CacheDedup: In-line Deduplication for Flash Caching

Wenji Li, *Arizona State University;* Gregory Jean-Baptise, Juan Riveros, and Giri Narasimhan, *Florida International University;* Tony Zhang, *Rensselaer Polytechnic Institute;* Ming Zhao, *Arizona State University*

https://www.usenix.org/conference/fast16/technical-sessions/presentation/li-wenji

This paper is included in the Proceedings of the
14th USENIX Conference on
File and Storage Technologies (FAST '16).
February 22–25, 2016 • Santa Clara, CA, USA
ISBN 978-1-931971-28-7

CacheDedup: In-line Deduplication for Flash Caching

Wenji Li
Arizona State University

Gregory Jean-Baptise
Florida International University

Juan Riveros
Florida International University

Giri Narasimhan
Florida International University

Tong Zhang
Rensselaer Polytechnic Institute

Ming Zhao
Arizona State University

Abstract

Flash caching has emerged as a promising solution to the scalability problems of storage systems by using fast flash memory devices as the cache for slower primary storage. But its adoption faces serious obstacles due to the limited capacity and endurance of flash devices. This paper presents CacheDedup, a solution that addresses these limitations using in-line deduplication. First, it proposes a novel architecture that integrates the caching of data and deduplication metadata (source addresses and fingerprints of the data) and efficiently manages these two components. Second, it proposes duplication-aware cache replacement algorithms (D-LRU, D-ARC) to optimize both cache performance and endurance. The paper presents a rigorous analysis of the algorithms to prove that they do not waste valuable cache space and that they are efficient in time and space usage. The paper also includes an experimental evaluation using real-world traces, which confirms that CacheDedup substantially improves I/O performance (up to 20% reduction in miss ratio and 51% in latency) and flash endurance (up to 89% reduction in writes sent to the cache device) compared to traditional cache management. It also shows that the proposed architecture and algorithms can be extended to support the combination of compression and deduplication for flash caching and improve its performance and endurance.

1 Introduction

Flash caching employs flash-memory-based storage as a caching layer between the DRAM-based main memory and HDD-based primary storage in a typical I/O stack of a storage system to exploit the locality inherent in the I/Os at this layer and improve the performance of applications. It has received much attention in recent years [7, 5, 2, 23], which can be attributed to two important reasons. First, as the level of consolidation—in terms of both the number of workloads consolidated to a single host and the number of hosts consolidated to a single storage system—continues to grow in typical computing systems such as data centers and clouds, the scalability of the storage system becomes a serious issue. Second, the high performance of flash-memory-based storage devices has made flash caching a promising option to address this scalability issue: it can reduce the load on the primary storage and improve workload performance by servicing I/Os using cached data.

There are however several key limitations to effective caching with flash memories. First, with the increasing data intensity of modern workloads and the number of consolidated workloads in the system, the demands on cache capacity have skyrocketed compared to the limited capacity of commodity flash devices. Second, since flash memories wear out with writes, the use of flash for caching aggravates the endurance issue, because both the writes inherent in the workload and the reads that miss the cache induce wear-out.

This paper presents CacheDedup, an in-line flash cache deduplication solution to address the aforementioned obstacles. First, deduplication reduces the cache footprint of workloads, thereby allowing the cache to better store their working sets and reduce capacity misses. Second, deduplication reduces the number of necessary cache insertions caused by compulsory misses and capacity misses, thereby reducing flash memory wear-out and enhancing cache durability. Although deduplication has been studied for a variety of storage systems including flash-based primary storage, this paper addresses the unique challenges in integrating deduplication with caching in an efficient and holistic manner.

Efficient cache deduplication requires seamless integration of caching and deduplication management. To address this need, CacheDedup embodies a novel architecture that integrates the caching of data and deduplication metadata—the source addresses and fingerprints of the data, using a separate Data Cache and Metadata Cache. This design solves two key issues. First, it allows CacheDedup to bound the space usage of metadata, making it flexible enough to be deployed either on the client side or the server side of a storage system, and implemented either in software or in hardware. Second, it enables the optimization of caching historical source addresses and fingerprints in the Metadata Cache after their data is evicted from the Data Cache. These historical deduplication metadata allow CacheDedup to quickly recognize duplication using the cached fingerprints and produce cache hits when these source addresses are referenced again.

Based on this architecture, we further study duplication-aware cache replacement algorithms that can exploit deduplication to improve flash cache performance and endurance. First, we present D-LRU, a duplication-aware version of LRU which can be efficiently implemented by enforcing an LRU policy on both the Data and Metadata Caches. Second, we present D-ARC, a duplication-aware version of ARC that exploits the scan-resistant ability of ARC to further improve cache performance and endurance. For both algorithms, we also prove theoretically that they do not lead to wastage in the Data and Metadata caches and can efficiently use their space.

CacheDedup is implemented atop block device virtualization [5], and can be transparently deployed on existing storage systems. We evaluate it using real-world traces, including the FIU traces [18] and our own traces collected from a set of Hadoop VMs. The results show that CacheDedup substantially outperforms traditional cache replacement algorithms (LRU and ARC) by reducing the cache miss ratio by up to 20%, I/O latency by 51%, and the writes sent to flash memories by 89%. It can effectively deduplicate data both within a workload and across multiple workloads that share the cache. We also measure the overhead of CacheDedup using the fio benchmark [1], which shows that the throughput overhead is negligible and the latency overhead from fingerprinting can be overlapped with concurrent I/O operations and dominated by the hit ratio gain from deduplication. In terms of space overhead, CacheDedup needs $< 4\%$ of the flash cache to store the deduplication metadata in order for our algorithms to achieve peak performance.

CacheDedup is among the first to study duplication-aware cache management for cache deduplication. Compared to the related work, which also considered data reduction techniques for server-side flash caching [19], we show that our approach can be naturally extended to support both duplication- and compression-aware cache management and that it can improve the read hit ratio by 12.56%. Our approach is not specific to flash-based caches—it leverages only flash devices' faster speed compared to HDDs and larger capacity compared to DRAMs. Therefore, it is also applicable to other non-volatile memory technologies used as a caching layer between DRAMs and the slower secondary storage. While the new technologies may have better endurance, they are likely to have quite limited capacity compared to NAND flash, and will still benefit greatly from CacheDedup, which can substantially reduce both cache footprint and writes sent to cache device.

The rest of the paper is organized as follow: Section 2 explains the background; Section 3 presents the architectural design of CacheDedup; Section 4 describes the duplication-aware cache management algorithms; Section 5 presents the evaluation results; Section 6 examines the related work; and Section 7 concludes the paper.

2 Background and Motivations

Need of Integrated Flash Cache Deduplication. The emergence of flash-memory-based storage has greatly catalyzed the adoption of flash caching at both the client side and server side of a network storage system [7, 5, 2]. However, flash caches still face serious capacity and endurance limitations. Given the increasingly data-intensive workloads and increasing level of storage consolidation, the size of commodity flash devices is quite limited. The use of flash for caching also aggravates the wear-out problem of flash devices, because not only the writes from the workloads cause wear-out, but also all the reads that are inserted into the cache due to cache misses.

Deduplication is a technique for eliminating duplicate copies of data and has been used to reduce the data footprint for primary storage [12, 15] and backup and archival storage [27, 22]. It often uses a collision-resistant cryptographic hash function [4, 28] to identify the content of a data block and discover duplicate ones [27, 18, 8]. Deduplication has the potential to solve the above challenges faced by flash caching. By reducing the data footprint, it allows the flash cache to more effectively capture the locality of I/O workloads and improve the performance. By eliminating the caching of duplicate data, it also reduces the number of writes to the flash device and the corresponding wear-out.

Although one can take existing flash caching and deduplication solutions and stack them together to realize cache deduplication, the lack of integration will lead to inefficiencies in both layers. On one hand, it is infeasible to stack a caching layer upon a deduplication layer because the former would not be able to exploit the space reduction achieved by the latter. On the other hand, there are also serious limitations to simply stacking a deduplication layer upon a caching layer. First, the deduplication layer has to manage the fingerprints for the entire primary storage, and may make fingerprint-management decisions that are detrimental to the cache, e.g., evicting fingerprints belonging to data with good locality and causing duplicate copies in the cache. Second, the caching layer cannot exploit knowledge of data duplication to improve cache management. In contrast, CacheDedup employs an integrated design to optimize the use of deduplication for flash caching for both performance and endurance.

The recent work Nitro [19] studied the use of both deduplication and compression for server-side flash caches. CacheDedup is complementary to Nitro in its new architecture and algorithms for duplication-aware cache management. Moreover, our approach can be applied to make the cache management aware of both compression and deduplication and improve such a solution that uses both techniques. A quantitative comparison to Nitro is presented in Section 5.6.

Deduplication can be performed in-line—in the I/O path—or offline. CacheDedup does in-line deduplication to prevent any duplicate block from entering the cache, thereby achieving the greatest reduction of data footprint and wear-out. Our results show that the overhead introduced by CacheDedup is small. Deduplication can be done at the granularity of fixed-size chunks or content-defined variable-size chunks, where the latter can achieve greater space reduction but at a higher cost. CacheDedup chooses deduplication at the granularity of cache blocks, which fits the structure of flash caches and facilitates the design of duplication-aware cache replacement. Our results also confirm that a good level of data reduction

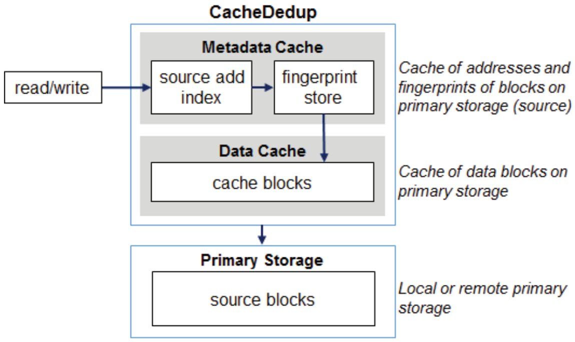

CacheDedup
Metadata Cache
read/write
source add index
fingerprint store
Cache of addresses and fingerprints of blocks on primary storage (source)
Data Cache
cache blocks
Cache of data blocks on primary storage
Primary Storage
source blocks
Local or remote primary storage

Figure 1: Architecture of CacheDedup

can be achieved with fixed-size cache deduplication.

Need for Deduplication-aware Cache Management. There exists a number of cache replacement schemes. In particular, the widely used LRU algorithm is designed to exploit temporal locality by always evicting the least-recently used entry in the cache. Theoretically, it has been shown to have the best guarantees in terms of its worst-case performance [30]. But it is not "scan resistant", i.e., items accessed only once could occupy the cache and reduce the space available for items that are accessed repeatedly. ARC [21] is an adaptive algorithm that considers both recency and frequency in cache replacement. It is "scan resistant" and shown to offer better performance for many real-world workloads.

However, cache replacement algorithms typically focus on maximizing the hit ratio, and disregard any issues related to the lifespan and wear-and-tear of the hardware device, which are unfortunately crucial to flash-based caches. Attempts to reduce writes by bypassing the cache invariably affect the hit ratio adversely. The challenge is to find the "sweet spot" between keeping hit ratios close to the optimum and lowering the number of write operations to the device. CacheDedup addresses this challenge with its duplication-aware cache replacement algorithms, designed by optimizing LRU and ARC and enabled by an integrated deduplication and cache management architecture.

3 Architecture

3.1 Integrated Caching and Deduplication

CacheDedup seamlessly integrates the management of cache and deduplication and redesigns the key data structures required for these two functionalities. A traditional cache needs to manage the mappings from the *source addresses* of blocks on the primary storage to the *cache addresses* of blocks on the cache device. A deduplication layer needs to track the fingerprints of the data blocks in order to identify duplicate blocks. There are several unique problems caused by the integration of caching and deduplication to the design of these data structures.

First, unlike a traditional cache, the number of source-to-cache address mappings in CacheDedup is not bounded by the size of the cache, because with deduplication, there is now a many-to-one relationship between these two address spaces. Second, even though the number of fingerprints that a cache has to track is bounded by the cache size, there is an important reason for CacheDedup to track fingerprints for data beyond what is currently stored in the cache. Specifically, it is beneficial to keep historical fingerprints for the blocks that have already been evicted from the cache, so that when these blocks are requested again, CacheDedup does not have to fetch them from the primary storage in order to determine whether they are duplicates of the currently cached blocks. Such an optimization is especially important when CacheDedup is employed as a client-side cache because it can reduce costly network accesses. However, fingerprint storage still needs to abide by the limit on CacheDedup's space usage.

To address these issues, we propose a new *Metadata Cache* data structure to cache source addresses and their fingerprints. This design allows us to solve the management of these metadata as a cache replacement problem and consider it separately from the *Data Cache* that stores data blocks (Figure 1). The Metadata Cache contains two key data structures. The *source address index* maps a source addess of the primary storage to a fingerprint in the Metadata Cache. Every cached source address is associated with a cached fingerprint, and because of deduplication, multiple source addresses may be mapped to the same fingerprint. The *fingerprint store* maps fingerprints to block addresses in the Data Cache. It also contains historical fingerprints whose corresponding blocks are not currently stored in the Data Cache. When the data block pointed to by a historical fingerprint is brought back to the Data Cache, all the source addresses mapped to this fingerprint can generate cache hits when they are referenced again. Each fingerprint has a reference count to indicate the number of source blocks that contain the same data. When it drops to zero, the fingerprint is removed from the Metadata Cache.

The decoupled Metadata Cache and Data Cache provide separate control "knobs" for our duplication-aware algorithms to optimize the cache management (Section 4). The algorithms limit the metadata space usage by applying their replacement policies to the Metadata Cache and exploiting the cached historical fingerprints to opportunistically improve read performance. Both caches can be persistently stored on a flash device to tolerate failures as discussed in Section 3.3. Moreover, our architecture enables the flash space to be flexibly partitioned between the two caches. For a given Data Cache size, the minimum Metadata Cache size is the required space for storing the metadata of all the cached data. Our evaluation using real-world traces in Section 5 shows that the "minimum" size is small enough to not be an issue in practice. More importantly, the Metadata Cache can be expanded by taking away some space from the Data Cache to store more historical fingerprints and potentially improve the performance. This tradeoff is studied in Section 5.4.

3.2 Operations

Based on the architecture discussed above, the general operations of CacheDedup are as follows. The necessary cache replacement is governed by the duplication-aware algorithms presented in Section 4.

Read. A read that finds its source block address in the Metadata Cache with a fingerprint pointing to a Data Cache block is a hit in the Data Cache. Otherwise, it is a miss. Note that the read may match a historical fingerprint in the Metadata Cache which does not map to any data block in the Data Cache, and it is still a miss. Upon a miss, the requested data block is fetched from the primary storage and, if it is not a duplicate of any existing cached data block, it is inserted into the Data Cache. The corresponding source address and fingerprint are inserted into the Metadata Cache if necessary. If the fingerprint already exists in the Metadata Cache, it is then "revived" by pointing to the new Data Cache block, and all the historical source addresses that point to this fingerprint are also "revived" because they can generate hits when accessed again.

Write. The steps differ by the various write policies that CacheDedup supports:

1. *Write invalidate*—the requested source address and the cached data for this address are invalidated in the Metadata and Data Caches if they exist. The write goes directly to the primary storage.
2. *Write through*—the write is stored in cache and at the same time submitted to the primary storage. The source address and fingerprint are inserted into the Metadata Cache if they are not already there. If the data contained in the write matches an existing cached block, no change needs to be made to the Data Cache, while the reference count of its fingerprint is incremented by one. If the previous data of this block is already in the Data Cache, the reference count of the previous fingerprint is decremented by one.
3. *Write back*—the write is stored only in the cache and submitted to the primary storage later, when the block is evicted or when the total amount of dirty data exceeds a predefined threshold. The steps are identical to the write-through policy except that the write is not immediately submitted to the primary storage.

3.3 Fault Tolerance

The nonvolatile nature of flash storage allows the cached data to persist when the host of the cache crashes, but to recover the cached data, their metadata also needs to be stored persistently. If the goal is to avoid data loss, only the source-to-cache address mappings for the locally modified data from using the write-back policy must be persistent. CacheDedup synchronously commits both the metadata and data to the cache device to ensure consistency. If the goal is to avoid warmup after the host restarts, the entire Metadata Cache, including the source block index and fingerprints, is made persistent. The time overhead of making the Metadata Cache persistent is not as significant as its space overhead on the flash device, because the metadata for clean blocks can be written in batches and asynchronously. For both fault tolerance goals, the Metadata Cache is also kept in main memory to speed up cache operations, and its memory usage is bounded. Finally, if the goal is to tolerate flash device failures, additional mechanisms [26] need to be employed. We can also leverage related work [17] to provide better consistency for flash caching.

3.4 Deployment

CacheDedup can be deployed at both the client side and server side of a storage system: client-side CacheDedup can more directly improve application performance by hiding the high network I/O latency, whereas server-side CacheDedup can use the I/Os from multiple clients to achieve a higher level of data reduction. When CacheDedup is used by multiple clients that share data, a *cache coherence* protocol is required to ensure that each client has a consistent view of the shared data. Although it is not the focus of this paper, CacheDedup can straightforwardly extend well-studied cache coherence protocols [13, 24] to synchronize both the data in the Data Cache and the fingerprints in the Metadata Cache, thereby ensuring consistency across the clients.

While the discussions in this paper focus on a software-based implementation of CacheDedup, its design allows it to be incorporated into the flash translation layer of specialized flash devices [29, 25]. The space requirement is bounded by the Metadata Cache size, and the computational complexity is also limited (Section 4), making CacheDedup affordable for modern flash hardware.

The discussions in the paper also assume the deployment of CacheDedup at the block-I/O level, but its design is largely applicable to the caching of filesystem-level reads and writes, which requires only changing the Metadata Cache to track (*file handle*, *offset*) tuples instead of source block addresses.

4 Algorithms

In this section we present two duplication-aware cache replacement algorithms. Both are enabled by the integrated cache and deduplication management framework described above. We first define some symbols (Table 1).

- *Data Cache*, D, stores the contents of up to d deduplicated data blocks, indexed by their fingerprints.
- *Metadata Cache*, M, holds a set of up to m source addresses and corresponding fingerprints, a function f that maps a source address to the fingerprint of its data block, and a function h that maps a fingerprint to the location of the corresponding data block in D. We denote the composition of f and h by the function h' that maps a source address to the location of the corresponding data block.

Symbol	Definition
D	Data Cache
d	Total number of data blocks in the Data Cache
M	MetaData Cache
m	Total number of source addresses in M
p_i	A source address in M
g_i	A fingerprint in fingerprint store
a_i	The content of a data block in D
$f(p_i)$	Function that maps a source address to a fingerprint
$h(g_i)$	Function that maps a fingerprint to a data block
$h'(p_i)$	Function that maps a source address to a data block

Table 1: Variable definitions

Thus, for a source address x in M, $f(x)$ is its fingerprint, and $h'(x) = h(f(x))$ is the corresponding data block in D. If $f(x) = f(y)$ and $x \neq y$, then x and y contain *duplicate* data. If $f(x) = g$, we will refer to x as an *inverse map* or one of the addresses of fingerprint g. If we have a source address in M for which the corresponding data is absent from D, we call it an *orphaned address*; if we have a block in D for which the mapping information is not available in M, then it is an *orphaned data block*. Instead of strictly disallowing orphaned addresses and data, we will require our algorithms to comply with the **no-wastage policy**, which states that the cache replacement algorithms are required to not have orphaned addresses and orphaned data blocks simultaneously. The no-wastage policy is important because "wastage" implies suboptimal use of the cache, i.e., instead of bringing in other useful items into cache, we are storing items in cache with incomplete information that would surely result in a miss if requested.

In the rest of this section, we describe the algorithms and analyze their no-wastage property and complexity. Note that the size of data structures required by the algorithms is bounded by the space required to store up to m source addresses and fingerprints; therefore, we omit the space complexity analysis.

4.1 D-LRU

4.1.1 Algorithm

We present D-LRU (pronounced "dollar-you"), a duplication-aware variant of LRU. The pseudocode (Algorithm 1) consists of two separate LRU policies being enforced first on the Metadata Cache (M) and then on the Data Cache (D). Insert-Using-LRU(x,A,n) inserts x in list A (with capacity n) only if it is not already in A, in which case the LRU item is evicted to make room for it.

Algorithm 1: D-LRU pseudocode

```
Remarks: $D$ is indexed by $f(x)$ and $M$ is indexed by $x$
Input: The request stream $x_1, x_2, \ldots, x_t, \ldots$
Initialization: Set $D = \emptyset$, $M = \emptyset$
for every $t \geq 1$ and any $x_t$ do
    Insert-Using-LRU$(x_t, M, m)$
    Insert-Using-LRU$(f(x_t), D, d)$
```

4.1.2 Analysis

The algorithm of D-LRU is rather simple, but our analysis shows that it is also quite powerful as it allows efficient use of both the Metadata and Data Caches with no wastage. We start the analysis with several useful observations. The first is that no duplicate addresses are inserted into M and no duplicate data blocks are inserted into D. However, every new address does result in an entry in M, even if it corresponds to a duplicate data block.

To discuss more observations, we introduce the following notation. Let $\{p_1, \ldots, p_m\}$ be the source addresses in the Metadata Cache M, ordered so that p_1 is the LRU entry and p_m the MRU entry. Let $\{g_1, \ldots, g_n\}$ be the corresponding fingerprints stored in the fingerprint store. Let $\{a_1, a_2, \ldots, a_d\}$ be the Data Cache contents, ordered so that a_1 is the LRU entry and a_d the MRU entry. For any data block a, let $max_M(a)$ be the position in the Metadata Cache of its most recently accessed source address. In other words, for any $a \in D$, $max_M(a) = \max\{i | h(f(p_i)) = a \ \wedge p_i \in M\}$, if a is not an orphan, and 0 otherwise. Next we observe that the order in D is the same as the order of their most recently accessed addresses. Finally, any orphans in D must occupy contiguous positions at the bottom of the DataCache LRU list. Any orphans in M need not be in contiguous positions but must occupy positions that are lower than $max_M(a_1)$, where a_1 is the LRU item in D.

To prove that D-LRU does indeed comply with the no-wastage policy, we propose the following invariants.

P1: If $\exists a \in D$ s.t. $max_M(a) = 0$, then $\forall q \in M, h'(q) \in D$.
P2: If $\exists q \in M$ s.t. $h'(q) \notin D$, then $\forall a \in D, max_M(a) > 0$.

Simply put, invariant P1 states that if there are orphaned data items in D, there are no orphaned addresses in M. Invariant P2 states the converse. When p is the only entry in M, then $p \in M$ and $h'(p) \in D$. The invariants hold. We then need to show that if these two invariants hold after serving a set of requests (inductive hypothesis), then it continues to hold after processing one more request. Let the next request be for source address x and fingerprint $f(x)$. We list the base case and then one of three cases must occur before the new request is processed.

Case 1: $x \in M$ and $h(f(x)) \in D$. D-LRU performs no evictions and the contents of the Metadata and Data Caches remain unchanged, as do the invariants.

Case 2: $x \in M$ and $h(f(x)) \notin D$. In this case D-LRU evicts an item from D to bring back the data $h(f(x))$ for the orphaned address x. Using the inductive hypothesis, and the fact that $x \in M$ is orphaned, we know that no orphaned items exist in

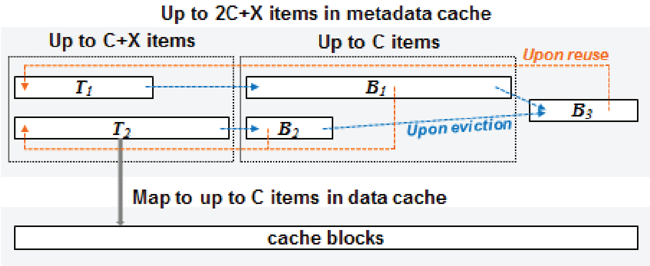


Figure 2: Architecture of D-ARC

D. Since D-LRU does not evict any entry from M, no new orphans will be created in D, leaving the invariants true.

CASE 3: $x \notin M$. In this case, D-LRU evicts p_1 and adds x as the new MRU in M. Also, if $h(f(x)) \notin D$, then D-LRU will evict a_1 and add $h(f(x))$ as MRU in D. Two possible cases apply for the analysis of this situation.

CASE 3.1: If there is at least one orphaned data item in D (and none in M) prior to processing the new request, then since all orphans in D are at the bottom, its LRU item a_1 is an orphan. Thus, the eviction of a_1 cannot create any orphans in M, and thus the invariants will hold.

CASE 3.2: If there are no orphans in D and $x \notin M$, we have two cases. First, if $h(f(x)) \notin D$, the algorithm must evict the LRU items from both M and D. If $max_M(a_1) > 1$ then the eviction of p_1 will leave no orphans. If $max_M(a_1) = 1$ then p_1 is the only address in M that maps to a_1 and since both will get evicted, no new orphans are created in the process. Second, if $h(f(x)) \in D$, the algorithm only evicts p_1 from M. If $max_M(a_1) > 1$ then the eviction of p_1 will not leave orphans in D. If $max_M(a_1) = 1$ then the eviction of p_1 makes a_1 an orphan. But because there can be no orphan addresses occupying positions lower than p_1 in M, the invariants still hold.

Thus, D-LRU complies with the *no-wastage policy*.

Complexity. D-LRU can service every request in $O(1)$ time because the LRU queues are implemented as doubly linked lists and the elements in the lists are also indexed to be able to access in constant time. Evicting an element or moving it to the MRU position can also be done in constant time.

4.2 D-ARC

4.2.1 Algorithm

Next we present a duplication-aware cache management algorithm, D-ARC, that is based on ARC [21], which is a major advance over LRU because of its scan-resistant nature. Similarly, the algorithm of D-ARC is more complex than D-LRU.

We start with a brief description of ARC. ARC assumes a Data Cache of size C. It uses four LRU lists, T_1, T_2, B_1, and B_2 to store metadata, i.e., cached source addresses, with total size $2C$. The key idea is to preserve information on frequently accessed data in T_2 and to let information on “scan” data (single-access data) pass through T_1. Together, the size of T_1 and T_2 cannot exceed C. When a new data block is added to the cache, its corresponding metadata is added to T_1, and it is moved to T_2 only when that address is referenced again. The relative sizes of the two lists, T_1 and T_2, are controlled by an adaptive parameter p. The algorithm strives to maintain the size of T_1 at p. When an item is evicted from T_1 or T_2, its data is also removed from the cache. ARC uses B_1 and B_2 to save metadata evicted from T_1 and T_2, respectively. Together they store an additional C metadata items, which help monitor the workload characteristics. When a source address from B_1 or B_2 is referenced, it is brought back into T_2, but triggers an adjustment of the adaptive parameter p.

Our ARC-inspired duplication-aware cache replacement algorithm is named D-ARC. The idea behind it is to maintain a duplication-free Data Cache D of maximum size C, and to use an ARC-based replacement scheme in the Metadata Cache M. If evictions are needed in D, only data blocks with no mappings in $T_1 \cup T_2$ are chosen for eviction. Figure 2 illustrates the architecture of D-ARC. The corresponding fingerprints are stored in the fingerprint store, which is omitted for clarity.

The first major difference from ARC is that the total size of T_1 and T_2 is not fixed and will vary depending on the duplication in the workload. If the workload has no duplicate blocks, then M will hold at most C source addresses, each mapped to a unique block in D, just as with the original ARC. In the presence of duplicates, D-ARC allows the total size of T_1 and T_2 to grow up to $C+X$, in order to store X more source addresses whose data duplicates the existing ones. A single block in D may be mapped from multiple source addresses in T_1 and T_2. X is a parameter that can be tuned by a system administrator to bound the size of M to store up to $2C+X$ items (source address/fingerprint pairs).

Second, when source addresses are evicted from T_1 or T_2 and moved into B_1 or B_2, as dictated by the ARC algorithm, D-ARC saves their fingerprints and data to opportunistically improve the performance of future references to these addresses. Moreover, D-ARC employs an additional LRU list B_3 to save source addresses (and their fingerprints) evicted from B_1 and B_2, as long as the lists T_1, T_2, and B_3 together store less than $C+X$ mappings. In essence, B_3 makes use of the space left available by T_1 and T_2. When a hit occurs in B_3, it is inserted into the MRU position of T_1, but does not affect the value of p. A future request to a source address retained in $B_1 \cup B_2 \cup B_3$ may result in a hit in D-ARC if its fingerprint shows that the data is in D. In contrast, any item found in B_1 or B_2 always results in a miss in the original ARC.

Third, when eviction is necessary in the Data Cache, D-ARC chooses an item with no mappings in $T_1 \cup T_2$. If no such data item is available, then items are evicted from $T_1 \cup T_2$ using the original ARC algorithm until such a data block is found. Note that at most $X+1$ items are evicted from $T_1 \cup T_2$ in the process.

The D-ARC pseudocode is shown in Algorithms 2 and

Algorithm 2: D-ARC(C, X) pseudocode

```
Input: The request stream $x_1, x_2, \ldots, x_t, \ldots$
ProcessRequest ()
    if $x_i \in T_1 \cup T_2$ then
        Move $x_i$ to MRU position on $T_2$
    if $x_i \in B_1 \cup B_2$ then
        Increase $p$, if $x_i \in B_1$; else Decrease $p$.
        if $h'(x_i) \notin D$ then
            InsertInDataCache()
        CheckMetadataCache()
        Move $x_i$ to MRU position in $T_2$
    if $x_i \in B_3$ then
        if $h'(x_i) \notin D$ then
            InsertInDataCache()
        Move $x_i$ to MRU position in $T_1$
    if $x_i \notin T_1 \cup T_2 \cup B_1 \cup B_2 \cup B_3$ then
        if $h'(x_i) \notin D$ then
            InsertInDataCache()
        CheckMetadataCache()
        Move $x_i$ to MRU position in $T_1$
```

Algorithm 3: D-ARC(C, X) subroutines

```
Subroutine InsertInDataCache()
    while no "victim" in D with no references in
    $T_1 \cup T_2$ do
        ManageMetadataCache()
    replace LRU "victim" in $D$ with $h'(x_i)$
Subroutine CheckMetadataCache()
    if $|T_1| + |T_2| + |B_3| = C + X$ then
        if $|B_3| = 0$ then
            ManageMetadataCache()
        evict LRU item from $B_3$
Subroutine ManageMetadataCache()
    if $|T_1| + |B_1| \geq C$ then
        if $|T_1| < C$ then
            move LRU from $B_1$ to $B_3$
            ReplaceInMetadataCache()
        else
            if $|B_1| > 0$ then
                move LRU from $B_1$ to $B_3$
                move LRU from $T_1$ to $B_1$
            else
                move LRU from $T_1$ to $B_3$
    else
        if $|B_1| + |B_2| \geq C$ then
            move LRU from $B_2$ to $B_3$
        ReplaceInMetadataCache()
Subroutine ReplaceInMetadataCache()
    if $|T_1| > 0 \wedge (|T_1| > p \vee (x_i \in B_2 \wedge |T_1| = p))$
    then
        move LRU from $T_1$ to $B_1$
    else
        move LRU from $T_2$ to $B_2$
```

3. In the main program (ProcessRequest), we have four cases, of which only the third ($x_i \in B_3$) is not present in ARC. In each of the other cases, we have at most two independent operations—one to insert into the Data Cache (if needed) and second to insert into the Metadata Cache (if needed). In InsertInDataCache, an appropriate victim to evict from D is one with no references in $T_1 \cup T_2$. However, to find such a victim, several items may have to be deleted from M, as indicated by the while loop. In CheckMetadataCache, if $T_1 \cup T_2 \cup B_3$ exceeds $C + X$, an item from B_3 is always evicted, possibly after moving something from $T_1 \cup B_1 \cup B_2$ (as achieved in ManageMetadataCache). Finally, ReplaceInMetadataCache is similar to the Replace operation in original ARC and creates space for the new metadata item to be placed.

4.2.2 Analysis

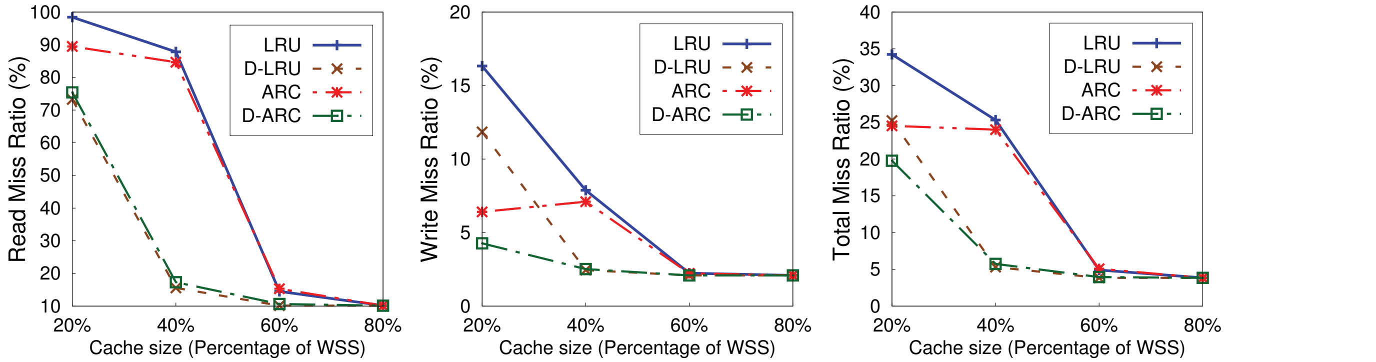
To show that D-ARC complies with the no-wastage policy, we show that no orphans are created in the metadata contents of $T_1 \cup T_2$. (The B lists store historical metadata by design as they do in ARC, so we exclude them from the analysis.) On one hand, if a duplicated item is requested, it does not change the Data Cache, and therefore cannot create orphans in $T_1 \cup T_2$. On the other hand, every time a non-duplicated item is requested, it results in an insert into the Data Cache, causing some item to be evicted. As per the algorithms, the only evictions that are allowed involve items that have no source addresses in $T_1 \cup T_2$. If one exists, we are done and no orphans are created in the Metadata Cache by this insertion. If none exists, we "clear" items from $T_1 \cup T_2$ until we find an item in the Data Cache with no source address in $T_1 \cup T_2$, and then that item becomes the victim to be evicted. This victim cannot create an orphan in the Metadata Cache because of the way it is identified. Thus, D-ARC complies with the *no-wastage policy*.

Complexity. The ARC-based insert to the Metadata Cache can be performed by D-ARC in constant time. However, an insert into the Data Cache may trigger repeated deletions from $T_1 \cup T_2$, which cannot be done in constant time. In fact, if $|T_1| + |T_2| = C + \delta$, for some number $\delta \leq X$, then at most δ evictions are needed for this operation. However, in order to have reached $C + \delta$ elements there must have been δ requests serviced in the past for which there were no evictions. So the amortized cost of each D-ARC request is still $O(1)$.

5 Evaluation

5.1 Methodology

Implementation: We created a practical prototype in Linux kernel space, based on block device virtualization [10]. It



(a) Read miss ratio **(b) Write miss ratio** **(c) Total miss ratio**

Figure 3: Miss ratio from WebVM

Name	Total I/Os	Working	Write-to	Unique
	I/Os (GB)	Set (GB)	-read ratio	Data (GB)
WebVM	54.5	2.1	3.6	23.4
Homes	67.3	5.9	31.5	44.4
Mail	1741	57.1	8.1	171.3
Hadoop	23.6	14.4	0.4	3.7

Table 2: Trace statistics

can be deployed as a drop-in solution on existing systems that run Linux (including hypervisors that use the Linux I/O stack [6, 3]). It appears as a virtual block device to the applications/VMs if deployed on the client side, or to the storage services (e.g., iSCSI, NFS) if deployed on the server side.

Storage setup: We evaluated the real I/O performance of CacheDedup as the client-side flash cache for an iSCSI-based storage system, a widely used network storage protocol. The client and server each runs on a node with two six-core 2.4GHz Xeon CPUs and 24GB of RAM. The client uses a 120GB MLC SATA SSD as the cache, and the server uses a 1TB 7.2K RPM SAS disk as the target. Both nodes run Linux with kernel 3.2.20.

Traces: For our evaluation, we replayed the FIU traces [18]. These traces were collected from a VM hosting the departmental websites for webmail and online course management (*WebVM*), a file server used by a research group (*Homes*), and a departmental mail server (*Mail*). To study the support for concurrent workloads, we also collected traces (*Hadoop*) from a set of Hadoop VMs used to run MapReduce course projects. The characteristics of these traces are summarized in Table 2 where every I/O is of 4KB size. The working set size of a trace is calculated by counting the number of unique addresses of the I/Os in the trace and then multiplying it by the I/O size.

Metrics: We use the following metrics to compare the different cache-management approaches.

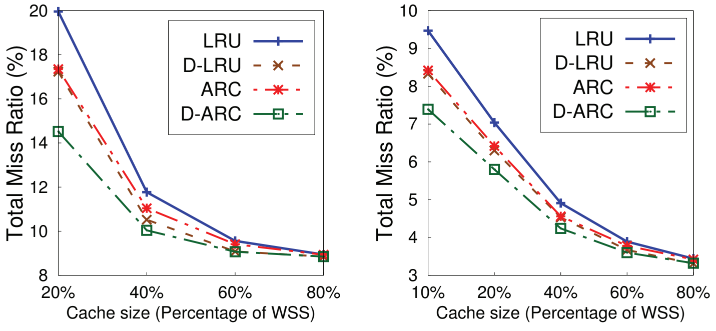
1) *Miss ratio:* We report both read miss ratio and write miss ratio, which are the numbers of reads and writes, respectively, over the total number of I/Os received by the cache. When the write-through policy is used, the read miss ratio is more important because writes always have to be performed on both cache and primary storage [11]. When the write-back policy is used, writes that hit the cache are absorbed by the cache, but the misses cause writes to the primary storage (when they are evicted). Therefore, the write miss ratio has a significant impact on the primary storage's I/O load and the performance of read misses [17, 5]. Therefore, we focus on the results from the write-back policy where the read miss ratio is also meaningful for the write-through policy. We omit the results from the write-invalidate policy which performs poorly for the write-intensive traces, although the deduplication-aware approaches still make substantial improvements as for the other two write policies.

2) *I/O latency/throughput:* To understand how the improvement in cache hits translates to application-perceived performance, we measure the latency of I/Os from replaying the traces on the storage system described above. To evaluate the overhead of CacheDeup, we measure both the I/O latency and throughput using an I/O benchmark, fio [1].

3) *Writes to flash ratio:* Without assuming knowledge of a flash device's internal garbage collection algorithm, we use the percentage of writes sent to the flash device, w.r.t. the total number of I/Os received by the cache manager, as an indirect metric of wear-out.

Cache configurations: We compare several different cache management approaches: 1) *LRU* (without deduplication); 2) *ARC* (without deduplication); 3) *D-LRU*; 4) *D-ARC*. To compare to the related work *Nitro* [19], we also created *CD-ARC*, a new ARC-based cache management approach that is aware of both duplication and compressibility.

For all the approaches that we considered, we show the results from the fault-tolerance configuration that keeps the entire metadata persistent (as discussed in Section 3.3). The



(a) Homes **(b) Mail**

Figure 4: Miss ratio from Homes and Mail

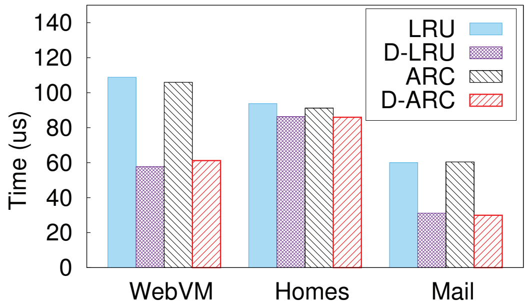


Figure 5: I/O latency from WebVM, Homes, and Mail with a cache size that is 40% of their respective WSS

same size of flash storage is used to store both data and metadata, so the comparison is fair. As discussed in Section 3.1, the Data Cache size can be traded for storing more historical fingerprints in the Metadata Cache as an optimization in the duplication-aware approaches. This tradeoff is studied in Section 5.4. In the other experiments, the Metadata Cache size is fixed at its minimum size (which is under 1% and 3% of the Data Cache size for D-LRU and D-ARC, respectively) plus an additional 2% taken from the Data Cache.

5.2 Performance

We evaluate the cache performance for each trace with different total cache sizes that are chosen at 20%, 40%, 60%, and 80% of its total working set size (WSS), which is listed in Table 2. Figure 3 compares the miss ratios of the WebVM trace. Results reveal that the duplication-aware approaches, D-LRU and D-ARC, outperform the alternatives by a significant margin for most cache sizes in terms of both read miss ratio and write miss ratio. Comparing the total miss ratio, D-LRU reduces it by up to 20% and D-ARC by up to 19.6%. Comparing LRU and ARC, ARC excels at small cache sizes, which is leveraged by D-ARC to also outperform D-LRU. For example, when the cache size is 20% of the trace's WSS, D-ARC has about 5% lower total miss ratio than D-LRU.

As discussed in Section 3, keeping historical source addresses and fingerprints in the Metadata Cache can help improve the hit ratio, because when the data block that a historical fingerprint maps to is brought back to the Data Cache, all the source addresses that map to this fingerprint can generate hits when they are referenced again. To quantify this benefit, we also measured the percentage of read hits that are generated by the historical metadata. For WebVM with a cache size that is 20% of its WSS, 83.25% of the read hits are produced by the historical metadata, which confirms the effectiveness of this optimization made by CacheDedup.

For the Homes and Mail traces, we show only the total miss ratio results to save space (Figure 4). Because the Mail trace is much more intensive than the other traces, we also show the results from the cache size that is 10% of its total WSS. Overall, D-ARC has the lowest total miss ratio, followed by D-LRU. D-ARC reduces the miss ratio by up to 5.4% and 3% compared to LRU and ARC, respectively in Homes and up to 2% and 1% in Mail. Compared to D-LRU, it reduces misses by up to 2.71% and 0.94% in Homes and Mail, respectively.

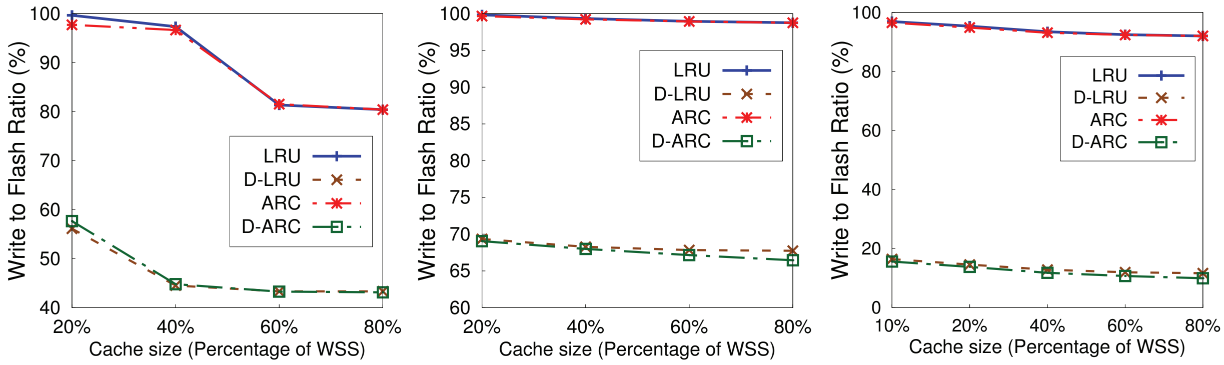
Figure 5 shows the average I/O latency from replaying the three traces with a cache size of 40% of their respective WSS. D-LRU and D-ARC deliver similar performance and reduce the latency by 47% and 42% compared to LRU and ARC, respectively for WebVM, 8% and 6% for Homes, and 48% and 51% for Mail. The improvement for Homes is smaller because of its much higher write-to-read ratio; the difference between a write hit and write miss is small when the storage server is not saturated. Note that the latency here does not include the fingerprinting time, which is $< 20\mu s$ per fingerprint, since the fingerprints are taken directly from the traces. But we cannot simply add this latency to the results here because many cache hits do not require fingerprinting. Instead, we evaluate this overhead in Section 5.7 using a benchmark.

5.3 Endurance

The results in Figure 6 confirm that the two duplication-aware approaches can substantially improve flash cache endurance by reducing writes sent to the flash device by up to 54%, 33%, and 89% for the WebVM, Homes, and Mail traces, respectively, compared to the traditional approaches. The difference between D-LRU and D-ARC is small. This interesting observation suggests that the scan-resistant nature of ARC does not help as much on endurance as it does on hit ratio. It is also noticeable that the flash write ratio decreases with increasing cache size, but the difference is small for Homes and Mail and for WebVM after the cache size exceeds 40% of its WSS. This can be attributed to two opposite trends: 1) an increasing hit ratio reduces cache replacements and the corresponding flash writes; 2) but when write hits bring new data into the cache they still cause flash writes, which is quite common for these traces.

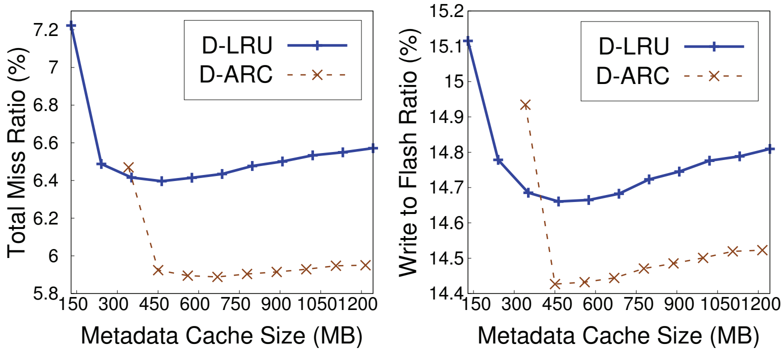
5.4 Sensitivity Study

We used the Mail trace to evaluate the impact of partitioning the shared flash cache space between the Data Cache and the



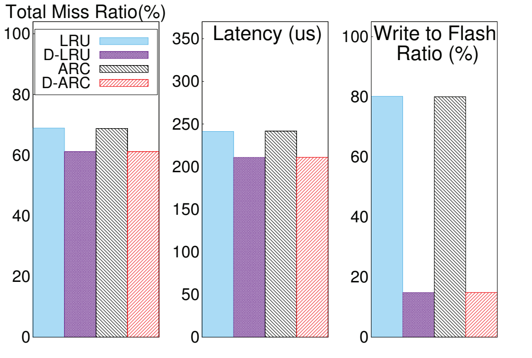
(a) WebVM **(b) Homes** **(c) Mail**

Figure 6: Writes to flash ratio



(a) Total miss ratio **(b) Writes to flash ratio**

Figure 7: D-LRU and D-ARC with varying Metadata/Data Cache space sharing for Mail



(a) **(b)** **(c)**

Figure 8: Results from concurrent Hadoop traces

Metadata Cache for the duplication-aware approaches. As discussed in Section 3.1, for a given Data Cache size, the minimum Metadata Cache size is what is required to store all the metadata of the cache data. But the Metadata Cache can take a small amount of extra space from the Data Cache for storing historical metadata and potentially improving performance. In this sensitivity study, we consider a cache size of 11GB which is 20% of Mail's WSS, and evaluate D-LRU and D-ARC while increasing the Metadata Cache size (and decreasing the Data Cache size accordingly). The results in Figure 7 show that both total miss ratio and wear-out initially improve with a growing Metadata Cache size. The starting points in the figure are the minimum Metadata Cache sizes for D-LRU and D-ARC (129MB for D-LRU and 341MB for D-ARC). Just by giving up 2% of the Data Cache space to hold more metadata (240MB of total Metadata Cache size for D-LRU and 450MB for D-ARC), the total miss ratio falls by 0.73% and 0.55% in D-LRU and D-ARC respectively, and the writes-to-flash ratio falls by 0.33% and 0.51%. The performance however starts to decay after having given up more than 3% of the Data Cache space, where the detrimental effect caused by having less data blocks starts to outweigh the benefit of being able to keep more historical metadata.

5.5 Concurrent Workloads

Next we evaluate CacheDedup's ability in handling concurrent workloads that share the flash cache and performing deduplication across them. We collected a set of VM I/O traces from a course where students conducted MapReduce programming projects using Hadoop. Each student group was given three datanode VMs and we picked three groups with substantial I/Os to replay. All the VMs were cloned from the same templates so we expect a good amount of duplicate I/Os to the Linux and Hadoop data. But the students worked on their projects independently so the I/Os are not identical. The statistics of these VM traces are listed as *Hadoop* in Table 2.

We replayed the last three days before the project submission deadline of these nine datanode VM traces concurrently, during which the I/Os are most intensive. The flash cache that they shared has a capacity of 40% of their total working set size. Figure 8 compares the performance of D-LRU and D-ARC to LRU and ARC. Overall, DLRU and DARC lower the miss ratio by 11% and the I/O latency by 12% while reducing the writes sent to the cache device by 81%. Notice that the reduction in flash write ratio is much higher than the reduction in miss ratio because, owing to the use of deduplication, a cache miss does not cause a write to the cache if the

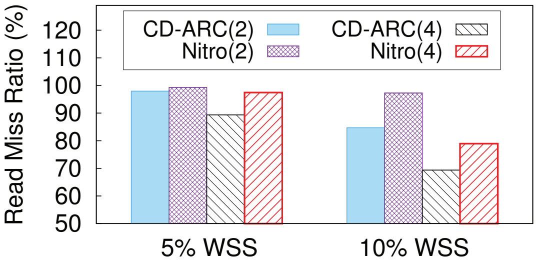


Figure 9: CD-ARC and Nitro for WebVM with a compression ratio of 2 and 4

requested data is a duplicate to the existing cached data.

5.6 Compression- and Duplication-aware Cache Management

Compression can be employed in addition to deduplication to further reduce the volume of data written to cache and improve cache performance and endurance. The recent work Nitro [19] was the first to combine these two techniques. It performs first deduplication and then compression on the data blocks. The compressed, variable-length data chunks (named extents) are packed into fixed-size blocks, named Write-Evict Units (WEU), and stored in cache. Cache replacement uses LRU at the granularity of WEUs. The size of a WEU is made the same as the flash device's erase block size to reduce garbage collection (GC) overhead. The fingerprints are managed using LRU separately from data replacement. If the primary storage also employs deduplication, Nitro can prefetch the fingerprints for identifying duplicates in future accesses.

CacheDedup is complementary to Nitro. On one hand, it can use the concept of WEU to manage compressed data in cache and reduce the flash device's internal GC overhead. On the other hand, CacheDedup can improve Nitro with its integrated cache management to reduce cache wastage and improve its performance and endurance. To prove this point, we created a version of D-ARC that is also compression-aware, named *CD-ARC*. It is still based on the integrated Metadata and Data Cache management architecture of CacheDedup. The main differences from D-ARC are that 1) the Data Cache stores WEUs; 2) the fingerprints in the Metadata Cache point to the extents in the WEUs; and 3) replacement in the Data Cache preferably uses a WEU with no mappings in $T_1 \cup T_2$.

We compare CD-ARC to a Nitro implementation without fingerprint prefetching because prefetching is an orthogonal technique that can be used by both approaches. But we extend Nitro to also cache historical fingerprints, so the comparison is fair. We also set the same limits on the two algorithms' data cache capacity and metadata space usage. We present the results from a 2MB WEU size with a compression ratio of 2 and 4, and report the read miss ratios in Figure 9 for the WebVM trace. CD-ARC improves the read hit ratio by up to 12.56% compared to Nitro. This improvement can be largely attributed to CD-ARC's scan-resistant and adaptive properties inherited from ARC, which is possible only because of the integrated cache and deduplication design of our approach.

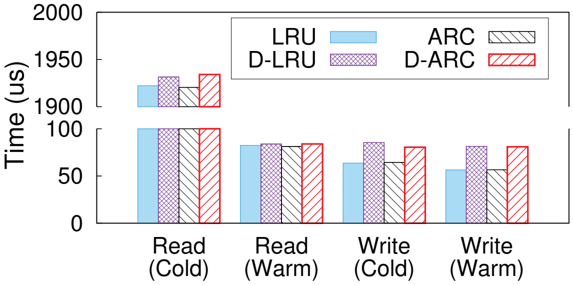


Figure 10: FIO latency with random reads and writes

Although compression can further reduce a workload's cache footprint and wear-out, it also adds additional overhead. Moreover, for a write-intensive workload which has a large number of updates, the use of compression also introduces wastage, because the updated data cannot be written in place in the cache. For example, for WebVM with a cache size that is 20% of its WSS and a compression ratio of 2, CD-ARC achieves only 8% higher read hit ratio, because in average 46% of the cache capacity is occupied by invalid data caused by updates.

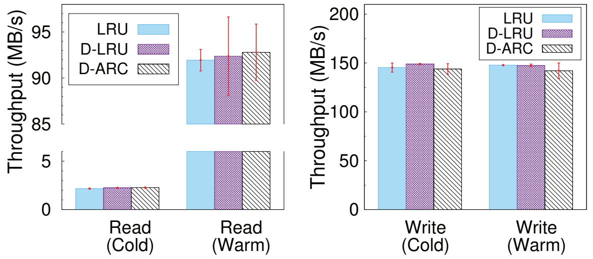
5.7 Overhead

Finally, we used an I/O benchmark fio [1] to measure CacheDedup's overhead compared to a stand-alone caching layer that does not use deduplication. The benchmark issued random reads or writes with no data duplication, so the results reveal the worst-case performance of CacheDedup. Direct I/O is used to bypass the main memory cache. First, we used a single fio thread with 1GB of random I/Os. Figure 10 shows that D-LRU and D-ARC adds a $10-20\mu$s latency to LRU and ARC for writes and for reads when the cache is cold, which is mainly the overhead of creating the fingerprint.

Although this fingerprinting overhead is considerable, in typical workloads, concurrent I/Os' fingerprinting operations can be overlapped by their I/Os and become insignificant in the overall performance. To demonstrate this, in the next experiment, we used eight concurrent fio threads each issuing 512MB of I/Os to evaluate the throughput overhead. Figure 11 shows that CacheDedup does not have significant overhead in terms of throughput. Moreover, CacheDedup's hit ratio gain and the corresponding performance improvement (as shown in the previous experiments) will significantly outweigh the fingerprinting overhead.

6 Related Work

A variety of research and commercial solutions have shown the effectiveness of flash caching for improving the performance of storage systems [7, 2, 5]. Compared to traditional main-memory-based caching, flash caching differs in its rel-



(a) Read throughput **(b) Write throughput**

Figure 11: FIO throughput with 8 concurrent threads

atively larger capacity and ability to store data persistently. Therefore, related work revisited several key design decisions for flash caching [11, 5]. In particular, it has been observed that write-through caching can provide good performance when the writes are submitted asynchronously to the backend storage, whereas write-back caching can reduce the I/O load on the backend and further improve the performance. The solution proposed here, CacheDedup, supports both write policies. Related work on the allocation of shared cache capacity among concurrent workloads [23, 20] is also complementary to CacheDedup's support for reducing each workload's footprint through in-line deduplication.

Deduplication has been commonly used to reduce the data footprint in order to save the bandwidth needed for data transfer and the capacity needed for data storage. Related solutions have been proposed to apply deduplication at different levels of a storage hierarchy, including backup and archival storage [27, 22], primary storage [12, 15], and main memory [33]. The wear-out issue of flash devices has motivated several flash deduplication studies [8, 9, 16] which show that it is a viable solution to improving endurance. Compared to these related efforts, CacheDedup addresses the unique challenges presented by caching. First, caching needs to process both reads and writes, while for primary storage only writes create new data and need to be considered. Second, the management of deduplication cannot be dissociated from cache management issues if localities have to be captured.

As discussed in Section 2, although a solution that simply stacks a standalone deduplication layer upon a standalone caching layer could also work, it would have a much higher metadata space overhead because both layers have to maintain the source block addresses, and the deduplication layer also has to manage the fingerprints for the entire source device. Moreover, such simple stacking cannot support a more sophisticated cache replacement algorithm such as ARC, which is made possible in D-ARC because of its integrated cache and deduplication management.

Nitro [19] is a closely related work which combines deduplication and compression to manage a flash cache employed at the server-side of a network file system. As discussed in Section 5.6, CacheDedup is complementary to Nitro in that our proposed architecture and algorithms can be incorporated to create a compression- and duplication-aware caching solution with further improved cache hit ratio and endurance.

As small random writes can decrease the throughput and device lifespan of a flash cache, related work RIPQ [32] proposed several techniques to address this problem, including aggregating the small random writes, which is similar to the WEU technique of Nitro, and our CD-ARC. It is also conceivable to apply WEU to D-LRU and D-ARC to aggregate small writes in order to sustain cache throughput and further improve flash endurance.

Related work studied cache admission policies to reduce flash wear-out [34, 14]. By not caching data with weak temporal locality, they showed improvements in endurance. Suei *et al.* [31] created a device-level cache partition design to distribute frequently-updated data into different erase blocks and lower the chances of blocks to be worn-out soon. These solutions are complementary to CacheDedup's focus on optimizing the use of deduplication for improving endurance.

7 Conclusions

This paper presents CacheDedup, a first study on integrating deduplication with flash caching using duplication-aware cache management. The novelties lie in a new architectural design that seamlessly integrates the caching of metadata and data, and new cache replacement algorithms D-LRU and D-ARC that allow the optimization for both performance and endurance. The paper offers an in-depth study of these algorithms with both theoretical analysis and experimental evaluation, which proves their no-cache-wastage property and shows the improvement on cache hit ratio, I/O latency, and the amount of writes sent to the cache device.

Between the two algorithms, D-ARC achieves the best performance, and D-LRU is attractive because of its simplicity. Both are efficient in terms of time and space usage. CacheDedup is a versatile framework for enabling various algorithms, including one (CD-ARC) that improves the use of compression with deduplication. As its design is not specific to flash devices, we believe that the CacheDedup approach can be also applied to new non-volatile memory technologies and improve their performance and endurance when used for caching.

8 Acknowledgements

We thank the anonymous reviewers and our shepherd, Geoff Kuenning, for their thorough reviews and insightful suggestions, and our colleagues at the VISA Research Lab, Dulcardo Arteaga, for his help with the caching framework, and Saman Biook Aghazadeh for his support of this paper including collecting the Hadoop traces. This research is sponsored by National Science Foundation CAREER award CNS-125394 and Department of Defense award W911NF-13-1-0157.

References

[1] Fio — Flexible I/O Tester Synthetic Benchmark. http://git.kernel.dk/?p=fio.git.

[2] Fusion-io ioCache. http://www.fusionio.com/products/iocache/.

[3] Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[4] Federal Information Processing Standards (FIPS) publication 180-1: Secure Hash Standard. National Institute of Standards and Technology (NIST), April 17, 1995.

[5] D. Arteaga and M. Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR)*, pages 7:1–7:11, New York, NY, USA, 2014. ACM.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, 2003.

[7] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST)*, Pacific Grove, CA, USA, 2012. IEEE.

[8] 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 *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, volume 11, 2011.

[9] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 91–103, 2011.

[10] E. V. Hensbergen and M. Zhao. Dynamic policy disk caching for storage networking. Technical Report RC24123, IBM, November 2006.

[11] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference (ATC)*. USENIX Association, 2013.

[12] B. Hong, D. Plantenberg, D. D. Long, and M. Sivan-Zimet. Duplicate data elimination in a SAN file system. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, pages 301–314, 2004.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[14] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceeding of 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2013.

[15] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of International Conference on Systems and Storage (SYSTOR)*, page 7. ACM, 2009.

[16] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H.-u. Lee, S. Kang, Y. Won, and J. Cha. Deduplication in SSDs: Model and quantitative analysis. In *Proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[17] R. Koller, L. Marmol, R. Ranganswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, 2013.

[18] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[19] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 501–512. USENIX Association, 2014.

[20] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-CAVE: Effective SSD caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 103–112. IEEE Press, 2013.

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[22] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of The Israeli Experimental Systems Conference (SYSTOR)*, page 8. ACM, 2009.

[23] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[24] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, 1988.

[25] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484. ACM, 2014.

[26] D. Qin, A. D. Brown, and A. Goel. Reliable write-

back for client-side flash caches. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 451–462. USENIX Association, 2014.

[27] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, volume 2, pages 89–101, 2002.

[28] R. Rivest. The MD5 message-digest algorithm. 1992.

[29] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, pages 267–280, New York, NY, USA, 2012. ACM.

[30] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communication. ACM*, 28(2):202–208, Feb. 1985.

[31] P.-L. Suei, M.-Y. Yeh, and T.-W. Kuo. Endurance-aware flash-cache management for storage servers. *IEEE Transactions on Computers (TOC)*, 63:2416–2430, 2013.

[32] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[33] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[34] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, page 10. ACM, 2013.