Opening the Chrysalis: On the Real Repair Performance of MSR Codes

Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, and Cyril Gyuot, ***WD Research;***
Eyal En Gad, ***University of Southern California;*** **Zvonimir Bandic,** ***WD Research***

https://www.usenix.org/conference/fast16/technical-sessions/presentation/pamies-juarez

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

Open access to the Proceedings of the 14th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Opening the Chrysalis: On the Real Repair Performance of MSR Codes

Lluis Pamies-Juarez*, Filip Blagojević*, Robert Mateescu*,
Cyril Guyot*, Eyal En Gad**, and Zvonimir Bandic*

*WD Research, **University of Southern California
{*lluis.pamies-juarez, filip.blagojevic, robert.mateescu, cyril.guyot, zvonimir.bandic*}*@hgst.com*,
engad@usc.edu

Abstract

Large distributed storage systems use erasure codes to reliably store data. Compared to replication, erasure codes are capable of reducing storage overhead. However, repairing lost data in an erasure coded system requires reading from many storage devices and transferring over the network large amounts of data. Theoretically, Minimum Storage Regenerating (MSR) codes can significantly reduce this repair burden. Although several explicit MSR code constructions exist, they have not been implemented in real-world distributed storage systems. We close this gap by providing a performance analysis of Butterfly codes, systematic MSR codes with optimal repair I/O. Due to the complexity of modern distributed systems, a straightforward approach does not exist when it comes to implementing MSR codes. Instead, we show that achieving good performance requires to vertically integrate the code with multiple system layers. The encoding approach, the type of inter-node communication, the interaction between different distributed system layers, and even the programming language have a significant impact on the code repair performance. We show that with new distributed system features, and careful implementation, we can achieve the theoretically expected repair performance of MSR codes.

1 Introduction

Erasure codes are becoming the redundancy mechanism of choice in large scale distributed storage systems. Compared to replication, erasure codes allow reduction in storage overhead, but at a higher repair cost expressed through excessive read operations and expensive computation. Increased repair costs negatively affect the Mean Time To Data Loss (MTTDL) and data durability.

New coding techniques developed in recent years improve the repair performance of classical erasure codes (e.g. Reed-Solomon codes) by reducing excessive network traffic and storage I/O. Regenerating Codes (RGC) [13] and Locally Repairable Codes (LRC) [19] are the main representatives of these advanced coding techniques. RGCs achieve an optimal trade-off between the storage overhead and the amount of data transferred (*repair traffic*) during the repair process. LRCs offer an optimal trade-off between storage overhead, fault tolerance and the number of nodes involved in repairs. In both cases, the repairs can be performed with a fraction of the read operations required by classical codes.

Several explicit LRC code constructions have been demonstrated in real world production systems [20, 35, 28]. LRCs are capable of reducing the network and storage traffic during the repair process, but the improved performance comes at the expense of requiring extra storage overhead. In contrast, for a fault tolerance equivalent to that of a Reed-Solomon code, RGCs can significantly reduce repair traffic [28] without increasing storage overhead. This specifically happens for a subset of RGCs operating at the Minimum Storage Regenerating tradeoff point, i.e. MSR codes. At this tradeoff point the storage overhead is minimized over repair traffic. Unfortunately, there has been little interest in using RGCs in real-world scenarios. RGC constructions of interest, those with the storage overhead below $2\times$, require either encoding/decoding operations over an exponentially growing finite field [8], or an exponential increase in number of sub-elements per storage disk [14, 31]. Consequently, implementation of RGCs in production systems requires dealing with complex and bug-prone algorithms. In this study we focus on managing the drawbacks of RGC-MSR codes. We present the first MSR implementation with low-storage overhead (under $2\times$), and we explore the design space of distributed storage systems and characterize the most important design decisions affecting the implementation and performance of MSRs.

Practical usage of MSR codes equals the importance of a code design. For example, fine-grain read accesses introduced by MSR codes may affect performance negatively and reduce potential code benefits. Therefore, understanding the advantages of MSR codes requires

characterizing not only the theoretical performance, but also observing the effect the code has on real-world distributed systems. Because of the complex and multi--layer design of distributed storage systems, it is also important to capture the interaction of MSR codes with various system layers, and pinpoint the system features that improve/degrade coding performance.

We implement an MSR code in two mainstream distributed storage systems: HDFS and Ceph. These are the two most widely used systems in industry and academia, and are also based on two significantly different distributed storage models. Ceph does *online* encoding while data is being introduced to the system. HDFS performs encoding as a *batch job*. Moreover, Ceph applies erasure codes in a per-object basis whereas HDFS does it on groups of objects of the same size. And finally, Ceph has an open interface to incorporate new code implementations in a pluggable way, while HDFS has a monolithic approach where codes are embedded within the system.

The differences between HDFS and Ceph allow us to cover an entire range of system design decisions that one needs to make while designing distributed storage systems. The design observations presented in this study are intended for designing future systems that are built to allow effortless integration of MSR codes. To summarize, this paper makes the following contributions: (i) We design a recursive Butterfly code construction —a two-parity MSR code— and implement it in two real-world distributed storage systems: HDFS and Ceph. Compared to other similar codes, Butterfly only requires XOR operations for encoding/decoding, allowing for more efficient computation. To the best of our knowledge, these are the first implementations of a low overhead MSR code in real-world storage systems. (ii) We compare two major approaches when using erasure codes in distributed storage systems: *online* and *batch*-based encoding and point the major tradeoffs between the two approaches. (iii) We examine the performance of Butterfly code and draw a comparison between the theoretical results of MSR codes and the performance achievable in real systems. We further use our observations to suggest appropriate distributed system design that allows best usage of MSR codes. Our contributions in this area include *communication vectorization* and a plug-in interface design for pluggable MSR encoders/decoders.

2 Background

In this section we introduce erasure coding in large-scale distributed storage systems. In addition we provide a short overview of HDFS and Ceph distributed filesystems and their use of erasure codes.

2.1 Coding for Distributed Storage

Erasure codes allow reducing the storage footprint of distributed storage systems while providing equivalent or even higher fault tolerance guarantees than replication. Traditionally, the most common type of codes used in distributed systems were Reed-Solomon (RS) codes. RS are well-known maximum distance separable (MDS) codes used in multiple industrial contexts such as optical storage devices or data transmission. In a nutshell, Reed-Solomon codes split each data object into k chunks and generate r linear combinations of these k chunks. Then, the $n = k + r$ total chunks are stored into n storage devices. Finally, the original object can be retrieved as long as k out of the n chunks are available.

In distributed storage systems, achieving long MTTDL and high data durability requires efficient data repair mechanisms. The main drawback of traditional erasure codes is that they have a costly repair mechanism that compromises durability. Upon a single chunk failure, the system needs to read k out of n chunks in order to regenerate the missing part. The repair process entails a k to 1 ratio between the amount of data read (and transferred) and the amount of data regenerated. Regenerating Codes (RGC) and Locally Repairable Codes (LRC) are two family of erasure codes that can reduce the data and storage traffic during the regeneration. LRCs reduce the number of storage devices accessed during the regeneration of a missing chunk. However, this reduction results in losing the MDS property, and hence, relaxing the fault tolerance guarantees of the code. On the other hand, RGCs aim at reducing the amount of data transferred from each of the surviving devices, at the expense of increased number of devices contacted during repair. Additionally, when RGCs minimize the repair traffic without any additional storage overhead, we say that the code is a Minimum Storage Regenerating (MSR) code.

LRCs have been demonstrated and implemented in production environments [20, 35, 28]. However, the use of LRCs in these systems reduces the fault tolerance guarantees of equivalent traditional erasure codes, and cannot achieve the minimum theoretical repair traffic described by RGCs. Therefore, RGCs seem to be a better option when searching for the best tradeoff between storage overhead and repair performance in distributed storage systems. Several MSR codes constructions exist for rates smaller that 1/2 (i.e. $r \geq k$) [26, 23, 29, 30], however, designing codes for higher rates (more storage efficient regime) is far more complex. Although it has been shown that codes for arbitrary (n,k) values can be asymptotically achieved [9, 30], explicit finite code constructions require either storing an exponentially growing number of elements per storage device [7, 31, 24, 14], or increasing the finite field size [27]. To the best of our knowledge, Butterfly codes [14] are the only codes that

allow a two-parity erasure code ($n-k=2$) over a small field (i.e. GF(2)), and hence, they incur low computational overhead. The relatively simple design and low computational overhead make Butterfly codes a good candidate for exploring the challenges of implementing a MSR code in real distributed storage systems.

2.2 Hadoop Filesystem

Hadoop is a scalable runtime designed for managing large-scale computation/storage systems. Hadoop supports a map-reduce computational model and therefore is very suitable for algorithms that target processing of large amounts of data. The Hadoop filesystem (HDFS) is its default storage backend, and was initially developed as an open source version of the Google filesystem [17] (GFS), containing many of the features initially designed for GFS. HDFS is currently one of the most widely used distributed storage systems in industrial and academic deployments.

In HDFS there are two types of physical nodes: the Namenode server and multiple Datanode servers. The Namenode server contains various metadata as well as the location information about all data blocks residing in HDFS, whereas Datanode servers contain the actual blocks of data. The "centralized metadata server" architecture lowers the system design complexity, but poses certain drawbacks: (i) Limited metadata storage This problem has been addressed by recent projects (such as Hadoop Federation [4]) that allow multiple namespaces per HDFS cluster. (ii) Single point of failure - In case of the Namenode failure, the entire system is unaccessible until the Namenode is repaired. Recent versions of HDFS address this problem by introducing multiple redundant Namenodes, allowing fast failover in case the Namenode fails.

Starting with the publicly available Facebook's implementation of a Reed-Solomon code for HDFS [2], Hadoop allows migration of replicated data into more storage efficient encoded format. The erasure code is usually used to reduce the replication factor once the data access frequency reduces. Hence, the encoding process in HDFS is not a real-time task, instead it is performed in the background, as a batch job. While the batch-based approach provides low write latency, it also requires additional storage where the intermediate data resides before being encoded.

2.3 Ceph's Distributed Object Store

Ceph [32] is an open source distributed storage system with a decentralized design and no single point of failure. Like HDFS, Ceph is self-healing and a self-managing system that can guarantee high-availability and consistency with little human intervention. RADOS [34] (Reliable, Autonomic Distributed Object Store) is Ceph's core component. It is formed by a set of daemons and libraries that allow users accessing an object-based storage system with partial and complete read/writes, and snapshot capabilities. RADOS has two kinds of daemons: monitors (MONs), that maintain consistent metadata, and object storage devices (OSDs). A larger cluster of OSDs is responsible to store all data objects and redundant replicas. Usually a single OSD is used to manage a single HDD, and typically multiple OSDs are collocated in a single server.

RADOS storage is logically divided into object containers named *pools*. Each pool has independent access control and redundancy policies, providing isolated namespaces for users and applications. Internally, and transparent to the user/application, pools are divided into subsets of OSDs named *placement groups*. The OSDs in a placement group run a distributed leader-election to elect a *Primary* OSD. When an object is stored into a pool, it is assigned to one placement group and uploaded to its Primary OSD. The Primary OSD is responsible to redundantly store the object within the placement group. In a replicated pool this means forwarding the object to all the other OSDs in the group. In an erasure encoded pool, the Primary splits and encodes the object, uploading the corresponding chunks to the other OSDs in the group. Hence, the encoding process in Ceph is performed as real-time job, i.e. the data is encoded while being introduced into the system. The placement group size directly depends on the number of replicas or the length of the code used. OSDs belong to multiple placement groups, guaranteeing good load balancing without requiring large amount of computing resources. Given the cluster map, the pool policies, and a set of fault domain constraints, RADOS uses a consistent hashing algorithm [33] to assign OSDs to placement groups, and map object names to placement groups within a pool.

3 Butterfly Codes

Vector codes are a generalization of classical erasure codes where k α-dimensional data vectors are encoded into a codeword of n α-dimensional redundant vectors, for $n > k$. As it happens for classical erasure codes, we say that a vector code is systematic if the original k vectors form a subset of the n codeword vectors, that is, the codeword only adds $n-k$ redundant vectors. In this paper we refer to the codeword vectors as code columns, and to the vector components as column elements.

Butterfly Codes are an MDS vector code construction for two-parities (i.e. $n-k=2$) of an explicit Regenerating Code operating at the minimum storage regenerating (MSR) point. This means that to repair a single disk failure, Butterfly codes require to transfer $1/2$ of all the remaining data, which is optimal. Additionally, Butterfly codes are binary vector codes defined over $GF(2)$, allow-

ing implementation of encoding and decoding operations by means of simple exclusive-or operations.

A preliminary construction of the Butterfly code was presented earlier [14], and in this section we provide a new recursive construction of the code. Compared to the original construction, the recursive approach has a simplified design that results in a simpler implementation. Furthermore, the recursive design partitions the problem in a way that allows for a better reuse of precomputed values, leading to better cache locality. Due to space limitations we omit the exhaustive cache behavior analysis.

3.1 Butterfly Encoder

Let D_k be a matrix of boolean values of size $2^{k-1} \times k$, for $k \geq 2$. D_k represents a data object to be encoded and stored in the distributed storage system. For the purpose of describing the encoding/decoding process, we represent D_k by the following components:

$$D_k = \begin{bmatrix} \mathbf{a} & A \\ \mathbf{b} & B \end{bmatrix}, \tag{1}$$

where A and B are $2^{k-2} \times k-1$ boolean matrices, and $\mathbf{a}$ and $\mathbf{b}$ are column vectors of 2^{k-2} elements.

Let D_k^j be the jth column of D_k, $j \in \{0,\ldots,k-1\}$. Therefore, the matrix D_k can be written as a vector of k columns $D_k = (D_k^{k-1},\ldots,D_k^0)$. From a traditional erasure code perspective each of the columns is an element of GF(2^{k-1}), and after encoding we get a systematic codeword $C_k = (D_k^{k-1},\ldots,D_k^0,H,B)$, where H and B are two vector columns representing the horizontal and butterfly parities respectively.

To describe how to generate H and B, we define two functions such that $H = \mathcal{H}(D_k)$ and $B = \mathcal{B}(D_k)$:

- if $k = 2$, then:

$$\mathcal{H}\left(\begin{bmatrix} d_0^1 & d_0^0 \\ d_1^1 & d_1^0 \end{bmatrix}\right) = \begin{bmatrix} d_0^1 \oplus d_0^0 \\ d_1^1 \oplus d_1^0 \end{bmatrix}; \tag{2}$$

$$\mathcal{B}\left(\begin{bmatrix} d_0^1 & d_0^0 \\ d_1^1 & d_1^0 \end{bmatrix}\right) = \begin{bmatrix} d_1^1 \oplus d_0^0 \\ d_0^1 \oplus d_0^0 \oplus d_1^0 \end{bmatrix}. \tag{3}$$

- if $k > 2$, then:

$$\mathcal{H}(D_k) = \begin{bmatrix} \mathbf{a} \oplus \mathcal{H}(A) \\ P_{k-1}\left[P_{k-1}\mathbf{b} \oplus \mathcal{H}(P_{k-1}B)\right] \end{bmatrix}; \tag{4}$$

$$\mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\mathbf{b} \oplus \mathcal{B}(A) \\ P_{k-1}\left[\mathbf{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)\right] \end{bmatrix}, \tag{5}$$

  where P_k represents a $k \times k$ permutation matrix where the counter-diagonal elements are one and all other elements are zero. Notice that left-multiplication of a vector or a matrix by P_k flips the matrix vertically.

It is interesting to note that the double vertical flip in (4) is intentionally used to simultaneously compute $\mathcal{H}$ and $\mathcal{B}$ over the same data D_k. Because of the double vertical flip, the recursion can be simplified, and encoding of D_k can be done by encoding A and $P_{k-1}B$. In Figure 1 we show an example of the recursive encoding for $k = 4$.

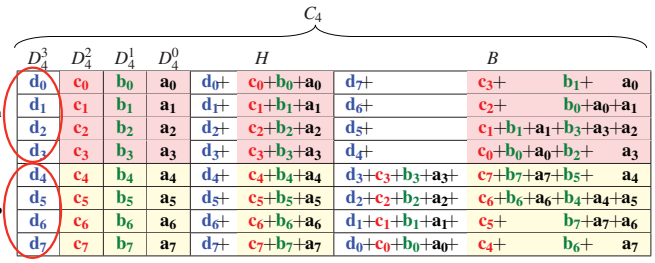


Figure 1: Butterfly codeword for $k = 4$, C_4. One can observe how C_4 can be computed by recursively encoding submatrix A (red highlight) and B (yellow highlight) from (1) and adding the extra non-highlighted elements.

3.2 Butterfly Decoder

In this section we show that Butterfly code can decode the original data matrix when any two of the codeword columns are missing, and hence it is an MDS code.

Theorem 1 (MDS)**.** *The Butterfly code can recover from the loss of any two columns (i.e. two erasures).*

Proof. The proof is by induction over the number of columns, k. In the *base case*, $k = 2$, one can carefully verify from (2) and (3) that the code can recover from the loss of any two columns. The *inductive step* proceed as follows. Let's assume that the Butterfly construction gives an MDS code for $k-1$ columns, for $k > 2$. We will prove that the construction for k columns is also MDS. We distinguish the following cases:

(1) The two parity nodes are lost. In this case we encode them again through $\mathcal{H}$ and $\mathcal{B}$ functions.

(2) One of the parities is lost, along with one data column. In this case we can use the remaining parity node to decode the lost data column, and then re-encode the missing parity node.

(3) Two data columns are lost, neither of which is the leftmost column. In this case we can generate from the parity columns the vectors $\mathcal{H}(A)$, $\mathcal{B}(A)$, by XOR-ing $\mathbf{a}$ and $P_{k-1}\mathbf{b}$. By using the inductive hypothesis, we can recover the top half of the missing columns (which is part of the A matrix). Similarly, we can generate by simple XOR the values $\mathcal{H}(P_{k-1}B)$ and $\mathcal{B}(P_{k-1}B)$. By the induction hypothesis we can recover the bottom half of the missing columns (which is part of the B matrix).

(4) The leftmost column along with another data column D_k^j, $j \neq k-1$, are lost. From the bottom half of the butterfly parity $\mathcal{B}(D_k)$ we can obtain $\mathcal{B}(P_{k-1}B)$, and then decode the bottom half of D_k^j. From the bottom half of the horizontal parity $\mathcal{H}(D_k)$ we can now decode $\mathbf{b}$. Following the decoding chain, from the top half of the butterfly

parity $\mathcal{B}(D_k)$ we can obtain $\mathcal{B}(A)$, and then decode the top half of D_k^j. Finally, from the top half of the horizontal parity $\mathcal{H}(D_k)$ we can obtain **a**. □

Theorem 2 (optimal regeneration)**.** *In the case of one failure, the lost column can be regenerated by communicating an amount of data equal to* $1/2$ *of the remaining data (i.e.,* $1/2$ *of* $k+1$ *columns). If the lost column is not the butterfly parity, the amount of communicated data is exactly equal to the amount read form surviving disks (i.e. optimal I/O access).*

Due to space constrains we do not provide a proof here. Instead, in the next section we provide the details required to regenerate any of the codeword columns.

3.3 Single Column Regeneration

The recovery of a single column falls under four cases. Note that in all of them, the amount of data that is transferred is optimal, and equal to half of the remaining data. Moreover, the amount of data that is accessed (read) is also optimal (and equal to the data that is transferred), except in case (4) when we recover the butterfly parity. Case (1) is the most common. The data to be transferred is selected by algebraic expressions, but there are more intuitive ways to understand the process. The indeces correspond to locations in the butterfly parity that do not require the additional elements; similarly, they correspond to inflexion points in the butterfly lines (v-points); also they correspond to 0 value for bit $j-1$ of the binary representation of numbers ordered by the reflected Gray code. Finally, the recovery in case (4) is based on a self-duality of the butterfly encoding.

(1) One column from $\{D_k^1,\ldots,D_k^{k-1}\}$ is lost Let D_k^j be the lost columnn. Every remaining column (systematic data and parities), will access and transfer the elements in position i for which $\left\lfloor \frac{i}{2^{j-1}} \right\rfloor \equiv 0 \mod 4$, or $\left\lfloor \frac{i}{2^{j-1}} \right\rfloor \equiv 3 \mod 4$. Let D_{k-1} be the matrix of size $(k-1)\times 2^{k-2}$ formed from the transmitted systematic data, and H_{k-1}, B_{k-1} the columns of size 2^{k-2} formed from the transmitted parity information. Let $h = \mathcal{H}(D_{k-1}) \oplus H_{k-1}$ and $b = \mathcal{B}(D_{k-1}) \oplus B_{k-1}$ (i.e., we use butterfly encoding on the matrix D_{k-1}). The data lost from D_j is now contained by h and b. More precisely, for $i \in \{0,\ldots,2^{k-2}\}$, let $p = \left\lfloor \frac{i+2^{j-1}}{2^j} \right\rfloor 2^j + i$, and let $r = p \mod 2^j$. Then $D_k^j(p) \leftarrow h(i)$, and $D_k^j(p-2r+2^j-1) \leftarrow b(i)$.

(2) Column D_k^0 is lost In this case, the columns $D_k^1,\ldots,D_k^{k-1},H$ will access and transfer the elements with even index, and the column B will access and transfer the elements with odd index. Similar to case (1), the vectors h and b are obtained by applying butterfly encoding, and they provide the even, repectively odd, index elements of the lost column.

(3) First parity column H is lost All the remaining columns access and transfer their lower half, namely all the elements with index $i \in \{2^{k-2},\ldots,2^{k-1}-1\}$. The horizontal parity over the systematic transmitted data provides the lower half of H, and the butterfly parity over $D_{k-1}^0,\ldots,D_{k-1}^{k-2}$ XOR-ed with data from B will provide the top half of H.

(4) Second parity column B is lost In this case D_k^{k-1} will access and transfer its top half, while H will do the same with its bottom half. The rest of the columns $D_k^0,\ldots,D_k^{k-2}$ will access all of their data, but they will perform XOR operations and only transfer an amount of data equal to half of their size. Each D_k^j for $j \neq k-1$ will compute and transfer values equal to their contributions in the bottom half of B. Therefore a simple XOR operation between the data transferred from the systematic columns will recover the bottom half of B. Interestingly, computing a butterfly parity over the data transferred from D_k^j, where $j \neq k-1$, and XOR-ing it correspondingly with the bottom half of H will recover the top half of B.

4 Butterfly Codes in HDFS

To avoid recursion in Java, and possible performance drawbacks due to non-explicit memory management, in HDFS we implement an iterative version of Butterfly [14]. Our implementation of Butterfly code in HDFS is based on publicly available Facebook's Hadoop [2] version. In this section we provide implementation and optimization details of our Butterfly implementation.

4.1 Erasure Coding in HDFS

We use the Facebook HDFS implementation as a starting point for the Butterfly implementation. Facebook version of Hadoop contains two daemons, RaidNode and BlockFixer, that respectively create parity files and fix corrupted data. Once inserted into the HDFS, all files are initially replicated according to the configured replication policy. The RaidNode schedules map-reduce jobs for erasure encoding the data. The encoding map-reduce jobs take groups of k newly inserted chunks, and generate $n-k$ parity chunks, as presented in Figure 2. The parity chunks are then stored back in HDFS, and the replicas can be garbage collected. Lost or corrupted data is detected and scheduled for repair by the BlockFixer daemon. The repair is performed using map-reduce decode tasks. Upon decoding completion, the reconstructed symbol is stored back to HDFS.

4.2 Butterfly Implementation in HDFS

The encoding and repair process in HDFS-Butterfly follows a 4-step protocol: (i) in the first step the encoding/decoding task determines the location of the data blocks that are part of the k symbol message; (ii) the sec-

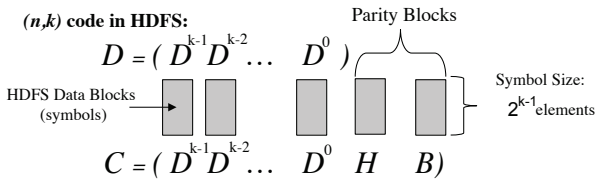


Figure 2: Erasure Coding (k,n) in HDFS: (i) each symbol is part of a separate HDFS block, (ii) k symbols represent a message, n symbols represent a codeword (initial message blocks + parity blocks), (ii) In Butterfly, each symbol is divided in $\alpha = 2^{k-1}$ elements.

ond step assumes fetching the data to the node where the task is running; (iii) in the third step the encoding/decoding computation is performed, and finally (iv) the newly created data is committed back to HDFS.

The first step, locating the data necessary for encoding/decoding process, is identical to the initial Facebook implementation: position of the symbol being built is used to calculate the HDFS file offset of the entire *k-symbol message/codeword* (data necessary for building the symbol). Calculating the offsets is possible because the size of the data being repaired equals the size of an entire HDFS block. In case the symbol being repaired is smaller than the HDFS block size, we rebuild all the symbols contained in that HDFS block. Therefore, we always fetch k consecutive HDFS blocks – *k-symbol message*. The location of the parity symbols/blocks during the decoding is determined using similar approach.

The second step, data fetching, is performed asynchronously and in parallel, from multiple datanodes. The size of the fetched data is directly related to the Butterfly message size, i.e. set of butterfly symbols spread across different datanodes. We allow the size of a Butterfly symbol to be a configurable parameter. We set the symbol element size to $\ell = symbol\ size/2^{k-1}$. The advantage of tunable symbol size is twofold: (i) improved data locality: size of the data chunks used in computation can be tuned to fit in cache; (ii) computation - communication overlap: "rightszing" the data chunk allows communication to be completely overlapped by computation.

The third step implements Butterfly encoding and decoding algorithms. While Section 3.2 presents formal definition of Butterfly, in Figure 3 we describe an example of Butterfly encoding/decoding schemes in HDFS. Figure 3 is intended to clarify the encoding/decoding process in HDFS-Butterfly through a simple example, and encoding/decoding of specific components might be somewhat different. Our encoding/decoding implementation is completely written in Java. While moving computation to a JNI module would significantly increase the level of applicable optimizations (including vectorization), these benefits would be shadowed by the cost of data movements between Java and JNI modules.

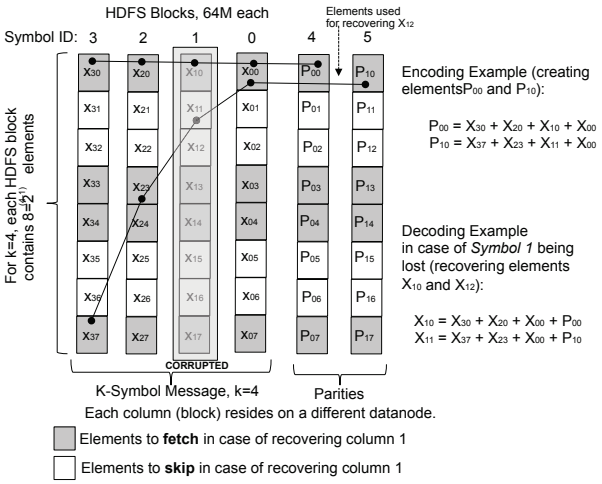


Figure 3: Regenerating column 1 for HDFS-Butterfly with $k = 4$. The line represents the elements that are xored to regenerate the first two symbols. The dark gray elements are the total required ones. White elements are skipped during communication.

Our Java computation is aggressively optimized through manual loop reordering and unrolling but we saw little benefits from these optimizations. By instrumenting the computational code, we found that the time spent in memory management significantly outweighs the benefits computational optimizations. We also parallelize computational loops in the OpenMP fashion. The degree of parallelization is a configurable parameter.

During the fourth step, the encoding/decoding task detects physical location of all the symbols (HDFS blocks) contained in the *k-symbol message*. The newly created symbol is placed on a physical node that does not contain any of the other message symbols, i.e. we avoid collocating two symbols from the same message on the same physical node. In this way we increase the system reliability in case of a singe node failure.

4.3 Communication Protocol

HDFS is a streaming filesystem and the client is designed to receive large amounts of contiguous data. If the data stream is broken, client assumes communication error and starts an expensive process of re-establishing connection with the datanode. However, the Butterfly repair process does not read remote data in a sequential manner. As explained in Figure 3, not all of the vector-symbol's elements are used during the decoding process (in Figure 3 only gray elements are used for reconstructing *Symbol 1*). The elements used for decoding can change, depending on the *symbol ID* being repaired. In our initial implementation, we allowed the datanode to *skip* reading unnecessary symbol components and send back only useful data (method `sendChunks()` in the HDFS datanode implementation). Surprisingly, this approach resulted in very low performance due to the interruptions

in client–datanode data stream.

To avoid the HDFS overhead associated with streaming non-contiguous data, we implement *vector communication* between the client and the datanode: datanode packs all non-contiguous data chunks into a single contiguous buffer (gray components in Figure 3) and streams the entire buffer to the client side. On the client side, the received data is extracted and properly aligned following the requirements of the decoding process. Vectorization of communication introduced multi-fold performance improvement in HDFS.

To implement vectorized communication, we introduce *symbol ID* parameter to client→datanode request. *symbol ID* represents the position of the requested symbol in the butterfly message. In HDFS, the client discovers data blocks (symbols) needed for the decoding process. Therefore, the client passes *symbol ID* to the datanode, and the datanode uses this information to read from a local storage and send back only useful vector components.

4.4 Memory Management

Butterfly decoding process requires an amount of DRAM capable of storing an entire k symbol message. When the symbol size equals the size of the HDFS block (64 MB), the amount of DRAM equals $(k+2)$x64M. In addition, unpacking the data upon completing vector communication requires additional buffer space. The RaidNode daemon assigns recovery of multiple corrupted symbols to each map-reduce task for sequential processing. Tasks are required to allocate large amounts of memory when starting symbol recovery, and free the memory (garbage collect) upon decoding completion. Frequent and not properly scheduled garbage collection in JVM brings significant performance degradation.

To reduce the garbage collection overhead, we implemented a memory pool that is reused across multiple symbol decoders. The memory pool is allocated during the map-reduce task setup and reused later by the computation and communication threads. Moving the memory management from JVM to the application level increases implementation complexity, but at the same time we measured overall performance benefits of up to 15%.

5 Butterfly Codes in Ceph

Starting from version 0.80 Firefly, Ceph supports erasure code data redundancy through a pluggable interface that allows the use of a variety of traditional erasure codes and locally repairable codes (LRC). Unlike HDFS, Ceph allows encoding objects *on-line* as they are inserted in the system. The incoming data stream is partitioned into small chunks, or *stripes*, typically around 4096 bytes. Each of these small chunks is again split into k parts and encoded using the erasure code of choice. The described stripe-based approach allows the encoding process to be performed simultaneously while the data is being streamed in the system. Acknowledgment confirming the completed write operation is sent to the user immediately upon the data uploading and encoding are completed. Figure 4 describes the encoding process.

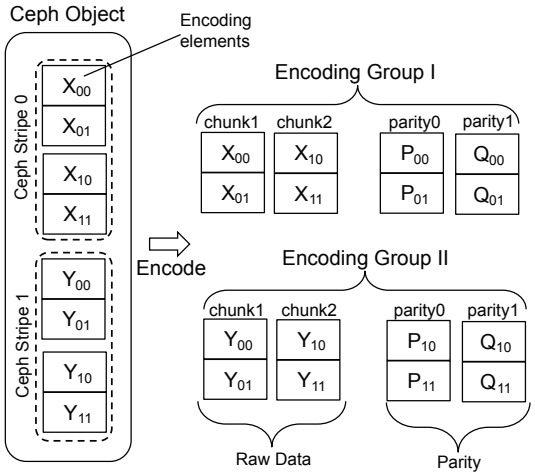


Figure 4: Ceph-Butterfly encoding example. For simplicity reasons $k = 2$. Object is divided in multiple stripes, and the Butterfly elements within each of the stripes are encoded.

The stripe size in Ceph determines the amount of memory required to buffer the data before the encoding process starts, i.e. larger stripe size requires more memory. Note that the entire encoding process takes place within a single server, therefore the memory is likely to be a scarce resource. In addition, having larger stripe sizes negatively affects the object write latency since one would get less benefits from pipelining the encoding process. Hence, from the performance point of view, it is desirable to use small stripe sizes. However, erasure code implementations benefit from operating on larger data chunks, because of being able to perform coarser computation tasks and read operations. In case of Butterfly codes, the performance is even more impacted by the stripe size, due to the large number of elements stored per column. As described in Figure 3, the Butterfly repair process requires accessing and communicating non-contiguous fragments of each code column. Small fragments incur high network and even higher HDD overhead. Due to internal HDD designs, reading random small fragments of data results in suboptimal disk performance. Because each Ceph stripe contains multiple Butterfly columns with $k \cdot 2^{k-1}$ elements per column, using large stripes is of great importance for the Butterfly repair process. In Section 6 we evaluate the effects that Ceph stripe size has on Butterfly repair performance.

5.1 Plug-In Infrastructure

To separate the erasure code logic from that of the OSD, RADOS uses an erasure code plug-in infrastructure that

allows dynamical use of external erasure code libraries. The plug-in infrastructure is designed for traditional and LRC codes. Third-party developers can provide independent erasure code implementations and use the plug-in infrastructure to achieve seamless integration with RADOS. We summarize the three main plug-in functions:
– *encode()*: Given a data stripe, it returns a list of n encoded redundant chunks, each to be stored in a different node in the placement group. This function is not only used to generate parity chunks, but it is also responsible to stripe data across n nodes.
– *minimum_to_decode()*: Given a list of available chunks, it returns the IDs of the chunks required to decode the original data stripe. During the decoding process, Ceph will fetch the chunks associated with these IDs.
– *decode()*: Given a list of redundant chunks (corresponding to the IDs returned by *minimum to decode*) it decodes and returns the original data stripe.

Because it was intended for traditional and LRC codes, the Ceph erasure coding plug-in infrastructure does not differentiate between repairing missing redundant chunks, and decoding the original data. When RADOS needs to repair a missing chunk, it uses the *decode* function to decode all the k original message symbols, and then uses the *encode* function to only regenerate the missing ones. The described model does not allow recovering a missing chunk by only partially downloading other chunks, a feature that is indispensable for efficient MSR implementation.

Efficient integration of Butterfly code with Ceph requires re-defining the plug-in interface. The interface we propose is a generalization of the existing one, and is usable across all types of RGCs as well as existing LRCs and traditional codes. Compared to the previous interface, the new plug-in provides the following two extra functions:
– *minimum_to_repair()*: Given the ID of a missing chunk, it returns a list of IDs of the redundant blocks required to repair the missing one. Additionally, for each of the required IDs, it specifies an additional list of the subparts that need to be downloaded (a list of offsets and lengths). Ceph will download all the returns subparts from the corresponding nodes.
– *repair()*: Given the ID of a missing chunk, and the list of chunk subparts returned by *minimum to repair*, the function reconstructs the missing chunks.

In order to implement the new plug-in infrastructure, parts of the OSD implementation had to be changed. These changes in Ceph do not allow to support both systems simultaneously. For back-compatibility with legacy erasure code plug-ins, we implemented a proxy plug-in that dynamically links with existing plug-ins. In practice, if the new plug-in system does not find a requested plug-in library, the legacy proxy plug-in is loaded.

5.2 Butterfly Implementation

Matching the previous plug-in interface, Butterfly is implemented as an external C library and compiled as a new-style RADOS erasure code plug-in. The level of algorithmic and implementation optimizations included in Ceph-Butterfly is significantly higher than HDFS--Butterfly, due to HDFS's dependency on Java. Our implementation of Butterfly in Ceph follows the recursive description provided in Section 3. Compared to HDFS, the recursive approach simplifies implementation. The recursive approach also achieves better data locality, which provides better encoding throughput.

6 Results

In this section, we evaluate the repair performance of our two Butterfly implementations, HDFS-Butterfly and Ceph-Butterfly.

6.1 Experimental Setup

To evaluate our Butterfly implementations we use a cluster of 12 Dell R720 servers, each with one HDD dedicated to the OS, and seven 4TB HDDs dedicated to the distributed storage service. This makes a total cluster capacity of 336TB. The cluster is interconnected via 56 Gbps Infiniband network using IPoIB. High--performance network ensures that the communication bandwidth per server exceeds the aggregated disk bandwidth. In addition to 12 storage nodes, we use one node to act as a metadata server. In HDFS a single DataNode daemon per server manages the seven drives and an additional NameNode daemon runs on the metadata server. In Ceph each server runs one OSD daemon per drive and the MON daemon runs separately on the metadata server.

For the erasure code we consider two different configurations: $k = 5$ and $k = 7$. Since Butterfly codes add two parities, these parameters give us a storage overhead of 1.4x and 1.3x respectively, with a number of elements per code column of 16 and 64 respectively. Having two different k values allows capturing the impact that the number of code columns (and hence the IO granularity) has on the repair performance. We compare the Butterfly code performance against the default Reed Solomon code implementations in HDFS and Ceph for the same k values. For both systems we evaluate the performance to repair single node failures. Upon crashing a single data node, the surviving 11 servers are involved in recreating and storing the lost data.

Our experiments comprise 2 stages: (i) Initially we store 20,000 objects of 64MB each in the storage system. Including redundant data, that accounts for a total of 1.8TB of total stored data. Due to the data redundancy overhead, on average each node stores a total 149.33GB for $k = 5$, and 137.14GB for $k = 7$. (ii) In the second stage we power-off a single storage server and let the 11

surviving servers repair the lost data. We log CPU utilization, IO activity and network transfers of each server during the recovery process.

6.2 Repair Throughput

In Figure 5 we show aggregate repair throughput across all 12 nodes, in MB/s. Figure 5(a) represents a comparison of Butterfly and Reed-Solomon on HDFS when $k = 5$ and $k = 7$. For HDFS we allow 12 reduce tasks per node (1 task per core), i.e. each node can work on repairing 12 blocks simultaneously. In Figure 5(a), it is observable that in all cases the repair throughput has a steep fall towards the end. In addition, RS($k = 7$) experiences very low repair throughput towards the end of the repair process. If the number of blocks to be repaired is lower than the number of available tasks, the repair system does not run at the full capacity, resulting in the steep decline of the aggregated repair throughput. Repair process for RS ($k = 7$) experiences load imbalance resulting in low throughput towards the end. Providing repair load-balancing in Hadoop is out of scope of this study. We focus on the sustainable repair throughput rather than the overall running time.

In HDFS, for $k = 5$, Butterfly reaches a repair throughput between 500 and 600MB/s, which is 1.6x higher than RS repair throughput. Although this is an encouraging result, Butterfly does not reach twice the performance of RS due to several reasons. The most important being the higher storage media contention in case of Butterfly. During the repair process Butterfly accesses small non-contiguous column elements (see gray components in Figrue 3). Larger number of relatively small IOs introduces more randomness in HDD behavior which in turn reduces the overall repair performance of Butterfly. Another performance degrading source is the data manipulation required by the vector-based communications we introduced (see Section 4.3). Although vector-based communication significantly improves the overall performance, certain overhead is present due to data packing and unpacking, i.e. high overhead memory manipulation operations in Java.

It is interesting to note that for $k = 7$, the difference in repair throughput between Butterfly and RS is $\sim$2x. With larger values of k communication bandwidth requirements increase due to large number of blocks required during the repair process. For $k = 7$ the benefits of reducing the network contention with Butterfly significantly outweigh possible drawbacks related to HDD contention and vector communication.

In Figure 5(b),(c) we depict the repair throughput for Ceph. In the case of 4MB stripe size, Figure 5(b), each stripe forms a Butterfly data matrix of 2^{k-1} rows and k columns. Consequently, the size of each data element is of 50KB and 9KB for $k = 5$ and $k = 7$ respectively. During the repair process, when non-contiguous elements are accessed, the small element size results in an inefficient HDD utilization and additional CPU operations due to element manipulation. This in turn leads to a degraded an inconsistent repair throughput as we can observe in Figure 5(b). Increasing the stripe size to 64MB results into having element sizes of 800KB, 143KB, large enough sizes to make a better utilization of the disk, and provide better repair throughput as we depict in Figure 5(c).

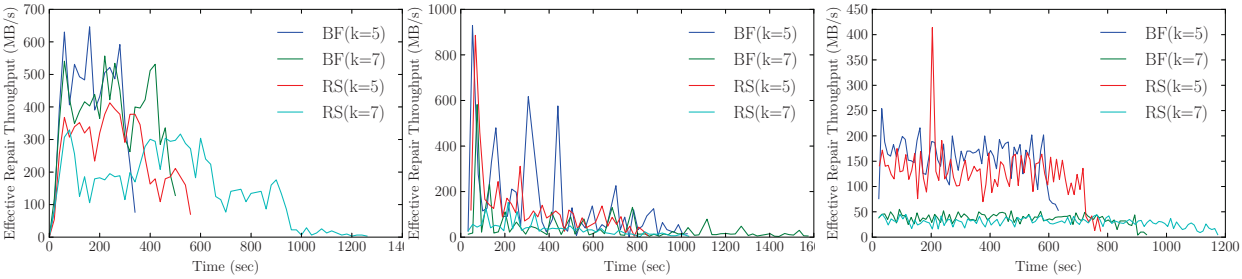
6.3 CPU Utilization

We measure CPU utilization of Butterfly/RS repair and evaluate the capability of each approach to possibly share in-node resources with other applications. CPU utilization understanding is of importance in distributed systems running in cloud-virtualized environments, or when the data repair processes share resources with other applications (e.g., map-reduce tasks). Figure 6 represents the CPU utilization of Butterfly and RS on a single node. The presented results are averaged across all nodes involved in computation.

For HDFS, in Figure 6(a) we observe that the CPU utilization for Butterfly exceeds RS CPU utilization by a factor of 3-4x, for both $k = 5$ and $k = 7$. Partially, this is due to the fact that RS spends more time waiting for network IO, because it requires higher communication costs compared to Butterfly. However, the number of total CPU cycles spent on computation in Butterfly is significantly higher than in RS. Compared to RS, Butterfly spends $\sim$2.1x and $\sim$1.7x more cycles, for $k = 5$ and $k = 7$ respectively. The observed CPU utilization is strongly tied to Java as the programming language of choice for HDFS. Butterfly implementation frequently requires non-contiguous data accesses and vector-based communication. Java does not have slice access to buffer arrays, requiring extra memory copies for packing and unpacking non-contiguous data.

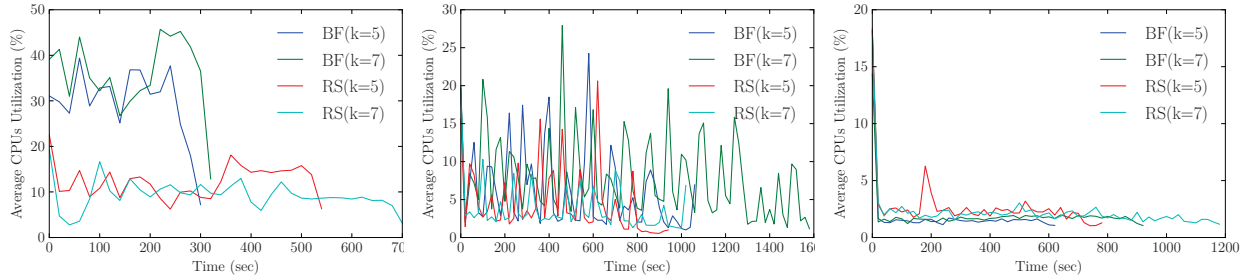
Figure 6(b),(c) represents the CPU utilization for Ceph-Butterfly and RS, when the Ceph stripe size is 4MB and 64MB. For Ceph stripe size of 4MB and $k = 5$, the Butterfly repair process operates on large number of elements that are only $\sim$50K in size. The fine granularity computation, together with frequent and fine granularity communication, causes erratic and unpredictable CPU utilization, presented in Figure 6(b). Similar observation applies for $k = 7$. Note that the CPU utilization for RS is somewhat lower compared to Butterfly, but still unstable and with high oscillations. The RS repair process operates on somewhat coarser data chunks, but the software overhead (memory management, function calls, cache misses, etc.) is still significant.

With the Ceph stripe size of 64M, Figure 6(c), the Butterfly element size as well as the I/O size increases sig-



(a) HDFS. (b) Ceph with 4MB stripes. (c) Ceph with 64MB stripes.

Figure 5: Repair throughput aggregated across all nodes involved in the repair process. Each system configuration we run with RS and Butterfly, with $k = 5$ and $k = 7$.



(a) HDFS. (b) Ceph with 4MB stripes. (c) Ceph with 64MB stripes.

Figure 6: Average CPU utilization per server. Each system configuration we run with RS and Butterfly, with $k = 5$ and $k = 7$. The graphs represent the average utilization across all 12 nodes involved in the repair process.

nificantly, resulting in lower and more predictable CPU utilization. Although Butterfly algorithm requires more operations compared to RS, our cache-aware implementation, carefully optimized with vector instructions in C/C++, achieves CPU utilization comparable to that of RS. Furthermore, both Butterfly and RS achieve ∼2-3% CPU utilization across all observed k values.

The presented results show that MSR codes over GF(2) achieve low CPU usage and are a good candidate for running in multi-user environments. However, achieving efficient CPU utilization requires a programming language that allows appropriate set of optimizations, and relatively coarse data chunks. In an *on-line* encoding system, such as Ceph, the size of the stripe size is of extreme importance for achieving efficient CPU usage and good repair performance.

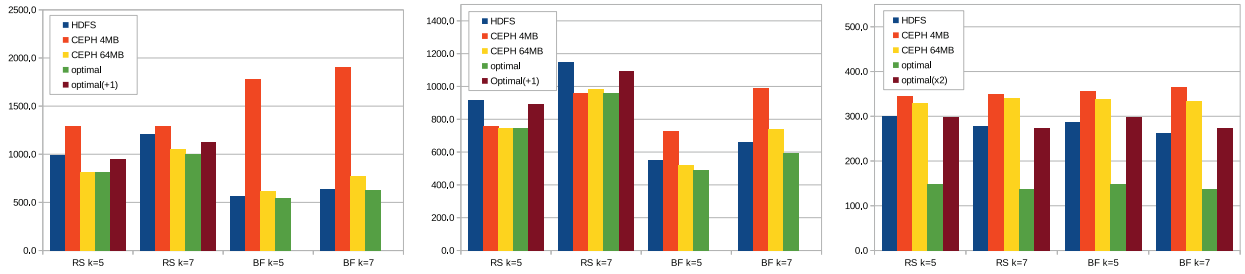
6.4 Network Traffic

In all systems used in this study we monitor network and storage traffic, and compare the observed results to the theoretical expectations. Figure 7 presents the results.

Figure 7(a) depicts the network traffic comparison. The *optimal* bars represent the lower bound on the amount of traffic. The *optimal + 1* bars represent the minimum increased by the size of a single HDFS block (we use 64MB block size). The original implementation of Reed-Solomon in Facebook - HDFS [2] unnecessarily moves an extra HDFS block to the designated repair node, causing somewhat higher network utilization. *optimal + 1* matches the amount of data pushed through the network in case of Reed-Solomon on HDFS.

We can observe in Figure 7(a) that HDFS-Butterfly implementation is very close to the theoretical minimum. The small difference between Butterfly and the *optimal* value is due to the impact of metadata size. Similarly, Reed-Solomon on HDFS is very close to *optimal + 1* with the additional metadata transfer overhead. In case of Ceph, the network traffic overhead is significantly higher. For Ceph-4MB, the large overhead comes from the very small chunks being transferred between the nodes and the per-message overhead introduced by the system. The communication overhead reduces for larger stripe sizes, i.e. Ceph-64MB. However, even with Ceph-64MB the communication overhead increases with k, again due to reduced message size and larger per-message overhead. Small message sizes in Ceph come as a consequence of the *on-line* encoding approach that significantly reduces the size of encoded messages, and hence the sizes of the symbol elements.

The results presented in Figure 7(a) reveal that if care-



(a) Overall Network Traffic in GB (b) Overall Disk Reads in GB (c) Overall Disk Writes in GB

Figure 7: The aggregate amount (across all 11 nodes included in the repair process) of network traffic and IOs during the repair process. We observe RS and Butterfly with $k = 5$ and $k = 7$.

fully implemented, MSR codes can reduce the repair network traffic by a factor of 2x compared to traditional erasure codes. Higher amount of network traffic in case of Ceph suggests that a specific system designs that avoid fine-grain communication is necessary.

6.5 Storage Traffic

Figures 7(b),(c) represent the observed HDD read/write traffic, as well as the theoretically optimal values predicted from the code design. We record the total amount of traffic across all HDDs in the system. HDFS-Butterfly achieves nearly optimal amount of read traffic, with the addition of metadata. HDFS-RS read traffic is also very close to *optimal + 1*, again the extra overhead comes from the HDFS metadata.

The results become somewhat more interesting with Ceph. It is noticeable that the difference between Ceph--Butterfly and *optimal* increases as we move from $k = 5$ to $k = 7$. Due to the small read I/O sizes with Ceph--Butterfly, reads suffer two drawbacks: (i) misaligned reads may cause larger data transfer (smallest HDD I/O is 4K and it needs to be 4K aligned), and (ii) read-ahead mechanism (128K by default) increases the amount of data transferred from an HDD. While read-ahead can be disabled, the entire system would suffer when reading sequential data, which is likely the most frequent scenario. The mentioned two drawbacks increase the influence on performance when the read size reduces, which is the case when we move from $k = 5$ to $k = 7$. With large enough Ceph stripes, the read I/O size increases in size, and the read overhead converges to zero. Example of large read I/Os is Ceph-RS, where the read overhead becomes negligible, Figure 7(b). In case of Butterfly, achieving large read I/Os requires impractically large Ceph stripe sizes.

For both systems and for both erasure code schemes, the overall amount of writes exceeds the *optimal* (lost data) amount by a factor of $\sim$2, as presented in Figure 7(c). Ceph allows updates of stored data, and for maintaining consistency in case of a failure, Ceph relies on journaling. In our experiments the journal for each OSD was co-located with the data, sharing the same HDD. In Ceph, all data being written have to pass through the journal and as a consequence the write traffic is doubled. Furthermore, the amount of data written in Ceph exceeds 2 x *optimal* because of data balancing. By examining the Ceph logs, we found that during the repair process many OSDs become unresponsive for certain amount of time. When that happens the recovered data is redirected to available OSDs, and load-balancing is performed when the non-responsive OSDs come back on-line. Note that the load-balancing can be performed among OSDs on the same server, therefore not affecting the network traffic significantly. Also, reads are not affected by load balancing since the data being moved around is "hot" and in large part cached in the local filesystem. Tracking down the exact reason for having OSDs temporarily unavailable is outside of scope of this study.

In case of HDFS, there is an intermediate local file where the recovered block is being written before committed back to the filesystem. This was the initial design in HDFS-RS, and our HDFS-Butterfly currently uses the same design. We will remove the extra write in the future. The intermediate file is not always entirely synced to HDD before the recovered data is further destaged to HDFS, resulting in the overall write traffic being sometimes lower than 2 x *optimal*.

7 Related Work

Traditional erasure codes, such as Reed-Solomon, have been implemented and tested in a number of large--scale distributed storage systems. Compared to replication, Reed-Solomon emerged as a good option for cost--effective data storage and good data durability. Most widely used open source distributed systems HDFS and Ceph implement Reed-Solomon variants [1, 5]. In addition, traditional erasure codes have been used in numerous other studies and production systems, including storage systems in Google and Facebook [3, 6, 15, 16, 18, 22]. Compared to the MSR code used in this study, the

traditional erasure codes initiate up to 2x more network and storage traffic during the repair process.

When it comes to practical usage of advanced erasure coding schemes, previous work have mostly been focused on LRC-based implementations in distributed storage systems. Sathiamoorthy et al. [28] introduce Xorbas, an LRC-based erasure coding that they also implement in HDFS. The study presents significant performance improvements in terms of both, disk and network traffic during the repair process. Xorbas introduce 14% storage overhead compared to RS.

Huang et al. [20] implement and demonstrate the benefits of LRC-based erasure coding in Windows Azure Storage. Khan et al. [21] designed a variation of Reed-Solomon codes that allow to construct MDS array codes with symbol locality, optimized to improve degraded read performance. In case of Butterfly codes, the degraded reads performance would be equivalent to RAID 5 degraded reads, and not as efficient as the work presented in by Khan. Xia et al. [35] present an interesting approach, where they combine two different erasure coding techniques from the same family, depending on the workload. They use the fast code for high recovery performance, and compact code for low storage overhead. They focus on two erasure coding families, product codes and LRC. All of the mentioned studies focus on the LRC erasure codes that cannot achieve the minimum repair traffic described by MSRs.

Rashmi et al. [25] implemented a MSR code construction with a storage overhead above $2\times$. In the regime below $2\times$, Hu et al. [11] presented functional MSR code, capable of achieving significant performance improvement over the traditional erasure codes when it comes to repair throughput. However, upon the first data loss and repair process, functional MSRs do not hold systematic (raw) data in the system any more. Consequently, the cost of reading systematic data increases significantly as the system ages.

In recent years, several erasure code constructions have attained the theoretical minimum repair traffic [7, 10, 12, 24, 31]. Although similar in the amount of repair traffic, Butterfly codes are systematic MDS array codes with node elements over GF(2). This small field size allows relatively simple implementation and high computational performance.

8 Discussion, Future Work, Conclusions

In this study we captured the performance of MSR codes in real-world distributed systems. Our study is based on Butterfly codes, a novel MSR code which we implemented in two widely used distributed storage systems, Ceph and HDFS. Our study aims at providing answers to important questions related to MSR codes: (i) can the theoretical reduction in repair traffic translate to an actual performance improvement, and (ii) in what way the system design affects the MSR code repair performance. Our analysis shows that MSR codes are capable reducing network traffic and read I/O access during repairs. For example, Butterfly codes in HDFS achieves almost optimal network and storage traffic. However, the overall encoding/decoding performance in terms of latency and storage utilization heavily depends on the system design, as well as the ability of the system to efficiently manage local resources, such as memory allocation/deallocation/movement. Java-based HDFS experiences significant CPU overhead mainly due to non-transparent memory management in Java.

The encoding approach is one of the most important decisions the system architect faces when designing a distributed erasure coding system. The initial decision of using real-time or batch-based encoding strongly impacts the overall system design and performance. The real-time approach achieves efficient storage utilization, but suffers high storage access overhead due to excessive data fragmentation. We show that in Ceph, for stripes of 4MB the repair network overhead exceeds many times the expected one, while the storage access overhead goes up to 60% higher than optimal (depending on code parameters). The situation improves with larger stripe sizes but the communication and storage overhead remains. Batch-based data encoding (implemented in HDFS) achieves better performance, but reduces storage efficiency due to the required intermediate persistent buffer where input data is stored before being encoded.

To address the design issues, we suggest a system with on-line data encoding with large stripes, able to use local non-volatile memory (NVM) to accumulate enough data before encoding it. The non-volatile device has to be low-latency and high-endurance which are important attributes of future NVM devices, some of which have already been prototyped. Part of our on-going effort is to incorporate this non-volatile and low-latency devices into a distributed coding system.

When it comes to the features required to efficiently implement MSR codes in distributed storage systems, our results indicate that communication vectorization becomes necessary approach due to the non-contiguous data access pattern. The interface between the system and the MSR codes requires novel designs supporting the specific requirements of these codes. In case of Ceph we showed the necessity for chaining the plug-in API, and we proposed a new model that is suitable for MSR codes.

While the overall performance of MSR codes in distributed storage systems depends on many factors, we have shown that with careful design and implementation, MSR-based repairs can meet theoretical expectations and outperform traditional codes by up to a factor of 2x.

References

[1] Ceph Erasure. http://ceph.com/docs/master/rados/operations/erasure-code/. Accessed: Apr 2015.

[2] Facebook's hadoop 20. https://github.com/facebookarchive/hadoop-20.git. Accessed: Jan 2015.

[3] Google Colossus FS. http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf. Accessed: Apr 2015.

[4] HDFS Federation. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html. Accessed: Jan 2015.

[5] HDFS RAID. http://wiki.apache.org/hadoop/HDFS-RAID. Accessed: Apr 2015.

[6] M. Abd-El-Malek, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Conference on USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[7] V. R. Cadambe, C. Huang, and J. Li. Permutation code: Optimal Exact-Repair of a Single Failed Node in MDS Code Based Distributed Storage Systems. In *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, 2011.

[8] V. R. Cadambe, S. Jafar, H. Maleki, K. Ramchandran, C. Suh, et al. Asymptotic Interference Alignment for Optimal Repair of MDS Codes in Distributed Storage. *Information Theory, IEEE Transactions on*, 2013.

[9] V. R. Cadambe, S. A. Jafar, and H. Maleki. Distributed Data Storage with Minimum Storage Regenerating Codes-Exact and Functional Repair are Asymptotically Equally Efficient. *arXiv preprint arXiv:1004.4299*, 2010.

[10] V. R. Cadambe, S. A. Jafar, and H. Maleki. Minimum Repair Bandwidth for Exact Regeneration in Distributed Storage. In *Wireless Network Coding Conference*, 2010.

[11] H. C. Chen, Y. Hu, P. P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. *IEEE Transactions on Computers*, 2014.

[12] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *Information Theory, IEEE Transactions on*, 2010.

[13] A. G. Dimakis, P. B. Godfrey, M. J. Wainwright, and K. Ramchandran. The Benefits of Network Coding for Peer-To-Peer Storage Systems. In *Third Workshop on Network Coding, Theory, and Applications*, 2007.

[14] E. En Gad, R. Mateescu, F. Blagojevic, C. Guyot, and Z. Bandic. Repair-Optimal MDS Array Codes Over GF (2). In *Information Theory Proceedings (ISIT), IEEE International Symposium on*, 2013.

[15] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing. In *Proceedings of the International Conference for High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

[16] 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 *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*, 2003.

[18] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, 2005.

[19] C. Huang, M. Chen, and J. Li. Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems. In *Network Computing and Applications (NCA), 6th IEEE International Symposium on*, 2007.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[21] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *FAST*, page 20, 2012.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. *SIGPLAN Not.*, 35(11):190–201, Nov. 2000.

[23] S.-J. Lin, W.-H. Chung, Y. S. Han, and T. Y. Al-Naffouri. A Unified Form of Exact-MSR Codes via Product-Matrix Frameworks. *Information Theory, IEEE Transactions on*, 61(2):873–886, 2015.

[24] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe. Repair Optimal Erasure Codes Through Hadamard Designs. *Information Theory, IEEE Transactions on*, 2013.

[25] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 81–94. USENIX Association, 2015.

[26] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction. *Information Theory, IEEE Transactions on*, 57(8):5227–5239, 2011.

[27] B. Sasidharan, G. K. Agarwal, and P. V. Kumar. A High-Rate MSR Code With Polynomial Sub-Packetization Level. *arXiv preprint arXiv:1501.06662*, 2015.

[28] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

[29] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *Information Theory, IEEE Transactions on*, 58(4):2134–2158, 2012.

[30] C. Suh and K. Ramchandran. Exact-repair MDS Code Construction Using Interference Alignment. *Information Theory, IEEE Transactions on*, 57(3):1425–1442, 2011.

[31] I. Tamo, Z. Wang, and J. Bruck. Zigzag Codes: MDS Array Codes with Optimal Rebuilding. *Information Theory, IEEE Transactions on*, 59(3):1597–1616, 2013.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 307–320. USENIX Association, 2006.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.

[34] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing'07*, pages 35–44. ACM, 2007.

[35] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A Tale of Two Erasure Codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 213–226. USENIX Association, 2015.