represent intra-procedural data/control dependencies between them. For uniform treatment of memory locations in subsequent steps of the slicer, this step also performs a mod-ref analysis[1] and creates additional nodes in each PDG denoting implicit arguments for heap locations and globals possibly read in the method, and return results for those possibly modified in the method.

The PDGs constructed for all methods are stitched into a System Dependence Graph (SDG) [24], which represents inter-procedural data/control dependencies. This involves creating extra edges (so-called linkage-entry and linkage-exit edges) linking actual to formal arguments and formal to actual return results, respectively.

In building PDGs, we handle Java native methods, which are built with JNI calls, specially. We implement simple stubs to represent these native methods for the static analysis. We examine the code of the native method and write a stub that has the same dependencies between the arguments of the method, the return value of the method, and the class variables used inside the method as does the native method itself. We currently perform this step manually. Once a stub for a method is written, the stub can be reused for further analyses.

Augmenting System Dependence Graph: This step uses the algorithm by Reps, Horwitz, Sagiv, and Rosay [33] to augment the SDG with summary edges, which are edges summarizing the data/control dependencies of each method in terms of its formal arguments and return results.

Two-Pass Reachability: The above two steps are more computationally expensive but are performed once and for all for a given program, independent of the slicing criterion. This step takes as input a slicing criterion and the augmented SDG, and produces as output the set of all statements on which the slicing criterion may depend. It uses the two-pass backward reachability algorithm proposed by Horwitz, Reps, and Binkley [24] on the augmented SDG.

Translation: As a final step, we translate the slicer code based on intermediate representation to bytecode.

Extra Steps for Executable Slices: A set of program

[1] This finds all expressions that a method may *mod*ify-*ref*erence directly, or via some method it transitively calls.

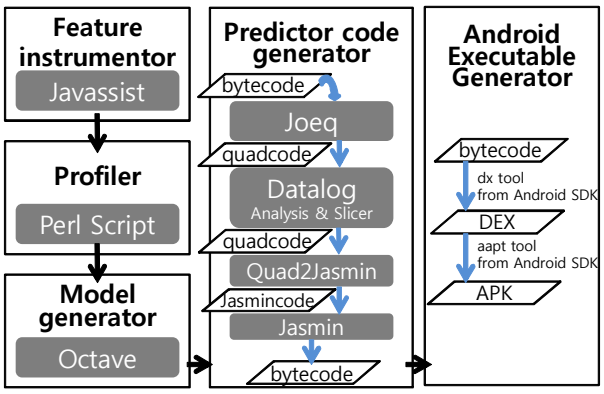


Figure 2: Mantis prototype toolchain.

statements identified by the described algorithm may not meet Java language requirements. This problem needs to be resolved to create executable slices.

First, we need to handle accesses to static fields and heap locations (instance fields and array elements). Therefore, when building an SDG, we identify all such accesses in a method and create formal-in vertices for those read and formal-out for those written along with corresponding actual-in and actual-out vertices. Second, there may be uninitialized parameters if they are not included in a slice. We opt to keep method signatures, hence we initialize them with default values. Third, there are methods not reachable from a main method but rather called from the VM directly (e.g., class initializers). These methods will not be included in a slice by the algorithm but still may affect the slicing criterion. Therefore, we do not slice out such code. Fourth, when a new object creation is in a slice, the corresponding constructor invocation may not. To address this, we create a control dependency between object creations and corresponding constructor invocations to ensure that they are also in the slice. Fifth, a constructor of a class except the Object class must include a call to a constructor of its parent class. Hence we include such calls when they are missing in a slice. Sixth, the first parameter of an instance method call is a reference to the associated object. Therefore if such a call site is in a slice, the first parameter has to be in the slice too and we ensure this.

6 Evaluation

We have built a prototype of Mantis implementing the instrumentor, profiler, model generator and predictor code generator (Figure 2). The prototype is built to work with Android application binaries. We implemented the feature instrumentor using Javassist [2], which is a Java bytecode rewriting library. The profiler is built using scripts to run the instrumented program on the test inputs and then the results are used by the model generator, which is written in Octave [4] scripts. Finally, we implemented our predictor code generator in Java and Datalog by extending JChord [3], a static and dynamic Java program-analysis tool. JChord uses the Joeq Java compiler framework to convert the bytecode of the input Java program, one class file at a time, into a three-address-like intermediate code called quadcode, which is more suitable for analysis. The predictor code generator produces the Joeq quadcode slice, which is the smallest subprogram that could obtain the selected features. Each quad instruction is translated to a corresponding set of Jasmin [1] assembly code, and then the Jasmin compiler generates the final Java bytecode.

We have applied the prototype to Android applications. Before Android applications are translated to Dalvik Executables (DEX), their Java source code is first compiled into Java bytecode. Mantis works with this bytecode and translates it to DEX to run on the device. Mantis could work with DEX directly, as soon as a translator from DEX to Joeq becomes available.

6.1 Experimental Setup

We run our experiments with a machine to run the instrumentor, model generator, and predictor code generator, as well as a smartphone to run the original and instrumented codes for profiling and generated predictor codes for slicing evaluation. The machine runs Ubuntu 11.10 64-bit with a 3.1GHz quad-core CPU, and 8GB of RAM. The smartphone is a Galaxy Nexus running Android 4.1.2 with dual-core 1.2Ghz CPU and 1GB RAM. All experiments were done using Java SE 64-bit 1.6.0_30.

The selected applications — Encryptor, Path Routing, Spam Filter, Chess Engine, Ringtone Maker and Face Detection — cover a broad range of CPU-intensive Android-application functionalities. Their execution times are sensitive to inputs, so challenging to model. Below we describe the applications and the input dataset we used for experiments in detail.

We evaluate Mantis on 1,000 randomly generated inputs for each application. These inputs achieve 95-100% basic-block coverage, only missing exception handling. We train our predictor on 100 inputs, and use the rest to test the predictor model. For each platform, we run Mantis to generate predictors and measure the prediction error and running time. The threshold is set to 5%, which means a generated predictor is accepted only if the predictor running time is less than 5% of the original program's completion time.

Encryptor: This encrypts a file using a matrix as a key. Inputs are the file and the matrix key. We use 1,000 files, each with its own matrix key. File size ranges from 10 KB to 8000 KB, and keys are 200×200 square matrices.

Path Routing: This computes the shortest path from one point to another on a map (as in navigation and game ap-

Application	Prediction error (%)	Prediction time (%)	No. of detected features	No. of chosen features
Encryptor	3.6	0.18	28	2
Path Routing	4.2	1.34	68	1
Spam Filter	2.8	0.51	55	1
Chess Engine	11.9	1.03	1084	2
Ringtone Maker	2.2	0.20	74	1
Face Detection	4.9	0.62	107	2

Table 1: Prediction error, prediction time, the total number of features initially detected and the number of chosen features.

Application	Selected features	Generated model
Encryptor	Matrix-key size (f_1), Loop count of encryption (f_2)	$c_0{f_1}^2 f_2 + c_1{f_1}^2 + c_2 f_2 + c_3$
Path Routing	Build map loop count (f_1)	$c_0{f_1}^2 + c_1 f_1 + c_2$
Spam Filter	Inner loop count of sorting (f_1)	$c_0 f_1 + c_1$
Chess Engine	No. of game-tree leaves (f_1), No. of chess pieces (f_2)	$c_0{f_1}^3 + c_1 f_1 f_2 + c_2{f_2}^2 + c_3$
Ringtone Maker	Cut interval length (f_1)	$c_0 f_1 + c_1$
Face Detection	Width of image (f_1), Height of image (f_2)	$c_0 f_1 f_2 + c_1{f_2}^2 + c_2$

Table 2: Selected features and generated prediction models.

plications). We use 1,000 maps, each with 100-200 locations, and random paths among them. We queried a route for a single random pair of locations for each map.

Spam Filter: This application filters spam messages based on a collective database. At initialization, it collects the phone numbers of spam senders from several online databases and sorts them. Then it removes white-listed numbers (from the user's phonebook) and builds a database. Subsequently, messages from senders in the database are blocked. We test Mantis with the initialization step; filtering has constant duration. We use 1,000 databases, each with 2,500 to 20,000 phone numbers.

Chess Engine: This is the decision part of a chess application. Similarly to many game applications, it receives the configuration of chess pieces as input and determines the best move using the Minimax algorithm. We set the game-tree depth to three. We use 1,000 randomly generated chess-piece configurations, each with up to 32 chess pieces.

Ringtone maker: This generates customized ringtones. Its input is a wav-format file and a time interval within the file. The application extracts that interval from the audio file and generates a new mp3 ringtone. We use 1,000 wav files, ranging from 1 to 10 minutes, and intervals starting at random positions and of lengths between 10 and 30 seconds.

Face Detection: This detects faces in an image by using the OpenCV library. It outputs a copy of the image, outlining faces with a red box. We use 1,000 images, of sizes between 100×100 and 900×3000 pixels.

6.2 Experiment Results

Accurate and Efficient Prediction: We first evaluate the accuracy and efficiency of Mantis prediction. Table 1 reports the prediction error and predictor running time of Mantis-generated predictors, the total number of features initially detected, and the number of features actually chosen to build the prediction model for each application. The "prediction error" column measures the accuracy of our prediction. Let $A(i)$ and $E(i)$ denote the actual and predicted execution times, respectively, computed on input i. Then, this column denotes the prediction error of our approach as the average value of $|A(i) - E(i)|/A(i)$ over all inputs i. The "prediction time" measures how long the predictor runs compared to the original program. Let $P(i)$ denote the time to execute the predictor. This column denotes the average value of $P(i)/A(i)$ over all inputs i.

Mantis achieves accuracy with prediction error within 5% in most cases, while each predictor runs around 1% of the original application's execution time, which is well under the 5% limit we assigned to Mantis.

We also show the effect of the number of training samples on prediction errors in Figure 3. For four applications, the curve of their prediction error plateaus before 50 input samples for training. For Chess Engine and Encryptor, the curve plateaus around 100 input samples for training. Since there is little to gain after the curve plateaus, we only use 100 input samples for training Mantis. Even for bigger input datasets of 10,000 samples, we only need about 100 input samples for training to obtain similar prediction accuracy.

Mantis generated interpretable and intuitive prediction

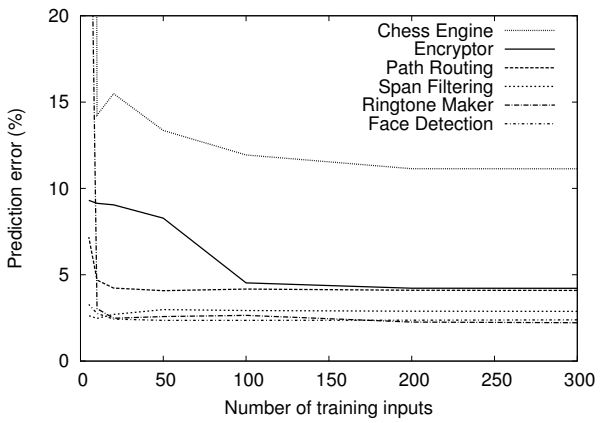


Figure 3: Prediction errors varying the number of input samples. The y-axis is truncated to 20 for clarity.

models by only choosing one or two among the many detected features unlike non-parametric methods. Table 2 shows the selected features and the generated polynomial prediction model for each application. In the model, c_n represents a constant real coefficient generated by the model generator and f_n represents the selected feature. The selected features are important factors in execution time, and they often interact in a non-linear way, which Mantis captures accurately. For example, for Encryptor, Mantis uses non-linear feature terms ($f_1^2 f_2$, f_1^2) to predict the execution time accurately.

Now we explain why Chess Engine has a higher error rate. Its execution time is related to the number of leaf nodes in the game tree. However, this feature can only be obtained late in the application execution and is dependent on almost all code that comes before it. Therefore, Mantis rejects this feature because it is too expensive. Note that we set the limit of predictor execution time to be 5% of the original application time. As the expensive feature is not usable, Mantis chooses alternative features: the number of nodes in the second level of the game tree and the number of chess pieces left; these features can capture the behavior of the number of leaf nodes in the game tree. Although they can only give a rough estimate of the number of leaf nodes in the game tree, the prediction error is still around only 12%.

Benefit of Non-linear Terms on Prediction Accuracy: Table 3 shows the prediction error rates of the models built by Mantis and Mantis-linear. Mantis-linear uses only linear terms (f_i's) for model generation. For Encryptor, Path Routing, and Face Detection, non-linear terms improve prediction accuracy significantly since Mantis-linear does not capture the interaction between features.

Benefit of Slicing on Prediction Time: Next we discuss how slicing improves the prediction time. In Table 4, we compare the prediction times of Mantis-generated predictors with those of predictors built with *partial execution*. Partial Execution (PE) runs the instrumented program only until the point where we obtain the chosen feature values.

Application	Mantis pred. error (%)	Mantis-linear pred. error (%)
Encryptor	3.6	6.6
Path Routing	4.2	13.8
Spam Filter	2.8	2.8
Chess Engine	11.9	13.2
Ringtone Maker	2.2	2.2
Face Detection	4.9	52.7

Table 3: Prediction error of Mantis and Mantis-linear. Mantis-linear uses only linear terms (f_i's) for model generation.

Application	Mantis pred. time (%)	PE pred. time (%)
Encryptor	0.20	100.08
Path Routing	1.30	17.76
Spam Filter	0.50	99.39
Chess Engine	1.03	69.63
Ringtone Maker	0.20	0.04
Face Detection	0.61	0.17

Table 4: Prediction time of Mantis and PE.

Mantis reduces the prediction time significantly for Encryptor, Path Routing, Spam Filter, and Chess Engine. For these applications, PE predictors need to run a large piece of code, which includes code that is unrelated to the chosen features until their values are obtained.

Spam Filter and Encryptor are the worst cases for PE since the last updates of the chosen feature values occur near the end of their execution. In contrast, Ringtone Maker and Face Detection can obtain the chosen feature values cheaply even without slicing. This is because the values for the chosen features can be obtained at the very beginning in the application's execution. In fact, the Mantis-generated predictors of these applications take longer than PE because the generated code is less optimized than the code generated directly by the compiler.

Benefit of Slicing on Prediction Accuracy: To show the effect of slicing on prediction accuracy under a prediction time limit, we compare our results with those obtained using *bounded execution*. Bounded Execution (BE) gathers features by running an instrumented application for only a short period of time, which is the same as the time a Mantis-generated predictor would run. It then uses these gathered features with the Mantis model generator to build a prediction model.

As shown in Table 5, the prediction error rates of the models built by BE are much higher than those of the

Application	Galaxy S2		Galaxy S3	
	Prediction error (%)	Prediction time (%)	Prediction error (%)	Prediction time (%)
Encryptor	4.6	0.35	3.4	0.08
Path Routing	4.1	3.07	4.2	1.28
Spam Filter	5.4	1.52	2.2	0.52
Chess Engine	9.7	1.42	13.2	1.38
Ringtone Maker	3.7	0.51	4.8	0.20
Face Detection	5.1	1.28	5.0	0.69

Table 6: Prediction error and time of Mantis running with Galaxy S2 and Galaxy S3.

Application	Mantis pred. error (%)	BE pred. error (%)
Encryptor	3.6	56.0
Path Routing	4.2	64.0
Spam Filter	2.8	36.2
Chess Engine	11.9	26.1
Ringtone Maker	2.2	2.2
Face Detection	4.9	4.9

Table 5: Prediction error of Mantis and BE.

models built by Mantis. This is because BE cannot exploit as many features as Mantis. For Spam Filter and Encryptor, no usable feature can be obtained by BE; thus, BE creates a prediction model with only a constant term for each of the two applications.

Prediction on Different Hardware Platforms: Next we evaluate whether Mantis generates accurate and efficient predictors on three different hardware platforms. Table 6 shows the results of Mantis with two additional smartphones: Galaxy S2 and Galaxy S3. Galaxy S2 has a dual-core 1.2Ghz CPU and 1GB RAM, running Android 4.0.3. Galaxy S3 has a quad-core 1.4Ghz CPU and 1GB RAM, running Android 4.0.4. As shown in the table, Mantis achieves low prediction errors and short prediction times with Galaxy S2 and Galaxy S3 as well. For each application, Mantis builds a model similar to the one generated for Galaxy Nexus. The chosen features for each device are the same as or equivalent (e.g., there can be multiple instrumented variables with the same value) to the chosen features for Galaxy Nexus, while the model coefficients are changed to match the speed of each device. The result shows that Mantis generates predictors robustly with different hardware platforms.

Prediction under Background Load: Finally, we evaluate how the predictors perform under changing environmental loads. Table 7 shows how much effect CPU-intensive loads have on the performance of Mantis predictors for Galaxy Nexus. The application execution times under the background CPU-intensive load are compared to the predicted execution times of Mantis predictors generated with an idle smartphone. The background load is generated by the SysBench package [5], which consists of a configurable number of events that compute prime numbers. For our evaluation, the load is configured to initially have a steady 50%, 75%, or 99% CPU usage. The test applications run in the foreground.

Application	Pred. error (%) for the x% background CPU load			
	x=0	x=50	x=75	x=99
Encryptor	3.6	7.5	10.5	21.3
Path Routing	4.2	5.3	5.8	6.7
Spam Filter	2.8	4.7	5.2	5.8
Chess Engine	11.9	13.5	15.3	15.8
Ringtone Maker	2.2	2.3	3.0	3.1
Face Detection	4.9	5.3	5.6	5.8

Table 7: Prediction error of Mantis-generated predictors for Galaxy Nexus under background CPU-intensive loads.

As shown in the table, in most cases background load has only a moderate effect on the accuracy of Mantis predictors. This is mainly due to Android's scheduling policy, which gives a higher priority to the process that is actively running on the screen, or foreground, compared with the other processes running in the background. We observed that when an application was started and brought to the foreground, the Android system secured enough CPU time for the process to run smoothly by reducing the CPU occupancy of the background load.

However, the prediction error for Encryptor increases as the CPU load increases. Unlike the other applications, Encryptor creates a larger number of heap objects and calls Garbage Collection (GC) more often. We also observed that GC slows down under the heavy load, resulting in a slowdown of Encryptor's total execution time. This in turn makes it difficult for the Mantis predictor to predict the Encryptor execution time accurately under a heavy load. An extension of Mantis is to include environmental factors (e.g., the background CPU load) as features in Mantis prediction models.

Mantis Offline Stage Processing Time: Table 8 presents

Application	Profiling	Model gen.	Slicing	Test	Total	Iterations
Encryptor	2373	18	117	391	2900	3
Path Routing	363	28	114	14	519	3
Spam Filter	135	10	66	3	214	2
Chess Engine	6624	10229	6016	23142	46011	83
Ringtone Maker	2074	19	4565	2	6659	1
Face Detection	1437	13	6412	179	8041	4

Table 8: Mantis offline stage processing time (in seconds).

Mantis offline stage processing (profiling, model generation, slicing, and testing) time for all input training data. The total time is the sum of times of all steps. The last column shows how many times Mantis ran the model generation and slicing part due to rejected features. For the applications excluding Chess Engine, the total time is less than a few hours, the profiling part dominates, and the number of iterations in the feedback loop is small. Chess Engine's offline processing time takes 12.8 hours because of many rejected features. We leave speeding up this process as future work.

Summary: We have demonstrated that our prototype implemenation of Mantis generates good predictors for our test programs that estimate running time with high accuracy and very low cost. We have also compared our approach to simpler, intuitive approaches based on Partial Execution and Bounded Execution, showing that Mantis does significantly better in almost all cases, and as well in the few cases where Partial Execution happened upon good prediction features. Finally, we showed that Mantis predictors are accurate on three different hardware platforms and are little affected by variability in background CPU load.

7 Related Work

Much research has been devoted to modeling system behavior as a means of prediction for databases [16, 21], cluster computing [8, 39], networking [12, 31, 41], program optimization [26, 42], etc.

Prediction of basic program characteristics, execution time, or even resource consumption, has been used broadly to improve scheduling, provisioning, and optimization. Example domains include prediction of library and benchmark performance [28, 45], database query execution-time and resource prediction [16, 21], performance prediction for streaming applications based on control flow characterization [6], violations of Service-Level Agreements (SLAs) for cloud and web services [8, 39], and load balancing for network monitoring infrastructures [7]. Such work demonstrates significant benefits from prediction, but focuses on problem domains that have identifiable features (e.g., operator counts in database queries, or network packet header values) based on expert knowledge, use domain-specific feature extraction that may not apply to general-purpose programs, or require high correlation between simple features (e.g., input size) and execution time.

Delving further into extraction of non-trivial features, research has explored extracting predictors from execution traces to model program complexity [17], to improve hardware simulation specificity [37, 38], and to find bugs cooperatively [32]. There has also been research on multi-component systems (e.g., content-distribution networks) where the whole system may not be observable in one place. For example, extracting component dependencies (web objects in a distributed web service) can be useful for what-if analysis to predict how changing network configuration will impact user-perceived or global performance [12, 31, 41].

A large body of work has targeted worst-case behavior prediction, either focusing on identifying the inputs that cause it, or on estimating a tight upper bound [22, 30, 35, 36, 47] in embedded and/or real-time systems. Such efforts are helped by the fact that, by construction, the systems are more amenable to such analysis, for instance thanks to finite bounds on loop sizes. Other work focuses on modeling algorithmic complexity [17], simulation to derive worst-case running time [34], and symbolic execution and abstract evaluation to derive either worst-case inputs for a program [11], or asymptotic bounds on worst-case complexity [19, 20]. In contrast, our goal is to automatically generate an online, accurate predictor of the performance of particular invocations of a general-purpose program.

Finally, Mantis's machine learning algorithm for prediction is based on our earlier work [25]. In the prior work, we computed program features manually. In this work, we introduce program slicing to compute features cheaply and generate predictors automatically, apply our whole system to Android smartphone applications on multiple hardware platforms, and evaluate the benefits of slicing thoroughly.

8 Conclusion

In this paper, we presented Mantis, a framework that automatically generates program performance predictors that

can estimate performance accurately and efficiently. Mantis combines program slicing and sparse regression in a novel way. The key insight is that we can extract information from program executions, even when it occurs late in execution, cheaply by using program slicing and generate efficient feature evaluators in the form of executable slices. Our evaluation shows that Mantis can automatically generate predictors that estimate execution time accurately and efficiently for smartphone applications. As future work, we plan to explore how to extend Mantis to predict other metrics like resource consumption and evaluate Mantis for diverse applications.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and our shepherd, Paul Leach, for his guidance.

References

[1] Jasmin. `jasmin.sourceforge.net`.

[2] Javassist. `www.csg.is.titech.ac.jp/~chiba/javassist`.

[3] JChord. `code.google.com/p/jchord`.

[4] Octave. `www.gnu.org/software/octave`.

[5] Sysbench: a system performance benchmark. `http://sysbench.sourceforge.net/`.

[6] F. Aleen, M. Sharif, and S. Pande. Input-Driven Dynamic Execution Behavior Prediction of Streaming Applications. In *PPoPP*, 2010.

[7] P. Barlet-Ros, G. Iannaccone, J. Sanjuas-Cuxart, D. Amores-Lopez, and J. Sole-Pareta. Load Shedding in Network Monitoring Applications. In *USENIX*, 2007.

[8] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *HotCloud*, 2009.

[9] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *COMPSTAT*, 2010.

[10] E. Brewer. High-Level Optimization via Automated Statistical Modeling. In *PPoPP*, 1995.

[11] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, 2009.

[12] S. Chen, K. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting. Link Gradients: Predicting the Impact of Network Latency on Multitier Applications. In *INFOCOM*, 2009.

[13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *EuroSys*, 2011.

[14] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *MobiSys*, 2010.

[15] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least Angle Regression. *Annals of Statistics*, 32(2), 2002.

[16] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, 2009.

[17] S. Goldsmith, A. Aiken, and D. Wilkerson. Measuring Empirical Computational Complexity. In *FSE*, 2007.

[18] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *OSDI*, 2012.

[19] B. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*, 2008.

[20] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*, 2009.

[21] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*, 2008.

[22] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *RTSS*, 2006.

[23] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2009.

[24] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI*, 1988.

[25] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In *NIPS*, 2010.

[26] Y. Jiang, E. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *CGO*, 2010.

[27] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An Interior-Point Method for Large-Scale l1-Regularized Least Squares. *IEEE J-STSP*, 1(4), 2007.

[28] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, 2006.

[29] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, School of Computer Science, McGill University, 2006.

[30] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwar Academic Publishers, 1999.

[31] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating Performance Prediction for Web Services. In *NSDI*, 2010.

[32] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *PLDI*, 2005.

[33] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up Slicing. In *FSE*, 1994.

[34] R. Rugina and K. E. Schauser. Predicting the Running Times of Parallel Programs by Simulation. In *IPPS/SPDP*, 1998.

[35] S. Seshia and A. Rakhlin. Game-Theoretic Timing Analysis. In *ICCAD*, 2008.

[36] S. Seshia and A. Rakhlin. Quantitative Analysis of Systems Using Game-Theoretic Learning. *ACM TECS*, 2010.

[37] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *PACT*, 2001.

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, 2002.

[39] P. Shivam, S. Babu, and J. S. Chase. Learning Application Models for Utility Resource Planning. In *ICAC*, 2006.

[40] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[41] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar. Answering What-If Deployment and Configuration Questions with WISE. In *SIGCOMM*, 2008.

[42] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA*, 2010.

[43] R. Tibshirani. Regression Shrinkage and selection via the Lasso. *J. Royal. Statist. Soc B.*, 1996.

[44] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3), 1995.

[45] K. Vaswani, M. Thazhuthaveetil, Y. Srikant, and P. Joseph. Microarchitecture Sensitive Empirical Models for Compiler Optimizations. In *CGO*, 2007.

[46] M. Weiser. Program Slicing. In *ICSE*, 1981.

[47] R. Wilhelm. Determining Bounds on Execution Times. *Handbook on Embedded Systems*, 2005.

[48] T. Zhang. Adaptive forward-backward greedy algorithm for sparse learning with linear models. In *NIPS*, 2008.

I/O Stack Optimization for Smartphones

Sooman Jeong[1], Kisung Lee[2,*], Seongjin Lee[1], Seoungbum Son[2,*], and Youjip Won[1]
[1] Hanyang University, Seoul, Korea
[2]Samsung Electronics, Suwon, Korea

Abstract

The Android I/O stack consists of elaborate and mature components (SQLite, the EXT4 filesystem, interrupt-driven I/O, and NAND-based storage) whose integrated behavior is not well-orchestrated, which leaves a substantial room for an improvement. We overhauled the block I/O behavior of five filesystems (EXT4, XFS, BTRFS, NILFS, and F2FS) under each of the five different journaling modes of SQLite. We found that the most significant inefficiency originates from the fact that filesystem journals the database journaling activity; we refer to this as the JOJ (Journaling of Journal) anomaly. The JOJ overhead compounds in EXT4 when the bulky EXT4 journaling activity is triggered by an `fsync()` call from SQLite. We propose (i) the elimination of unnecessary metadata journaling for the filesystem, (ii) external journaling and (iii) polling-based I/O to improve the I/O performance, primarily to improve the efficiency of filesystem journaling in the SQLite environment. We examined the performance trade-offs for each combination of the five database journaling modes, five filesystems and three optimization techniques. When we applied three optimization techniques in existing Android I/O stack, the SQLite performance (inserts/sec) improved by 130%. With the F2FS filesystem, WAL journaling mode (SQLite), and the combination of our optimization efforts, we improved the SQLite performance (inserts/sec) by 300%, from 39 ins/sec to 157 ins/sec, compared to the stock Android I/O stack.

1 Introduction

Smart devices, e.g., smartphones, tablets, and smart TVs, have become mainstream computing devices and are quickly replacing their predecessor, PCs. Smartphones and tablets have become the dominant source of DRAM consumption [17] and account for 45% of Internet web browsing [18]. They are becoming *the* personal computing devices for a variety of applications, including social network services, games, cameras, camcorders, mp3 players, and web browsers.

The application performance of a smartphone is not governed by the speed of its airlink, e.g., Wi-Fi, but rather by the storage performance, which is currently utilized in a quite inefficient manner [11]. Furthermore, one of the main sources of this inefficiency is an excessive I/O activity caused by uncoordinated interactions between EXT4 journaling and SQLite journaling [14]. Despite its significant implications for the overall smartphone performance, the I/O subsystem behavior of smartphones has not been studied nearly as thoroughly as those in enterprise servers [26, 23], web servers [4, 10], OLTP servers [15], and desktop PCs [34, 8].

In this work, we present extensive measurement results to understand Android's I/O behavior and propose techniques to optimize the individual layers so that the overall Android I/O stack behaves much more efficiently when the layers are integrated. The Android I/O stack is a collection of elaborate and mature software layers (SQLite, EXT4, the interrupt-driven I/O of the Linux kernel, and NAND-based storage), each of which has gone through several years of development and refinement. When the layers are integrated, the resulting I/O behavior is not well-orchestrated and leaves a substantial room for an improvement. We overhaul the I/O stack of the Android platform from DBMS to a storage device and propose several techniques to improve the performance. Our contributions are as follows:

• Starting from EXT4, we performed an extensive performance study of five filesystems (BTRFS, XFS, NILFS, and F2FS [1]) in one of the most recent Android-based smartphones and examined how they interact with each journaling mode of SQLite. We found that SQLite journaling interacts with the EXT4 journaling layer in an unexpected way and, the EXT4 filesystem stresses the storage device in a way that was rarely seen before.

* This work was done while the author was a graduate student at Hanyang University.

We found that recently introduced F2FS can be a good remedy for Journaling of Journal anomaly which current stock Android I/O stack suffers from.

• Examining five journal modes of SQLite, we found that Write-Ahead-Logging mode(WAL) yields the best performance since it generates smallest amount of the synchronous random writes amongst all SQLite journal modes.

• We propose the use of external journaling, in which the filesystem journal is maintained in a separate storage device to explicitly preserve the access locality induced by the filesystem journal file access. This approach enables the FTL layer of the NAND storage to more effectively exploit the locality of the incoming I/O stream so that it can reduce the NAND flash management overhead, e.g., garbage collection in page mapping and the log block merge operation in hybrid mapping.

• We found that SQLite triggers a significant amount of synchronous random write traffic when it commits its journal file to the filesystem, a significant fraction of which is not required. We tuned SQLite to eliminate this random write I/O by employing `fdatasync()` in place of `fsync()`.

• NAND-based storage is sufficiently fast, and state-of-the-art smartphones are equipped with a sufficient number of CPU cores. We developed a polling-based I/O system for Android storage devices and studied its effectiveness.

Combining these optimization efforts with the optimal choices for the filesystem and database journaling mode of SQLite (i.e., by F2FS, WAL journaling mode in SQLite, using external journaling, eliminating unnecessary metadata commits, and polling-based I/O), we achieved a 300% improvement in the SQLite performance (inserts/sec) compared to the stock Android platform.

The remainder of the paper is organized as follows: Section 2 presents the background. The I/O characteristics of Android is briefly described in Section 3, and the current Android I/O stack is examined in Section 4. Section 5 explores various filesystem choices for Android. Section 6 provides optimization techniques for the Android I/O stack. Section 7 presents the results of our integration of the proposed schemes. Section 8 describes other works related to this study. Our conclusions are presented in Section 9.

2 Background

2.1 Android I/O Stack

Google Android is an open-source Linux-based operating system designed for mobile devices. Figure 1 illustrates the architecture of Android. Android applications are written in Java and packaged as `.apk` (Android Application Package) files. Android provides a set of libraries used extensively by various system components and applications; some of the most widely used libraries are SQLite, libc, and the media libraries. The Linux kernel provides core services, such as memory management, process management, security, networking, and a driver model. Android uses the Dalvik virtual machine (VM) with just-in-time compilation to run `.dex` (Dalvik Executable) files, and an application runs on top of the Dalvik VM.

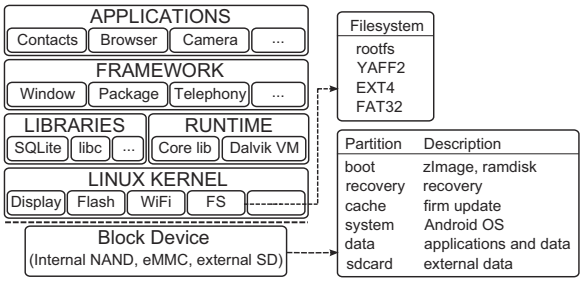


Figure 1: Android Architecture and Storage Partition

We define the Android I/O stack as a set of software and hardware layers used by applications for persistent data management. The I/O stack of the Android platform consists of the DBMS, filesystem, block device driver, and NAND flash based storage device. SQLite and EXT4 are the default DBMS and filesystem, respectively. The Android platform uses interrupt driven I/O with a CFQ I/O scheduling scheme. The eMMC and SD card are used as internal and external storage devices, respectively.

Most Android applications use SQLite to manage data in a persistent manner. SQLite interacts with the underlying filesystems through the usual system calls, such as `open()`, `unlink()`, `write()`, and `fsync()`. SQLite uses journaling for recovery. It records rollback information at `.db-journal` file. The database file and journal file are frequently synchronized with the storage device using `fsync()`.

Since the release of Android 4.0.4 (Ice Cream Sandwich), Android only uses EXT4 to manage its internal storage, eMMC.

2.2 AndroStep: Android Storage Analyzer

We use Androstep [9] to collect, analyze and replay the trace in this study. AndroStep is a collection of tools developed for analyzing the storage stack behavior of Android. It consists of Mobibench[1], MOST[2], and Mobigen[3]. Mobibench (mobile benchmark) is an I/O workload generator which is specifically designed for An-

[1] `https://github.com/ESOS-Lab/Mobibench`, available at Google playstore

[2] `https://github.com/ ESOS-Lab/MOST`

[3] `https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen`

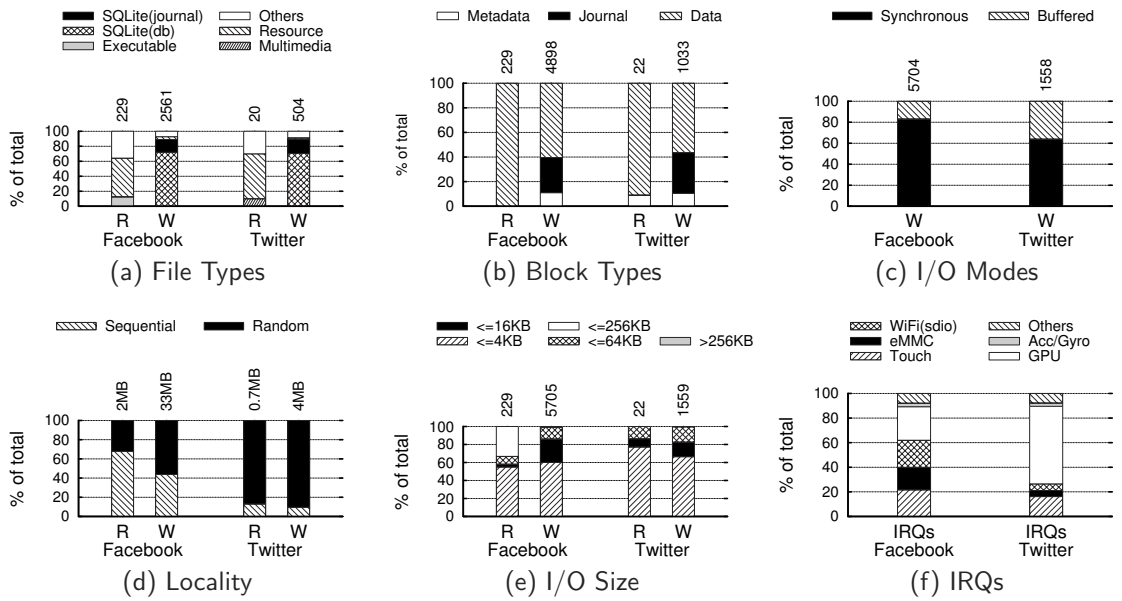


Figure 2: I/O distribution of file types, block types, I/O modes, randomness, and I/O size. The number on the top of each bar indicates the number of respective block I/O for R (Read) and W (Write), respectively.

droid workload generation. It can generate SQLite workload (insert, update, and delete) and filesystem workload (read, write, create, unlink etc). User can configure SQLite journaling option, filesystem journaling operation, and various filesystem I/O options, e.g., direct vs. buffered I/O, synchronous I/O, and etc. The accuracy of Mobibench is validated against existing widely used benchmark IOZONE [9].

MOST (mobile storage analyzer) is a tool to collect and to analyze the block level trace. From the block trace, MOST can identify the block type (metadata, journal, and data), and the file type of the respective block such as SQLite journal/database, apk, and etc. The salient feature of MOST is that it keeps track of this information for deleted files.

In addition to Mobibench and MOST, Androstep has a tool to record and to replay the system call trace, Mobigen (Mobile Workload Generator). Mobigen is used to collect the system calls generated from the human user behavior for using a given application. By replaying the system call trace, Mobigen can reproduce the human driven I/O activities without actual human intervention.

3 I/O Characteristics of Android Applications

Prior works performed extensive study of Android I/O characterization and found that a significant fraction of the I/O's are generated by SQLite operation [11, 14]. Kim et.al. [11] found that most I/Os in Android platform are related to SQLite database operations. Lee et.al [14] performed extensive I/O characterization study and found that dominant fraction of Andriod I/O is synchronous random write caused by misaligned interaction between SQLite and EXT4 filesystem. We analyzed the I/O behavior of Facebook and Twitter apps, both of which are highly popular smartphone applications. We present the analysis results only for Facebook and Twitter apps because the results are well aligned with our prior study on fourteen popular Android apps and exhibits similar characteristics [14].

The results of the study presented here are based on the Galaxy S3 (Samsung Exynos 4412 1.4 GHz Quadcore, 2 GB RAM, 32 GB eMMC, Android 4.0.4 with Linux kernel 3.0.15)[4]. We use MOST (Mobile Storage Analyzer) [9] to collect and analyze the I/O trace. Figure 2 illustrates the results of the analysis. The numbers on the top of each bar represent the number of I/O requests. We briefly summarize our findings as follows:

- *90% of the write requests are to the SQLite database and journal.* We categorize the files into six categories: database file (`.db`), journal file (`.db-journal`), executables (`.so`, `.apk`, and `dex`), resources (`.dat` and `.xml`), and others. We found that SQLite and its journal file are responsible for approximately 90% of the write I/O requests in both Facebook and Twitter apps (Figure 2(a)).
- *Writes to the EXT4 journal block constitute 30% of all writes.* We categorized the blocks in the filesystem partition into three types: metadata, journal,

[4]We have also tested earlier smartphone models, the Nexus S (Android 2.3 "Gingerbread", 2010 Nov.) and Galaxy S (Android 2.1 "Eclair", 2010 Mar.). We only show the Galaxy S3 results to save space. I/O behaviors of earlier smartphone models are similar to that of the Galaxy S3.

and data. 10% and 30% of all writes are for the metadata and journal, respectively (Figure 2(b)).

- *Of all writes, 70% are synchronous.* Figure 2(c) shows the number of buffered and synchronous writes. 70% of all writes are synchronous I/O operations, initiated primarily by SQLite.
- *75% of all writes are random.* Figure 2(d) shows the spatial characteristics of the write operations. In general, random writes are unfavorable for NAND storage devices and are considered to be a source of performance degradation.
- *64% of the I/O operations involve data with size less than 4 KB.* Figure 2(e) shows the I/O size distribution. A dominant fraction (64%) of the I/O requests has sizes of 4 KB. This is because in SQLite on EXT4, every update to the database table and the respective journaling activity are synchronized with the storage device.
- *The interrupt requests issued by the eMMC comprise 18% of all interrupts.* Figure 2(f) shows the interrupt requests from each device driver. We found that the eMMC is responsible for 18% of the interrupt requests on average.

4 Analysis of the Android I/O Stack

In this section we examine SQLite journaling and EXT4 file system journaling. We focus on how Android storage system is affected subject to the SQLite journaling mode, especially SQLite journaling and EXT4 journaling are both active.

4.1 Journaling in SQLite

SQLite is the most popular persistent data management module on the Android platform. Even multimedia players use SQLite to store configuration options such as the speaker volume. SQLite uses journaling to provide transactional capabilities for its database operations. There are six journaling modes in SQLite: DELETE, TRUNCATE (default in Android 4.0.4), PERSIST, MEMORY, write-ahead logging (WAL), and OFF. The differences among these modes are both subtle and profound.

In DELETE, SQLite creates a journal file at the start of a transaction and deletes it upon completion of the transaction. After the journal file is created, SQLite inserts journal records and calls `fsync()` to make the journal file persistent.

In TRUNCATE mode, SQLite truncates the journal file to zero instead of unlinking it when the transaction completes. This truncation is performed to relieve the burden of updating the metadata (for example, the directory block and inode bitmap) involved in creating and deleting the database journal file.

PERSIST mode takes a more aggressive approach than TRUNCATE mode to more efficiently reduce the journaling. In PERSIST mode, SQLite writes zeros at the beginning of the database journal when the transaction completes instead of truncating the file. When inserting a new record into journal file, PERSIST mode uses the existing blocks (zero-filled block), whereas TRUNCATE mode allocates a new block. The amount of metadata committed to filesystem journal is smaller in PERSIST mode compared to TRUNCATE mode.

In MEMORY mode, the journal records are kept in memory. Since MEMORY mode does not rely on filesystem service to maintain journal records, MEMORY mode does not have any variants different from the filesystem based journal modes.

WAL journaling mode creates a separate WAL file (`.wal`) and logs the database updates to the log file. When the `.wal` file reaches a specified threshold size, the outstanding logs of the `.wal` file are checkpointed to the database file (`.db`). In WAL mode, I/O operations tend to be sequential; therefore, this mode is good for exploiting the nature of NAND flash storage. The OFF journaling mode does not use journaling.

4.2 EXT4 Journaling and fsync()

EXT4 has long been the default filesystem on the Android platform. For efficiency, EXT4 journaling maintains the log records for multiple system calls as a single unit called a *journal transaction* and uses this as the unit at which the log records in memory are committed to filesystem journal. Normally, the overhead of journaling is negligible in EXT4 because a journal transaction consists of a large number of log records and a journal transaction is committed to the EXT4 journal file at relatively long intervals, e.g., every 5 sec. In the Android platform, however, the journaling overhead of EXT4 becomes rather significant because of its frequent `fsync()` calls. As we show in Section 4.3, the insert operation in SQLite issues two or more `fsync()` calls within 2 msec. Each `fsync()` call triggers the commit of a journal transaction in which the journal transaction consists of very few (often one or two) log records that represent the updated in-core metadata for the respective file. Consequently, EXT4 journaling becomes very inefficient when it is triggered by `fsync()`.

Let us physically examine the effect of `fsync()` in EXT4 journaling (ordered mode). We generated a 4 KB write followed by an `fsync()` call. Figure 3(a) illustrates the results. In ordered mode, filesystem first updates the file and commits the respective file metadata to filesystem journal. As a result of `fsync()`, there occurs three write operations to the storage. The first write in the lower range of LBA is the data update. The second and the third writes are for committing updated metadata to filesystem journal; writing the journal descriptor, inserting the log record(s) and writing a journal commit mark. In Figure 3(a), the journal descriptor and log record are

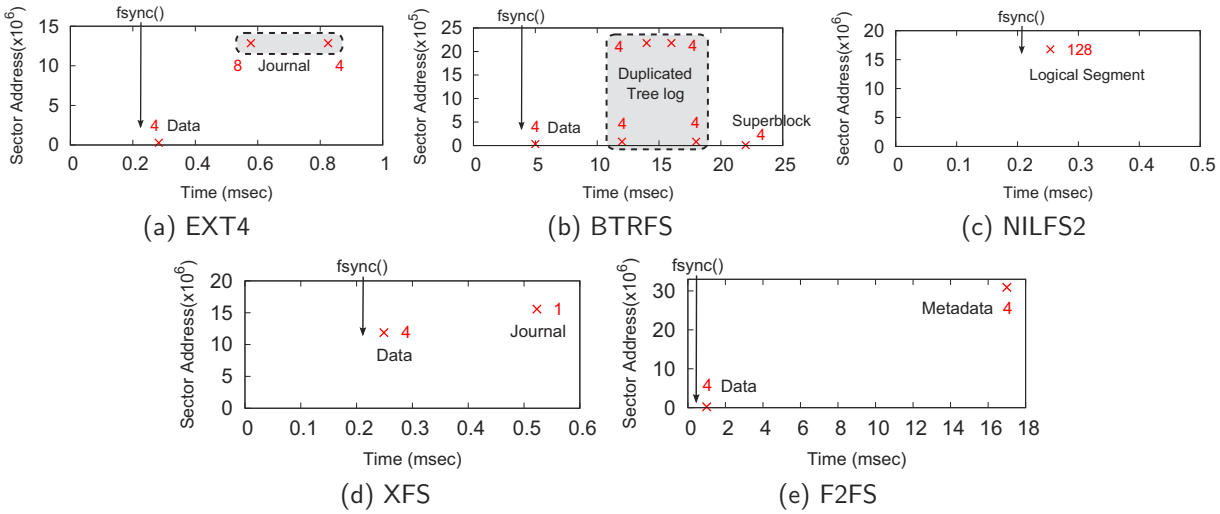


Figure 3: Block I/O pattern: 4 KB `write()` followed by `fsync()` EXT4, BTRFS, NILFS2, XFS, and F2FS. Number at each block I/O denotes I/O size in KB.

written in a single write operation. A single `write()` system call entails two additional block write operations, which are for updating the filesystem journal. The journaling overhead is 200% in this experiment.

`fsync()` not only creates additional write operations but also disintegrates the locality of the underlying workload. `fsync()` introduces more randomness to the underlying traffic because of frequent journal commits; consequently, `fsync()` significantly degrades the performance as well as lifetime of NAND-based storage.

4.3 Journaling of Journal: Interaction between SQLite and EXT4

Previous study [14] reported that the excessive I/O behavior of Android-based smartphones is due to the uncoordinated interaction between SQLite and EXT4, but the detailed mechanism has not been studied. We performed an in-depth analysis of the block-level I/O activity caused by SQLite and EXT4 (ordered mode). The application inserted one record (100 Byte) into the SQLite database table in this experiment. For comprehensiveness of the study, we examined four journaling modes of SQLite: DELETE, TRUNCATE, PERSIST, and WAL. Figure 4 shows the results. We denote the time of I/O, the respective starting LBA, and the size. Additionally, we specified the file where the I/O is designated.

In SQLite, the insert operation primarily consists of two phases: (i) it logs the insert operation at the SQLite journal, and (ii) it inserts the record to the actual database table. SQLite calls `fsync()` at the end of each phase to make the results persistent. Each `fsync()` call makes EXT4 filesystem update the file (database journal or database table) and write the updated metadata to the EXT4 journal.

Let us begin with DELETE mode (Figure 4(a)). SQLite creates the journal file (`.db-journal`), enters the journal entry for the insert operation and then calls `fsync()`. Upon `fsync()`, EXT4 writes `.db-journal` to storage and commits the updated metadata for `.db-journal` to the EXT4 journal. Then, SQLite inserts the record into the database table (`.db`) and calls `fsync()` to force the record into storage. When `fsync()` is called again, the same steps are repeated as in the first phase. Finally, SQLite calls `unlink()` to delete the `.db-journal` file. *A single insert operation results in nine I/Os to the storage device.*

The differences among three different journaling modes of SQLite, DELETE, TRUNCATE and PERSIST, lie in how SQLite treats the database journal file (`.db-journal`). These differences affect the amount of metadata committed to the EXT4 journal. When SQLite reuses the journal file (TRUNCATE mode), EXT4 is relieved from the burden of committing the metadata updates caused by the creation and deletion of SQLite journal. In PERSIST mode, SQLite not only reuses the existing journal but also reuses the data blocks of the journal file. Consequently, when SQLite operates in PERSIST mode, EXT4 is further relieved from the burden of committing the updated metadata involved in allocating a new data block to SQLite journal. Let us look at our experiment results (Figure 3). The first write operation designated to filesystem journal in each of Figure 4(a), Figure 4(b) and Figure 4(c) is for committing the updated metadata for the SQLite journal (`.db-journal`) to the EXT4 journal. The sizes of these operations are 24 KB, 16 KB, and 8 KB in DELETE, TRUNCATE, and PERSIST modes, respectively.

In PERSIST mode, however, SQLite generates addi-

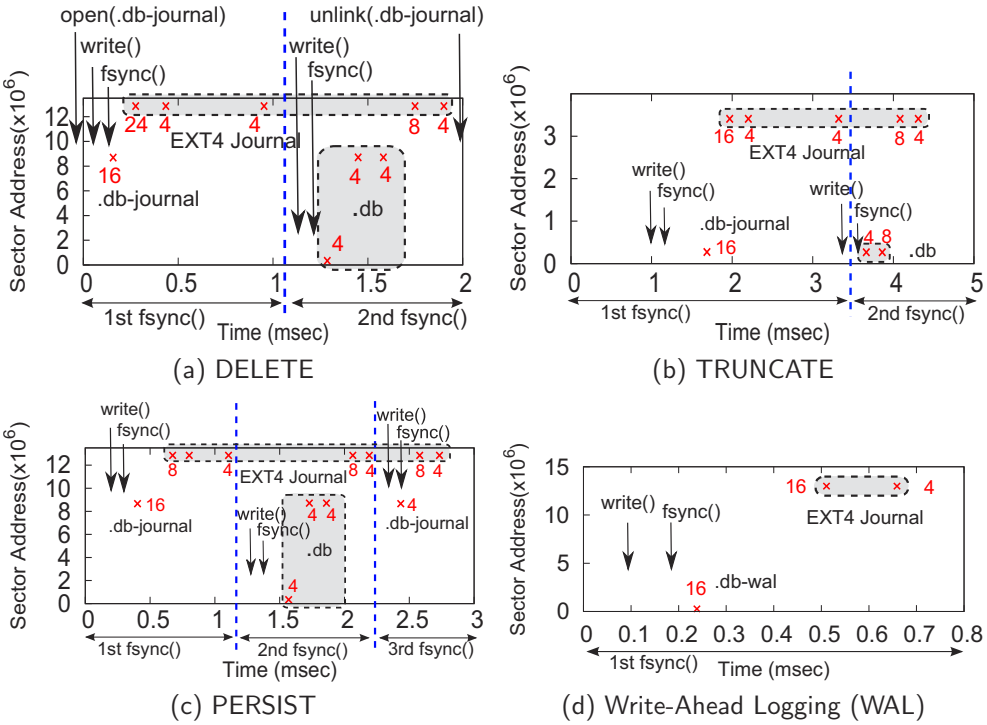


Figure 4: Block I/O accesses of the SQLite insert operation on EXT4 in Galaxy S3 (For four journal modes in SQLite 3.7.5: DELETE, TRUNCATE, PERSIST, and WAL mode). Number at each block I/O denotes I/O size in KB.

tional `fsync()` call at the end of a transaction (Figure 4(c)). This is to synchronize the zero fill operation to the SQLite journal in the storage. PERSIST mode generates the largest number of I/O (twelve I/O operations) among the four SQLite journaling modes.

In write-ahead logging (WAL), SQLite logs insert or update operations in the log file (`.db-wal`) and calls `fsync()`. Then, EXT4 filesystem updates the file (`.db-wal` file) and commits the updated metadata to the EXT4 journal. Because there is only one `fsync()` call, the overhead of filesystem journaling is the least severe and the database operation becomes the most efficient in WAL mode among five journaling modes of SQLite. Figure 4(d) shows the I/O trace in WAL mode. Because SQLite must maintain a sequence of logs in the log file, WAL mode may consume more storage space.

With an extensive analysis of the Android platform, we observed that the EXT4 filesystem *journals the database journaling activity* via `fsync()` calls from SQLite. The bulky journaling mechanism (4 KB log record) of EXT4 very frequently commits the metadata of SQLite database (`.db`) and SQLite journal (`.db-journal`). As a result, EXT4 filesystem, when used by SQLite generates excessive amount of small writes and stresses the storage in a way that has rarely been observed before. The overhead of the filesystem journaling and database journaling compounds when the operations are used together. We call this phenomenon JOJ (journaling of journal). We also found that none of the SQLite journaling modes are free from JOJ phenomenon, but WAL mode puts the least stress on the filesystem.

The ideal and classic remedy for the JOJ phenomenon is to have SQLite directly manage the storage without filesystem's assistance or to have Android apps use the filesystem primitive to maintain their data instead of using SQLite. These approaches mandate overhauling the SQLite stack or asking numerous Android application developers to use the inconvenient filesystem primitive when writing software for Android.

5 Alternative Filesystems on Android

We analyzed the behavior of four most popular filesystems to observe behavior on the Android platform: BTRFS [24], NILFS2 [13], XFS [29], and the recently introduced[5] F2FS [1]. We ported these filesystems to the Galaxy S3 (running Android 4.0.4). We examined the block-level I/O behavior and the overall performance of these filesystems.

5.1 Details of Filesystem Behavior

BTRFS [24] uses B+ tree to maintain both the data and metadata and adopts copy-on-write to update its content.

[5] Oct 5, 2012

Despite the filesystem's promising features (e.g., file and subvolume snapshots, online defragmentation, and TRIM support for SSD [28]), these two properties, copy-on-write and B+ tree, make BTRFS the worst filesystem on the Android platform. BTRFS suffers from the wandering tree problem, where an update in a tree node triggers cascading updates to the root of the tree [5]. Figure 3(b) shows the I/O behavior when `fsync()` is called after a 4 KB write. With `fsync()`, BTRFS writes four B+ tree logs and synchronizes the superblock to the storage at the end. For a 4 KB write, BTRFS generates five additional write operations when `fsync()` is called.

NILFS2 [13] is a log-structured file system. It merges a set of data writes and all updated metadata into a segment and synchronizes the segment to the storage. The size of a segment is 128 KB in NILFS2. The `fsync()` operation in NILFS2 is implemented to flush the entire logical segment. Figure 3(c) illustrates the result of a 4 KB write followed by `fsync()`. Each `fsync()` generates a 128 KB write. Despite its log-structured nature, NILFS2 does not exploit its structural advantages because of its large segment size and inefficient segment flush mechanism.

XFS [29] is a journaling filesystem that was originally designed for massive-scale enterprise storage. It is expected to handle as many files in a directory as the storage can hold, with a maximum file size of 8 EByte (8×2^{60}). XFS uses the B+ tree-based directory structure and supports sparse file for scalability. Despite its original design objective of massive-scale systems, XFS exhibits very good (the second best) performance in `write` followed by `fsync()`. Figure 3(d) shows the block access pattern of XFS. The performance advantage of XFS arises from two sources: the number of journal writes and the size of each journal write. The `fsync()` operation triggers only one journal write, which is half the number of journal writes in EXT4. Furthermore, the size of a journal write in XFS is 1 KB, whereas it is at least 4 KB in EXT4.

Flash-Friendly Filesystem (F2FS) is the youngest filesystem among the five filesystems that we studied [1]. It is a log-structured filesystem specifically designed for flash storage. F2FS categorizes incoming write requests with similar characteristics together to mitigate the overhead of garbage collection in flash-based storage. Unlike the existing log-structured filesystems that collect a sequence of writes in a single large write for filesystem updates, F2FS can also update the storage in small units, e.g., 4 KB. This feature makes F2FS behave very efficiently in the corner-case workload, such as `write()` followed by `fsync()`. Figure 3(e) illustrates the I/O trace in F2FS. It has two writes: one for data and one for inodes. The size of a write is 4 KB, whereas another log-structured filesystem, NILFS2 generates a 128 KB I/O in the same situation (Figure 3(c)).

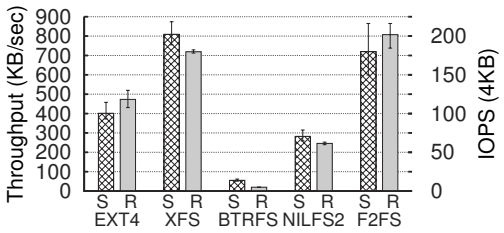


Figure 5: Sequential and random write using `fsync()` on 16GB Transcend SD card. S: Sequential (KB/sec), R: Random (IOPS), File size: 10MB, I/O size: 4 KB.

5.2 Summary: write() Followed by fsync()

We now compare the performance of five filesystems in a typical workload in Android platform: 4 KB write followed by `fsync()`. Figure 5 shows the results. XFS and F2FS yield the best performance among the five filesystems. F2FS yields the best random write performance while the edge goes to XFS in a sequential write. The key factor governing the performance of `fsync()` is the efficiency of the filesystem journaling, which we carefully studied in Section 5.1. In XFS, the size of a log record is 1 KB, and it generates one write per one journal commit. In EXT4, the size of a log record is 4 KB, and it generates at least two writes for each journal commit operation. For random writes, XFS and F2FS surpass EXT4 by approximately 50% and 70%, respectively. BTRFS exhibits the worst performance in both sequential and random writes. We will see in Section 6 that the performance of SQLite operations in each filesystem is precisely proportional to the performance of `write()` followed by `fsync()` demonstrated in Figure 5.

6 Optimization of the I/O Stack

In this section, we introduce optimization techniques to improve inefficiency caused by JOJ phenomenon, and examine the performance effect of individual techniques.

6.1 Eliminating Unnecessary Metadata Flushes

Our first effort of the optimization is to reduce the amount of metadata committed to a filesystem journal, which is caused by `fsync()` call in SQLite. The `fsync()` operation flushes both the metadata and data to storage. We found that `fdatasync()` operation is a good alternative to `fsync()` [2] because it does not flush metadata unless it is required to allow a subsequent data retrieval to be correctly treated. In Android platform, the filesystem is mounted with `noatime` option, and SQLite states that it only cares the files size, not the other attributes. Guaranteed that the underlying OS and filesystem support `fdatasync()` correctly, the use of `fdatasync()` does not affect the filesystem integrity.

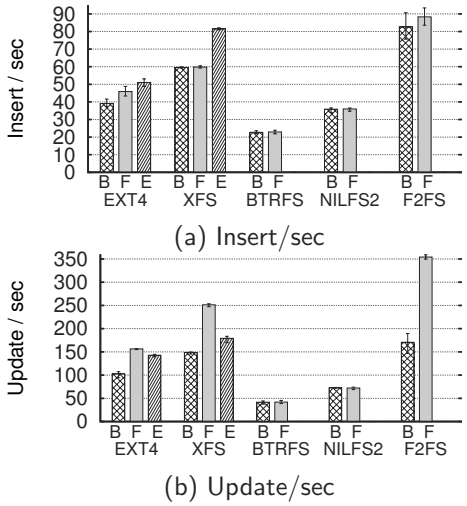


Figure 6: SQLite Insert and update/sec for 1,000 database items on 16GB Transcend SD card. B: baseline, F: `fdatasync()`, E: External Journal.

We examined the performance of SQLite operations (insert and update) using five filesystems after replacing `fsync()` with `fdatasync()`. Figure 6 displays the results. The B, F, and E labels on the X-axis denote the baseline (`fsync()` only), `fdatasync()` enhanced version, and filesystem with external journaling, respectively. Details regarding external journaling will be presented in Section 6.3.

By using `fdatasync()`, we achieved 17% performance improvement with EXT4 for an insert operation. For an insert operation, SQLite performs the best with F2FS. SQLite exhibits a 111% faster insert rate (inserts/sec) with F2FS than with EXT4. In BTRFS and NILFS2, the advantage of using `fdatasync()` is marginal. This is because in BTRFS and NILFS2, an insert operation causes an allocation of new blocks, in which the metadata are flushed even with `fdatasync()`. Figure 6(a) illustrates the result.

The advantage of using `fdatasync()` is more significant for an update operation than for an insert operation. Figure 6(b) illustrates the result. An update is an overwrite on the existing database record from filesystem's point of view. In EXT4 and XFS, update operation does not bring any changes on the metadata such as file size, indirect blocks, free block bitmaps and etc. Therefore, using `fdatasync()` in place of `fsync()` saves significant amount of metadata flushes. In EXT4 and XFS, update/sec increases by 50% and 66% when `fsync()` is replaced with `fdatasync()`, respectively. In contrast, for copy-on-write based filesystems, e.g., BTRFS and NILFS, `fdatasync()` yields little improvement because an update operation triggers the allocation of new blocks and subsequent metadata updates, which are flushed even when using `fdatasync()`. For an update operation, F2FS yields the best performance among the five filesystems. When we used F2FS with `fdatasync()`, the SQLite performance improved by 250% compared to the baseline platform (SQLite on EXT4 with `fsync()`).

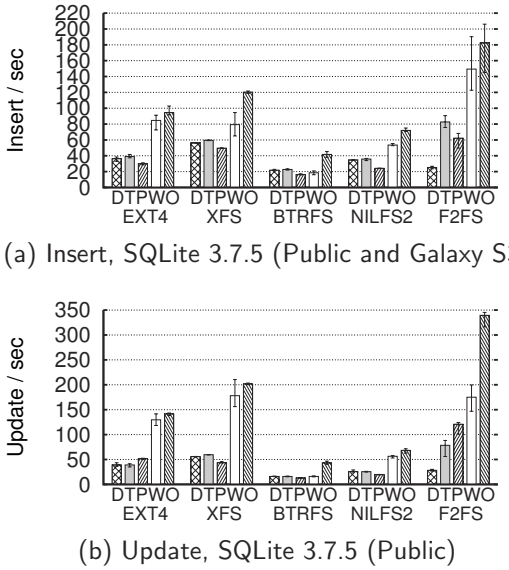


Figure 7: SQLite performance (with `fsync()`) under varying journal modes, 1,000 database items on 16GB Transcend SD card. D: DELETE, T: TRUNCATE, P: PERSIST, W: WAL, O: OFF

6.2 Using the Optimal Journaling Mode in SQL

The I/O performance is very sensitive to the journaling mode of the underlying DBMS. We tested five journaling modes (DELETE, TRUNCATE, PERSIST, WAL, and OFF) of SQLite on each of the five filesystems (EXT4, NILFS2, XFS, BTRFS, and F2FS) and measured the performance of SQLite (insert and update). Figure 7 shows the results. The performance of an insert operation decreased by more than 50% when we used one of DELETE, TRUNCATE or PERSIST compared to when we turned off the journal. In all filesystems, WAL mode yields the best insert/sec performance among four journaling modes (Figure 7(a)).

In an update operation, WAL mode yields three times better performance compared to the other journaling modes in all filesystems (Figure 7(b)). It should be noted that different from the publicly available SQLite, Galaxy S3 version of SQLite does not create any journal file in update operation. This yields significantly better performance, but the update operation can be unrecoverable[6].

For insert and update operations, F2FS is the best-performing filesystem for most of the journaling modes.

[6]We do not include the performance result due to space limit

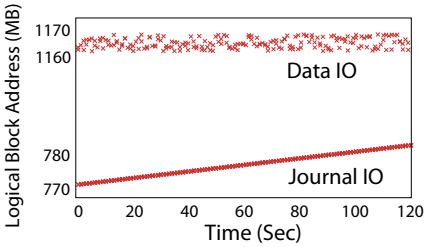


Figure 8: 4 KB random write followed by `fsync()`(EXT4)

When we replace EXT4 with F2FS, SQLite performance increases at least by 67%. This is because F2FS only generates two 4KB I/Os, one for data and the other for metadata, whereas EXT4 generates 3 to 12 random I/Os depending on the journal modes. For all filesystems, WAL mode yields the best performance because all the log data created from insert and update operations are appended to `.db-wal` file and there occurs only one `fsync()` call. BTRFS exhibits the worst performance for both insert and update operations because BTRFS induces more `write()` calls than any other filesystems due to wandering tree behavior.

In summary, the WAL mode is the optimal journaling mode for the Android platform from performance point of view. Despite its performance benefits, Write-Ahead-Logging has some issues, space requirement and recovery time. These need to be dealt with in the separate context.

6.3 External Journaling

EXT4 and XFS have an option of storing journal blocks on a separate block device. This option is called external journaling. We now show that external journaling can be a viable option to remove randomness in the aggregate traffic and to cluster correlated writes together to the same storage region such that the underlying NAND storage can easily exploit the locality in the traffic [16, 20].

In Figure 8, we plot the I/O traces from a 4 KB random write followed by `fsync()` in the EXT4 filesystem. The data file is in the 1160 to 1170 MB range, whereas the EXT4 journal blocks are in the 770 to 780 MB range. We can clearly see that the aggregate traffic consists of an interleaved mixture of two different I/O streams; the locality in the data region is random, whereas that in the journal region is sequential. Separating the data and journal I/Os appears to be an obvious next step, which allows the FTL of the underlying NAND-based storage to easily identify and to exploit the locality in the incoming I/O stream. The recent eMMC interface standard [3] allows physical partitioning of the internal storage. Thus, external journaling can indeed be a practical option for future smartphone storage.

We examined the effectiveness of external journaling in EXT4 and XFS. We used a 16 GB Transcend SD card and internal eMMC for the data storage and the external journal, respectively. The results are shown in Figure 6, where E stands for external journal. External journaling yields a significant performance improvement in EXT4; the insert rate is improved by 30%, and the update rate is improved by 39%. The improvement in XFS was not as great as that in EXT4 because the journaling overhead in XFS is not as significant as in EXT4.

Table 1: Throughput of 4 KB random write followed by `fsync()` on Internal eMMC with EXT4

	# of thread	Scenario	Idle		HD Record	
			base	poll	base	poll
eMMC	1	KIOPS	1002	981	667	756
		CPU (%)	7.5	10.9	26.4	30.2
	10	KIOPS	2609	2705	2136	2351
		CPU (%)	11.1	12.9	30.1	33.1

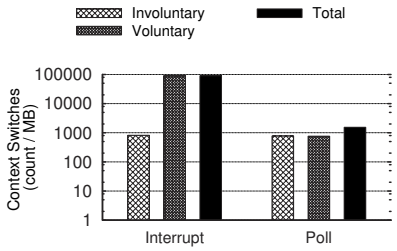


Figure 9: Number of context switches performed in interrupt-driven I/O (baseline) and polling-driven I/O

6.4 Polling-based I/O

Increasing number of CPU cores and decreasing I/O latency of a block device have led to a rediscovery of the value of polling-based I/O [32, 27]. State-of-the-art smartphones contain quad-core CPUs and NAND-based storage latency is an order of magnitude smaller than that of legacy hard disk drive. In this environment, interrupt-driven I/O may hinder the performance of a system due to context switches. When many small I/Os are generated from the block I/O layer, the I/O daemon for the eMMC, `mmcqd`, is subject to significant context switch overhead. Our results below show that this can indeed be the case and also show that polling-based I/O can provide a superior I/O performance to interrupt driven one without sacrificing the overall system performance.

We modify the I/O subsystem for the Android platform so that the `mmcqd` uses polling to access the storage device. There are two issues in polling-based I/O: CPU monopolization and power consumption. We perform an experiment if the polling based I/O interferes with the ongoing application, particularly CPU intensive one. We ran a HD-quality (1920x1080 at 30 fps) video recording application concurrently with our benchmark process. We found the soft real-time requirement of

video recording is well preserved even when the I/O subsystem is driven by polling. We perform another set of experiment to examine the power consumption behavior of polling driven I/O subsystem. Polling-based I/O may consume more CPU cycles and may reduce the opportunity for the CPU to stay in low-power mode. According to our experiment, CPU utilization increases by 4% when we use polling based I/O. In smartphone, dominant source of energy consumption in LCD and Wi-Fi [6, 33]. We carefully argue that energy overhead of polling based I/O is marginal and therefore polling based I/O is not an infeasible option.

Figure 9 shows the number of context switches made by the `mmcqd` daemon for the baseline and poll-driven I/O. We observed that the number of voluntary context switches is reduced to 1/100 and the total number of context switches is reduced to 1/50.

We examined the I/O performance under the polling-based I/O subsystem. We ran two experiments, one for single thread and the other for ten threads, where each thread in the experiment generates 4 KB random write followed by `fsync()`. We created ten threads to examine how polling-based I/O behaves when there are frequent TLB misses. Table 1 shows the results. In the single-thread case, the performance gain shows marginal gain of 1-2% when CPU is idle; the performance gain in the polling-based I/O is 13% when the smartphone is recording HD video in the background. When there were ten threads, the performance gain is slightly smaller, but it still shows 10.1% performance gain while recording HD video. As discussed in Yang et al. [32], the performance gain will be more significant with a faster storage medium.

6.5 Replay of Real Workload

As the final step to verify the effectiveness of the optimization, we examined the performance of each optimization technique under a real workload. We collected the system call trace and then replayed it with `Mobigen`. We captured traces from two widely used Android applications: Twitter and Facebook.

By replaying captured I/O traces of Twitter and Facebook, we extracted the duration of I/Os processed in the two applications (Figure 10). The results of this study exhibit similar characteristics to the results that we obtained from the SQLite performance and `write()` followed by `fsync()` performance. In both Twitter and Facebook execution, F2FS performed the best.

7 Combining All the Improvements

We examined the SQLite performance on three filesystems (EXT4 as the baseline, XFS, and F2FS) when applying the aforementioned three techniques[7] in combination. SQLite journaling mode was set to TRUNCATE (default).

[7]External Journaling is not applicable to F2FS since it is a log-structured filesystem

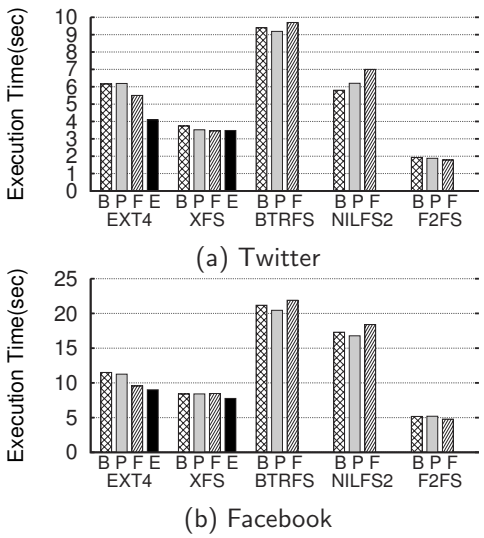


Figure 10: Compare execution-time of replaying script using Mobigen/Mobibench. B: Baseline, P: Polling, F: `fdatasync()`, E: External Journal

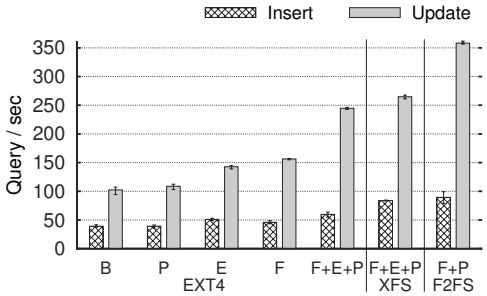


Figure 11: SQLite Performance for 1,000 database items. 16GB Transcend SD card. B: Baseline, P: Polling, F: `fdatasync()`, E: External Journal. TRUNCATE mode

Figure 11 illustrates the results. The baseline performance represents the current I/O performance: 39 inserts/sec and 102 updates/sec. Applying `fdatasync()`, external journaling, and polling-based I/O all together, SQLite on EXT4 showed 53% and 130% performance gains for insert and update operations, respectively. XFS and F2FS bring greater performance enhancements. F2FS with `fdatasync()` and polling-based I/O yields the best SQLite performance: the performance of the insert and update operations improved by 130% and 250%, respectively, compared to the baseline.

Finally, we combined all of the proposed techniques. We used WAL (write-ahead logging) SQLite journaling mode and examined the SQLite performance on three filesystems. Applying everything (`fdatasync()`, external journaling, polling-based I/O, and WAL SQLite

Table 2: Performance measurements of vertical Android I/O Stack. Measurement shows performance of SQLite insert/sec and update/sec on 16GB Transcend SD card.

Optimizations	EXT4		XFS		F2FS	
	Insert/sec	Update/sec	Insert/sec	Update/sec	Insert/sec	Update/sec
Baseline	39	102	60	149	83	171
fdatasync() (F)	46	156	60	251	88	354
External Journal (E)	51	143	82	179	-	-
Polling (P)	39	109	61	153	85	226
WAL mode (W)	76	100	75	153	149	155
F + E + P	60	**245**	84	**265**	89	**358**
F + E + P + W	**92**	113	**86**	188	**157**	175

journaling mode), we achieved a 150% performance improvement (from 39 inserts/sec to 92 inserts/sec) for SQLite on EXT4. When we used F2FS instead of EXT4 in the Android I/O stack, applying everything, we achieved a spectacular 300% performance improvement for SQLite (from 39 inserts/sec to 157 inserts/sec). Table 2 summarizes the results.

8 Related Work

Storage I/O characterization has been extensively studied in various computing environments. Ruemmler et al. [26] analyzed the disk I/O in three different HP-UX systems and demonstrated that a majority of the I/O operations are writes and that the majority of writes (67-78%) are for metadata, with user-data I/O representing only 3-41% of all accesses. Roselli et al. [25] reported that file accesses follow a bimodal distribution: some files are written repeatedly without being read, whereas other files are almost exclusively for reading. Zhou et al. [34] found that the read/write ratio in the filesystem is 80%/20% and that the majority of write I/Os are random. Harter et al. [8] studied the I/O behavior of the Mac OS filesystem and demonstrated that sequential I/O on a file rarely results in sequential I/O on a block device because of the complex XML-based document format. Prabhakaran et al. [22] provided a thorough analysis of journaling filesystems, such as EXT4, ReiserFS, JFS, and NTFS, and explained the events that cause data and metadata to be written to the journal. Piernas et al. [21] suggested separating the metadata from the data and demonstrated that this separation may improve the filesystem's performance.

There are a variety of interesting studies regarding smartphones, ranging from analyzing user behavior [7] to measuring power consumption [6], security [31, 30], and storage performance [11]. Kim et al. [11] demonstrated that the conventional wisdom that storage bandwidth is higher than network bandwidth must be reconsidered for smartphones. They demonstrated that storage performance does indeed affect the performance of application and operating system because the network bandwidth has increased significantly. Kim et al. [12] proposed a new buffer cache replacement scheme that provides a better sequential access in NAND storage devices. Lee et al. [14] analyzed the I/O behavior of eleven smartphone applications and found that the journaling efforts of SQLite and EXT4 compound with each other and result in excessive random write operations. To mitigate the overhead of random writes in NAND-based storage, Min et al. [19] proposed merging multiple random writes into a single write in the log-structured filesystem. This approach does not work on the Android platform where individual random writes are synchronized to the storage.

Yang et al. [32] demonstrated that in ultra-low latency devices using next-generation non-volatile memory, polling can deliver a higher performance than the traditional interrupt-driven I/O.

9 Conclusions

Modern OSes adopt a layered architecture that guarantees the independent operation of each layer; however, neglecting the underlying mechanisms produces a considerable amount of overhead related to the storage device. The well-designed SQLite and EXT4 components have unexpected effects on NAND-based storage devices when combined together because they produce many small, random, and synchronous write I/Os due to their misaligned interaction. We thoroughly analyzed the I/O stack (DBMS, filesystem, and block device driver) of Android. We examine the block level I/O behavior of SQLite operation under its five journal modes with five different filesystems in combinatorial manner. By removing frequent updates of the metadata, dislocating the EXT4 journal to separate storage, and using polling-based I/O, we have achieved a significant performance improvement in the insert and update rates. With the F2FS filesystem, WAL journaling mode (SQLite), and the combination of our improvements, we have observed an overall performance increase of 300% in SQLite performance.

10 Acknowledgements

We would like to thank our shepherd Steve Ko, and anonymous reviewers for insightful comments and sug-

gestions. This work is sponsored by IT R&D program MKE/KEIT. [No.10035202, Large Scale hyper-MLC SSD Technology Development], and by IT R&D program MKE/KEIT. [No. 10041608, Embedded system Software for New-memory based Smart Device].

References

[1] F2FS patch on LKML. `https://lkml.org/lkml/2012/10/5/205`.

[2] Linux programmer's manual for fdatasync. `http://www.kernel.org/doc/man-pages/online/pages/man2/fsync.2.html`.

[3] EMBEDDED MULTI-MEDIA CARD(e-MMC), ELECTRICAL STANDARD (4.5 Device), June 2011.

[4] ARLITT, M., AND WILLIAMSON, C. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. on Networking (ToN) 5*, 5 (1997), 631–645.

[5] BITYUTSKIY, A. Jffs3 design issues, Nov. 2005.

[6] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *Proc. of the USENIX Annual Technical Conference* (Boston, MA, US, June 2010).

[7] FALAKI, H., MAHAJAN, R., KANDULA, S., LYMBEROPOULOS, D., GOVINDAN, R., AND ESTRIN, D. Diversity in smartphone usage. In *Proc. of the 8th international conference on Mobile systems, applications, and services* (2010), ACM, pp. 179–194.

[8] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: understanding the I/O behavior of apple desktop applications. In *Proc. of SOSP* (2011), T. Wobber and P. Druschel, Eds., ACM, pp. 71–83.

[9] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Androstep: Android storage performance analysis tool. In *ME13: In Proc. of the First European Workshop on Mobile Engineering, Aachen, Germany* (Feb. 26 2013), vol. 215 of *Lecture Notes in Informatics*, pp. 327–340.

[10] KANT, K., AND WON, Y. Server capacity planning for web traffic workload. *IEEE Trans. on Knowledge and Data Engineering 11*, 5 (Sep 1999), 731–747.

[11] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proc. of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February, 2012*.

[12] KIM, H., RYU, M., AND RAMACHANDRAN, U. What is a good buffer cache replacement scheme for mobile flash storage? In *Proc. of the 12th ACM SIGMETRICS/PERFORMANCE, London, UK* (2012), ACM, pp. 235–246.

[13] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev. 40*, 3 (July 2006), 102–107.

[14] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *EMSOFT 2012: In Proc. of International Conference on Embedded Software, Tampere, Finland* (Oct. 7-12 2012).

[15] LEE, S., MOON, B., AND PARK, C. Advances in flash memory ssd technology for enterprise database applications. In *Proc. of the 35th SIGMOD international conference on Management of data, Providence, USA* (2009), ACM, pp. 863–870.

[16] LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. Last: Locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev. 42*, 6 (2008), 36–42.

[17] LEIMBACH, C. Dram share in tablets growing to the detriment of pcs. DRAM Dynamics, issue 23, Sep 2012.

[18] MEEKER, M. Kpcb internet trends year-end update. Kleiner Perkins Caufield & Byers, Dec 2012.

[19] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y.-I. SFS: Random write considered harmful in solid state drives. In *Proc. of the 10th USENIX conference on File and storage technologie* (San Jose, CA, USA, Feb. 2012).

[20] PARK, D., AND DU, D. Hot data identification for flash-based storage systems using multiple bloom filters. In *Proc. of Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on* (may 2011), pp. 1 –11.

[21] PIERNAS, J., CORTES, T., AND GARCIA, J. M. The design of new journaling file systems: The dualfs case. *IEEE Trans. on Computers 56*, 2 (2007), 267–281.

[22] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *Proc. of the USENIX Annual Technical Conference, General Track, Anaheim, CA, USA* (2005), pp. 105–120.

[23] RISKA, A., AND RIEDEL, E. Disk drive level workload characterization. In *Proc. of the USENIX Annual Technical Conference, General Track* (2006), USENIX, pp. 97–102.

[24] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *IBM Research Report* (July 2012).

[25] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proc. of the USENIX Annual Technical Conference* (Berkeley, CA, June 18–23 2000), pp. 41–54.

[26] RUEMMLER, C., AND WILKES, J. UNIX Disk Access Patterns. In *Proc. of Winter USENIX* (1993), pp. 405–20.

[27] SALAH, K., AND QAHTAN, A. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Computer Communications 32*, 1 (2009), 179–188.

[28] SHIN, D. About SSD. In *Proc. of the USENIX Linux Storage and Filesystem Workshop (LSF08), San Jose, CA* (2008).

[29] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability In The Xfs File System. In *Proc. of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), USENIX Association, pp. 1–1.

[30] VENNON, T. A study of known and potential malware threats. Tech. rep., SMobile Global Threat Center, Feb 2010.

[31] VIDAS, T., VOTIPKA, D., AND CHRISTIN, N. All your droid are belong to us: A survey of current android attacks. In *Proc. of the 5th USENIX conference on Offensive technologies, San Francisco, CA* (2011), USENIX Association, pp. 10–10.

[32] YANG, J., MINTURN, D., AND HADY, F. When poll is better than interrupt. In *Proc. of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February, 2012*.

[33] YOON, C., KIM, D., JUNG, W., KANG, C., AND CHA, H. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proc. of the USENIX Annual Technical Conference* (Boston, MA, US, June 2012).

[34] ZHOU, M., AND SMITH, A. J. Analysis of personal computer workloads. In *Proc. of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS* (1999), pp. 208 –217.

How to Run POSIX Apps in a Minimal Picoprocess

Jon Howell, Bryan Parno, John R. Douceur
Microsoft Research, Redmond, WA

Abstract

We envision a future where Web, mobile, and desktop applications are delivered as isolated, complete software stacks to a minimal, secure client host. This shift imbues app vendors with full autonomy to maintain their apps' integrity. Achieving this goal requires shifting complex behavior out of the client platform and into the vendors' isolated apps. We ported rich, interactive POSIX apps, such as Gimp and Inkscape, to a spartan host platform. We describe this effort in sufficient detail to support reproducibility.

1 Introduction

Numerous academic systems [5, 11, 13, 15, 19, 22, 25–28, 31] and deployed systems [1–3, 23] have started pushing towards a world in which Web, mobile, and desktop applications are strongly isolated by the client kernel. A common theme in this work is that guaranteeing strong isolation requires simplifying the client, since complexity tends to breed vulnerability.

Complexity evicted from the client kernel takes up residence in the apps themselves. This shift is beneficial: It lets each app vendor decide independently which complexity is worth the risk of vulnerability, and one vendor's decision in favor of complexity does not undermine another's decision to favor security. Of course, requiring each app vendor to implement a complete software stack is impractical, so we expect this complexity to migrate to app frameworks that app vendors can choose among, just as web developers choose among an ever evolving set of app frameworks on the server.

The minimality of the client interface must not inhibit the richness required by applications such as desktop productivity apps. New client application models often fail due to the burden of migrating every app–and every library–to run under a new model. Thus, we argue that shifting app delivery to a minimal-client model requires an inexpensive app migration path from complex-host frameworks such as POSIX and Windows.

On the other hand, support for richness should not sacrifice the small size and tight specification of the isolation interface. The web's current client execution interface has repeatedly failed to achieve strong app isolation, due to an interface bloated with HTML, DOM, JPG, PNG, JavaScript, Flash, etc. in pursuit of richness.

The recent Embassies system provides a concrete example of how to achieve both security and richness simultaneously [16]. It pushes the minimal client host interface to an extreme, proposing a client host without TCP, a file system or even storage, and with a UI constrained to simple pixel blitting (i.e., copying pixel arrays to the screen). In support of rich apps, Embassies's minimal interface specifies execution of native binary code. Native code is an important practical choice, because, we assert, it is the lack of native code that has forced each prior system based on language safety to evolve a complex trusted interface that provides access to native libraries [8, 10, 17, 20]. This complexity undermines the intent to provide strong security.

		Libraries	
Application	**Function**	**#**	**Examples**
Abiword	word processor	63	Pango,Freetype
Gimp	raster graphics	55	Gtk,Gdk
Gnucash	personal finances	101	Gnome,Enchant
Gnumeric	spreadsheet	54	Gtk,Gdk
Hyperoid	video game	6	svgalib
Inkscape	vector drawing	96	Magick,Gnome
Marble	3D globe	73	KDE, Qt
Midori	HTML/JS renderer	74	webkit

Table 1: *A variety of rich, functional apps transplanted to run in a minimal native picoprocess. While these apps are nearly fully functional, plugins that depend on* `fork()` *are not yet supported (§3.9).*

While native code is a target that every compiler can hit, it seems daunting to port arbitrary POSIX apps to such a minimal interface. Such apps expect to run on a complex host with hundreds of system calls and dozens of system services, reflecting decades of development.

However, our experience suggests this task is far easier than one might expect. Interactive apps use relatively little of the complexity available in modern host platforms. More importantly, rather than alter the app, the functions that are required can often be emulated behind the POSIX interface. This technique works without even recompiling the hundreds of libraries involved. The emulation work can be shared easily across many applications, making the porting work scalable. The broad selection of rich apps that our system supports (see Table 1) demonstrates the generality of the approach.

Contributions. This paper demonstrates the tractability of porting rich POSIX apps to a minimal environment, thus enabling them to run on a multitude of minimal client hosts [13, 16, 18, 22, 31]. We give a full account-

ing of the porting task, including which functionality is required and where corners can be cut. This includes low-level details, such as an exhaustive list of syscalls handled, to enable reproducibility and to eliminate any ambiguity about complexity hidden under the hood. Ultimately, we hope that this will expedite other efforts to adopt these techniques and hence achieve rich applications atop minimal, strongly-isolating client kernels.

2 Background: Minimal Client Facilities

In this work, we aim to transplant apps from a rich POSIX interface to a minimal client kernel. To ground the discussion, we target the minimal Embassies *picoprocess* interface [16], since it takes minimality to an extreme. If we can port an app to Embassies, we can certainly port it to a client with a richer interface.

The Embassies application binary interface (ABI) provides *execution* primitives that support an app's internal computation, *cryptographic* primitives to facilitate privacy and integrity, primitives for IPC and network *communication*, and *user interface* (UI) primitives for user interaction.

Execution. The execution primitives include:

- Calls to `allocate_memory` and `free_memory`. To simplify the specification and to make the ABI portable to most host environments, the app specifies only the amount of memory required; it has no control over the addresses returned by the allocator.
- `create_thread` accepts only the thread's initial program counter and stack pointer; the application provides the stack and any execution context.
- `exit_thread` destroys the current thread.
- A simplified futex-like [6] synchronization scheduling primitive, the *zutex*. `zutex_wake` is a race-free scheduling primitive that supports app-level efficient synchronization primitives. The corresponding `zutex_wait` is the only blocking call in the ABI; it allows an app to yield the processor.
- `clock` returns a rough notion of wall-clock time.
- `set_timer` sets a timer, in clock coordinates, that wakes a zutex on its expiration. Each picoprocess has only one timer; the app must multiplex it.
- `get_alarms` returns a list of three distinguished zutexes representing external events, one for each of `receive_packet`, `ui_event`, and `timer_expired`. Waiting on these zutexes is how threads block on external activity.
- A call to create a new picoprocess.

Cryptographic Infrastructure.

- `random` provides a supply of cryptographically strong entropy.
- `app_key` provides a machine-specific, application-specific secret. Apps use this key, along with cryptographic libraries, to store and recover private information despite starting from a public binary.

Communication. All communication outside the process, whether IPC to another process on the local machine, or remote to an Internet host, follows IP semantics: Data is transferred by value (a logical copy), so that the suspicious recipient needn't worry about concurrent modification; addressing is non-authoritative; delivery admits loss and duplication; packet privacy and integrity are not guaranteed. Just like servers on the Internet, apps build up integrity and privacy themselves using cryptography. To underscore these semantics, all communication in Embassies–remote or local–is done via IP.

- `get_addresses` assigns the process one IPv4 and one IPv6 address.
- `allocate_packet` allocates memory for an outgoing packet; this allocation is distinguished from `allocate_memory` to enable zero-copy transfer.
- `send_packet` delivers a packet, interpreting its argument as an IP header and payload.
- `receive_packet` returns an allocated and dequeued packet, or NULL if the queue is empty.
- `free_packet` frees an allocated packet.

User Interface.

- `ui_event` returns a dequeued UI event (keystroke or pointer motion), or NULL if the queue is empty.
- Some calls that manage viewports, letting them be transferred among applications, or letting one application sublet a region of its viewport to another application. In every case, even where nested, each viewport is owned by a single app; no app can inspect or modify the pixels of another app's viewport. Details can be found elsewhere [16].
- `map_canvas` allocates a framebuffer to back a viewport. This allocation is distinguished from `allocate_memory` to enable fast pixel blitting.
- `update_canvas` informs the client kernel that a region of the framebuffer has been updated, and that its pixels should be blitted to the display.

These calls comprise the entire Embassies ABI; all of the functionality described in the rest of the paper is implemented in terms of these primitives.

3 The POSIX Emulator

A conventional POSIX application employs dozens of libraries, access to a rich system call interface, and by way of those system calls, access to other rich services, such as the `X` server's graphics functions and the `dbus` desktop configuration object broker.

To execute applications expecting this rich POSIX environment, our POSIX emulator cleverly repurposes existing libraries and programs atop the execution environ-

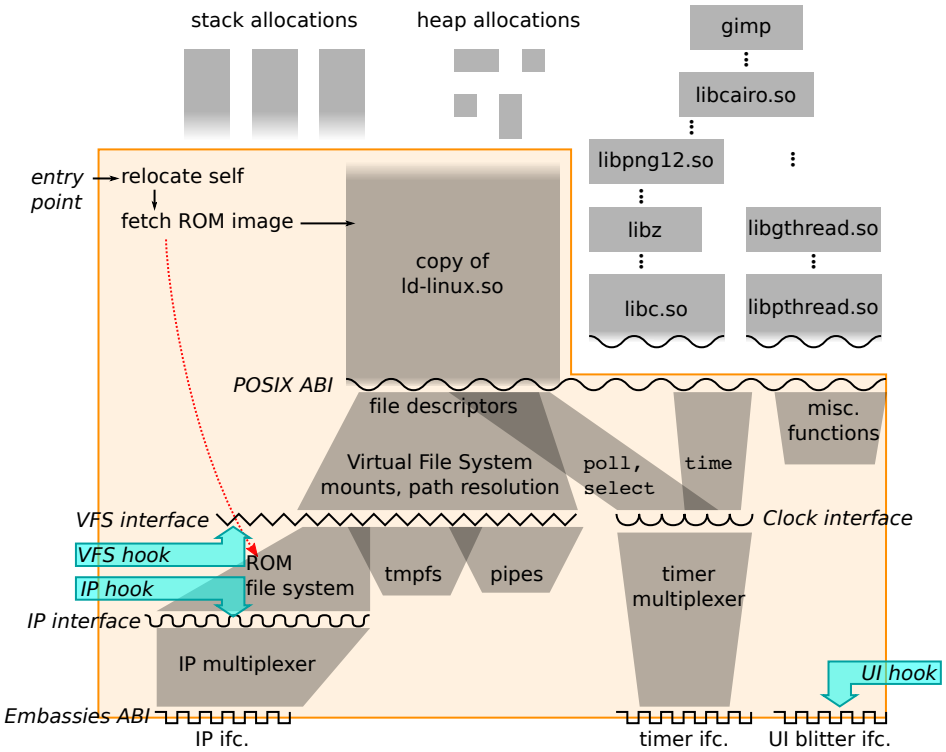


Figure 1: **The POSIX Emulator.** *To Embassies, the emulator (the large, L-shaped boundary) is a binary string whose entry point is its first byte, and which may call back into a set of low-level interfaces provided by the Embassies ABI. Internally, the emulator loads the app's read-only image, maps it into a virtual filesystem, and calls into a copy of* `ld-linux.so`*. That loader, using the emulated POSIX ABI, reads the app executable and additional ELF libraries into memory. The* `glibc` *libraries' syscalls are redirected to the emulator's POSIX interface. Non-POSIX hooks provide connections for UI and TCP services implemented outside of the emulator (Figure 2).*

ment's minimal services (§2). Figure 1 gives a structural overview of how the emulator maps the entire POSIX interface down to Embassies's picoprocess interface.

Below, we provide a functional exposition of this emulation, starting with application launch.

3.1 Application Launch

Embassies provides minimal support for app launch, merely loading and starting a vendor-specified *boot block* of code. Specifically, the host (1) maps the applications' boot block into an arbitrary region of address space, (2) sets up a minimal stack, and (3) places in a register the address of a dispatch table for the Embassies ABI (§2).

Within its boot block, the POSIX emulator (1) relocates its symbols, using a small piece of position-independent code, (2) allocates an adequate stack, and (3) establishes a dispatch function to emulate the POSIX syscall interface (§3.2) and virtual file system (§3.3).

Next, the emulator must load the app and its libraries into memory. In a full Linux implementation, the kernel would interpret the app's ELF binary format, map the app binary into memory, map the loader `ld-linux.so` into memory, and then jump to the loader. The loader would then enumerate dynamic library references within the ELF image, map these libraries into memory, link the images together (resolving symbolic references), and then jump to the app's entry point.

Embassies, however, provides neither a file system from which to map files nor a kernel willing to parse ELF binaries. Thus, our emulator must perform these tasks, which it does by invoking `ld-linux.so`, an image of which is included in the emulator's boot block. The emulator calls `ld-linux.so` and passes the app's path as an argument, which instructs the loader to map the app (and its libraries) into memory. POSIX calls made by `ld-linux.so` are serviced by the emulator (§3.2).

To call the loader, the emulator creates a suitable `argv` (naming the ELF executable), an `envp` (e.g. pointing `DISPLAY` at 127.0.0.1:6), and an `auxv` (some constants to convince libraries they're running on Linux).

3.2 Intercepting System Calls

The loader, as well as other libraries in the glibc suite, are at the bottom of the library stack; these are the libraries that make actual POSIX syscalls. In principle, other libraries could also include direct syscall instructions, but in practice, we have never observed this; instead, they simply use `libc`'s `syscall` symbol.

We want to exploit the functionality of the glibc suite, but glibc's system calls will fail in an Embassies process; they must be intercepted and replaced with calls to the syscall emulation layer. In principle this can be achieved by creating an alternate "sysdep" personality for glibc. In practice, at least for the x86 architecture, we found it easiest to apply a binary rewriting pass to each of the libraries in the glibc suite, patching every system call invocation (i.e., each occurrence of `int $0x80`) with a call to a dispatch function that we inserted at the end of the library.

The dispatch function in each library must, in turn, be patched dynamically to call into the emulator's syscall dispatcher. To identify libraries in need of such dynamic patching, we modified the libraries' ELF headers to label the dispatch function. As libraries are `mmap`ed into the app's address space, a filter file system in the VFS layer (§3.3) detects the modified ELF signature and transparently updates the dispatch function to point at the emulator's syscall dispatcher.

3.3 Virtual File System

Much of the POSIX ABI concerns file naming and file descriptors, which provide access to a variety of functions. Thus, like a Unix POSIX implementation, the emulator contains a virtual file system (VFS) abstraction.

VFS components include a read-only app image, a RAM-based writable temporary filesystem (`tmpfs`) that implements POSIX scratch directories like `/tmp`, and named pipes (Unix-domain `sockets`). The writable `tmpfs` directories provide the namespace for the Unix-domain sockets. There are also the virtual files that emulate POSIX special files. These comprise the `/proc` files of Section 3.8.1 and an emulated `/dev/random` which passes entropy up from the client kernel's `random` facility.

The emulated VFS contains an overlay mount table to weave these file systems together.

3.3.1 The Read-Only Application Image

The most important VFS component is the read-only binary image, whence libraries and data files are fetched.

A Linux app expects to fetch its libraries and read-only data files by name from a (shared) file system via `read` and `mmap`. In Embassies, such files come from a private app image whose integrity has been verified.

To support this, the developer packages every file the app requires into a single tar-style image file. The emulator fetches this file from an untrusted cache on the local machine, delegating to the cache the complexity of fetching the image from an upstream cache or origin server and exploiting commonality with other apps [14]. The reply appears in memory as a single (jumbo) IP packet. The emulator ensures integrity by comparing the image's hash to a fixed hash value embedded in the boot block.

`accept`	`recvfrom`
`bind`	`recvmsg`
`connect`	`send`
`getpeername`	`sendmsg`
`getsockname`	`sendto`
`getsockopt`	`setsockopt`
`listen`	`shutdown`
`recv`	`socket`

Table 2: **Socket Calls.** These calls are plumbed through the VFS interface to either the Unix named pipes implementation or the TCP stack.

The image file transmission protocol supports partial fetches, so that the app can start with only a subset of the image, and then later page in additional components.

3.3.2 Supported Interfaces

POSIX defines a wide, complex interface for interacting with the file system, so implementing the entire interface would be quite labor intensive. Fortunately, to support the varied applications from Table 1, it suffices for the VFS to support the following functions.

First, there is the core interface `open`, `close`, `ftruncate`, and `ftruncate64`; and the metadata interface `stat`, `lstat`, `fstat`, and `access`. VFS file descriptors track file pointers for `read`, `write`, `writev`, and `lseek`. Directory functions `mkdir`, `getdents`, `getdents64`, (hard) `link`, and `unlink` are only implemented in the `tmpfs`. The socket calls (Table 2) are routed through the VFS to the Unix pipe and TCP (§4.2) implementations.

The emulator also implements file handle functions `dup`, `dup2`, `pipe`, and `pipe2`. `pipe` connects two file descriptors with a blocking pipe with no presence in the VFS namespace. Functions `fsync` and `fdatasync` are no-ops. Most of `fcntl` and `fcntl64` are no-ops, except `F_DUPFD`, which calls the `dup` implementation.

3.4 Mmap Support

POSIX `mmap` is versatile, but in practice it is used in only a few idiomatic ways.

First, `mmap(MAP_ANONYMOUS)` is used to allocate blank memory at an address chosen by the kernel. The emulator transforms these calls into Embassies memory allocations.

Second, apps use `mmap` explicitly to map in non-executable data files. These calls also give the emulator freedom to choose the target address, so the emulator allocates fresh memory and uses a `memcpy`ing `read` implementation to simulate the effect of the `mmap`.

Finally, apps use `mmap` implicitly when they dynamically link executable libraries, either at load time via `ld-linux.so` or at runtime via `dlopen`. Some of these calls *do* expect to control the resulting data placement, a degree of control that Embassies does not provide when allocating memory.

Fortunately, the loader does not *really* care where a given library ends up; it just requires that the data segment of the library appears at the correct offset from the text segment. To this end, the loader's first `mmap` call does not specify a target address; instead, it specifies a length sufficient to reserve enough address space to cover all the segments in the file. The loader's subsequent `mmap` calls (e.g., for the data segment) do specify a target address, but the target address is always within the memory range allocated by the initial `mmap` call.

Thus, the emulator can support this final class of `mmap` calls by simply using the Embassies interface to allocate the initial memory region (which does not specify a particular address), and then confirming that subsequent `mmap`s (that do specify an address) fall within the initial memory allocation. As long as they do, the emulator can take the appropriate action, e.g., it can zero-fill the specified region for the binary's `.bss` section or copy in the contents of the `mmap`ed file.

This approach is clearly "less portable", in the sense that a POSIX app could in theory call `mmap` with an address outside of any preexisting allocation. Fortunately, we have not yet encountered any applications that rely on this functionality.

3.4.1 Fast `mmap`

The approach above is adequate for correct POSIX emulation, but for the apps we tested, where the bulk of the image comprises `mmap`-loaded libraries, it incurs many megabytes of `memcpy`s, adding noticeable delay (150 ms) to the app start time. We corrected this performance problem by page-aligning `mmap`able libraries in the image tar file (§3.3.1), and servicing `mmap` requests by yielding the memory region from the VFS to the app.

Of course, this means that the region can not be `read` or `mmap`ed later in the program's execution; if a program needs to map a file multiple times, we either store multiple copies in the image file (often worth the space), or mark the region "precious", inhibiting the optimization.

Fast-`mmap` files must be stored in the image in their in-memory layout, not their on-disk ELF layout, including necessary blank space to position the data and bss segments. The blank spaces are, of course, easy to compress during transmission.

3.4.2 Other Memory Calls

Most POSIX memory allocations appear as anonymous `mmap` calls. The emulator tracks such requested regions, freeing the underlying Embassies allocation once the entire region has been `munmap`ped.

Embassies provides no read/write/execute memory protections, so the emulator simply ignores `mprotect`, `madvise`, and `msync`. It also rejects `mremap`.

Unfortunately, `ld-linux.so` and `libc` both make initial memory allocations with the ancient `brk` interface. Why? We cannot say; the best solution would be to eradicate these deprecated calls. Instead, as a workaround, the emulator assumes that virtual memory has no cost, generously over-allocates on the initial `brk(0)` call, and services each subsequent `brk` extension by releasing more of the initial allocation.

3.5 Clock and Timers

The emulator provides the various flavors of POSIX time: `time`, `gettimeofday`, and `clock_gettime`. It translates all of these from the nanosecond precision clock supplied by the client kernel. That clock provides rate but no offset information; hence all of our apps think the current time is 2011. We use ntpdate to acquire a clock offset, although we have not yet attended to the security implications.

Embassies supplies the process with a single timer, which signals the process by firing a zutex, and thus can be reset in a race-free way. The emulator has the responsibility to multiplex this one timer into as many alarms as it needs to implement POSIX timeout interfaces. It does so using a tree of upcoming deadlines, for scalability. We found the clock multiplexer to be surprisingly subtle, with many race conditions that lead to deadlocks. It was helpful to diagram the detailed mapping between the host timer state and the state of the guest timer list.

3.6 Synchronization Primitives

The Embassies client kernel provides a single unified synchronization abstraction, the `zutex`, that is used both for internal waiting on other threads and waiting on external events (the network or the clock). This central abstraction is a simplified `futex` [6]. Like the futex, the zutex is actually a race-free *scheduling* primitive in support of efficient synchronization.

The basic POSIX `futex` maps readily onto the zutex, with the emulator folding in timeout behavior (§3.5). Many extra POSIX behaviors are neutered. For example highly concurrent servers use FUTEX_CMP_REQUEUE to avoid convoys, but our emulator simply wakes the requested threads and lets them requeue themselves. The emulator rejects FUTEX_WAKE_OP and FUTEX_WAIT_BITSET with an error, alerting `libpthread` to revert to the basic behavior.

The `nanosleep` call and POSIX multiple-wait primitives `select`, `newselect`, and `poll` are all mapped into `zutex_wait` operations, again with timeout behavior constructed by the emulator. POSIX blocking operations, like a `read` on an empty pipe, wait on zutex-signaled events.

3.7 Network Multiplexing

Embassies provides each process with a single `zutex` to signal the arrival of IP traffic. Thus, the emulator must collect incoming IP packets and multiplex them

inside the app. The emulator itself uses IP to fetch its image (§3.3) and for querying time servers (§3.5). The emulator's network stack demultiplexes IP and UDP, and delivers TCP packets to the LWIP library (§4.2).

3.8 Threads

POSIX uses `clone` to express both thread creation and process fork (§3.9). The emulator pattern-matches the thread-creation idiom and sets up the new thread's initial thread-local store (TLS). Because Embassies' `create_thread` conveys only a stack pointer, the emulator constructs a stub stack to pass the POSIX parameters and the caller's designated stack to the new thread. It records metadata about the new thread to correctly implement `CLONE_CHILD_CLEARTID` upon POSIX's thread-exit call.

The POSIX process-exit call, `exit_group`, signals the zone host (§4) that a zone has exited.

3.8.1 Supplying the Stack Address

Several applications rely on garbage collection libraries that need to know the address of the top (but not the bottom) of the current thread's stack. This is exposed in Linux POSIX through pseudofiles in `/proc`.

At first blush, it appears that the stack bottom address is also needed. For example, Libwebkit's JavaScriptCore garbage collector queries `libpthreads` for the stack bottom address. However, the GC does not use the bottom address directly; instead, it adds `RLIMIT_STACK` to yield the top address. Since `libpthreads` determines the bottom address by subtracting `RLIMIT_STACK` from the top address it obtains from `/proc/self/maps`, any sane value for `RLIMIT_STACK` will work correctly. We used 8 MB.

The stack top value returned by `/proc/self/maps`, on the other hand, does matter: It is how a conservative garbage collector learns the extent of the stack. Another garbage collector, `libgc`, looks for the stack top in `/proc/stat/self`. We install special VFS nodes at those names which return the appropriate stack top value for the current thread.

To identify which thread is querying the interface, the emulator snoops the app's thread-local store (TLS) register; that is, it uses grey-box assumptions about how glibc manages the TLS. For all threads other than the main thread, the emulator records each stack address as its thread is created by the `clone` syscall. For the main thread, the emulator allocated the stack (§3.1) and thus knows its address.

3.9 Unimplemented: Fork

Some apps employ the `fork/exec` pattern; e.g., Inkscape uses it for its plug-in modules. This pattern does not translate well to the minimal Embassies environment, since Embassies's memory management facilities are far too simple. The current emulator implementation does not support `fork` at all, leaving Inkscape's plug-ins inoperative.

An expedient approach, if the code is sufficiently idiomatic, is to emulate the fork with a thread, and perhaps intercept and neuter `close` calls from the child "process" preparing to `exec`. The `exec` call would launch a new zone (§4), or if fault-containment is desired, a new picoprocess (§2).

Alternatively, since the `fork/exec` pattern is usually implemented in a widely-used library, such as glib's `g_spawn`, one could modify this higher-level library to map fork's semantics cleanly onto the creation of a new zone or picoprocess.

`chmod`	`sched_setparam`
`chown`	`sigaction`
`fchmod`	`sigprocmask`
`rename`	`umask`
`sched_get_priority_max`	`sched_setscheduler`
`sched_get_priority_min`	

Table 3: **Failure-oblivious calls** return either EINVAL or ENOSYS, which the caller handles gracefully.

`fchown`	`set_tid_address`
`flock`	`setitimer`
`fstatfs`	`setpriority`
`inotify_init`	`setrlimit`
`inotify_init1`	`shmget`
`ioctl`	`statfs`
`ipc`	`sysfs`
`readlink`	`times`
`sched_getaffinity`	`xi_sched_yield`
`set_robust_list`	`xi_timer_create`
`xi_sched_rr_get_interval`	

Table 4: **Neutered calls** simply return 0 (success).

3.10 Neutered System Calls

The remaining syscalls are either unused by interactive apps, or can be simply rejected or neutered. This section identifies such syscalls in the interest of completeness.

Many calls (Table 3) can be rejected, returning `ENOSYS` or `EINVAL`, and the libraries that call them either handle the failure gracefully, fall back to an alternate POSIX mechanism, or ignore the result and trundle along obliviously [24].

Other syscalls can be neutered with brazen lies: When the caller actually checks the return code, we may need to return 0 ("success") even if we don't actually emulate the promised semantics (Table 4). Other functions require slightly more credible lies: The emulator fills in some plausible constant values to placate the caller (Table 5). For instance, the `clock_getres` call should provide some information about clock quality (§3.5), but we just claim a 500 ms resolution. As another example, we found no software that used `chdir`, so `getcwd` simply returns "/".

clock_getres	getpid
getcwd	getppid
getegid	getresgid32
getegid32	getresuid32
geteuid	getrusage
geteuid32	getuid
getgid	getuid32
getgid32	sched_getparam
getpgrp	uname
sched_getscheduler	

Table 5: **Deluded calls** return slightly fancier lies than 0.

3.11 Additional Program Requirements

Emulating the POSIX ABI is minimally intrusive to the apps, but a few conflicts remain.

3.11.1 Address Freedom

We have already seen that Embassies's refusal to let apps specify specific locations for allocated memory requires the boot block to relocate itself (§3.1) and requires a delicate hand in servicing `mmap` (§3.4).

It also means that every executable must be relocatable or position independent. Every Linux shared library is relocatable, but for no discernible reason, executables are not relocatable by default. We address this by rebuilding each app's top-level executable with the `-pie` ("position-independent executable") compiler flag. Although this requires tampering with the app's build system (§6.1), it is required only for the top-level application, not any libraries; and in most cases, passing `DEB_CFLAGS=-pie` to `dpkg-buildpackage` does the job. The change is nowhere near as invasive as trying to change to static linkage (§6.1).

3.11.2 The TLS Register

On the architecture we experimented on, the arcane x86-32 instruction set architecture, a paucity of general-purpose registers leads POSIX compilers to employ a disused segment register `%gs` as a thread-local storage (TLS) pointer. This usage gets compiled into every library and application binary. Since the idiom has no security-sensitive semantics, we opted to provide a `store-gs` call in the x86-32 Embassies ABI; the emulator uses it to implement `set_thread_area`.

A better solution would be to either recompile or binary rewrite every binary to eliminate `%gs` references.

4 Zones: Programs as Libraries

Besides kernel services, POSIX apps often expect access to higher-level services provided by daemon programs like `X` windows, a window manager (e.g., `twm`), or a configuration manager, like the `dbus` desktop bus. We satisfy such apps by including these services *inside* the apps that need them, rather than in the client kernel's TCB (which would add them to every app's TCB).

`X`, `twm`, and `dbus` are designed as independent

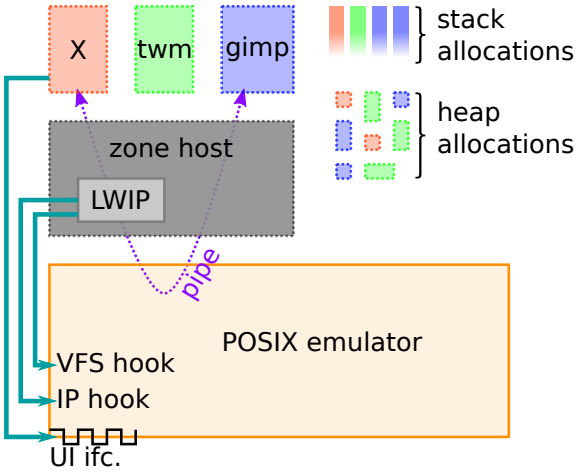


Figure 2: **Multiple POSIX apps coexist in one picoprocess as *zones*.** *Each zone comprises a noncontiguous partition of the address space. Each has its own copies of libraries, like `libc`, and its own stack and heap allocations. Programs that expect POSIX pipe IPC, such as an `X` session, see the same behavior within the picoprocess.*

POSIX processes. Rather than convert them into libraries, we found it expedient to create a general mechanism for loading multiple programs into a single picoprocess. This is easier than it sounds, because each program separately allocates memory and file descriptors, which carves the resource namespaces into interleaving partitions. We call such partitions "zones" (Figure 2).

Embassies's refusal to allow memory allocations at specific addresses works to our advantage when implementing zones, since it precludes zones from demanding overlapping allocations. It is zones that use the emulated Unix pipes (§3.3). For example, the `X` zone listens on `/tmp/.X11unix/X0`, and the xlib client library in the main application zone binds to it there.

The vestigial `brk` interface (§3.4.2), however, presents a hurdle. Two threads in different zones may concurrently extend different `brk` heaps. The `brk` interface assumes hidden per-process state, which becomes per-zone state. The good news is that we can infer which zone is making the request, and hence which per-zone state to consult, because each request should appear within the address space set aside for that zone's brk.

The bad news is that, on 32-bit hardware, virtual address space is scarce enough to warrant preserving, which means allocating only appropriately-sized `brk` regions for each zone. This is tricky because the initial call from each zone is a stateless `brk(0)`, from which the emulator cannot infer the identity of the calling zone. Our expedient solution forces the zones to start up sequentially. A more elegant solution would identify the calling zone by its TLS or stack pointer, or (better yet) eliminate `brk` calls from libc.

4.1 Example Zones

The X server presents a complex security boundary, and this complexity conflicts with Embassies's goal of a minimal client kernel TCB. Therefore we use X only inside the picoprocess, in a zone, disregarding its security-sensitive multiplexing functions and exploiting only its rasterization function. The rendered frame buffer that X produces is blitted to the user's display through Embassies's pixel-level UI interface.

Some apps, like Gimp, use a plethora of palette windows. For expediency, we add a twm window manager zone into such apps, to allow manipulation of the palettes within the surface of the app's single display region. With more effort, one could coordinate multiple windows via Embassies's window management, perhaps using a technique like Nitpicker's [9].

Gnome desktop apps expect to connect to the dbus daemon to find other components and learn configuration settings. This tight coupling among applications has no cost in a trusted-everything system, but is too risky for mutually untrusting apps. Hence we do not reproduce the connected dbus; instead, we link a copy of the daemon into each app to expediently satisfy the client library. With more effort, one could strip the dbus dependencies out of each app.

4.2 Extension Hooks

The emulator sits below libc, and hence cannot exploit libc. Coding without libc is painful; thus where possible, we push functionality out of the emulator into layers above. To facilitate this modularity, the emulator exports four hooks via unused syscall numbers.

Specifically, as alluded to above, we use an X zone to translate app UIs into easily blitted pixel regions. A modified X server within the zone supplies the graphical user interface. It uses one extension hook, ex_get_dispatch_table, to gain access to the raw Embassies UI functions. It uses a second extension hook, ex_open_zutex_as_fd, to wrap the UI notification zutex in a POSIX file descriptor, enabling the extension to smoothly integrate into X's existing poll loop.

All unhandled IP traffic, including TCP traffic, is handed off to a TCP stack based on lwIP [7] that resides in the zone host. The lwIP stack is a loadable module, attaching to the emulator's IP multiplexer via ex_add_default_handler and servicing requests for SOCK_STREAM sockets via ex_mount_vfs.

5 Debugging Strategies

The key premise of this work is that most apps use only a fraction of POSIX functionality. This paper catalogs these functions in detail precisely because the challenge is in discovering *which* functions matter.

Most of the effort in emulating the right subset of POSIX involves figuring out why a segfault occurred in a library dozens of layers below the app. To assist the practitioner who wishes to extend this approach, this section identifies our most valuable debugging strategies.

It is important to plumb error messages out of the picoprocess. Our insecure debug-mode Embassies monitor offers an extended ABI with debug channels that record to files. The emulator routes stdout and stderr to them.

Since most of our changes occur behind the POSIX interface, it is very effective to compare system call traces; divergences often identify root causes. We capture a reference trace in Linux with strace, and add a corresponding debug facility at the emulator's entry point. It emits a trace file using another debug output channel.

Of course, a debugger is invaluable. Our debug-mode monitor runs apps as Linux processes. It routes Embassies syscalls out through a pipe to a coordinating process, but leaves the conventional POSIX syscall interface intact, enabling gdb to connect to the process.

However, gdb has no access to symbols. The emulator does not use POSIX mmap to map in ELF files, so gdb's inspection of Linux-provided metadata in /proc/*pid*/maps is fruitless. To bridge this gap, the emulator records a trace of file open and mmap operations via another debug channel. A script transforms the trace into a gdb add-symbol-file script, solving the symbol problem.

Similarly, gdb's usual mechanism for discovering new threads fails when thread creation is handled by the emulator. Thus, the debug monitor provides another extension by which the emulator signals thread creation, and the debug monitor generates the appropriate trap (int $0x3) to alert gdb.

We haven't yet implemented gdb stubs for our secure monitors, because once an app runs correctly in the debug monitor, it rarely fails in the secure monitors. In the rare failure cases, we found it sufficient to study a core file (a snapshot at the moment of failure). Each secure monitor has a debug mode in which a picoprocess exception generates an ELF-format core dump.

The debug monitor also provides an extension to query CPU time (POSIX times()), and a sampling profiler, for diagnosing performance problems. An example discovery was that the emulator was returning bogus stat values, causing a font library to deem its cache file invalid, causing it to re-scan thousands of individual font files at app start.

Finally, gathering the appropriate file set for the read-only app image is tedious. To expedite, the emulator can start in "gullible mode", where rather than fetch an image, it passes every open request path out to a lookup server located on the development machine where the original POSIX app is installed. That server hashes the corresponding file, injects the file contents into the cache,

and returns the path to the emulator. By this means, the emulator demand-loads the app's required files; it also captures a trace of these loads, which serves as a manifest for generating the app image.

6 Discussion

Our goal is to reuse conventional interactive desktop applications in a new minimal runtime environment. Ideal reuse would use unmodified binaries; required modifications can be ranked based on their invasiveness.

Transparently emulating required behavior below the POSIX interface has proven to be very inexpensive; the main cost is discovering which features actually warrant implementation (§5). Our experience suggests that the emulator is asymptotically nearing completeness.

The choice to give apps no control over their memory layout, which makes Embassies implementable on any host, is slightly invasive; it requires relinking the top-level app binary, which is easy in practice (§3.11.1).

The Embassies environment demands some point changes higher in the software stack, including the binding of X to the Embassies UI interface (§4.1) and the replacement of implicit kernel communication with explicit protocols (§6.3). Such changes do require source modification of specific packages, but very few such changes are required, compared with the hundreds of library packages ported.

The Embassies impositions do preclude running some unmodified binaries, such as closed-source apps. Closed-source libraries with Embassies-compatible semantics, such as a PDF-rendering library, may be usable, though.

6.1 Dynamic vs. Static Linking

In our previous experience with the Xax project [13], we found that modifying a package's build system was frustratingly difficult, generally much harder than modifying the source code and using the package's build system to remake it. Most packages use common source languages such as C or C++, but it seems every package uses a different build scheme.

Engineering choices in Xax required statically linking each app with all of its libraries. Because that changed how apps and libraries build, the task ranged from difficult to all-but-impossible, and required new work for nearly every package.

Thus, in the present work, we elected instead to keep applications dynamically linked, and to press `ld-linux.so` into service for runtime linking. We found that this expedient substantially reduces the invasiveness of porting, as essentially every intermediate library is readily usable in binary form.

6.2 Limitations

Our experience iterating the emulator to support several apps suggests that the emulator is asymptotically nearing completeness, ready to support most desktop productivity apps. In most cases where we have introduced a lie or neutered behavior into the emulator, it is because we have examined the corresponding call site in libc, and we were able to conclude that the lie completely satisfies that code path. This approach occasionally backfires when a different call site finds the lie unconvincing, but these occurrences are rare.

System configuration tools are unlikely to port well, since our approach destroys tight application coupling, for example by neutering `dbus`. We accept this limitation as fundamental to Embassies's goal of making apps more autonomous.

Embassies presently has only paper designs for audio and GPU facilities. Apps that integrate multiple programs with `fork()` are not currently supported (§3.9).

6.3 Inter-Application Protocols

This paper focuses on moving apps from a rich, trusting, shared environment to the isolated picoprocess. However, interesting apps still communicate with the outside world. Some inter-app communication is already based on IP: The apps we used discover printers and send jobs with the Internet Printing Protocol [12], so printing works correctly without special support.

However, how should apps replace communication patterns once done locally? For example, suppose one app produces data another app wishes to read. We expect such communications, once supplied by a complex trusted platform (e.g., the OS), to be replaced by IP-based protocols. Just as in the Internet, IP-based protocols are bilateral: Both participants have the opportunity to decide how much of the protocol they are willing to implement, and to select vulnerability-resistant implementations. The Embassies paper [16] addresses this question in greater detail.

7 Evaluation

7.1 Porting Effort

The most salient proof of effectiveness for our techniques is in the results: We are able to run many rich apps *without even recompiling* them (Figure 3). Instead, we binary-rewrite glibc to redirect the POSIX interface, use libraries as unmodified binaries, and relink the top-most executable to make it relocatable. That such non-invasive techniques are successful with eight interactive apps built on disparate library stacks is strong evidence that they will generalize easily to most interactive apps.

Figure 4 shows lines of code [30] in the components and patches to existing programs. Most of the effort is in the VFS implementation in the emulator.

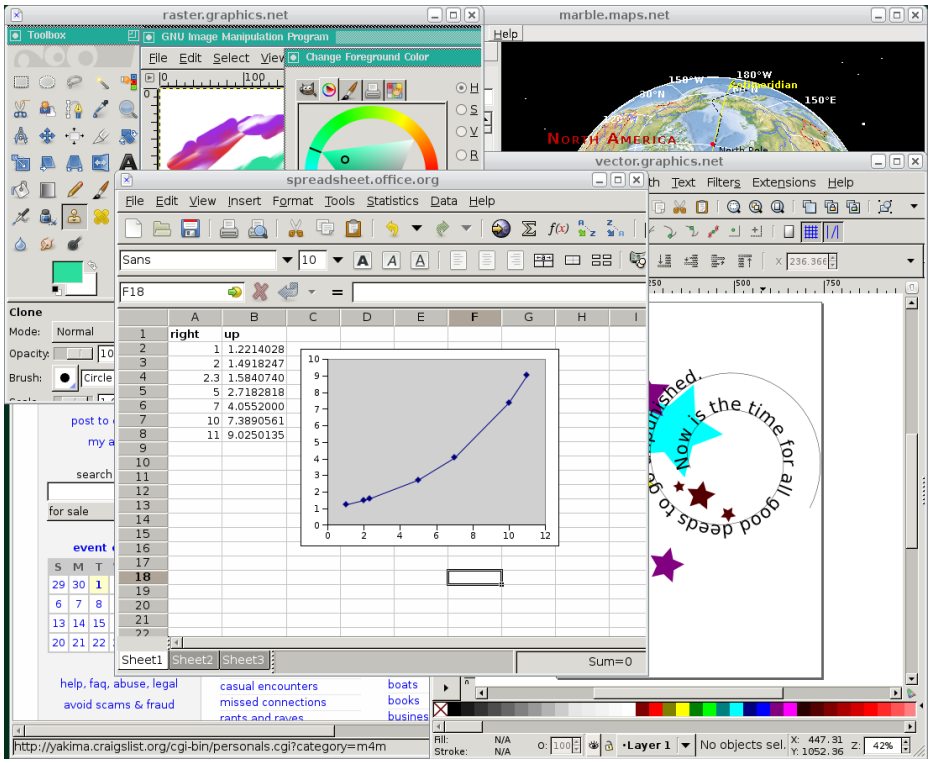


Figure 3: **POSIX emulation handles diverse, rich applications,** *e.g., the Midori Web renderer, Gimp, Marble, Inkscape, and Gnumeric. Not shown are Abiword, Gnucash, and Hyperoid.*

component	SLOC
emulator	29156
zone host	1328
lwIP patches	477
X patches	660
twm patches	0

Figure 4: **Lines of code** in system components.

7.2 Performance

For compute-bound tasks, the emulator is not involved, and apps run at native speeds. We verified this by running, on both Linux and Embassies, image rotations in Gimp and a subset of the SunSpider JavaScript benchmark [29] in Midori. As anticipated, in both cases the difference is negligible, within 2% ($c_v = 1\%$).

Informally, we have observed some emulated activities run faster than their Linux equivalents. For example, filesystem interactions with temporary files outperform Linux because they avoid a kernel-mode transition.

The application launch mechanism precludes the use of the OS buffer cache, but we recapture much of that performance in the Embassies environment [14, Fig. 14]. App starts are 50–100 ms slower than Linux; the largest bottleneck is verifying the integrity of fetched content.

7.3 Coverage

We have demonstrated a layer that emulates a small subset of Posix behavior, and we have shown this to be sufficient to run a diverse set of productivity apps. However, perhaps exercising the apps more aggressively or running additional productivity apps would require substantially more Posix-level emulation. To bound the degree to which more emulation could be required, we compare the set of syscalls visited dynamically with the set reachable statically. This analysis is approximate, because some syscalls (e.g., `ioctl`) aggregate multiple behaviors, and our static analysis tool is rather coarse.

Figure 5 shows the results. Columns are syscall numbers (sorted for contiguity); rows are applications. The upper eight apps are those we support (Table 1); the lower eight are other Linux apps to aid extrapolation. System calls in region (d) are supported as described in this paper: as meaningful, failure-oblivious, neutered, or deluded calls. Syscalls in region (x) are observed dynamically when the app is run on Linux but not when run on our emulator. For example, because `shmget` is neutered in the emulator, `shmat` and `shmctl` never appear dynamically. If the lower eight apps in Figure 5 were run on our emulator, these syscalls might be obviated for analogous reasons, but we do not know this for certain.

The 27 syscalls in region (s) are reachable statically, but not observed dynamically. Five are never called because a better version is emulated (`stat64` for `stat`). Another 7 are trivial variations of existing emulation (`ppoll` for `poll`), and 12 (those unique to muse and stella) are neuterable (`mlock`, `setgid32`). We saw 3 calls in the lower eight apps that likely require implementation: `utimes`, `symlink`, and `mknod`.

Our static analysis is imperfect, as evidenced by ten calls reached dynamically but not discovered statically.

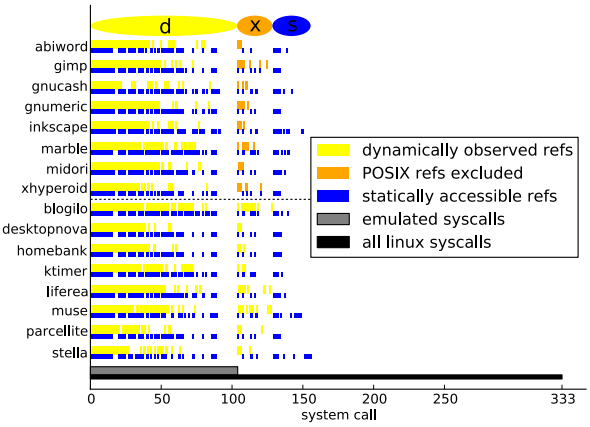


Figure 5: **An approximate static coverage check validates** *that emulating most Linux system calls is not necessary.*

Nonetheless, the broad agreement between static analysis and dynamic observation suggests that our emulation is largely complete. Of Linux's 333 syscalls, 203 are observed neither by dynamic tracing nor by static analysis. This lends credence to our claim that a broad range of productivity apps can be supported without emulating the majority of Linux functionality.

8 Related Work

8.1 Application Models

Java was offered as an alternative to the clunky mid-1990s web programming interface [10]. Absent native code, Java had to either rewrite every framework an app could want, or import and abstract existing frameworks as native libraries. Practicality demanded applying the latter technique; even the early UI toolkit AWT [33] abstracted over the host UI at a high level. The result was a Java client with a complex implementation that shared the host's vulnerabilities, and isolation that depended on a complex and growing security interface [21].

As Java largely failed to replace the HTML web app model, HTML thrived, evolving a notion of isolation [15, 32] fundamental to web apps. However, pressure to enhance functionality has progressively grown client complexity, undermining the promise of isolation [16].

The Slinky system proposed distributing POSIX apps as static binaries, enabling app developers to precisely specify their dependencies [4]. They extended the Linux kernel to detect and exploit implicit page sharing while preserving the semantics of static executables. Their approach treats shared libraries as a configuration problem. It inspired our work; we extend the Slinky insight to autonomy-preserving isolation against adversarial neighboring apps. This not only requires avoiding late-bound library sharing, but also demands eliminating the complex shared graphics stack (X or an HTML DOM renderer). Since simplicity is a priority, we eliminate even the shared buffer cache, requiring a sharing implementation different than that used in Slinky [14].

8.2 Porting Applications

Several years ago, our Xax project [13] demonstrated that rich stacks of libraries could be readily transplanted from a conventional operating system environment to provide useful functionality even from inside a picoprocess attached to a web browser. This paper reports on a more thorough implementation that supports complete, rich, interactive applications. Xax gave a high-level overview of the porting effort, enumerating five categories of techniques used to emulate the missing OS or to trigger alternative behavior in the transplanted library. This paper aims to completely demystify the process.

The Drawbridge effort demonstrated that similar techniques could be used for code based on the Windows commodity OS stack [22]; that project required introducing additional techniques, such as hoisting the GDI graphics rasterizing library from the OS kernel to become a library inside the picoprocess. The Drawbridge system assumes a non-minimal host that includes a file system, buffer cache, and TCP stack.

The task at hand is reminiscent of the Exokernel's motto, "exterminate all operating system abstractions" [18]. Like Exokernel, Embassies minimizes abstractions in the host platform; but where the Exokernel evicted abstractions to expose new performance opportunities, Embassies aims to produce a simple, rarely-changing host with a minimal attack surface. Therefore, Exokernel techniques, such as those for sharing storage, do not translate well to Embassies apps.

Google's Native Client system [31] includes ports of dozens of libraries, but does not support complete interactive applications. The difference in target assumption–that applications will run as web plug-ins, rather than replacing web apps altogether–has led the project to a different ABI, security model, and execution model. These choices necessitate a modified C compiler, which in turn requires fussing with libraries' build environment (§6.1), a task we found difficult to scale. However, once those issues are resolved, the approach in the present paper should readily enable the conversion of POSIX apps into NaCl plug-ins.

9 Conclusion

This paper showed how to support rich POSIX applications on top of a minimal picoprocess interface. Such support can be achieved by providing a POSIX emulation layer and by binding existing programs, like lwIP, `X`, and `twm` into the application itself. The POSIX emulation layer is not nearly as complicated as a conven-

tional POSIX implementation (e.g., Linux); in fact, this paper exhaustively lists every syscall emulated and every program adaptation required. Such emulation is possible in part because many POSIX functions exist to support scalability and performance more relevant to server applications (e.g., databases and web servers) and hence are unused by interactive apps. Thus, not only is it feasible to adapt POSIX applications to a sparse environment, it is reproducible. We hope these results will encourage others to adapt the existing world of rich POSIX-based applications to even the most minimal of client execution environments.

References

[1] ANDROID OS. http://www.android.com/.

[2] APPLE. iOS6, 2013. http://www.apple.com/iphone/.

[3] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the Chromium browser. http://www.adambarth.com/papers/2008/barth-jackson-reis.pdf, 2008.

[4] COLLBERG, C., HARTMAN, J. H., BABU, S., AND UDUPA, S. K. Slinky: static linking reloaded. In *USENIX ATC* (2005).

[5] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *IEEE Symp. on Security & Privacy* (2006).

[6] DREPPER, U. Futexes are tricky. Tech. rep., Red Hat, Nov. 2011.

[7] DUNKELS, A. lwIP - a lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, 2013.

[8] ECMA. Standard ECMA-262: ECMAScript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm, June 2011.

[9] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *IEEE ACSAC* (2005).

[10] GOSLING, J., JOY, B., AND STEELE, G. *Java™ Language Specification*. Addison-Wesley, 1996.

[11] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy* (2008).

[12] HASTINGS, T., HERRIOT, R., DEBRY, R., ISAACSON, S., AND POWELL, P. Internet Printing Protocol/1.1: Model and Semantics. RFC 2911 (Proposed Standard), Sept. 2000. Updated by RFCs 3380, 3382, 3996, 3995.

[13] HOWELL, J., DOUCEUR, J. R., ELSON, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008).

[14] HOWELL, J., ELSON, J., PARNO, B., AND DOUCEUR, J. R. Missive: Fast appliance launch from an untrusted buffer cache. Tech. Rep. MSR-TR-2013-9, Microsoft Research, Jan. 2013.

[15] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: Operating system abstractions for client mashups. In *HotOS* (May 2007).

[16] HOWELL, J., PARNO, B., AND DOUCEUR, J. Embassies: Radically refactoring the web. In *NSDI* (2013).

[17] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the crossdomain policies of Flash applications. In *IEEE Web 2.0 Security and Privacy Workshop (W2SP)* (2011).

[18] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *SOSP* (1997).

[19] MICKENS, J., AND DHAWAN, M. Atlantis: Robust, extensible execution environments for Web applications. In *SOSP* (2011).

[20] MICROSOFT. Silverlight. http://www.microsoft.com/silverlight/.

[21] NEVILLE, P. S. Mastering Java security policies and permissions. http://www2.sys-con.com/itsg/virtualcd/java/archives/0501/neville/index.html, 2004.

[22] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. In *ASPLOS* (2011).

[23] REIS, C., AND GRIBBLE, S. D. Isolating Web Programs in Modern Browser Architectures. In *ACM EuroSys* (2009).

[24] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., , AND BEEBEE, JR., W. S. Enhancing server availability and security through failure-oblivious computing. In *OSDI* (2004).

[25] TANG, S., MAI, H., AND KING, S. T. Trust and Protection in the Illinois Browser Operating System. In *OSDI* (2010).

[26] WANG, H. J., FAN, X., JACKSON, C., AND HOWELL, J. Protection and communication abstractions for web browsers in MashupOS. In *SOSP* (Oct. 2007).

[27] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symposium* (2009).

[28] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of desktop and web applications on a multi-service OS. In *USENIX HotSec Workshop* (2009).

[29] WEBKIT. SunSpider JavaScript Benchmark. Version 0.9.1 at http://www.webkit.org/perf/sunspider/sunspider.html, 2012.

[30] WHEELER, D. A. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.

[31] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security & Privacy* (2009).

[32] ZALEWSKI, M. Browser security handbook: Same-origin policy. Online handbook. http://code.google.com/p/browsersec/wiki/Part2.

[33] ZUKOWKSI, J. *Java AWT Reference*. O'Reilly, 1997.

Network Interface Design for Low Latency Request-Response Protocols

Mario Flajslik
Stanford University

Mendel Rosenblum
Stanford University

Abstract

Ethernet network interfaces in commodity systems are designed with a focus on achieving high bandwidth at low CPU utilization, while often sacrificing latency. This approach is viable only if the high interface latency is still overwhelmingly dominated by software request processing times. However, recent efforts to lower software latency in request-response based systems, such as memcached and RAMCloud, have promoted network interface into a significant contributor to the overall latency. We present a low latency network interface design suitable for request-response based applications. Evaluation on a prototype FPGA implementation has demonstrated that our design exhibits more than double latency improvements without a meaningful negative impact on either bandwidth or CPU power. We also investigate latency-power tradeoffs between using interrupts and polling, as well as the effects of processor's low power states.

1. Introduction

Historically, network card latency has been overshadowed by long wide area links and slow software, both of which easily bring the overall latency into the millisecond range. More recently, datacenter applications have emerged with more rigorous latency requirements, thus inspiring efforts to reach low latency on commodity systems. An example of such datacenter applications are various database and caching services that operate from main memory, such as memcached [6]. Moreover, there are ongoing efforts to build ultra low latency software, such as the RAMCloud project [22]. RAMCloud is a durable storage system boasting latency goals of 5-10 us round trip, inside a commodity datacenter.

There are economic reasons that lead us to believe that the commodity low latency trend will continue for the foreseeable future, and will not be limited to the high performance niche. The ability to do evermore processing and continuously add new features is essential to many software companies' differentiation strategies, thus being the driving force behind generating revenue. These companies' services must run within a small latency budget, demanding commodity technology that provides low latencies.

On the academic front there has been an analogous, and likely correlated, interest in low latency. Some software efforts were already mentioned, but other disciplines also contribute. For example, network and interconnect research yielded ideas for bufferless switching [1] and new network topologies [16]. We position our work between network and software research, right on the interface connecting the host and the network.

We adopt a clean slate approach to the problem and build the lowest latency request-response system that we can. Following the clean slate approach, we developed a very simple, and very fast, minimal object store evaluation application that supports only two operations: GET and SET. The minimal object store application exhibits low absolute latency, but it also has low latency variability. Due to this application choice, the latency focus shifts onto the network interface design, which is our paper's main contribution. We focus on two latency sources in the network interface: 1) control communication and data transfer between the CPU and the network card; 2) processor idle state wakeup times and power management.

Our interface, NIQ (Network Interface Quibbles), was designed after a detailed investigation of reasons behind latency inefficiencies in current network cards, as described in Section 2. We found that one of our key objectives must be to minimize the number of transitions over the PCIe interconnect. This is especially true for small packets, which are prevalent in request-response protocols. In Section 3 we describe how combining existing techniques (e.g. embedding small packets inside descriptors) with new ideas (e.g. custom polling, creative use of caching policies) leads to a low latency interface that does not sacrifice bandwidth.

To evaluate NIQ in detail and compare it to other possible solutions, we built a configurable FPGA-based network card. This network card is configured and controlled by a user-space NIQ driver that provides zero copy capabilities and offers direct application access through bypassing the kernel stack. Evaluation system is described in Section 4, together with bandwidth and latency performance analysis. CPU idle state power measurements and power-latency tradeoffs of interrupts and polling are presented in Section 5.

Ideally, total network latency would be dominated by

wire propagation delay which is limited by the speed of light to around 5 nanoseconds per meter. Assuming total round trip distances of under 200 meters inside the datacenter, total time spent on the wire is under one microsecond. The ultimate challenge is to bring network card latencies into the same range. In the meantime, our NIQ achieved the best round trip latency (client-server-client, with a network cable in between) of 4.65 us. However, much of that time is inherent to the hardware components we had available. In Section 7 we discuss the possibility of round trip latencies under 2.3 us with state of the art components and an ASIC implementation.

2. Network Interface Card Design

When implementing request-response protocols on commodity systems, attempts at low latency often run into a system designed for a wide area network where latency is secondary to achieving high bandwidth. We find current designs to be lacking in latency performance, even though they are well suited for high bandwidth systems that are dominated by software latency. Our motivation are new low latency systems, such as the RAMCloud project [22], that have software overheads in the one microsecond range. As a comparison, one round trip time through an idle linux kernel networking stack measures at 32-37 us. This measurement includes one receive and one transmit path between the NIC driver and the user application for UDP (32 us) and TCP (37 us) packets. In this section we highlight the network controller challenges to achieving low request-response latency on existing systems.

To illustrate the problems addressed in this paper, we examine in detail how one would build a request-response based system on top of a current commodity 10G Ethernet NIC. We assume an Intel 10G x520-DA2 adapter [14] because we have access to one, but other network controllers, such as Broadcom NetXtreme, exhibit similar behavior [11, 4].

A typical network card is connected to the host system over PCI Express (PCIe) and contains these components: DMA engine, ring buffers, Ethernet MAC and PHY, plus additional features (offload engines, QoS, virtualization, etc.). The DMA engine connects directly to the PCIe interface and transfers data between the host memory and the ring buffers on the NIC. Performance of the MAC and the PHY, as well as the PCIe interconnect, contribute to overall performance, but in this paper we focus on the interface between the NIC and the host. This interface is defined by the functionality of the ring buffers and how they are managed by the NIC and the host driver. We describe the interactions between the host and the NIC by stepping through the packet transmit and the packet receive process, shown in Figure 1.

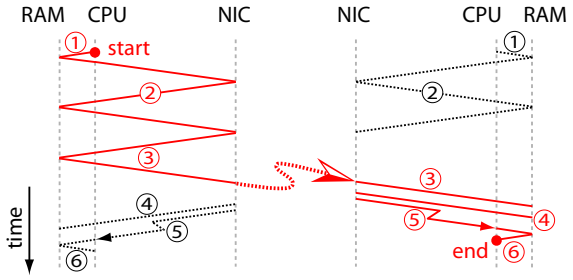


Figure 1: Timing of TX (left) and RX (right) steps.

2.1. Transmit steps

When a client application decides to make a request, it formats a request packet and sends it to the server. The NIC transmit interface requires adding packet metadata (i.e. the packet descriptor) to the main memory descriptor ring shared between the CPU and the NIC (step 1 in Figure 1 left). To inform the NIC of a newly available transmit descriptor, the CPU does an uncached I/O write to the "doorbell" register on the NIC (first part of step 2). The NIC, then, uses its DMA engine to fetch the next transmit descriptor (last part of step 2). The DMA engine is also employed to fetch the contents of the packet that is ready for transmission, based on information obtained in the packet descriptor (step 3). Overall, it takes two and a half PCIe round trips to get from software transmit initiation to the packet being available in the NIC.

Described use of the descriptor ring buffer provides decoupling between the CPU and the NIC, thereby allowing the CPU to get ahead of the NIC in the number of transmitted packets. Bursts of packets from the CPU are effectively queued in the ring buffer which is drained by the NIC as fast as the network allows. This decoupling is key to achieving high bandwidth.

At this point the packet is on the wire and on its way to the server, but there is still some necessary client NIC bookkeeping. When a packet is handed to the NIC for transmission, its memory cannot be reused until the DMA is completed. Upon DMA completion, the NIC sets a flag in host memory indicating that the packet buffer may be reused (step 4). Typically, there is space reserved for this flag in the packet's descriptor entry in host memory. Following step 4, an interrupt is generated (step 5) that triggers the CPU completion handling (step 6) and thus completing the transmit process. Bookkeeping (steps 4-6) is not in the critical latency path of the request-response protocol, but it must be done promptly to prevent the client from running out of resources (e.g. memory space, or flow control credits).

2.2. Receive steps

Before the NIC can receive any packets, it must be loaded with information about available receive packet buffers in main memory. A descriptor ring buffer is used

to keep the available receive descriptors, similar to the transmit descriptor ring. Prior to any packets arriving, the driver allocates multiple receive packet buffers, creates receive descriptors pointing to those buffers, and transfers the descriptors to the NIC (steps 1 and 2 in Figure 1 right).

Once the client's request packet finds its way through the network, it arrives at the server's NIC. Upon packet arrival, the NIC reads the next available descriptor entry to determine where to deposit the packet. After depositing the packet into the host memory using the DMA engine (step 3), the NIC notifies the CPU of the packets arrival. The appropriate packet descriptor ring entry is repurposed as a completion entry, now containing packet length and a flag indicating there is a new valid packet (step 4).

Just as they did in the transmit case, the ring buffer structures allow the NIC to run ahead of the CPU and deposit packets faster than the CPU can process them, at least in bursts. The CPU must read the ring completion entry to discover the location and size of the arrived packet. To avoid dedicating a CPU core to monitoring the completion ring, operating systems prefer to configure the card to interrupt the CPU as a form of a completion notification (step 5). Under high loads, the NIC might generate too many interrupts for the host to handle, thereby putting the receiver at risk of a livelock. One of the mechanism that mitigates this problem is interrupt coalescing in the NIC [26], where the NIC can coalesce several received packets together by deferring receive notifications and eventually triggering only one interrupt for several received packets. Under low load this method adds significant latency (even hundreds of microseconds) because interrupt generation is in the critical receive latency path. As an alternative to interrupts, the host can operate in a polling mode, continuously reading the next expected completion ring entry until it becomes valid. Modern operating systems alternate between polling and interrupt modes depending on the load, thus avoiding the receiver livelock issue [21] regardless of interrupt coalescing.

Finally, the CPU reads the receive descriptor (step 6) and the received packet, which is then forwarded to the application layer for processing. Also, at this point the CPU allocates a new receive buffer and updates the NIC with a new receive descriptor to replace the one that was just used. After processing the request, the server application formats a reply packet and sends it back to the client following the same transmit-receive sequence that was just described, thus completing one request-response round trip. Overall, 16 one-way transitions over PCIe links take place between client initiating a request and receiving a response from the server. Out of the 16 one-way transitions, 12 are synchronous, which means they

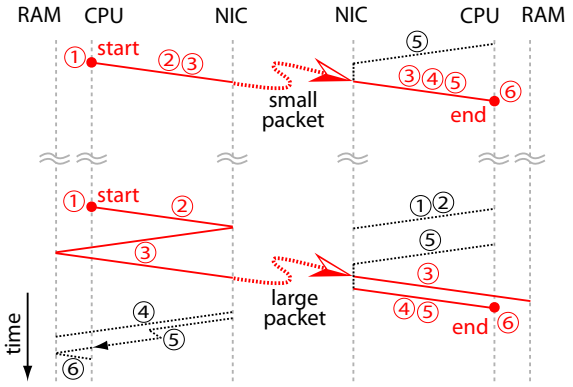


Figure 2: NIQ timing of small and large packets.

have to be completed before the next one can begin, thus affecting the overall latency.

3. NIQ: Interface Design

To resolve our Network Interface Quibbles, we propose a new network interface design, NIQ, that enables low latency implementations of request-response protocols. We achieve that goal by focusing on small packets and reducing the number of PCIe transitions.

The key insight with regard to the nature of request-response protocols is that at least half of the network packets are very small. This is true because either the read request, or the write response contain no data. Another important general insight is that network latency of short packets is generally more critical than the latency of large packets. Whenever large chunks of data are involved, inherent serialization and software processing latencies are higher, lessening the importance of other latency sources.

Network card design described in Section 2 and illustrated in Figure 1 is particularly inefficient for small packets because it requires a total of eight PCIe transitions for send and receive combined. Crossing over the PCIe interconnect is costly in terms of latency (over 0.9 us round trip is our system), therefore we focus on minimizing the number of PCIe transitions. In the best case scenario, only two transitions would suffice: one for transmission and one for reception. We present an interface that accomplishes the best case for small packets, but also provides good results for larger packets.

3.1. Small Packets

NIQ exploits the fact that modern processors are optimized around cache-line size transfers. Minimum size Ethernet packets are 60 bytes in length, thereby fitting into a 64 byte cache line. Note that minimum Ethernet size is usually said to be 64 bytes, but that includes the 4-byte FCS (Frame Check Sequence), although it does not include the preamble and the start delimiter. All mod-

ern network cards generate and strip the FCS in hardware and never expose it to higher layers, reducing the effective minimum packet size to 60 bytes. Observation that minimum size packets fit inside a 64 byte cache line serves as an important guideline to designing the small packet interface. Processors are already optimized for communication using cache lines, so we assume that small packets are minimum size 60 byte Ethernet packets. Expanding the interface to support, for example, two cache-line size packets is very much possible, and we discuss it in Section (Section 7). Timing of the small packet transmit and receive sequences is illustrated in the top part of Figure 2, with step numbers corresponding to Figure 1.

To achieve the ideal goal of one PCIe transition on transmit and one transition on receive, NIQ interface folds all critical steps from Figure 1 into just a single step. Folding of the transmission steps into one is accomplished by embedding the entire small packet within the transmit descriptor. Moreover, we do not employ the DMA engine to transfer the descriptor, but instead the descriptor is transferred by the CPU directly to the network card. Our transmit descriptors are one cache line wide, with flags indicating whether an entire small packet is embedded in it, or it is a traditional descriptor with packet address and length.

On the receive side, folding of the steps is achieved by embedding the entire small packet inside the completion entry. Upon reception, the entire small packet is transferred within a cache line wide completion, instead of copying it into a host buffer via the DMA engine. Assuming the use of polling (discussed later in this section), the completion containing the data also serves as a notification, thus successfully folding all critical receive steps.

As an additional benefit of transferring small packets in this fashion, no host memory buffers are consumed on transmit or receive. Since no host memory is consumed, there is no need to allocate or free any buffers, thus reducing the total amount of software bookkeeping. It is still necessary to exchange flow control credit information between the network card and the host, but that can be batched and done less frequently.

As was already mentioned, both transmit descriptors and receive completions are 64 bytes wide. To efficiently communicate with the network card in cache line size units, we utilize the cache hierarchy and write-gathering buffers. All memory that is written by the CPU is mapped as write-gathering (also called write combining), while the memory that is read by the CPU is mapped as cacheable. This is a departure from standard practices of mapping I/O memory as uncacheable, but it is similar to graphics cards practice of using write-gathering policy for mapping frame buffers.

Any write by the CPU made to a write-gathering address bypasses the cache hierarchy and goes into one of CPU's write-gathering buffers. Once the entire cache line is written, or a memory-ordering instruction (such as *sfence*) is executed, the entire cache line is flushed over PCIe to the network card [12]. Combining the writes in a buffer close to the CPU core improves CPU bandwidth and PCIe bandwidth. In fact, it would not be feasible for the CPU to use uncached writes to write the descriptor over PCIe 8 bytes at a time. Each PCIe packet incurs up to 28 bytes of header overhead across all layers [23], resulting in a 77% overhead. The number of overhead bytes is the same regardless of data payload size, making 64 byte transfers more efficient.

The cacheable mapping of completion entries enables us to issue cache misses to the network card, transferring the entire 64 byte completion in one PCIe packet. Moreover, the method of polling the network card puts the completion entry all the way into the first level cache, where it is ready for immediate processing. In order to force the cache miss, an appropriate *clflush* instruction is executed before the polling read. One of the implications of using a cacheable memory type in this fashion is that read side-effects are not allowed in the NIQ. For example, a NIQ address location might be read multiple times, either speculatively or due to a cache eviction. Therefore, we cannot rely on reads to inform us when the CPU has processed the data, but instead we require explicit flow control notifications.

3.2. Large Packets

For large packets NIQ still uses cache lines for communication, but entire packets no longer fit within a descriptor. Instead, we follow the traditional approach of employing the DMA engine to transfer packet data, as discussed in Section 2. Descriptors and completions are still cacheable and 64 bytes wide, but instead of embedding the entire packet (like in the small packet case), they only contain the first 48 bytes of the packet, which includes headers. Putting the headers inside the descriptor/completion enables an efficient implementation of header splitting on transmit/receive.

Header splitting on transmit is necessary to enable zero-copy techniques often used in low latency system implementations. If the header cannot be split from the payload, it is necessary to copy the packet data into an intermediate memory buffer before transmit. To avoid copying the header and the data into a single buffer, the network card's DMA engine can be programmed to transfer them separately and join them before they leave the card. However, programming the DMA engine to perform two transfers typically requires two separate transmit descriptors. Our interface enables header splitting with just a single transmit descriptor. Splitting the header on reception can also be beneficial, because the

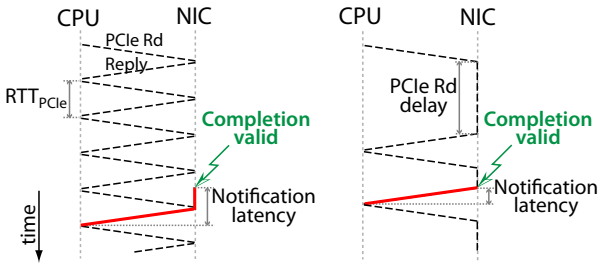


Figure 3: Notification timing diagram without (left) and with (right) PCIe read delay.

header is available for CPU processing as soon as the completion is read, as they are in the same cache line.

We have explored and evaluated the option of using the CPU to transfer entire large packets through the write-gathering buffers, but we found the CPU bandwidth penalties increase significantly with packet size (up to 70% penalty for largest packets). Even though the small packet interface provides lower latency, it is entirely possible to send small packets through the large packet interface. This is recommended when strict packet ordering is important because a small packet that is sent through the small packet interface is allowed to pass a large packet and leave the network card out of order. Allowing small packets to bypass large packets is meant as a feature, since it can easily save over 1 us of transmit time for the small packets. It is entirely possible to enforce strict ordering on the interface, but since the underlying network doesn't guarantee packet ordering, we choose to allow reordering.

There is some necessary bookkeeping of host memory buffers taking place off the latency critical request-response path, much like we discussed in Section 2. On the transmit path, this processing is batched for efficiency reasons and an interrupt scheme employing interrupt coalescing is used to notify the CPU of the necessary bookkeeping. NIQ uses interrupts for bookkeeping notification, but for critical notifications it employs a custom polling scheme, which we describe next.

3.3. NIQ polling

For the NIQ interface we designed a custom polling technique that we refer to as NIQ polling. Instead of polling the host memory, NIQ polling repeatedly issues reads over PCIe to the network card, thus avoiding communicating through the main memory. Additionally, the replies from the network card are put directly into the first level cache, thereby avoiding a cache miss.

The goal of the NIQ polling notification scheme is to minimize *notification latency*. We define *notification latency* as the time between a new valid completion entry being ready in the NIQ and when that completion is ready for processing in the CPU. When repeatedly polling the network card (illustrated on the left of Figure 3), *notification latency* is between half a PCIe round trip and one and a half round trip. The expected *notification latency* value is one PCIe round trip because it may take up to one whole round trip time between the completion being ready and the CPU polling read arriving at the network card.

To lower the expected *notification latency* we introduce a *PCIe read delay* time inside the network card. When there are no new valid completions, the network card holds on to the polling read for one *PCIe read delay* time before eventually replying with an invalid entry, instead of replying immediately. However, if a new completion is ready, a reply is generated immediately, thereby significantly reducing the expected *notification latency* (up to 2x reduction). This process is illustrated on the right of Figure 3. As an added benefit of *PCIe read delay*, fewer invalid entries get transferred over PCIe, also reducing the number of invalid entries the CPU must process.

Choosing the correct value for *PCIe read delay* is a balancing act between minimizing expected *notification latency* and avoiding triggering any deadlock prevention or watchdog mechanisms that could be triggered by a delayed memory read. Expected *notification latency* is inversely proportional to 1+(*PCIe read delay*), and we found that a *PCIe read delay* value of about 20 PCIe round trips gives good latency performance. This delay value is well clear of triggering any operating system watchdog mechanisms, but we have encountered an interesting interaction with Intel's implementation of simultaneous multithreading, known as hyper threading. While a hyper thread is waiting on the polling read to return, one would expect its sibling thread to make full use of the execution units and other shared resources, leading to NIQ polling being an even more attractive solution. However, we found that when using NIQ polling and delaying the read response for longer than around 4 us (or 10000 cycles), the sibling hyper thread makes no forward progress. We attribute this phenomenon to a deadlock/starvation prevention mechanism that detects the polling thread has not made progress for a long time (while waiting on the poll read), and upon detection preventatively stalls the sibling thread. For this reason we do not use hyper threading.

Polling in this way permanently consumes one PCIe read credit, but is unlikely to cause any issues on the PCIe bus because the PCIe spec allows for read delays of at least 10 ms [23]. Next, we discuss alternatives to our NIQ polling technique.

3.3.1. Interrupts vs. host polling vs. mwait

Historically, there was a huge speed mismatch between the I/O devices and the CPU, making interrupt

schemes necessary and efficient. It would be unfeasible for the single core CPU to wait several milliseconds for an I/O device to complete an operation. However, modern network cards are much faster and modern CPUs have multiple cores, thus changing the balance. One would still like the CPU core to be able to do other processing while waiting on the network device, but often there is nothing else to do. This reasoning coupled with poor latency performance of interrupts is what makes polling an attractive option.

We find three main reasons for poor interrupt latency performance. Firstly, we measure a 1.4 us delay between when the interrupt controller is instructed to generate an interrupt, and the linux interrupt handler is executed. The second latency penalty comes from the necessary inter-thread communication between the interrupt handler and the user application thread. The inter-thread communication is necessary because implementing the entire user application (i.e. consuming and generating packets) inside an interrupt handler is impractical at best. The third reason interrupts have a bad latency reputation is power management, and specifically CPU idle states. While waiting for an interrupt the CPU is not busy (unlike when polling) and often reaches a deep idle state with an exit time in tens of microseconds. We investigate the power management tradeoffs further in Section 5.

Polling on the host memory location is a good low latency alternative to interrupts. When using host memory polling, the CPU reads the memory location of the next expected completion entry in a tight loop until that location becomes valid. Because the reads are done in a tight loop, the memory location is cached and reads return quickly for as long as the completion entry is invalid. Due to reads hitting in the cache, the CPU must do a lot of useless work processing invalid entries, realizing they are invalid, and reading them again. When the network card actually updates the completion entry, the cache line gets invalidated and the CPU incurs a cache miss when it tries to read it. To avoid spinning on a memory location in a tight loop, the CPU can issue a *monitor* instruction for that cache line, followed by an *mwait* instruction that halts the processor to save power [12]. As soon as the monitored cache line is invalidated by the network card's write, the CPU is woken up to process the completion. Section 5 provides experimental evaluation of latency-power tradeoffs between interrupts and polling.

4. Object Store Evaluation

In this section we evaluate latency and bandwidth performance of a NIQ implementation. Our implementation is based on a dual socket system with Intel Westmere processors and a NetFPGA [29] board, as shown in the block diagram in Figure 4. The system has a minimum PCIe read round trip latency of 930 ns, which includes

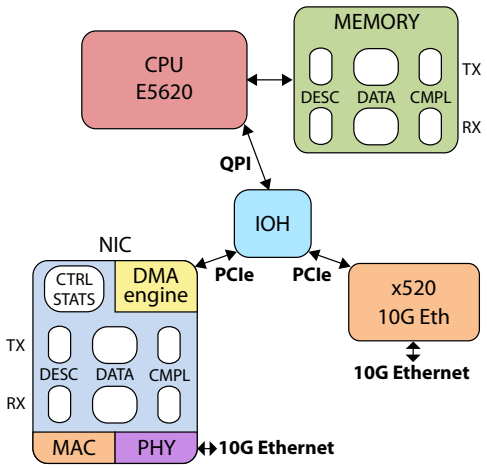


Figure 4: Block diagram of the test system.

the FPGA and the host machine, but is mostly dominated by the FPGA. NetFPGA Ethernet PHY and MAC round trip latency is 868 ns + 0.8 ns/byte, giving a range of 920 ns to 2.08 us for smallest to largest packets. All measurements in this section are based on a request-response roundtrip of a single-threaded minimal object store application. The object store application was built to provide only the essential set of features for conducting the evaluation, thus shifting the evaluation focus onto the NIQ. For latency experiments, client and server are running on different processors, but on the same system, both using the same NetFPGA card with a copper twinax network cable providing a loopback link. Running on the same system, and thus in the same time domain, makes it possible to obtain detailed latency breakdowns. To avoid interference under high load, bandwidth experiments are run on two separate systems, with the client using a commodity network card, while the server still runs on NIQ.

NIQ is directly compared to an Intel x520 network card. To make the comparison fair, the x520 user-space driver was modified and integrated into the object store application so it can be used in the same way as the NIQ driver. For the reasons of implementation simplicity, physical addresses are made available to the application by using a kernel module that allocates 1 MB chunks of contiguous physical memory, which are further fragmented and managed by the application itself. Our application protocol runs on top of Ethernet and IP to correctly emulate a datacenter environment where a packet must travel through switches and IP routers. Most measurements are conducted on GET requests, except where otherwise indicated, because GETs are assumed to be an order of magnitude more frequent [3].

4.1. Latency evaluation

Here we present latency measurements on an unloaded system. In the unloaded system each request is sent in

		A	B	C	D	E	F	G	H	I	J	K
NIC	NIQ	•	•	•	•	•	•	•	•	•		•
	x520										•	
RX	small pkt	•	•	•	•		•	•	•	•		
	niq poll	•				•	•	•	•	•		
	host poll		•								•	•
	interrupt			•								
	mwait				•							
TX	small pkt	•	•	•	•	•	•		•	•		
	hdr split	•	•	•	•	•	•	•	•		•	•
	no DMA						•					
	doorbell								•		•	•

Table 1: Design configuration variants.

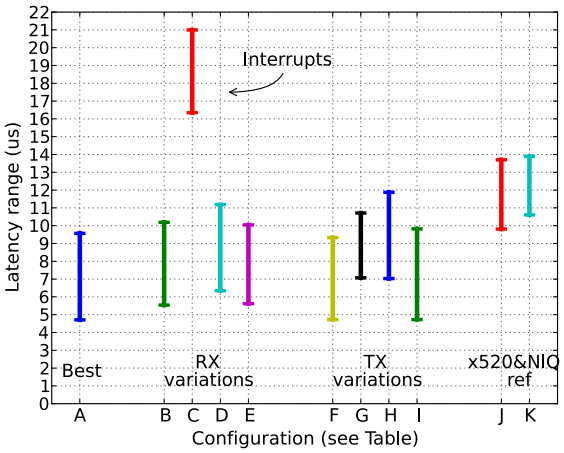


Figure 5: Design configurations' latency ranges for minimum to maximum size packets.

isolation from other requests in an effort to get consistent best case latency breakdown. Experiments with a loaded system are presented in the next subsection. All of the NIQ latency experiments are conducted with both client and server using the NIQ interface. Reference x520 experiments are done with client and server using the Intel x520 network card.

Our NIQ prototype can easily be configured to operate in different modes (e.g. polling, interrupts, header splitting on/off, small packet optimization on/off). We present latency performance of configurations listed in Table 1. For each configuration in Table 1, a matching latency range can be found in Figure 5. The bottom end of each latency range corresponds to a GET request round trip of a small object (4 bytes), while the upper end of the latency range corresponds to a large object (1452 bytes). Configuration A is our best NIQ configuration, as described in Section 3. Reference configurations J and K correspond to the Intel NIC design described in Section 2, with the exception that they use host memory polling instead of interrupts. In configuration K, NIQ prototype is configured to behave the same as the x520 NIC (configuration J) and they both exhibit similar latencies. This validates our choice of comparing our design with the x520 card, since for the same configuration NIQ and x520 perform similarly.

Configurations B through I are all unique and differ from configuration A in only one parameter, enabling us to quantify the benefits of each configuration parameter. In Figure 5 we demonstrate that our NIQ polling technique has the lowest latency, especially compared to an interrupt scheme (configuration C) that exhibits the highest latency. The same Figure 5 shows latency effects of small packet optimizations, as well as the effect of using a doorbell register scheme to initiate a descriptor DMA transfer. Even though, for large requests only, configuration F has a marginal latency advantage over configuration A, we dub configuration A as best. This is because configuration F uses the CPU to transfer data to the NIQ, instead of utilizing the DMA engine, which incurs a CPU bandwidth hit of over 3.5x for large requests. Header splitting provides small latency gains (up to 0.24 us) and it enables zero copy transmit, thereby improving bandwidth by over 20% for the largest requests.

To obtain further insight into the latency breakdown, we instrumented the userspace driver code to timestamp the request-response pairs at various points using the *rdtsc* instruction. For the timestamps to be accurate, we place every *rdtsc* instruction in-between two memory barriers, thus causing a total instrumentation overhead of 0.35 us compared to times presented in Figure 5. We are able to get one way request time measurements because the server and the client run on the same physical machine, but on different processors.

Figure 6 shows latency breakdowns for GET request (subplot a) and SET request (subplot b) running on NIQ. GET and SET breakdowns are similar, with a difference that the server returns a SET reply before touching any object data, leading to a constant server application latency with respect to object size. Replies containing objects that are 12 bytes or smaller, fit into a minimum size packet (after accounting for the 48 byte header), thus allowing the server to use the small packet transmit path and achieve lower latency (shown in the inset of Figure 6a).

While NIQ is optimized for small packets, it also performs well with large packets. A comparison between Figures 6a and 6c illustrates that NIQ and the x520 card latencies scale similarly with object size, but NIQ consistently provides lower latency. Figure 6d illustrates why we choose to transfer large packets using the DMA engine (subplot a), rather than stream writes using the CPU (subplot d). In subplot d, the server driver time dramatically increases with packet size, reducing the server bandwidth by up to 3.5 times. However, streaming CPU

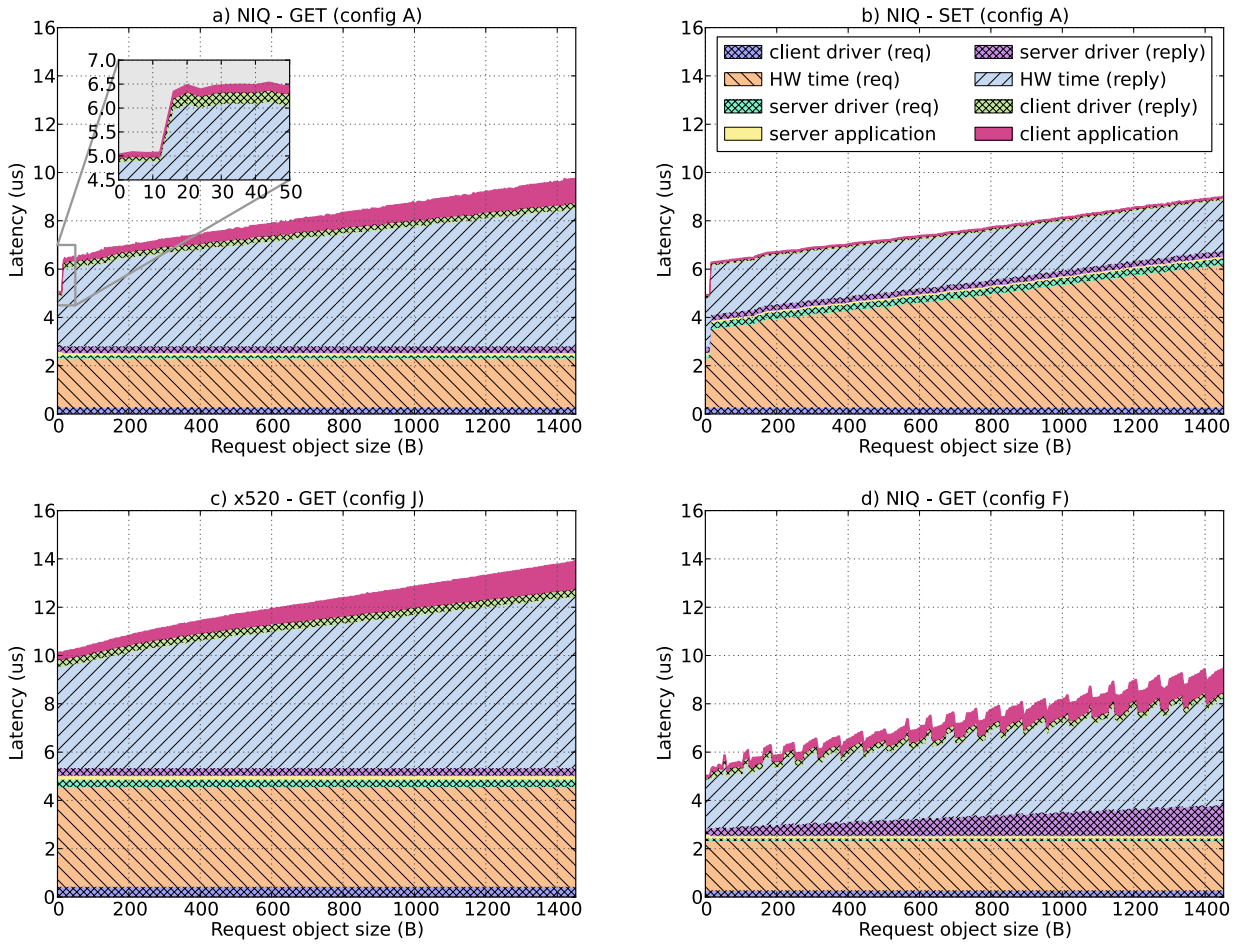


Figure 6: Detailed request latency breakdown.

writes might be a practical solution for transmitting packets that are just over the minimum size, where latency benefit is large and bandwidth penalty low. The saw-like artifacts in subplot d are very specific to our implementation and are the result of the host running out of PCIe credits when the object size is not an integer number of cache lines. Writing a partial cache line is implemented using multiple 8-byte writes, thus requiring more PCIe credits.

4.2. Bandwidth evaluation

Another important performance aspect is how NIQ behaves under heavy load, which we explore next. To generate enough load to saturate the server that is running the NIQ based single-threaded object store application, we use a multithreaded client running on top of the x520 card. This way we are able to make sure the bottleneck is the server running NIQ, therefore measuring NIQ bandwidth and NIQ latency under heavy load.

Figure 7 shows the server throughput on the outgoing link, which is also the bottleneck link since it carries larger packets than the inbound link. For large packets the throughput is limited by the physical link speed of 10 Gbps, but small and medium packet throughput is limited by the server processing power. There is an initial NIQ throughput drop in Figure 7 that occurs when the request object size is over 12 bytes. The drop is caused by the necessary descriptor assembly and additional bookkeeping required by objects that don't fit in small packets. As object size increases further, physical link bandwidth limitation causes a drop in throughput inversely proportional to the object size, for both NIQ and x520. Medium size packets' throughput is roughly constant with variations that are within one cache miss, and thus difficult to account for.

Interestingly, NIQ manifests higher throughput than the x520 card implementation. This observation implies that even though engineering for high bandwidth often compromises latency, the converse is not true; engineering for low latency is very favorable to achieving high bandwidth.

To further explore latency impact of high request loads, we plot the latency vs. load graph in Figure 8. For load generation, we model request arrivals as a Poisson

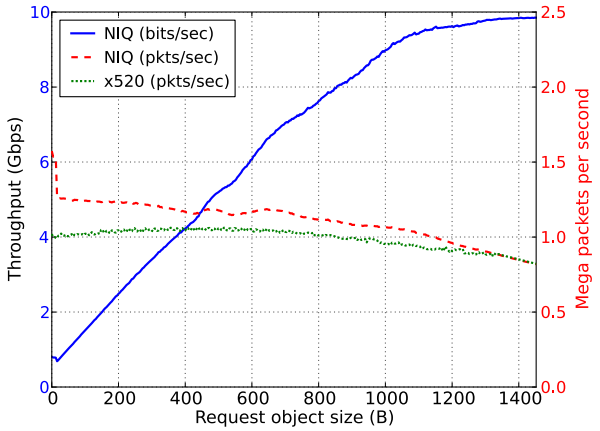


Figure 7: Server throughput results.

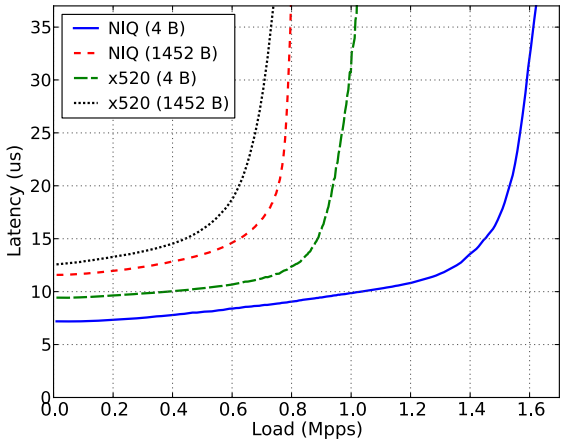


Figure 8: Server latency vs. load.

process by making the time between generated requests follow an exponential distribution. Figure 8 shows load-latency curves for minimum and maximum size objects, for both NIQ and x520 implementations. As predicted by the queueing theory, latency exponentially increases with load, and it does so in a similar fashion for all experimental setups.

5. Latency vs. Power Analysis

In this section we investigate latency and power impacts of processor power management. Our experiments have demonstrated that it is critical to properly use power management states to achieve low latency. We focus on a subset of Intel's power management states [10] that are relevant to our application, namely core idle states (c-states), package idle states (pc-states) and performance states (p-states). These states are Intel's implementation of platform independent mechanisms defined in the ACPI specification [8].

Core and package idle states lower the CPU power consumption during the idle periods at a cost of incurring

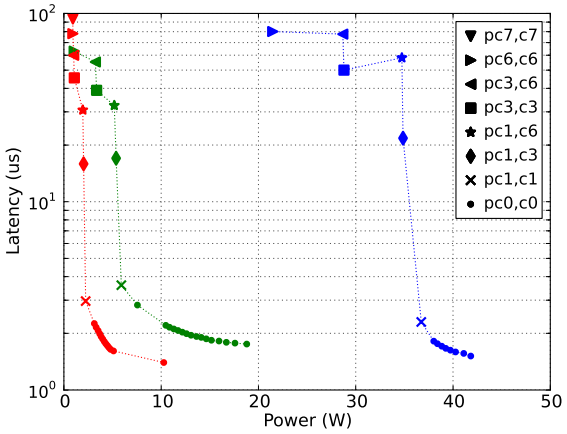


Figure 9: CPU latency-power tradeoffs for: mobile class Ivy Bridge (left/red), desktop class Sandy Bridge (middle/green) and server class Westmere (right/blue).

some wakeup latency when the CPU transitions back from idle to active. Package idle states control the power of the "uncore" logic (e.g. memory controller, shared caches, QPI links), while core idle states manage the cores (c0-active; c1-clock gated; c3-local cache flushed; c6-power gated). Wakeup latencies are on the order of 50 us for the deepest c-states, and up to 100 us for deep pc-states. Processor performance states (p-states) are active states tied to different processor clock frequencies that result from frequency-voltage scaling. Lower frequencies burn less power and incur higher execution time, but they do not add to wakeup latency because the processor can execute instructions in any p-state.

5.1. Experimental evaluation

We experimentally determine wakeup latencies and power consumption for three classes of Intel processors (mobile Ivy Bridge 3610QM, desktop Sandy Bridge 2600K and server Westmere E5620). Power is measured on the 12 volt rail that feeds into the CPU voltage regulators and thus includes any inefficiencies those regulators might introduce (typical regulator efficiency is around 85%). Wakeup latencies are measured from an FPGA board connected to a PCIe slot to avoid any instrumentation code interfering with the measurements. The FPGA generates a ping request to an idle CPU (in low power state) and measures how long it takes to receive a reply. Overall latency displayed in Figure 9, thus, includes one wakeup latency plus some small code execution time and one PCIe round trip overhead (overall overhead is less than 1 us). In this experiment the FPGA generates one ping request every 5 ms, allowing the processor plenty of time to completely reach any idle state. Each point in Figure 9 represents power and wakeup latency for a given (*package pc-state, core c-state*) pair. Power value is the processor idle power in the given idle-state pair,

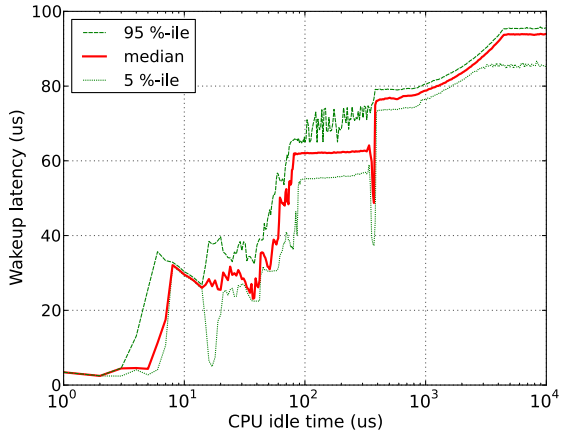


Figure 10: Wakeup latency for a mobile Ivy Bridge processor in transition from active to PC7 state.

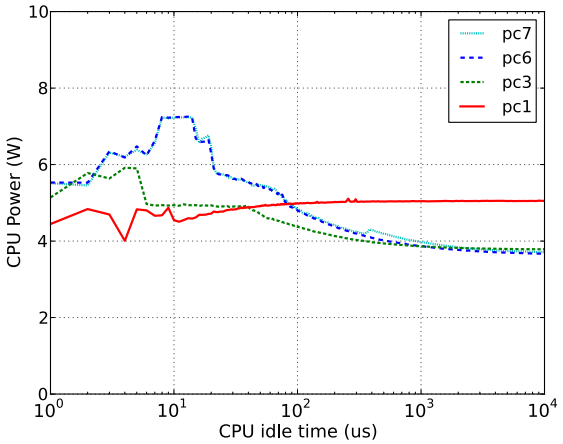


Figure 11: Average CPU power of a mobile Ivy Bridge processor in transition from active to idle state, but being interrupted after *CPU idle time*.

while wakeup latency is the time it takes the processor to exit its idle state and be ready to execute instructions. Multiple active-state points (pc0, c0) correspond to different performance states (i.e. different clock frequencies). Active states have zero wakeup latency, so (pc0, c0) points in Figure 9 show latency measurement overheads and corresponding power consumptions.

It can take several milliseconds for a processor with complex power management (e.g. mobile Ivy Bridge) to go from being active to being steadily in the deepest idle state. Since it can take milliseconds to reach a stable state, we must explore the wakeup latency and power of a processor that is in transition form active to deep idle. In Figure 10, a mobile Ivy Bridge processor is in transition from active state to pc7 (power gated cores, memory controller shut off, shared cache flushed and disabled). The figure shows how the wakeup latency depends on the actual idle time of the processor, as the processor is interrupted after *CPU idle time* into the transition from active to pc7. One can notice several plateaus corresponding to major processor components being shut down, as well as the overall complexity of the low power transition process. It is evident that the wakeup latency is lower if the transition process is interrupted early (e.g. only 4 us wakeup if interrupted 5 us into the pc7 transition). However, the proposition of frequently waking the processor before it completely reaches its intended idle state can be costly in terms of power, as shown in Figure 11. This figure shows the average power of a mobile Ivy Bridge processor that is trying to reach an idle state (pc1, pc3, pc6, pc7), but is always interrupted after *CPU idle time*. One can see from the figure that deep idle states are more efficient only if the CPU is actually idle for longer periods of time. Power numbers presented in Figure 11 were obtained from the RAPL interface registers [12] and each experiment's *CPU idle time* in Figure 10 was randomized to avoid unwanted artifacts due to periodicity.

Effects shown in Figure 10 can also affect DMA latencies on processors with integrated memory controllers. The memory controllers are put into an idle state by the processor, but a DMA read requires them to access main memory. Any delay of over 10 us between the processor starting a transition into pc6 and the DMA engine initiating a read is likely to increase DMA latency by over 10x.

5.2. Application notification performance

Upon new packet reception, the NIC generates a notification to the application that a new packet is available. This notification can be an interrupt, a write to a predefined memory location that is actively polled by the CPU, or a combination of both. The notification mechanism is at the center of power-latency tradeoffs discussed here. To achieve lower power consumption while waiting for a notification, the CPU can enter an idle state. However, if the CPU running the application is idle when the notification is received, wakeup latency is incurred, as discussed previously.

In Figure 12 we present latency and idle power measurements for interrupt and polling notifications on a desktop class system. We consider a case of using interrupts and placing the entire application code inside the kernel interrupt handler. However, this is feasible only for the simplest of cases where the application processing can be done extremely quickly. A more plausible setup is the one where the kernel interrupt handler wakes up a user process that does the bulk of the processing. We find that signaling from the interrupt handler to the user process introduces more than several microseconds of additional latency, even though both interrupt based mechanisms are equally power efficient.

The lowest latency is achieved using *mwait*/polling from inside the user space process. With this mecha-

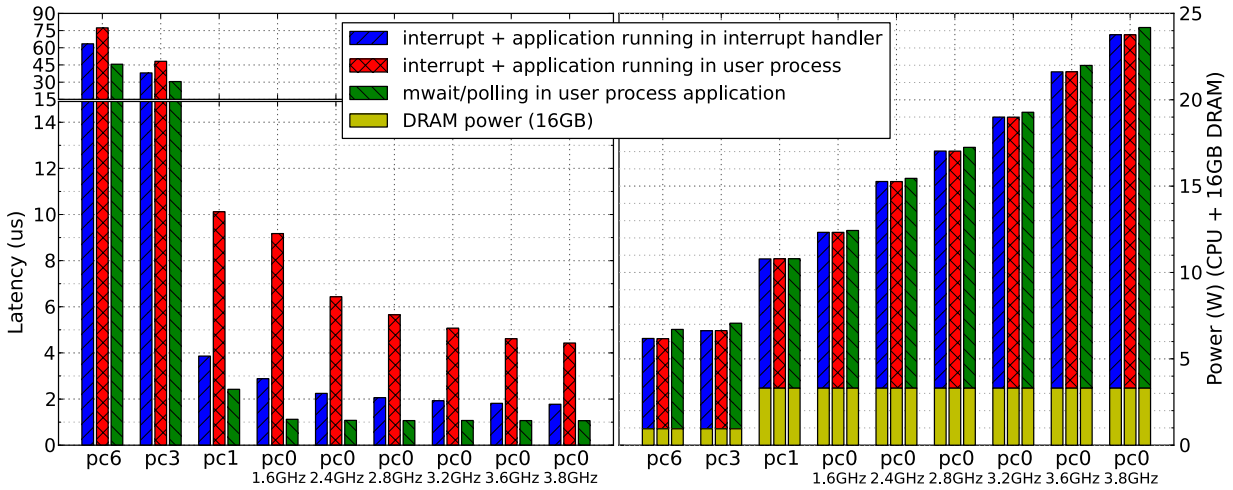


Figure 12: Latency vs. power tradeoffs of different notification mechanisms.

nism, in non-active states (pc6, pc3, pc1 in Figure 12) the user application makes calls to a kernel module to use *monitor/mwait* on a memory address that the NIQ writes to. In the active states (pc0 in Figure 12) the application polls that same memory address in a tight loop. For non-deep idle states (pc0, pc1), the *mwait*/polling approach increases power consumption by less than 2% over interrupt mechanisms, while offering more than 2-4x latency improvements. Deeper idle states (pc6, pc3) offer over 50% reduction in combined CPU and memory power over non-deep idle states (pc0, pc1), but they are unusable due to a huge latency penalty.

One disadvantage of the *mwait* approach is that the application process appears to the operating system as if it is constantly running without voluntarily yielding the CPU. This makes it impossible to take full advantage of the tickless kernel [27], resulting in marginally higher power usage in deep idle states, as shown in Figure 12 for states pc3 and pc6.

In conclusion, *mwait*/polling mechanism combined with the pc1 idle state is a viable way to save power (55% savings compared to highest performance states). However, deeper idle states incur too high of a latency penalty to be used with ultra low latency applications. While there is a power penalty to using *mwait*/polling instead of interrupts, it is not as dramatic as initially expected. We believe that the poor latency reputation of interrupts and the high power consumption reputation of polling are somewhat unjustified. While waiting for an interrupt, processors are generally allowed to enter deep idle states (e.g. pc6), but polling always keeps them busy in an active state (pc0). This means pc6-state power and latency are associated with interrupts, while pc0-state power and latency are associated with polling. However, one can select the CPU idle state for either interrupts or polling, and Figure 12 is meant to help with that choice.

6. Related Work

Engineers have been building low latency RPC systems since long before Ethernet was a dominating link layer protocol, as demonstrated in [32]. In that paper authors identify network controllers as having a significant influence on the overall RPC latency. They also express concern with the trend of NIC latencies not improving nearly as fast as their bandwidth capabilities. These trends have continued with NIC bandwidth increasing 1000-fold since then, but round trip RPC latencies improved only 10-100 times. Ironically, most of the latency improvement came from the decrease in serialization latency, which is directly tied to bandwidth (e.g. 64 byte packet at 10 Mbps has serialization latency of 51.2 us, while at 10 Gbps it is only 51.2 ns). The same authors [32] compared DMA based Ethernet controllers with FIFO-based ATM controllers. They concluded that DMA based controllers work well for large packets, but for small packets they prefer a simple FIFO interface that is directly accessed by the host. Similarly to our approach, they go on to advocate building future network controllers with a hybrid interface to best suit both small and large packets. Twenty years later, with computing shifting into datacenters, we find that many of the good ideas from the past are being adapted to fit current needs.

Even though Infiniband has traditionally been favored in high performance computing, recent work has shown that 10G Ethernet can also achieve good latency performance [24]. To complement this finding, high performance switch and NIC vendors have built 10G Ethernet switches capable of 0.3-0.5 microsecond latencies [7, 2] and network adapters capable of 1.3 microsecond latencies [20]. These proprietary components have promptly found their way into vertically integrated solutions for low latency applications, such as online trading systems [30]. We view our work as an integral part of the over-

all efforts to openly discuss, understand and build low latency datacenter systems on commodity hardware.

Researching the interactions between the cache hierarchy and I/O data transfers [31, 9, 13] has resulted in an implementation by Intel that claims to improve their network controller latency by 10-15% [13]. Previous work on power management of datacenter servers [18, 19] has been promising, but it assumes that latencies in hundreds of microseconds are acceptable. On the software research front, there are efforts that improve the average and tail latencies of existing datacenter applications, such as memcached and web search [15]. With a clean slate approach, the RAMCloud project [22] is driving the total round trip request-response latency into the 5-10 us range, applying significant efforts to reducing the software overheads. Kernel network stack overheads are typically removed by circumventing the network stack completely and accessing the network card through a user-level driver [17, 5, 28]. User-level drivers are typically hardware specific and do not provide the OS protection mechanisms that applications are accustomed to. There is ongoing work on the netmap framework [25] that provides low latency of a user level access in conjunction with the benefits of an operating system. This software research complements our work by accentuating the need for low latency network interfaces.

7. Discussion and Conclusion

Our approach in this paper is a clean slate approach where we assume the applications are written in a way to take advantage of NIQ. For an existing application to benefit from NIQ, it would need to be modified to use the NIQ user space library to send and receive packets. It is also possible to implement the NIQ driver as a kernel module, thus enabling application multiplexing, but we do not explore this option in the paper. Some of the software low latency techniques used in the NIQ implementation can also be used with most other network cards (e.g. polling, kernel bypass, header splitting). To make the evaluation comparison fair, we have implemented those techniques for both NIQ and the Intel x520 card. On the other hand, some of the techniques are unique to NIQ and require support form the driver, hardware and the host system (e.g. delayed PCIe reads, mapping NIQ memory as cacheable).

NIQ performance results presented in Section 4 are limited by the FPGA implementation. We implemented NIQ prototype on an FPGA development board utilizing available standard components, such as MAC, PHY and the PCIe core. However, those available components exhibit relatively high latencies, when compared to the lowest possible with today's technologies. We are able to extrapolate what the overall request-response latency would be, if the system is built with state of the art components. One small packet round trip through our PHY and MAC components measures at 920 ns, while the best ASIC components take only 300 ns, indicating 620 ns of possible improvement in the Ethernet path. For the PCIe estimation we take the minimum PCIe latency seen on the Intel's x520 card (560 ns) and compare it to the minimum NIQ PCIe latency (930 ns). Additionally, we measure an extra 190 ns of possible savings when running on one of the newest available server processors with an on-chip PCIe controller, for a total PCIe round trip savings of 560 ns. Since our minimum request-response latency (4.65 us) includes two Ethernet and two PCIe round trips, we infer possible request-response times of under 2.3 us, thereby doubling our performance.

We demonstrate that it is possible to fit a basic GET request within a minimum size (60 B) packet using binary object keys. Many existing object store systems use string keys instead of binary object keys, which generally makes GET requests bigger than a single cache line. One solution to this would be to use a hash on the string object keys to convert them into binary keys. However, it is also possible to extend the NIQ small packet interface to support larger packets (e.g. two cache line size packets) on receive and transmit. On the transmit side, writing a two-cache-line packet using the write-gathering memory mapping is as simple as writing two cache lines back-to-back. On the receive side, however, the change would be more extensive. One simple solution to support larger packet on the small packet interface is not to communicate complete headers between NIQ and the CPU. Instead, NIQ checks and drops redundant header fields on receive (e.g. destination mac address, type fields, etc.), thus enabling bigger Ethernet packets that still fit into one cache line. Another possible solution is to use two sibling hardware threads (hyperthreads) to issue two polling cache misses, with replies eventually ending up in the same L1 cache ready for processing. A similar solution might involve explicit prefetches (using the SSE instruction), or even automatic adjacent line prefetching, to get more than one cache line at a time from NIQ to the CPU. Finally, one might simply use the host polling technique; especially on a new SandyBridge-EP platform with DDIO [13], where it would perform well.

In its current prototype form, NIQ interface can only be accessed by a single thread at a time, thus limiting possibilities for parallelism. As future work we intend to extend the NIQ interface to support multiple queues, thus supporting multithreading and increasing performance through parallelism.

We also discuss two extreme solutions for building a low latency request-response applications (e.g. object store). One solution is to implement the entire application in the NIQ. This becomes problematic as soon as two or more memory accesses are needed, such as a hash

table lookup and a data read, because memory accesses over PCIe exhibit high latency. The second solution is to completely integrate the network card with the processor. This is an attractive option from the latency point of view, with an open question of how exactly would the integrated controller be connected to the processor. Unless the latency of that interconnect is negligible, such design would still benefit from our findings.

In conclusion, we have designed, implemented and evaluated a network interface solution for low latency request-response protocols. We demonstrate significant latency gains by focusing on minimizing the number of transitions over the PCIe interconnect, particularly for small packets. Moreover, we designed and made a case for a custom polling notification technique by evaluating its latency and power performance. We also investigated processor power management implementation and the impact it has on latency. Finally, our system's latency gains did not come at the expense of bandwidth, but there was an increase in implementation complexity.

References

[1] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND M., Y. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of USENIX NSDI conference* (2012).

[2] ARISTA NETWORKS. 7100 Series Switches: `http://www.aristanetworks.com/en/products/7100series`, July 2012.

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev. 40*, 1 (June 2012), 53–64.

[4] BROADCOM CORPORATION. Broadcom NetXtreme® 57XX User Guide, April 2012.

[5] DERI, L. ncap: wire-speed packet capture and transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on* (may 2005), pp. 47 – 55.

[6] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004).

[7] FULCRUM MICROSYSTEMS. FM4000 Switches: `http://www.fulcrummicro.com/documents/products/FM4000_Product_Brief.pdf`, July 2012.

[8] HP, INTEL, MICROSOFT, PHOENIX, TOSHIBA. ACPI: Advanced Configuration and Power Interface Specification, December 2011.

[9] HUGGAHALLI, R., IYER, R., AND TETRICK, S. Direct cache access for high bandwidth network I/O. In *Proceedings of the 32nd annual international symposium on Computer Architecture* (Washington, DC, USA, 2005), ISCA '05, IEEE Computer Society, pp. 50–59.

[10] INTEL CORPORATION. Intel and Core i7 (Nehalem) Dynamic Power Management, 2008.

[11] INTEL CORPORATION. ixgbe - Intel 10 Gigabit PCI Express Linux driver, 2010.

[12] INTEL CORPORATION. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3, April 2011.

[13] INTEL CORPORATION. Intel® Data Direct I/O Technology (Intel® DDIO): A Primer, February 2012.

[14] INTEL CORPORATION. Intel® Ethernet Converged Network Adapter X520, 2012.

[15] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 9:1–9:14.

[16] KIM, J., DALLY, W. J., SCOTT, S., AND ABTS, D. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2008), ISCA '08, IEEE Computer Society, pp. 77–88.

[17] KRASNYANSKY, M. uio-ixgbe: `https://opensource.qualcomm.com/wiki/UIO-IXGBE`, July 2012.

[18] MEISNER, D., GOLD, B. T., AND WENISCH, T. F. Powernap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 205–216.

[19] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 319–330.

[20] MELLANOX TECHNOLOGIES. ConnectX-2EN NIC: `http://www.mellanox.com/related-docs/prod_software/ConnectX-2_RDMA_RoCE.pdf`, July 2012.

[21] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15*, 3 (Aug. 1997), 217–252.

[22] OUSTERHOUT, J., ET AL. The case for RAMCloud. *Commun. ACM 54*, 7 (July 2011), 121–130.

[23] PCI SPECIAL INTEREST GROUP. *PCI Express Base Specification Revision 1.1*. PCI-SIG, 2005.

[24] RASHTI, M., AND AFSAHI, A. 10-gigabit iwarp ethernet: Comparative performance analysis with infiniband and myrinet-10g. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (Mar 2007), pp. 1 –8.

[25] RIZZO, L. Revisiting network I/O APIs: the netmap framework. *Commun. ACM 55*, 3 (Mar. 2012), 45–51.

[26] SALAH, K. To coalesce or not to coalesce. *AEU - International Journal of Electronics and Communications 61*, 4 (2007), 215 – 225.

[27] SIDDHA, S., PALLIPADI, V., AND VEN, A. Getting maximum mileage out of tickless. In *Linux Symposium* (2007), vol. 2, Citeseer, pp. 201–207.

[28] SOLARFLARE COMMUNICATIONS. OpenOnload: `http://www.openonload.org`, July 2012.

[29] STANFORD NETFPGA GROUP. NetFPGA 10G, `http://netfpga.org`, July 2012.

[30] SUBRAMONI, H., PETRINI, F., AGARWAL, V., AND PASETTO, D. Streaming, low-latency communication in on-line trading systems. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops* (2010), 1–8.

[31] TANG, D., BAO, Y., HU, W., AND CHEN, M. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *2010 IEEE 16th HPCA* (Jan 2010), pp. 1 –12.

[32] THEKKATH, C. A., AND LEVY, H. M. Limits to low-latency communication on high-speed networks. *ACM Trans. Comput. Syst. 11*, 2 (May 1993), 179–203.

DEFINED: Deterministic Execution for Interactive Control-Plane Debugging

Chia-Chi Lin[1] Virajith Jalaparti[1] Matthew Caesar[1] Jacobus Van der Merwe[2]

[1]*University of Illinois at Urbana-Champaign*
[2]*University of Utah*

Abstract

Large-scale networks are among the most complex software infrastructures in existence. Unfortunately, the extreme complexity of their basis, the control-plane software, leads to a rich variety of nondeterministic failure modes and anomalies. Research on debugging modern control-plane software has focused on designing comprehensive record and replay systems, but the large volumes of recordings often hinder the scalability of these designs. Here, we argue for a different approach. Namely, we take the position that *deterministic network execution* would vastly simplify the control-plane debugging process. This paper presents the design and implementation of *DEFINED*, a user-space substrate for *interactive debugging* that provides *deterministic execution* of networks in highly distributed and dynamic environments. We demonstrate our system's advantages by reproducing discovery of known *ordering* and *timing bugs* in popular software routing platforms, XORP and Quagga. Using Rocketfuel topologies and routing data from a Tier-1 backbone, we show *DEFINED* is practical and scalable for interactive fault diagnosis in large networks.

1 Introduction

Large-scale networks such as enterprise and ISP networks consist of a complex intertwining of systems and protocol implementations distributed over wide distances. At the basis of these networks lies the control-plane software that is responsible for controlling and managing data flows. Like other complex software systems, control-plane software is prone to defects or bugs introduced through human error. Indeed, studies have shown that control-plane traffic accounts for 95 – 99% of the observed bugs in networks [1].

One school of research has focused on applying *fully automated* techniques to analyze, test, and debug control-plane software. However, while automated techniques have been developed to localize memory faults [5] and avoid concurrency bugs [21], the larger class of *logical* or *semantic* errors seems to fundamentally require human knowledge to solve. To address this, most research has employed recording mechanisms to assist human troubleshooters in debugging large-scale control-plane software. Packet capture and replay tools such as `tcpdump` [30] record and replay packets at individual nodes. Friday [10] correlates recordings across distributed nodes to provide system-wide reproducibility for control-plane software. OFRewind [32] enables centrally controlled record and replay of control-plane software by leveraging the structure of OpenFlow controller domains. Unfortunately, while these schemes improve troubleshooter's ability to analyze control-plane software, the large volumes of recordings hinder the scalability of these systems. In fact, even the authors of Friday and OFRewind point out that a comprehensive recording of all events in an entire production network is infeasible.

Consequently, troubleshooters often enable only partial recordings in production networks, e.g., logging only packet headers or logging only at specific network locations. However, solutions based on partial recordings can fail to reproduce bugs triggered by nondeterministic behaviors, e.g., message orderings or unsynchronized clocks. Troubleshooting these nondeterministic bugs is challenging. Since a bug may happen only when certain messages arrive at a specific node in a specific ordering, if troubleshooters didn't select the node to record messages beforehand, it is difficult to reproduce the bug. Two types of nondeterministic bugs are particularly notorious in control-plane software: *ordering bugs* that appear only when certain messages occur in specific orderings and *timing bugs* that appear only when certain messages are processed at specific timings.

To address these nondeterministic bugs, in this paper, we present a new system, *DEFINED*, a debugger that allows a troubleshooter to analyze control-plane bugs after detecting erroneous behaviors of a system. *DEFINED* simplifies interactive control-plane software debugging through deterministic network execution. Namely, given the same set of external events (e.g., messages from external routers, failures of links and routers), we make every node in the network always receive messages in a deterministic ordering and timing. Accordingly, with *DEFINED*, troubleshooters can adopt partial recordings and still be able to reproduce nondeterministic bugs.

To enable deterministic network execution, *DEFINED* eliminates all sources of nondeterministic *internal* events in a network, and relies on partial recordings to record and replay nondeterministic *external* events. Specifically, *DEFINED* ensures each node receives messages and fires local timers in a deterministic fashion. To provide more debugging functionality without introducing prohibitive overheads to control-plane software, *DEFINED* consists of two components: *DEFINED-RB* (*RB*: RollBack) that instruments production networks, and *DEFINED-LS* (*LS*: LockStep) that manages debugging networks.

DEFINED-RB introduces minimum overheads to control-plane software with an "optimistic" approach: each node independently decides on a pseudorandom sequence of events, and then lets the network execute in an arbitrary fashion. If the order in which events execute is different from the pseudorandom sequence, the network is "rolled back" to an earlier state, and played forward with the correct ordering. We introduce a novel pseudorandom sequence to reduce to the number of rollbacks, and hence minimize the overheads.

DEFINED-LS provides an interactive stepping functionality to troubleshooters in a debugging network. It allows troubleshooters to investigate and manipulate state, and slowly step through the operation of individual messages. *DEFINED-LS* does so by forcing debugging networks to execute in a *lockstep* fashion. To reproduce nondeterministic bugs, *DEFINED* guarantees that *DEFINED-LS* deterministically reproduces *DEFINED-RB*'s execution.

Deterministic execution [4, 6, 12, 27] and interactive stepping [11, 13] have been widely applied in non-control-plane software to ease interactive debugging. These techniques, however, to our best knowledge, cannot operate efficiently and effectively with control-plane software. Our system, *DEFINED*, is a debugger for control-plane software that address the problem of interactive debugging in modern wide-area networks. To demonstrate the utility of *DEFINED* in assisting troubleshooters analyzing ordering and timing bugs, we use *DEFINED* to reproduce the discovery of known bugs in two popular open-source control-plane implementations, XORP and Quagga. Our evaluation on Emulab [31] with Rocketfuel topologies [28] and Tier-1 ISP traces shows that very little overhead is required to make production networks deterministic and that performance in debugging networks has sufficiently low response time for interactive use.

2 System Design

In this section, we describe the details of the design of *DEFINED*. We first give an overview of the system (Section 2.1). Then, we describe *DEFINED-RB* (Section 2.2), which instruments a production network to make its execution deterministic. We next show *DEFINED-LS* (Section 2.3), which allows a debugging network to be "stepped" through in a manner controlled by a human troubleshooter. We then conclude the section by discussing some properties and limitations of our design (Section 2.4 and Section 2.5). To keep the description concise, in this section, we focus on how our system ensures deterministic message events, and in Section 3, we will describe how *DEFINED* can be extended to provide deterministic timer events.

2.1 Interactive Network Debugging

We first clarify the benefits of a tool for interactive control-plane software debugging. Under our design, control-plane software runs on top of *DEFINED*, a user-space substrate, instead of directly on an operating system. Complementing existing log-based systems [10, 30, 32] that passively record software activities, we instrument control-plane software in a production network and actively manipulate ordering and timing of internal message receptions. The manipulation ensures network-wide execution is deterministic. When human troubleshooters observe any control-plane software bug in a production network, they can reproduce the bug deterministically in a debugging network with only partial recordings, and analyze it through the *debugging coordinator* with the interactive stepping functionality.

Our design consists of two key components, *DEFINED-RB*, which makes control-plane execution deterministic by *masking internal nondeterminism* and *DEFINED-LS*, which introduces distributed lockstep execution for interactive debugging.

***DEFINED-RB*:** Debugging a control-plane system becomes much easier if the operation of that system is deterministic. Unfortunately, existing control-plane software incorporates a high degree of internal randomness in its execution, arising from varying message orderings, delay and jitter, and other variables arising from distributed execution. To address this, our design manipulates the operation of a production network itself, to remove all internal nondeterminism and cause it to run in a deterministic manner.

Each node intercepts message and timer events before delivering them to the control-plane software, and then uses a pseudorandom ordering function to determine the exact orderings and timings at which to send the events up to the software. Instead of adopting a stop-and-wait design [12], we employ speculative execution to reduce overheads: upon each event occurrence, a node uses its pseudorandom ordering function to check whether the order in which the events appeared so far follows the computed pseudorandom sequence. If the order is the same as the pseudorandom sequence, the node delivers the event to the control-plane software. On the other

hand, if the order is different from the pseudorandom sequence, the network is "rolled back" to an earlier state, and played forward with the correct ordering. To further optimize the performance, we construct the pseudorandom ordering function according to the network topology, so that the computed pseudorandom sequence matches the event sequence that most frequently occurs. Consequently, the number of rollbacks is minimized.

***DEFINED-LS*:** The tremendous load, scale, and rates of change of modern networks make it hard for a human troubleshooter to build an understanding of the entire control-plane system's state. Debugging such a system becomes much easier if the troubleshooter has time to investigate and manipulate state, and slowly step through the operation of individual messages.

To achieve such interactive stepping of software in wide-area networks, a runtime coordinator manages execution across the distributed set of processes making up the control-plane software. This is done by logically dividing the software's execution into a series of *steps*. These steps may be chosen at various levels of granularity (per-event or per-path-change). The coordinator then runs the software in virtual time, executing it in a "lockstep" fashion across nodes (alternating between event-sending and event-processing phases). Distributed user-space substrates replay events to their local software and collect outbound events to be sent in the next cycle. Nodes use a distributed semaphore to coordinate. This approach controls execution, and deterministically reproduces the software's behavior by adopting the exact pseudorandom ordering function used by the production network.

2.2 Interfacing with Production Networks

To remove internal nondeterminism from a production network, *DEFINED-RB* uses speculative execution to ensure determinism while not significantly slowing down network execution. It does this by speculatively letting messages be sent to the software in the order in which they are received. To make sure the ordering can be reproduced, nodes in the network locally compute a pseudorandom ordering over these messages, and if messages do not arrive in the computed pseudorandom order, the node is *rolled back* to the point at which the first message arrived out of sequence, and messages are then played back in the correct order. Rolling back slows down processing, but with appropriate selection of the pseudorandom sequence, we can make rolling back rare. In particular, we design an optimized pseudorandom sequence to match the common-case ordering of events we would expect to see in the production network. Overall, we need to solve two problems: (i) we need to come up with a pseudorandom ordering that matches the common-case ordering of events; (ii) we need to perform the rollback when the predicted ordering is violated.

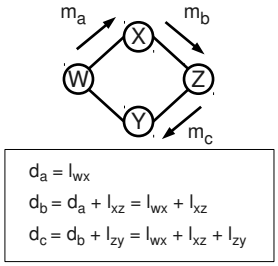


Figure 1: Example: calculating d_i.

Computing a message ordering: There are many ways to compute the pseudorandom ordering, for example using straightforward hashing and permutation. However, to ensure correctness, the pseudorandom ordering needs to maintain causal relationships between messages. In addition, every time the pseudorandom ordering diverges from the production network operation, *DEFINED-RB* requires a rollback. Hence, for efficiency reasons, we would like a pseudorandom ordering function that minimizes the number of rollbacks that are needed.

To do this, we construct an ordering function that reflects the expected ordering of message arrivals. The function takes as input a set of messages $\{m_1, m_2, \ldots, m_k\}$ received at a node n, where each message m_i is annotated with (i) n_i, the identifier of the *originating node* that generated the first message of the causal chain; (ii) s_i, a strictly increasing *sequence number* assigned by the originating node; (iii) d_i, a deterministic estimate of the delay from the originating node n_i to the local node n.

To clarify the meaning of each field, Figure 1 illustrates how *DEFINED-RB* calculates n_i, s_i, and d_i for three causally related messages. For each link (n_i, n_j), *DEFINED-RB* measures the average link delay l_{ij} before launching the control-plane software. When a node generates a message due to external events (e.g., a withdraw message when a link goes down), it is called the *originating node* of the message. The node annotates the message with n_i equal to its id, s_i equal to the current value of a strictly increasing counter, and d_i equal to the average link delay of the outgoing link. On the other hand, assume a node generates a message m_i due to another internal message m_j (e.g., a route update when receiving a message from another node in the system), where m_j is annotated with n_j, s_j, and d_j. The node annotates message m_i with n_i equal to n_j, s_i equal to s_j, and d_i equal to d_j plus the average link delay of the outgoing link. In the figure, we assume m_a is generated due to external events, while m_b is generated due to m_a, and m_c is generated due to m_b. Then, all messages have the same originating node W and sequence number. In addition, d_a equal to l_{wx}, and d_b and d_c are calculated by increasing d_a and d_b by l_{xz} and l_{zy}, respectively.[1]

[1] We use d_i to retain causal relationships by never rolling back mes-

When receiving messages from others, a node uses the ordering function to first sort the messages by d_i values. It then sorts messages with identical d_i values by their n_i values, and messages with identical n_i values with their s_i values. We sort messages by d_i before s_i, since for control-plane software, messages originating from a node can take different paths. The resulting function has three key properties: (i) it is *deterministic*, as it will always compute the same outputs given the same external events; (ii) it is *consistent*, as it retains causal relationships between messages; (iii) it is closely matched to the *common-case ordering* when originating nodes send out messages at roughly the same time, since d_i indicates the average arrival time of a message.

To further avoid long chains of rollbacks, *DEFINED-RB* makes sure the ordering function is applied independently to messages originated at roughly the same time. To do this, we divide time into distinct steps, group external events appearing in a single timestep together, and then independently impose the ordering function mentioned above on the messages corresponding to each timestep. We further bound the length of each causal chain within a timestep: messages over this bound are assigned to the next timestep. All messages in a single timestep correspond to a single group. Each group is associated with a distinct *group number*. One node is selected to periodically broadcast special packets called *beacons* which specify the group numbers to be used by the rest of the nodes in the network. (Leader election algorithms [22] are used to make sure the system can tolerate failures.) Group numbers are strictly increasing. Messages triggered by an external event are tagged with the current group number, while output messages generated due to an internal message are assigned the same group number as the internal message.[2]

In our implementation, we assign fixed values of d_i based on average link delay between nodes rather than dynamically estimating it. However, if desired, the link delay values may be periodically re-estimated, as long as they are applied and recorded at group boundaries.

Detecting if a rollback is necessary: Each node maintains a sliding window history of messages it received since the last group number update, as well as a list of messages it sent since the last group number update. The history is sorted by the ordering function. An entry in the history can be removed after all messages that might be ordered before it have arrived. Depending on the node

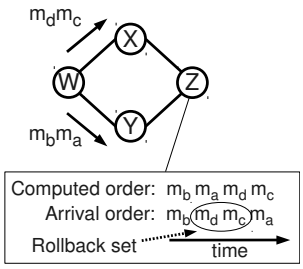


Figure 2: Example: detecting and performing rollback. In this example, we assume all messages originate from node W, and all links have the same expected delay. Thus, the order of the messages are determined by the sequence numbers. We assume messages m_b, m_a, m_d, and m_c have sequence numbers 0, 1, 2, and 3, respectively.

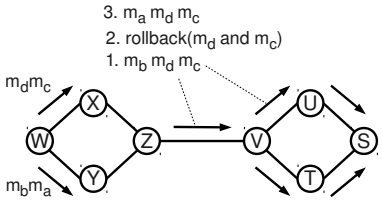


Figure 3: Example: rolling back across nodes.

performance, our experience shows that in a modern production network, we can generally remove an entry after two times the maximum propagation time across the network.[3] Whenever a node receives a new message, it passes the contents of this window to the ordering function to determine if the new message has arrived in the correct pseudorandom sequence.

If the message is received in the correct order, it is sent to the software. Otherwise, the node must roll back the node's state to the first point where the sequences diverge and replay received messages in the correct order. There are two scenarios in which a node needs to roll back its state: (i) when receiving messages that have earlier group numbers; (ii) when receiving messages that have the current group number, but don't arrive in the correct pseudorandom sequence. For example in Figure 2, if node Z receives messages in the order $\{m_b, m_d, m_c, m_a\}$ (message m_b received first), but upon receiving m_a it computes the sequence $\{m_b, m_a, m_d, m_c\}$, it would need to roll back to the point just before it received message m_d (i.e., it would need to roll back messages m_d and m_c). Note the pseudorandom ordering is computed on every message arrival, for example, here, the node would compute $\{m_b\}$ after receiving message m_b, then compute $\{m_b, m_d\}$ after receiving message m_d, then compute $\{m_b, m_d, m_c\}$ after receiving message m_c, but then compute the final sequence $\{m_b, m_a, m_d, m_c\}$ after receiving m_a.

sages with lesser d_i values. Therefore, to retain causal relationships for a message with multiple causal parents, we only need to record the largest d_i value among all its parents.

[2] A large variation in distances from nodes to the beacon source may cause unnecessary rollbacks. We can address this by dividing the network into smaller subnetworks, and applying *DEFINED-RB* to each subnetwork independently.

[3] We only need an upper bound of the maximum propagation time. In our implementation, we estimate this bound with the sum of the average propagation time and four times its standard deviation.

Performing the rollback: Finally, performing a rollback at a node may require "unsending" messages previously sent by it to its neighbors. To do this, the node keeps a history of previous messages sent within the last few group intervals. On rolling back, the node informs neighbors of the range of messages that should be rolled back. In the example in Figure 3, node Z had previously sent node V messages $\{m_b, m_d, m_c\}$. On performing the rollback to before receiving m_d, node Z tells node V to roll back the messages m_d and m_c, and then sends messages $\{m_a, m_d, m_c\}$ in the correct order. This process continues downstream: since node V had previously forwarded messages $\{m_d, m_c\}$, it must instruct node U and node T to roll them back as well. Messages at node S are rolled back in a similar manner.

To roll back misordered messages, we restore the state of the control-plane software, and if required, inform neighbors about such a rollback to ensure that all messages that are *causally related* to the rolled back messages are themselves rolled back.

2.3 Stepping through Debugging Networks

In a network instance created to support network debugging, mechanistic delays and overheads are not significant concerns. Therefore, in this environment, we introduce *DEFINED-LS* which allows interactive stepping by forcing the network to execute in a lockstep fashion. This is done by explicitly queuing messages and timer events received by a node, and playing them at coordinated intervals using a predetermined *ordering function* that is exactly the same as that used in the production network (Section 2.2) to ensure determinism. To make the network run in lockstep, our system instructs each node to cycle through two phases: a *transmission* phase and a *processing* phase. The system coordinates all nodes with a mechanism similar to a distributed semaphore to make sure they are in the same phase at the same time. To ensure determinism, *DEFINED-LS* replays partially logged external events according to the group numbers they received in the production network. In a debugging network, one group of events is replayed at a time. When all messages are output, the next group is replayed. The nodes use TCP for communication in order to ensure that messages are not lost, which is necessary for determinism.[4]

Transmission phase: In this phase, each node transmits all messages generated in the previous processing phase. That is, the node sends out messages in a *send buffer* (filled in the processing phase) and stores all messages received in a *receive buffer*. *DEFINED-LS* then uses the same ordering function used in the production network over the received messages to compute the order in which the messages are to be delivered to the application. Hence, the messages in the receive buffer are sorted in the same way as they are received in the production network, thereby ensuring the same ordering of events. This results in the debugging network reproducing the execution of the production network. To indicate readiness to transition to the processing phase, a node sends a marker packet when it has no further messages to send.

Processing phase: In this phase, each node processes all messages received during the previous transmission phase. In particular, the node sends all messages in the receive buffer up to the control-plane software, and enqueues the software's generated messages into the send buffer. The node moves to the transmission phase after the control-plane software processes all messages in the receive buffer.

2.4 System Properties

DEFINED has two provable properties: (i) *DEFINED-LS* exactly reproduces the execution of the production network instrumented by *DEFINED-RB*; (ii) even with the presence of cascading rollbacks (i.e., rollbacks across nodes), *DEFINED-RB* eventually terminates. The first property is the core of our system, as it provides reproducibility of network execution. The second property guarantees there will be no deadlocks when a production network is instrumented by *DEFINED-RB*. We present proofs of these two properties in a technical report [18].

2.5 Limitations

Supporting incremental deployment: *DEFINED* assumes control over all devices that need to be debugged. For example, when using our design to debug an Open Shortest Path First (OSPF) network, all OSPF-speaking routers should be instrumented with *DEFINED*. This may pose a challenge in environments in which the network operator can only instrument subsets the network, or needs to interface with adjacent networks not under the operator's control. Similar issues can also occur between the interactions of control plane and data plane, e.g., external rollbacks might be required when the control-plane software attempts to modify the data-plane forwarding table. To deal with these situations, *DEFINED* records inputs at interfaces with external systems. Our system can then replay these partial recordings at a later point in time to reproduce execution. In addition, we can avoid external rollbacks by employing buffers at border nodes as proposed in earlier work [14].

Inferring causality in closed-source software: Another assumption of our design is that the source code of the software is available, as our design requires the ability to infer causal relationships between incoming mes-

[4] Alternatively, we can also record these message-loss events, and replay them in the debugging network.

sages and outgoing ones. Despite this assumption, *DEFINED* is still highly useful for control-plane software developers as we will demonstrate with case studies in Section 4. In fact, it took a graduate student only one day to instrument the control-plane software in these case studies. In addition, if developers are willing to incorporate *DEFINED* in their software, their customers can still experience the benefits of deterministic network execution even when the shipped software is closed-source. Moreover, work on tracing information flow through application binaries [7] can help enable our design directly on closed-source software.

Imposing determinism on a single node: To provide deterministic network execution, *DEFINED* also needs to eliminate internal nondeterminism triggered by events on a local node (e.g., thread scheduling, memory reordering). In Section 4, we describe a specific implementation that removes internal nondeterminism triggered by local events from XORP and Quagga. Fortunately, existing works [2–4] provide more general solutions to this problem, and *DEFINED* can be combined with these works to ensure determinism of general control-plane software.

3 Implementation

To simplify deployment and operate with existing software bases, we implement *DEFINED* as a user-space "shim layer" in the form of a library consisting of function wrappers to intercept message sending, message receiving, and timer calls.

Our implementation addresses three key challenges:

Providing interfaces to mark causal relationships: As discussed in Section 2.2, *DEFINED-RB* must determine which messages *sent* by the control-plane software are causally related to messages that are *received* by the software. This information is used to determine which messages need to be "unsent" when the pseudorandom sequence is violated. Our implementation overcomes this challenge by providing interfaces for developers to tag a message with a unique identifier when it is originated, and extract the identifier from the message when it is received. With our design in Section 2.2, developers only need to mark "immediate" causal relationships between messages (causal relationships of messages that are triggered by the same external event). Then, *DEFINED* will use these immediate relationships to generate the correct annotated fields and sort messages in the correct order. When instrumenting XORP and Quagga, we track all immediate causal relationships by passing the identifier of an incoming message from message receiving functions, to message processing functions, and finally, to message sending functions. This is done by instrumenting these message related functions in the application software with an extra parameter.

Rolling back: After obtaining a pseudorandom ordering, a node needs to rollback the state of the software when the ordering is violated. To do this, nodes perform three steps: (i) check-point states between message receipts, (ii) restore a particular state, and (iii) play back messages in the given pseudorandom ordering.

To accomplish these steps, *DEFINED* employs the `fork()` system call. When a message is received, the node inserts the message into the history as described in Section 2.2 and, at the same time, checks if the pseudorandom ordering is violated. If the message arrival complies with the ordering, the node invokes the `fork()` system call. Then, a piece of shared memory is established between the parent and child processes for notifications of possible rollbacks. If the received message violates the ordering, the node uses the shared memory to instruct the process ID it wishes to roll back to. As discussed in previous literature [24], a normal `fork()` is not sufficient to ensure determinism. Specifically, *DEFINED* also saves the state of any open files and pending signals, and manipulates process and thread IDs. After restoring its state, the node plays back the received messages according to the pseudorandom ordering.

While using `fork()` may seem somewhat heavyweight, we found its overhead to be low enough in our implementation that pursuing other techniques did not seem necessary for common environments (modern OSs use copy-on-write to reduce overheads). If desired, the overhead of rollback may be reduced further, for example by only calling `fork()` for every several messages, and rolling back to the last `fork()` before the sequences diverge, or by using standard application-specific checkpointing techniques (we investigate some optimizations in Section 5).

Dealing with timers: The mechanisms described above are sufficient to reproduce message events. However, to reproduce timer events in our design, we need to ensure the rate at which the process perceives time as progressing is the same, every time the system is run. To do this we run control-plane software in *virtual time*: instead of triggering timers with the system clock, we make timers expire according to a counter that advances deterministically with respect to the message events. This enables timer events to be reproduced in our system. However, we would like to ensure that we do not substantially change behavior of the protocol when doing this. For example, consider the flap damping algorithm in Border Gateway Protocol (BGP) [23], which "holds down" unstable routes for a certain period of time. When we run flap damping in virtual time, we would like BGP to hold down routes for a similar amount of time, to avoid making the network less or more stable. To achieve this, we use a virtual time that is deterministically reproducible, yet progresses at a rate similar to "real" wall-

clock time. We do this by using a deterministic counter for virtual time that is advanced on receipt of every beacon message, with the rate of advancement between beacons equal to the configured beacon inter-arrival time. In our implementation, we broadcast one beacon message every 250 ms, corresponding to one unit of virtual time.

4 Case Studies

To demonstrate the practicality of *DEFINED*, we instrument the BGP module in XORP 0.4 and the Routing Information Protocol (RIP) module in Quagga 0.96.5 with our system. We use *DEFINED* to reproduce discovery of two known bugs in these control-plane implementations: an ordering bug in the BGP path selection process and a timing bug in the RIP timer refresh procedure. These case studies demonstrate how an operator might utilize *DEFINED* to troubleshoot control-plane bugs after observing erroneous behaviors. We then conclude this section with a discussion of our experience.

Ordering bug in XORP BGP path selection: A BGP module should select the best path among all paths it receives from its peers. To do so, it checks all valid paths against a list of rules. There are dozens of rules in the BGP path selection process, but to understand the XORP bug, we need to know only three of them. First, the selection process compares the AS path length of each path, and those with shortest AS path length are selected as preferable paths. Then, these preferable paths are grouped by the neighboring AS of each path. Within each group, paths with lowest multi-exit discriminator (MED) are selected. All selected paths are checked against the last rule that compares the interior gateway protocol (IGP) distance of the paths. Finally, the path with the lowest IGP distance is selected as the best path.

One peculiar aspect of this process is the MED rule. Because the rule checks the MED attribute only within a group of paths that have the same neighboring AS, it creates a non-transitive ordering among paths. For example, as illustrated in Figure 4, an AS with three routers $R1$, $R2$, and $R3$ peers with another two ASs at external routers $ER1$, $ER2$, and $ER3$. These external routers advertise three paths p_1, p_2, and p_3. Through neighboring routers $R1$ and $R2$, these paths eventually arrive at $R3$. All three paths have the same AS path length, while p_1 and p_2 have the same neighboring AS. In addition, p_1 has a MED attribute of 10, p_2 has 5, and p_3 has 20. Finally, p_1 has an IGP distance of 10, p_2 has 30, and p_3 has 20. Under these settings, when $R3$ considers only a pair of paths each time, p_2 wins over p_1, p_3 wins over p_2, but p_1 wins over p_3. Thus, to avoid choosing a less preferable path, a BGP module on $R3$ should compare all valid paths whenever the process is executed and select p_3 as the best path.

Version 0.4 of the XORP BGP module, however, makes a mistake here. When receiving an incoming path, it only compares the path with the current best path. As a result, the outcome of the selection process implementation can differ across executions: if the ordering of incoming paths at $R3$ is p_1, p_2, and p_3, then p_3 is selected as the best path; unfortunately, if the ordering of the incoming path is p_1, p_3, and p_2, then p_2 is incorrectly selected as the best path.

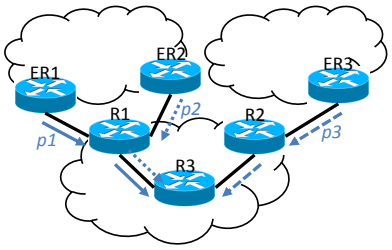


Figure 4: An illustration of a known bug in the BGP module of XORP 0.4.

Using only partial recordings on border routers and *gdb* to troubleshoot this bug, an operator enables logging for both external and internal network nondeterminism at $R1$ and $R2$ in Figure 4. When the bug is triggered, the operator replays log contents to reproduce the bug within a debugging network. However, because internal nondeterminism is recorded only at border routers, the set of paths can still reach router $R3$ in a nondeterministic fashion. The operator faces complications when experimenting with execution in the debugging network due to the inability to mirror behavior of the production network.

To address this, we use *DEFINED* to troubleshoot this XORP bug. We first use six machines to emulate the network depicted in Figure 4 and load them with the version of XORP containing the bug. We intercept nondeterministic system calls from XORP to remove internal nondeterminism triggered by local events. We then run the production network until the bug occurs. During the process, we are only required to enable partial recordings of external events at $R1$ and $R2$ but not recordings of internal events. Upon identifying the bug, we then activate *DEFINED-LS* in the debugging network. Since our system ensures that execution of both these networks match precisely, when we replay the logged external events and run the debugging network, the bug immediately occurs. We then use *DEFINED-LS*'s stepping functionality, to find the exact point at which XORP begins behaving incorrectly. After understanding the bug, we implement a patch for XORP and validate it in the debugging network. Finally, we install the patch in the production network. Deterministic execution again guarantees that all workarounds we create in the debugging network will behave the same way in the production network.

Timing bug in Quagga RIP timer refresh: To handle network dynamics, RIP maintains a timer for each route

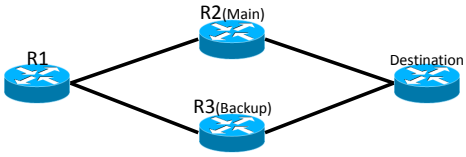


Figure 5: An illustration of the known timing bug in the RIP module of Quagga 0.96.5.

in its routing table. When receiving route announcements, if the route is already in the routing table, RIP updates the timer for the route. When a timer expires, RIP removes the route from the routing table. This mechanism ensures that the routing table contains only valid paths.

One subtle point of this process is that when comparing a route announcement with a route in the routing table, RIP must check both the destination field as well as the next-hop field. The Quagga RIP module, however, makes a mistake by only considering the destination field. As a result, the implementation contains a timing bug that triggers a black hole when certain route announcements are received at particular timings.

As illustrated in Figure 5, a router R_1 connects to two other routers R_2 and R_3. Both R_2 and R_3 provide R_1 routes to the same destination, and R_2 serves as the main router, while R_3 is the backup. All routers are running RIP, so what should happen is that R_1 maintains the route through R_2 in its routing table and refreshes the timer only when receiving announcements from R_2. When R_2 goes down, due to the lack of periodic announcement, the timer for the route will eventually time out. Then, R_1 will remove the route through R_2 from its routing table and pick up the route through R_3.

However, because the RIP implementation in Quagga 0.96.5 checks only the destination field when comparing announcements with routes in the routing table, R_1 will refresh the timer for the route through R_2 when receiving announcements from not only R_2 but also R_3. In this case, when R_2 goes down, two scenarios can happen. If announcements from R_3 reach R_1 after the route through R_2 times out, then R_1 correctly picks up the new route through R_3. Unfortunately, if announcements from R_3 reach R_1 before the route through R_2 times out, then R_1 will incorrectly refreshes the route through R_2 in the routing table. Even worse, the periodic announcements from R_3 will keep the invalid route through R_2 in the routing table and effectively create a black hole.

Using partial recordings of only message events and *gdb* to troubleshoot this bug can take a lot of time and resources, due to nondeterministic timer events embedded in control-plane software. For example, when using *gdb*, a human troubleshooter will experience timers going off unexpectedly while stepping through one instance of the Quagga RIP module on one of the routers. Moreover, to be able to reproduce the bug, it is also challenging for the human troubleshooter to manually coordinate the timing of message receipt and timer expirations.

We use four machines to emulate the network in Figure 5 and load them with the version of Quagga containing the bug. Fortunately, the same approach we used to troubleshoot the BGP path selection bug can address the RIP timer refresh bug, since timing events were also triggered deterministically in networks instrumented by *DEFINED*. As a result, during the debugging process, timers will not go off unexpectedly even when we step through the network execution at different paces.

Discussion: As shown in these case studies, *DEFINED* actively manipulates the ordering and timing of internal network events, and it makes control-plane software easier to test and debug. Another property that comes with the active manipulation, however, is that some network execution paths will never occur, and hence, some bugs will never appear in an instrumented network. For example, as we were debugging the XORP bug, we noticed that if the ordering function in *DEFINED* sorted the paths in the order of p_1, p_2, and p_3, the bug would not happen in the production network nor in the debugging network. This property, though, still protects instrumented networks from the bug, since the deterministic network execution guarantees that the bug will never appear.[5] Nevertheless, a troubleshooter may choose to not instrument the production network with *DEFINED-RB*, but to still leverage the interactive stepping functionality of *DEFINED-LS*. Fortunately, we can apply different ordering functions in *DEFINED-LS*, and then we will be able to examine all possible execution paths in the debugging network.

5 Evaluation

While *DEFINED* simplifies the task of control-plane debugging, it comes with several costs. In order to measure this overhead, we leverage Emulab [31] and take a two-pronged approach. First, to evaluate the performance of *DEFINED* in a practical setting, we perform experiments using topologies from Rocketfuel [28] and traces from a Tier-1 ISP (Section 5.2). Then, to study scalability of our system, we present results under a wide range of topologies and workloads (Section 5.3).

5.1 Methodology

We first give an overview of our experimental approach:

Topologies and traces: To improve the realism of our evaluation, we leverage topologies measured with Rocketfuel and OSPF traces collected at a Tier-1 ISP network.

[5] On the other hand, it is possible that *DEFINED* avoids some network execution paths with particular performance characteristics. In this case, an operator can modify the ordering function to force such paths to occur, potentially trading performance for more rollbacks.

We use PoP-level topologies from Rocketfuel including Sprintlink (43 nodes), Ebone (25 nodes), and Level3 (52 nodes). (Results from these topologies are similar, so we only present Sprintlink results due to space constraints.) Then, the OSPF traces are collected from a Tier-1 ISP area 0 network consisting of 324 nodes during a 2 week period (November 1st to 14th, 2009), resulting in 651 OSPF network events. We post-process these traces to reproduce the network dynamics over time, and then replay this workload in our experiments by randomly mapping events onto Rocketfuel topologies. Finally, to investigate the performance of *DEFINED* at scale, and over a wider range of topologies, we consider synthetic graphs constructed by the BRITE topology generator. Overall, we focus most of our experiments on intra-domain routing as opposed to inter-domain routing, as the lower propagation delays and tighter requirements on fast reaction make our overheads more visible. Unless otherwise specified, we run our implementation with the XORP OSPF router daemon, version 1.6.

Metrics: Since *DEFINED-RB* and *DEFINED-LS* have different purposes, we are concerned with different "success metrics" for each. As *DEFINED-RB* instruments a production network, we measure its *control*, *delay*, and *memory* overheads. On the other hand, as *DEFINED-LS* is designed for use in a debugging network for interactive stepping, overheads are of less concern (though retain some importance). Hence, we measure its response time for user-driven commands (e.g., a step command).

5.2 Performance

To characterize the performance overheads of *DEFINED*, we replay network traces against our implementation deployed on Emulab. We evaluate the design on several scales. First, we collect *network-level* results on our implementation in the Rocketfuel Sprintlink topology. We then gather *node-level* microbenchmarks to uncover the sources of bottlenecks in our implementation.

Network-wide experiments: First, we replay the Tier-1 ISP workload against our XORP-based implementation and measure the control overhead per node, for each event in the trace. Figure 6a shows that a small number of nodes experience more control overhead than others, as the rollback procedure requires additional control packets to be exchanged between nodes. Fortunately, in all cases, the percentage of these nodes is less than 1%.

Then, we measure the time for the network to converge (the time from when a failure is detected, to when all nodes are updated with their correct routing state). To stress our design, we reduce XORP's *hello* and *retransmit* intervals to be as small as possible (1 second). We compare against an unmodified XORP implementation. We found no statistically significant difference between the two. However, to improve stability, XORP's default OSPF configuration introduces a 1-second delay between when routing messages are received and when they are propagated on (due to the retransmit timer). To investigate whether this delay was the reason why our performance overheads could not be seen, we modified the XORP code to eliminate this 1-second delay. After doing this, the delays became more apparent: Figure 6b shows the network-wide convergence time is still close between the two, but our implementation has additional delay in a small number of cases, resulting in a longer tail. In addition, this figure also demonstrates that our technique in imposing local determinism on control-plane software (Section 4) has negligible overhead.

Finally, we measure the response time of *DEFINED-LS*, as it is designed to support *interactive* debugging and should respond quickly to commands from the human troubleshooter. Figure 6c shows the cumulative distribution function of the response time to execute a single *step* command of *DEFINED-LS* (where a single step is measured as the time to complete a transmission phase and a processing phase as described in Section 2.3). In this scenario every step requires less than a second.

Single-node experiments: To investigate the source of the tail, we instrumented a single node of our implementation to collect microbenchmarks. We compare XORP running under *DEFINED-RB* with an unmodified instance of XORP. In particular, we measure the amount of time required to perform a rollback (Figure 7a), as well as the time required to process packets without rollbacks (Figure 7b). We found that rollback code was triggered rarely, but, as expected, when it was triggered, it introduced overhead. To reduce the rollback overhead, instead of using `fork()` calls as described in Section 3 (FK), we manually intercepted memory writes (MI) using */proc/<pid>/mem* to directly access the memory of the process to emulate application-specific memory management, and measured the overhead to only copy changed bytes between the processes.[6] With this optimization, the median overhead for rollback is reduced to around 0.6 ms (Figure 7a), making non-rollback overhead the bottleneck. The variance observed in this figure comes from the variance of `fork()` calls and the number of events to be rolled back. Note that even the unoptimized implementation of rolling back may be tolerable for certain protocols, e.g., BGP uses a timer to intentionally slow convergence for scalability purposes.[7]

To reduce the non-rollback overhead, we investigate two optimizations. First, we try *pre-forking* (PF): instead of performing the fork when the new packet arrives (TF),

[6] We use this optimization to identify the optimal bound of rollbacks. It is not necessary for a system to do so to adopt *DEFINED*.

[7] The MRAI timer determines the minimum time between advertisements of routes to a particular destination from a single BGP device.

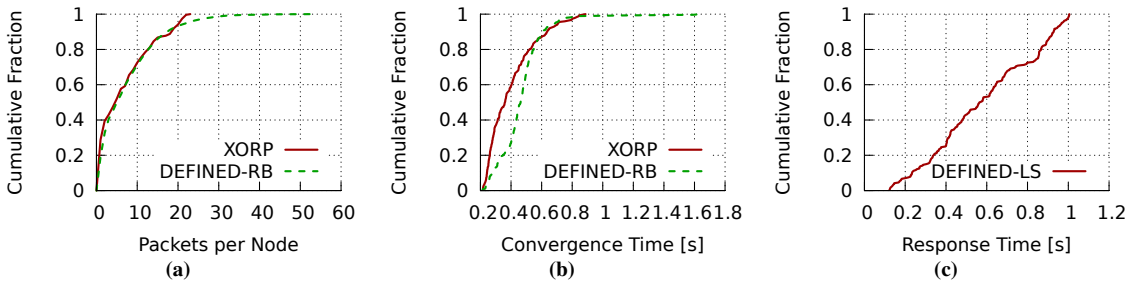


Figure 6: Network-level results of Sprintlink topology with Tier-1 OSPF traces: (a) control message overheads of *DEFINED-RB*; (b) delay of *DEFINED-RB*; (c) response time of *DEFINED-LS*.

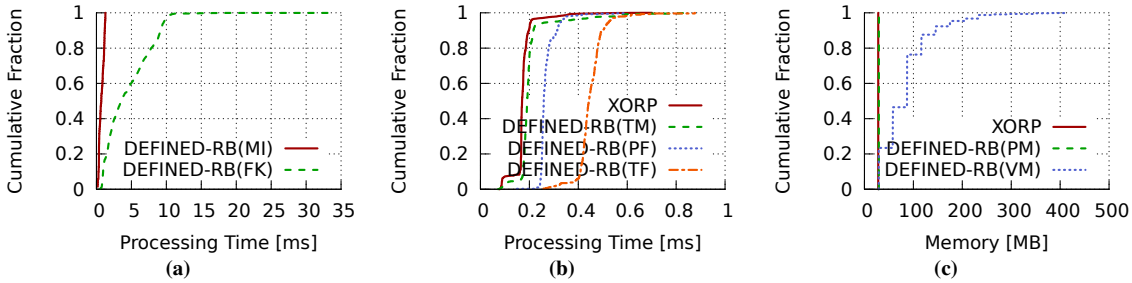


Figure 7: Node-level results of Sprintlink topology with Tier-1 OSPF traces: (a) rollback overhead, (b) non-rollback overhead, and (c) memory overhead of *DEFINED-RB*.

we perform the fork after the packet is processed (to prepare for the next packet). This causes forking to be performed during idle cycles. However, this does not completely remove the forking overhead, as due to copy-on-write, the memory copy associated with the fork is still delayed until the next packet is received. Hence, as a heuristic, we overload `malloc()` to manually touch memory (TM) on the heap when performing the pre-fork. This improves performance further (Figure 7b).

Finally, to achieve its benefits, our approach also incurs some additional memory overhead. Figure 7c shows the amount of virtual memory allocated to each process (VM). We find it increases linearly with the number of forked processes. However, some of this memory is not instantiated in practice due to page sharing. To measure the precise amount of physical memory allocated, we monitor memory writes in */proc/<pid>/mem* in Linux (PM), and plot the memory actually instantiated by the process. Since these processes share the vast majority of memory contents, the amount of memory inflation is small (less than 2% during the entire run).

5.3 Scalability

To investigate the performance of *DEFINED* at scale, and over a wide variety of workloads, we leverage BRITE topologies and synthetic events to investigate the sensitivity of our results to network size and event rate.[8]

Control overhead: We first measure the control overhead with BRITE topologies of varying sizes (Figure 8a). We found that the delay-sensitive pseudorandom ordering optimization described in Section 2.2 (OO) significantly reduces the number of rollbacks (and hence message overhead) of *DEFINED-RB* compared to random orderings (RO). Regardless of the network size, each node only needs to process at most 2 additional packets on average when using the optimized ordering (compared to the unmodified XORP instance).

Delay overhead: Figure 8b shows the network-wide convergence time of *DEFINED-RB* compared to the unmodified XORP instance. Overall, we find that while *DEFINED-RB* has a longer tail in its convergence time distribution (Figure 6b), the average convergence time between the two instances is comparable. In addition, the optimized ordering (OO) again outperforms random orderings (RO).

Response time: To evaluate *DEFINED-LS*, we measure how its response time scales with network size. Figure 8c shows that while the delay of *DEFINED-LS* increases with network size, it increases slowly. In addition, even when the network size grows to 80 nodes, the average delay remains below 0.8 seconds.

Event rates: Finally, to investigate how *DEFINED-RB* scales with event rates, we vary the number of events per second and measure the convergence time. Figure 8d il-

[8] Based on the nature of the bug, debugging can become difficult extremely fast as the network size increases. For nondeterministic bugs, a dozen nodes can already make debugging difficult.

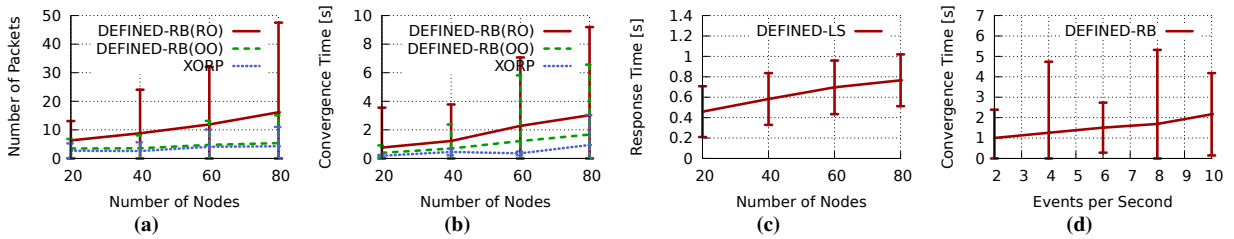


Figure 8: Scalability over network size: (a) control message overheads, and (b) delay of *DEFINED-RB*; (c) response time per step of *DEFINED-LS*. Scalability over event rate: (d) convergence time of *DEFINED-RB*.

lustrates that the convergence time increases slowly as the number of events per second increases, and the average convergence time is only a little bit over 2 seconds when there are 10 events per second. This event rate can easily cover all scenarios we observe in the Tier-1 traces. Nevertheless, when dealing with a higher rate of events, *DEFINED-RB* can decrease its beacon intervals to reduce the number of rollbacks and provide better scalability (as described in Section 2.2).

6 Related Work

DEFINED builds upon existing works and provide new primitives to support debugging of control-plane software in large-scale networks. We leverage works on distributed algorithms [14, 22] to construct the foundations of our design. Our work builds on two key areas:

Deterministic execution: DDOS [12] is the closest work to our design. Similar to *DEFINED*, DDOS introduces deterministic network execution by manipulating message orderings. DDOS runs the distributed software in virtual time, annotates each message with a virtual timestamp, and orders the messages by the source nodes' predefined identification numbers. When the distributed software tries to read a message from the network, DDOS blocks the read request until the correct message arrives. While DDOS provides deterministic network execution to general software, the blocked reads introduced by the algorithm can slow down software that requires constant communications, such as control-plane software. *DEFINED* improves the software's performance in a production network by leveraging speculative execution and introducing an innovative message ordering that minimizes the number of rollbacks.

Jefferson introduced the concept of Virtual Time [14] to provide synchronization for distributed software. Virtual Time is used to determine the ordering of messages, and rollbacks are used to make sure that messages are indeed processed in that order. However, the concept of Virtual Time cannot directly and efficiently be generalized to all software. In *DEFINED*, the message ordering that uses group numbers and estimated delays solidifies and optimizes the Virtual Time idea in the context of control-plane software.

Mechanisms enabling deterministic execution of parallel programs have long been the focus of extensive research. DPJ [6], Dthreads [19], Kendo [25], Tern [8], Determinator [2], and dOS [4] have focused on providing deterministic execution of parallel software with different approaches. DPJ supports determinism at the language level, which ensures more control over the software, but sacrifices generality. On the other hand, Dthreads, Kendo, and Tern offer determinism at the library level, and Determinator and dOS provide determinism at the OS level. These designs allow the system to handle a wider range of software. *DEFINED* takes a step further and guarantees deterministic execution of distributed control-plane software. We leverage a user-space library design, which allows us to support a wide range of control-plane software.

Instead of providing deterministic execution, several works such as Flashback [29], Friday [10], OFRewind [32], Pip [26], ReVirt [9], TTVM [16], and WiDS Checker [20] use comprehensive recordings to ensure reproducibility of execution. However, as the authors of Friday and OFRewind point out, the large storage requirements for logs are one of the limitations of these works. This limitation hinders these works from scaling to large systems, because processing a large amount of logs is prohibitively expensive. Our work targets large-scale networks, where maintaining comprehensive logs may not be tractable.

Finally, *DEFINED* leverages speculative execution, which has been previously used in many systems, for example databases [15] and multi-processor environments [17]. Our work studies the applicability and efficiency of such speculative techniques in large-scale networks. We, further, give several optimizations to reduce the overhead of rollbacks.

Interactive control: *DEFINED* not only ensures that a production network executes in a reproducible fashion, but also enables the network operators to control the execution in a debugging network. Interactive control has been used previously in several works. PDB [13] combines *gdb* with another tool, DISH, to interactively launch, manage, and troubleshoot distributed processes. The effect is similar to using multiple *gdb*

instances to troubleshoot multiple processes simultaneously. Similarly, Clairvoyant [33] supports source-level troubleshooting in wireless sensor networks by binding one *gdb* instance to each node. ndb [11] leverages the OpenFlow architecture to provide debugging primitives to software in Software-Defined Networks. Our work complements these techniques by introducing interactive debugging primitives targeting large-scale control-plane software and enabling deterministic network execution.

7 Conclusion

The high complexity of large-scale networks coupled with the rich variety of faults they undergo will require humans to be "in-the-loop" to diagnose complex problems for the foreseeable future. To address this, we proposed techniques for interactive debugging of control-plane software. We specifically addressed two key challenges, namely, deterministic network execution and interactive stepping. Our solution draws from previous work and also proposes new algorithms. We validated our work through a user-space "shim-layer" implementation and extensive evaluation using topologies from Rocketfuel and traces from a Tier-1 ISP. Our results show the practical feasibility and scalability of our approach. Specifically, we leveraged our system to reproduce discovery of known bugs in XORP and Quagga, and showed its benefits over the common debugging method that uses partial recordings and *gdb*.

Acknowledgements

We would like to thank our shepherd Meg Walraed-Sullivan for insightful comments and suggestions. We would also like to thank anonymous reviewers for valuable feedback. This work was supported by NSF CNS-10-40391 and DARPA MRC.

References

[1] G. Altekar and I. Stoica. Focus Replay Debugging Effort on the Control Plane. In *HotDep*, 2010.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, 2004.

[8] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable Deterministic Multithreading through Schedule Memoization. In *OSDI*, 2010.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[10] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI*, 2007.

[11] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the Debugger for my Software-Defined Network? In *HotSDN*, 2012.

[12] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. DDOS: Taming Nondeterminism in Distributed Systems. In *ASPLOS*, 2013.

[13] IBM. PDB parallel debugger. `http://www-03.ibm.com/systems/software/parallel/index.html`.

[14] D. R. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[15] D. R. Jefferson and A. Motro. The Time Warp Mechanism for Database Concurrency Control. In *ICDE*, 1986.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC*, 2005.

[17] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, 2010.

[18] C.-C. Lin, V. Jalaparti, M. Caesar, and J. Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. Technical report, UIUC, 2013.

[19] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, 2011.

[20] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[21] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[22] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[23] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route Flap Damping Exacerbates Internet Routing Convergence. In *SIGCOMM*, 2002.

[24] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[26] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[27] S. R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *DSN*, 2006.

[28] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM*, 2002.

[29] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A LightWeight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*, 2004.

[30] The tcpdump Team. tcpdump. `http://www.tcpdump.org/`.

[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

[32] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.

[33] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks. In *SENSYS*, 2007.

Improving Server Application Performance via Pure TCP ACK Receive Optimization

Michael Chan
Department of Computer Science
Stanford University
mcfchan@stanford.edu

David R. Cheriton
Department of Computer Science
Stanford University
cheriton@cs.stanford.edu

Abstract

Network stack performance is critical to server scalability and user-perceived application experience. Per-packet overhead is a major bottleneck in scaling network I/O. While much effort is expended on reducing per-packet overhead for data-carrying packets, small control packets such as pure TCP ACKs have received relatively scarce attention. In this paper, we show that ACK receive processing can consume up to 20% cycles in server applications. We propose a simple kernel-level optimization which reduces this overhead through fewer memory allocations and a simplified code path. Using this technique, we demonstrate cycles savings of 15% in a Web application, and 33% throughput improvement in reliable multicast.

1 Introduction

Performance of the endhost networking stack is critical to server scalability and perceived user application experience. Increases in network link speeds raise concerns over CPU utilization in keeping up with wireline data rates. This has led to various techniques such as TCP segmentation offloading (TSO) and generic receive offloading (GRO), which reduce overhead for packets carrying application payload. In contrast, relatively little attention has been given to offloading processing of small control packets such as pure TCP ACKs. (We define a pure ACK as a TCP segment which does not contain any payload, only has the ACK flag set and, if options are present, has only the timestamp option.) While control packets represent a minor portion of network bandwidth, the *number* of such packets received by a server can far outweigh that of packets containing application payload.

In a Web video streaming application, the client sends a small HTTP request to the server. The server responds with megabytes of video data encapsulated in TCP data segments. Software updates, patches and service deployment in datacenters require regular large-scale file distribution to thousands of machines. In return, the data source receives mostly pure ACKs.

Receive-side ACK processing predominantly incurs per-packet overhead, namely the cost of per-packet interactions between the driver and NIC, traversing the network stack and protocol processing. We find that per-packet overhead is significant — 20% CPU cycles of a video-chunk serving workload are expended on processing received packets, 99% of which are pure ACKs.

We propose a *network fastpath* architecture for processing small control packets. The fastpath interface provides an entry point to a significantly simplified networking stack for light-weight protocol processing. In optimizing pure ACK processing, the key insight is that ACKs convey only control metadata to the associated TCP socket, so they need not be delivered as packets. With fastpath processing, pure ACK header values are extracted from received packets and delivered directly to the TCP layer. This contrasts with conventional processing, in which the received packet is encapsulated in a packet buffer[1] and then passed up the stack. Fastpath processing allows packet buffers to be recycled for DMA, reducing memory operations. Moreover, bypassing the bulk of the conventional network stack reduces the number of CPU cycles expended.

We implemented one instance of a fastpath optimization for receive processing of pure ACKs, named TCP-PARO (Pure ACK Receive Optimization), in a recent Linux kernel. We show that it achieves real benefits for server workloads. TCP-PARO lowers application overhead by saving CPU cycles. By reducing per-ACK processing latency, TCP-PARO also improves the throughput achievable by reliable multicast.

[1] For example, *sk_buff* in Linux, commonly referred to as SKB.

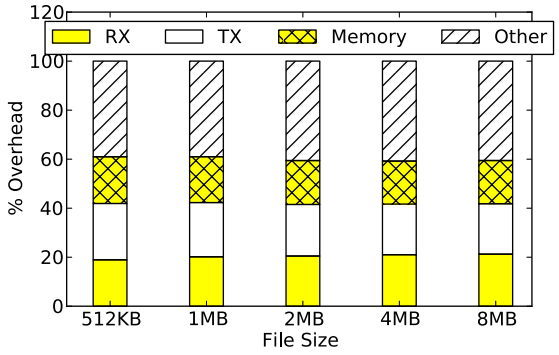


Figure 1: Breakdown of CPU cycles per Web request.

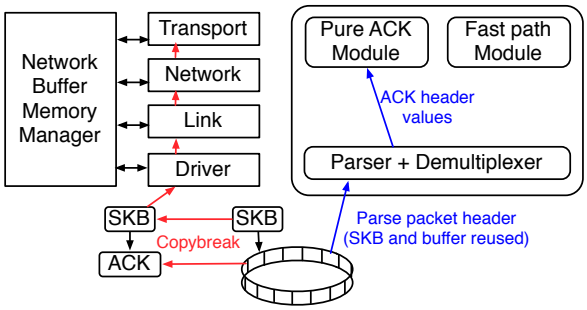


Figure 2: Network fastpath architecture.

2 ACK Processing Overhead Analysis

We quantify network processing overhead with a video chunk serving workload and show that a significant portion of the overhead is due to receive-side network stack processing of pure ACKs.

The tests are run on a testbed with two machines. Each machine is equipped with a quad-core Intel Nehalem processor, 4GB RAM, and a Neterion X3210 10GE adapter. One machine runs the nginx Web server. The 4 NIC hardware DMA engines are bound on 4 different CPU cores, and 4 nginx worker processes are assigned to the cores. This configuration is sufficient to saturate the 10GE link with the generated HTTP traffic. The other machine simulates multiple Web clients using curl-loader. 400 clients are run over 8 CPU threads. Each client requests a random file from a pool of 100 files from the server, receives the file to completion and then repeats. We perform system-wide profiling with oprofile on the server.

Figure 1 presents a breakdown of CPU cycles spent per Web request across various file sizes. The measurements are grouped by system components, where *RX* and *TX* represent receive-side and transmit-side network stack overhead, and *Memory* represents memory operations, including SKB allocations. The figure shows that 20% cycles are spent on network receive processing. Moreover, memory and SKB operations account for 18% of all cycles. The receive processing overhead is significant, considering that the server is mostly sending data, and the bulk of actual application payload is processed by the TX path. However, the receive overhead is comparable to transmit overhead, and is even bigger when serving 4MB and 8MB files. We found that pure ACKs comprised more than 99% of all received packets for all file sizes, thus network receive overhead is indeed dominated by pure ACK processing.

3 Pure ACK Receive Optimization with Network Fastpath

We propose a *network fastpath* architecture which provides efficient packet parsing, packet demultiplexing, and light-weight protocol processing. We describe its design and implementation in Linux, and optimize pure ACK receive processing with this architecture.

3.1 Fastpath Design and Implementation

Figure 2 shows the overall structure of the network fastpath architecture. The fastpath is a light-weight, parallel stack with optimized components for packet parsing, demultiplexing and protocol processing.

The fastpath adopts a modular design. The parser and demultiplexer module reads and parses a packet buffer from the receive-side DMA ring. It serves as a single entry point for network device drivers to deliver packets into the fastpath stack. Based on header fields obtained from the packet, the demultiplexer looks up the fastpath processing module and the socket for which the packet is destined. The fastpath processing module is then invoked on the socket with relevant fields from the received packet.

The fastpath provides a simple, single-function API to device drivers:

```
enum net_fastpath_verdict
net_fastpath_in(struct sk_buff *skb, u32 flags)
```

A verdict is returned to the driver to indicate processing outcomes. If the fastpath is able to process the packet, the *CONSUMED* verdict is returned. Otherwise, the *FALLBACK* verdict is returned. The driver RX routine is patched as follows:

```
if (net_fastpath_in(skb, flags) == CONSUMED)
    goto next_skb;
else
    rx_with_original_stack(skb);
```

Fastpath informs the driver if the packet has been processed. If not, the original stack is the fallback. This allows gradual adoption of light-weight processing for additional packet types without sacrificing protocol support.

Fastpath processing has two main advantages over traditional stack processing. First, the fastpath parses a given SKB, extracts required data and leaves the SKB untouched. Therefore, the SKB and the associated packet buffer can be reused for DMA. This reduces memory allocations. In contrast, existing receive processing clones both the packet buffer and the SKB to ensure isolation of the buffer from concurrent DMA writes by the NIC; Second, the fastpath stack bypasses much of the original stack, and hence reduces cycles expended per packet.

Synchronizing with other kernel threads: Packet receive processing is performed in the soft-interrupt context, which runs concurrently with other kernel threads. A per-socket mutex is held by a concurrent thread accessing the socket. When that thread releases the mutex, it processes received packets in the per-socket backlog. Fastpath similarly introduces a per-packet backlog ring. Parsed header fields are deposited into ring entries. When the mutex is released, the ring entries is flushed. The ring is statically allocated on socket creation. If the ring is full, the earliest entry is overwritten. In our experiments, we found that a ring size of 8 was sufficient. Because each entry is only 32 bytes, the ring adds a modest 256 bytes to socket size.

We note that network taps and Netfilter do not work for pure ACKs when TCP-PARO is used. However, TCP-PARO can also be disabled via *sysfs* should debugging with traditional tools like *tcpdump* be required. We believe this is a reasonable tradeoff between performance and functionality. Moreover, it would be feasible to interface the fastpath stack with existing debugging facilities, though this is beyond the scope of the paper.

3.2 Pure ACK Receive Optimization

We implement TCP-PARO (**P**ure **A**CK **R**eceive **O**ptimization) on top of the fastpath architecture.

Pure ACK parsing and demultiplexing: The parser identifies packets as pure TCP ACKs, extracts the salient header fields and looks up the TCP socket associated with the ACK using the source and destination address/-port pairs. The ACK processing fastpath module is invoked with 5 header fields from the packet — TCP sequence number, ACK number, receiver-advertised window and two timestamp option values. Only pure TCP ACKs are delivered to the processing module.

File size	Copybreak Cycles	PARO Cycles	Savings (%)
512KB	0.80	0.69	14.0
1MB	1.36	1.16	14.7
2MB	2.40	2.04	15.3
4MB	4.60	3.86	16.1
8MB	9.10	7.58	16.7

Table 1: Million cycles per HTTP request.

ACK processing fastpath module: This component quickly processes a pure ACK at the TCP socket level. All link and IP layer processing are omitted, because the ACK has already been parsed and demultiplexed. ACK processing consists of updating the RTT estimate for the connection, removing acknowledged segments from the retransmission queue, performing congestion control and transmitting new segments.

The NIC driver delivers to the fastpath packets that have (1) passed IP checksum test, and (2) have the correct length. In our prototype, a packet is a potential pure ACK if its length is either 54 bytes or 66 bytes, which are lengths of pure ACKs with or without TIMESTAMP option on Ethernet. Some NICs can identify specific packet types, such as pure TCP ACKs, and indicate so in the receive descriptors. The driver can indicate this in a flag when passing the SKB to the fastpath stack, which can skip redundant checks. We added TCP-PARO support to two NIC drivers — e1000e for Intel's PCI-E Gigabit Ethernet controllers and vxge for the Neterion X3120 10GE adapter. Each driver patch consists of 10 lines of code. The pure ACK processing fastpath module is written in 240 lines.

Handling a mixture of pure ACKs and other TCP segments: Whenever a non-pure ACK, such as a SACK, is received, the original path is used for receive processing. Moreover, if TCP-PARO discovers the socket backlog is non-empty, it delegates pure ACKs to the original receive path (by returning the *FALLBACK* verdict to the driver). When the mutex is released, the fastpath backlog ring is flushed, followed by the socket backlog. This scheme ensures all packets for the socket are processed in receive order.

3.3 Performance Evaluation

We repeated the experiments from Section 2 with TCP-PARO. Table 1 shows the number of CPU cycles expended per Web request. *Copybreak* refers to the unoptimized stack which performs SKB and packet buffer cloning (see Section 3.1). With TCP-PARO enabled,

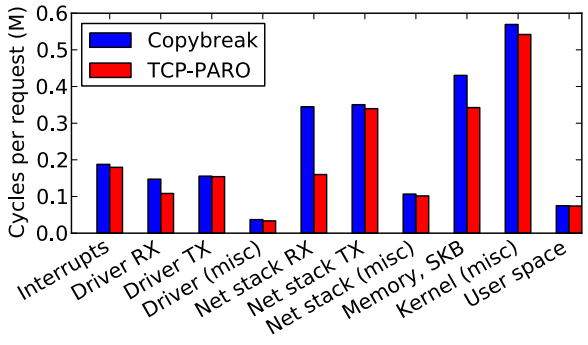


Figure 3: Breakdown of CPU cycles per Web request with and without TCP-PARO. File size is 2MB.

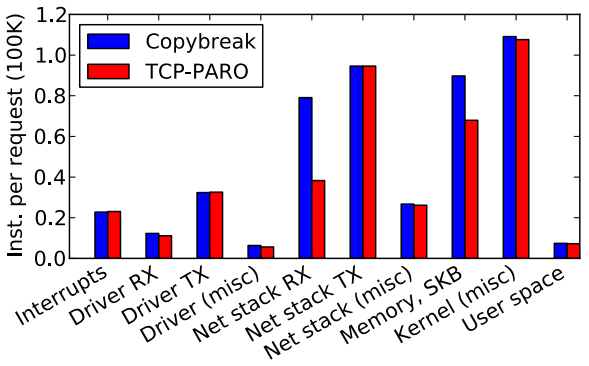


Figure 4: Breakdown of instructions executed per Web request with and without TCP-PARO. File size is 2MB.

CPU cycles are saved consistently across various response sizes. The savings increases with file size. At 8MB, cycles per request is reduced by 16.7%. This is expected because with larger files more data segments are transmitted, eliciting more pure ACKs from the clients.

To understand the sources of cycles savings, we profile cycle expenditure in various system components. Figure 3 shows the profile for 2MB files. TCP-PARO is effective in reducing cycles expended in Driver RX (by 25%), Network RX (by 53%) and memory operations (by 20%). Driver RX includes cycles spent in the driver function for polling packets and setting up fresh RX descriptors for DMA. With TCP-PARO, no SKB is allocated, hence the driver receive routine is lightweight. Network RX includes cycles spent in delivering a packet up the stack and protocol processing. Savings here are due to executing the fastpath processing module, instead of traversing multiple network layers. Our tests were conducted with minimal network stack features, i.e. no network taps or Netfilter modules. Therefore, Network RX for Copybreak represents the minimum cycles expended in receive processing of pure ACKs. The savings in memory operations can be explained by fewer SKB allocations, as SKBs are reused for DMA. Unlike Copybreak, no SKB cloning is necessary for the parsing and ACK header delivery.

Figure 4 presents a similar functional breakdown for number of instructions executed. A major source of saved cycles is from reduced instruction executed. 51% and 24% fewer instructions respectively are executed for network receive processing and memory operations.

4 ACK Optimization for Reliable Multicast

TCP-PARO can be readily integrated with TCP-based reliable multicast protocols, such as TCP-SMO [6]. TCP-SMO is a receiver-driven single-source reliable multicast extension to TCP. The sender maintains a TCP control block (TCB) for each receiver, and aggregates information across all TCBs to produce the multicast TCP state. This state tracks the slowest receiver's earliest unacknowledged number and the minimum congestion and flow control windows. Multicast data segments are ACKed by all receivers. ACK processing at the sender entails updating both the receiver TCB state and the aggregate multicast state.

TCP-PARO integration at the sender is straightforward. The fast path processes ACKs to update the receiver TCBs and then the multicast state. The thread multicasting data is synchronized with the kernel threads performing ACK processing in soft-interrupt. A single pre-allocated ring is used for backlogging pure ACKs in the fast path.

4.1 Performance Evaluation

We study the benefits of integrating TCP-PARO with TCP-SMO. A gigabit Ethernet network of 9 machines connected with a single gigabit switch is used for this study. Each machine is equipped with a Xeon E3-1230v2 processor, 16GB RAM and an Intel 82579LM on-board gigabit NIC. The *e1000e* NIC driver is augmented with TCP-PARO support. One machine acts as the multicast sender and the rest host the receivers.

We ran the sender process and all network processing on a single core. Figure 5 presents a boxplot comparing the total data transfer time with and without TCP-PARO integration into TCP-SMO. Each data point represents 50 experiment trials. TCP-PARO enables near-linear scaling of reliable multicast. The average total transfer time grows from 2.93s to 3.09s (5.6% increase) when number of receivers nearly doubles from 88 to 168. With more receivers, the sender needs to maintain more per-receiver

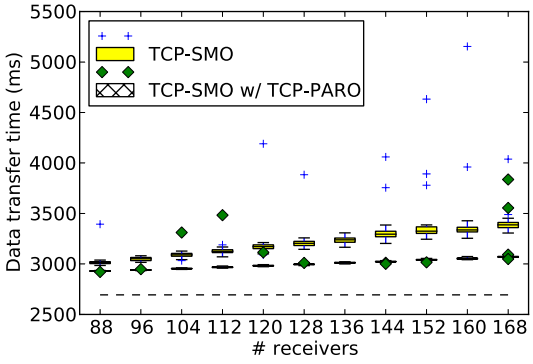


Figure 5: Data transfer time. Sender process and network processing on one core.

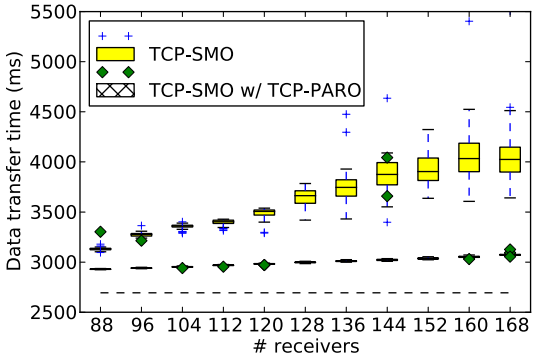


Figure 7: Data transfer time. Sender process on one core. Network processing on multiple cores.

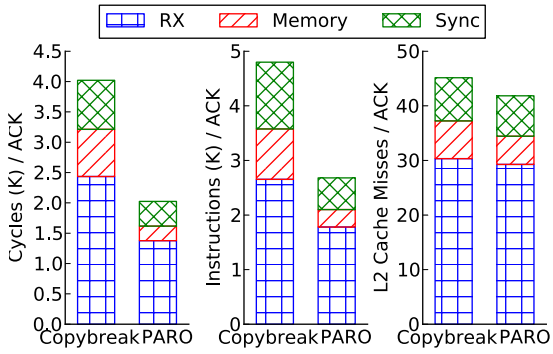


Figure 6: Per-ACK overhead (160 receivers). Sender process and network processing on one core.

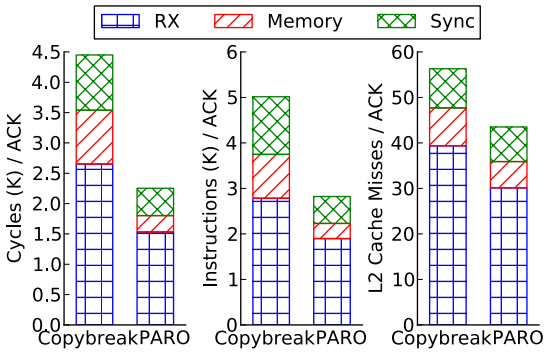


Figure 8: Per-ACK overhead (160 receivers). Sender process on one core. Network processing on multiple cores.

state, and the multicast state updates are more expensive. Moreover, more ACKs are received and processed by the sender. For reference, the black dotted line shows the best possible transfer time on the 1GE network with linear scaling and zero protocol overhead (2.7s). In contrast, without TCP-PARO, the total transfer time grows from 3.02s to 3.4s, a 12.6% increase. Figure 6 breaks down the per-ACK processing overhead into its dominating components. With TCP-PARO, nearly 50% cycles and instructions are saved. The savings in *RX* and *Memory* are due to lightweight fastpath processing and reduced memory allocations. *Sync* includes synchronization operations such as atomic primitives, spinlocks and RCU. TCP-PARO reduces synchronization overhead by 50%, because fastpath execution minimizes lock acquisitions in the original stack which are unnecessary for pure ACK processing. The residual synchronization cost is due to the per-socket mutex. By processing ACKs faster, the sender is able to absorb the burst of ACKs from receivers, thus reducing the transfer time.

We next investigate the potential of parallel network processing on multiple cores to alleviate ACK processing cost. Figure 7 shows a similar boxplot for this setting. The total transfer time with TCP-PARO stays the same, the time penalty growth is similar to the single-core case, and throughput is improved by 33%. In contrast, without optimization, the transfer time grows much more rapidly. The performance difference can be explained by examining per-ACK overhead. In Figure 8, we observe similar cycles savings with TCP-PARO, but L2 cache performance worsens. L2 cache misses increase by about 4% with TCP-PARO, and by 24% with Copybreak. The difference is due to TCP-PARO's reusing SKBs, hence reducing cache miss penalities when cores attempt to deallocate SKBs from other cores' slab caches. Increased cache misses contributed to 12% extra cycles with Copybreak, which in turn translated into 19% increase in the median transfer time and higher variance. On the other hand, the increase in TCP-PARO cycles was modest, hence it did not adversely affect the sender's ability to quickly process ACK bursts. These results show that,

with cheap onboard NICs which do not offer multiple DMA engines for parallel processing, ACK optimization can effectively mitigate cache effects in network processing, while preserving the ability to share the NIC among multiple cores. The trend towards smaller computing units in the datacenter suggest that ACK-heavy workloads can expect to benefit from the reduced interactions with memory managers by employing fastpath-based optimizations like TCP-PARO.

The small testbed limited the number of receivers to 168, beyond which the receivers become the bottleneck. Nonetheless, contrary to conventional wisdom, our results indicate that ACK-based reliable multicast can be scaled to non-trivial receiver group sizes, and that the ACK implosion problem can be mitigated fairly well at the end host. We are working on further characterizing the behavior and performance of TCP-SMO with TCP-PARO on 10GE and other environments.

5 Related Work

Per-packet overhead has been identified as a significant overhead in network I/O [3] [7]. Batching optimizations have been proposed to amortize receive-side overhead for data packets [4][8]. Our work focuses on receive processing of small control packets, which are not suitable for batching. Instead, we propose a fastpath architecture which bypasses the original stack and reduces memory allocations.

Alternative packet I/O schemes [9][5] focus on delivering packets from NICs to software. The main motivation of these schemes is to provide a high-performance platform for building software packet processors, such as software routers. However, they do not address packet multiplexing to sockets, buffering, synchronization with multiple threads and protocol processing. They are thus orthogonal to our work.

Partial network offloading techniques such as segmentation offloading and checksum offloading [2], are widely available. Modern NICs implement even more features in the hardware to assist with software stack processing. Examples include filtering packets based on network protocol and performing packet steering to multiple cores [1]. Future NICs can further enhance fastpath performance by implementing packet header parsing in hardware. This could further simplify and improve the performance of the fastpath.

6 Conclusions

In this paper, we investigated optimizing pure TCP ACK receive processing to improve server performance. Pure ACKs consume a small portion of network bandwidth, and have thus received relatively scarce attention in network optimization. However, the rapid increase in network bandwidth, the resultant expectation of higher client-server ratios and the potential benefits of reliable multicast suggest that improving ACK processing efficiency is a real concern in achieving high application performance.

We designed a network fastpath architecture for efficient packet delivery and protocol processing. We implemented TCP-PARO on fastpath, and demostrated 16% cycles savings in an HTTP workload. Moreover, reliable multicast throughput improved by 33% due to reduced ACK processing time.

With the deployment of 40GE and then 100GE, and the desire of lowering server power and space footprint, it is becoming more compelling to improve efficiency of server resource usage. In particular, processing of control packets as a CPU and memory intensive operation is likely to increase in relative cost as link speeds increase and servers handle more clients. Optimizations such as TCP-PARO can be expected to play an important role in reducing the impact of control packet processing on application performance, both on average and in overload cases, as can arise in reliable multicast. We are currently studying other applications of the fastpath architecture, such as SYN flood attack detection, packet accounting and traffic filtering.

References

[1] *Section 7, Intel i350 Gigabit Ethernet Controller Datasheet*. April 2012.

[2] CHASE, J., GALLATIN, A., AND YOCUM, K. End system optimizations for high-speed tcp. *Communications Magazine, IEEE 39*, 4 (apr 2001), 68 –74.

[3] FOONG, A. P., HUFF, T. R., HUM, H. H., PATWARDHAN, J. R., AND REGNIER, G. J. TCP Performance Revisited. ISPASS '03, pp. 70–79.

[4] GROSSMAN, L. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Ottawa Linux Symposium* (2005), pp. 195–200.

[5] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. SIGCOMM '10, pp. 195–206.

[6] LIANG, S., AND CHERITON, D. TCP-SMO: extending TCP to support medium-scale multicast applications. INFOCOM '02, pp. 1356 – 1365.

[7] LIAO, G., ZNU, X., AND BNUYAN, L. A new server I/O architecture for high speed networks. HPCA '11, pp. 255–265.

[8] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. USENIX ATC'08, pp. 85–98.

[9] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. USENIX ATC'12, pp. 101–112.