הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

Evaluating Zigzag Code in a Distributed Storage System

Matan Liram

Evaluating Zigzag Code in a Distributed Storage System

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Matan Liram

Submitted to the Senate of
the Technion — Israel Institute of Technology
Iyar 5777 Haifa May 2017

The research thesis was done under the supervision of Dr. Gala Yadgar, Prof. Eitan Yaakobi and Prof. Assaf Schuster in the Computer Science Department.

I would sincerely want to thank Gala Yadgar for investing extraordinary amount of time to help me overcome research obstacles, directing me to make the right research decisions and helping me with extensive task and time planning throughout our research period. Every submission and presentation, and even the bare thesis hand-in happened on time and in the best quality I could provide thanks to her tremendous research experience and superior planning strategies. I would like to thank Itzhak Tamo and Eitan Yaakobi for helping me with the theoretical parts of the paper, without their help I wouldn't be able to evaluate the experiments correctly, nor to 'speak the language' of Zigzag so fluently. In addition, I would sincerely want to thank Ido Hakimi for his tremendous help with implementing an amazingly neat version of the Zigzag encoder and decoder, which saved me tons of debugging time. I would have spent hours of compile time without his thoroughly tested code. I would also like to thank Assaf Schuster for pointing out essential tips and issues with the research directions, and helping us do it the best way possible. His vast experience in the research field has been striking.

I would like to thank my parents for supporting me throughout this period and gently forcing me non-stop to keep up with my hard work. The top-priority I gave to the goal of finishing the research was mainly inherited from their encouragement and massive support. I would like to thank my office partner Lev Yohananov for pleasing my time at the office, for being curious and providing me with ideas to complex problems I've tackled throughout my research, for which he has shown surprising curiosity.

I wouldn't be able to finish this research without the help mentioned above. In some stages it almost seemed as if I'm filling my small role in an oiled research machine. I feel very lucky to have spent the recent two years with such an efficient, professional and fun team.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Erasure codes protect data in large scale data centers against multiple concurrent failures. However, in the frequent case of a single node failure, the amount of data that must be read for recovery can be an order of magnitude larger than the amount of data lost. Some existing codes successfully reduce these recovery costs but increase the storage overhead considerably. Others, which are theoretically optimal, minimize the amount of data required for recovery, but incur irregular I/O patterns that may actually increase overall recovery time and cost. Thus, while the theoretical results in this context continue to improve, many of them are inapplicable to realistic system settings, and their benefit remains theoretical as well.

This gap between theory and practice has been observed in previous studies that applied theoretically optimal techniques to real systems. In this paper, we present a novel system-level approach to bridging this gap in the context of reducing recovery costs. We optimize the *sequentiality* of the data read, at the cost of a minor increase in its *amount*. We use Zigzag—a family of erasure codes with minimal overhead and optimal recovery—and trade its theoretical optimality for real performance gains. Our implementation of Zigzag and its optimizations in Ceph reduces recovery costs with two, three and four parity nodes, for large and small objects alike. We were able to cut down recovery time by up to 28% compared to that of Reed-Solomon, and to reduce the amount of data read and transferred by 18% to 39%.

Abbreviations and Notations

n	—	Code length
k	—	Data blocks
r	—	Redundancy blocks
rr	—	Rebuilding ratio - the portion of the surviving nodes' data that must be read during recovery
m	—	Each data and parity block is composed of a few elements, the number of elements in each block is m
s	—	The duplication factor of the code, $\frac{k}{k'}$ where k' is of the original zigzag code
v	—	The number of virtual blocks logically filled with zeroes
k'	—	number of blocks in each duplicated code instance
MDS	—	Minimum distance seperable - a MDS code with r parity blocks can recover from r erasures
MTTDL	—	Mean time to data loss, the time between two failures in the system, repair time must be shorter
HDD	—	Hard disk drive, a magnetic storage device
G	—	A generator matrix of an array-code, such as zigzag
OSD	—	Object storage device - stores a single block of data or parity elements for objects in corresponding placement groups
PG	—	A placement group, logical collection of objects that are replicated across the same set of OSDs
GF	—	Galois finite field
LRC	—	Local Reconstruction Codes

Chapter 1

Introduction

1.1 Disk Failures

In modern data centers that host hundreds to tens of thousands of nodes, node failures are the norm, and correlated failures are not uncommon. As a result, large scale storage systems are configured with high degrees of redundancy, storing three replicas of each data object, or dedicating parity nodes to each data array to withstand up to four node failures. The frequent node failures trigger recovery processes that impose significant load on the system. These cause excess disk I/O and network transfer to server up-time and respective power and cooling costs.

A recent study on Facebook's data centers showed that recovery in erasure-coded arrays incurred an order of 180TB of data transfer between racks each day [26]. Thus, significant research effort has gone into reducing recovery costs in such systems. The above study showed that 98.08% of failure events constitute exactly one unavailable node, and only 1.87% of failure events constitute two unavailable nodes. Indeed, although erasure-coded systems are designed to withstand several concurrent failures, much of the related effort has focused on optimizing recovery from failures of a single node.

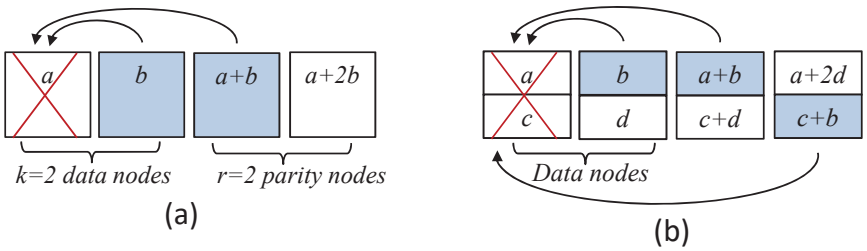


Figure 1.1: Recovery of a single failed node in an (n,k) erasure-coded system. With the widely used Reed-Solomon codes (a), k entire nodes must be read. With Zigzag codes with r parities (b), the data can be recovered using $\frac{1}{r}$ of each surviving node.

1.2 Resilience Approaches

For example, Local Reconstruction Codes (LRC) support recovery by reading only a small subset of the surviving nodes [9, 15, 16, 30, 33], but they increase the storage overhead of the system. Other codes reduce the amount of data transferred during recovery, but still require reading most of the surviving data from disk [7, 27, 24].

Several approaches have been suggested for reducing the amount of data read from each node in existing codes or special variations of the read data [12, 17, 25]. The latter result in highly irregular I/O patterns and are thus useful for systems storing extremely large objects. For example, in *Hitchhiker* code, presented in [25] in order to utilize small objects the code requires to utilize a *strided read pattern* — a pattern in which we read non sequentially by taking constant sized steps between segments of the read data. The example will be further detailed in Chapter 6.

A promising approach involves codes that minimize the *rebuilding ratio*—the portion of the surviving nodes' data that must be read during recovery—and achieve the lower theoretical bound on this ratio via explicit constructions [10, 18, 40]. However, these codes are characterized by high internal fragmentation: encoded objects are composed of hundreds of *elements*. The low rebuilding ratio is achieved by reading only a subset of the elements stored on each surviving node, but these elements are necessarily non-contiguous on the low level storage device.

Hard disks are the dominating storage technology in most data centers,

especially in storage pools that store infrequently accessed (*cold*) data and provide data durability and redundancy with erasure codes. This is due to the high cost per GB that SSDs incur [2]. Nonsequential I/O accesses such as those described above can be detrimental to hard disk performance. Even *strided* accesses, which read contiguous data units but skip some of them at regular intervals, significantly reduce I/O throughput. A recent study demonstrated that the cost of irregular I/O accesses may cancel the benefit from reducing the rebuilding ratio, and may even increase the amount of data read, as well as the overall recovery time [20].

1.3 Our Contribution

In this work, we show how to resolve this tension between theory and practice, to achieve near-optimal recovery in real system settings. We use Zigzag codes that have both optimal storage overhead and an optimal rebuilding ratio [34], along with an inherent flexibility that trades the rebuilding ratio for I/O sequentiality. We describe a set of optimizations made possible by this flexibility, and show how a small increase in the code's theoretical rebuilding ratio can significantly reduce recovery costs in a real system. Some of these optimizations can be applied to other codes as well.

The novelty of our optimizations lies in their underlying objective: while previous approaches aim to *reduce the amount* of data required for recovery, our optimizations are designed to *increase the sequentiality* of this data, possibly at the cost of reading more of it. We employ *code duplication*, a technique for extending a stripe to span more nodes without increasing its internal fragmentation, at the cost of increasing the amount of data that must be read during recovery [34]. We combine duplication with the notion of *virtual nodes*: nodes that do not store any data but increase the code's parameter space and allow system designers to choose the construction that best suits their goals. The concept is known in the coding literature as *code-puncturing*. Finally, we exploit the generic design of Zigzag codes to choose code constructions as well as recovery schemes with the most contiguous I/O accesses.

The second set of optimizations is orthogonal to the specific code construction, and targets the low-level disk accesses. First, we add *virtual padding* at the end of each object so that the fragments that are read for

recovery are aligned to 4KB sector boundaries. Second, we examine two approaches for coalescing disk reads, possibly reducing the amount of I/Os issued at the cost of increasing the amount of data read.

We implement Zigzag with the mentioned optimizations in *Ceph* — a highly scalable, open-source storage system that supports both replication and erasure coding [36, 38] — and compare it to Reed-Solomon — the system's default erasure code. In addition, we include a detailed analysis of how Zigzag compares with LRC in the settings we chose for our experiments, using the basic Pyramid construction [15].

Our evaluation of Zigzag includes populating the cluster with data, followed by a node failure which triggers the system's recovery process. We measure and report the CPU usage, network transfers and disk accesses during each such process. We do this for a variety of configurations using each of our optimizations. The evaluation shows that by reducing internal fragmentation, Zigzag codes can significantly reduce the amount of data read and transferred during recovery (by 18% to 39%), as well as the overall recovery time (by up to 28%). Unlike previous approaches, Zigzag achieves similar reductions for all common array configurations, as well as for small (4MB) objects.

Chapter 2

Background

2.1 Erasure Codes in Theory

In an *(n,k) erasure code*, *k data blocks* are encoded into a *stripe* of *n code blocks*. The code is *systematic* if the data blocks are included in the n code blocks. In this case, the remaining $r = n-k$ blocks in the stripe are referred to as *parity blocks*. *Coding equations* define the computation of the parity blocks. In *linear* codes, which are the most common, each parity block is a linear combination of some or all of the data blocks, where the coefficients of the equation are picked from some finite field $\mathbb{F}$. The data blocks that appear in a parity block's coding equations make up its *dependency set*. For a given (n,k) pair, a code *construction* is defined by the coding equations for all $r = n - k$ parity blocks.

An (n,k) erasure code is called *MDS (maximum distance separable)* if it can recover any block as long as up to r blocks are unavailable, where r=n-k. MDS codes are desirable because they can recover the maximum number of unavailable blocks given their storage overhead.

In *scalar codes*, the coding equations are defined over a set of entire blocks, as seen in Figure 1.1(a). In *array codes*, each parity block is calculated from a set of sub-blocks, which we refer to as *elements*. When a stripe is viewed as an array, each block represents a *column*, and the number of elements in a block is the number of *rows* in the array. For example, the array code in Figure 1.1(b) has $n = 4$ code blocks, among them $k = 2$ are data blocks, and $r = 2$ are parity blocks. Each block consists of 2 rows. Array codes were initially motivated by the goal of avoiding computationally

expensive finite field operations during decoding and encoding, replacing them with simple XOR operations.

When a block is unavailable, it can be recovered using the content of some or all of the remaining blocks in its stripe. Thus, we distinguish between two related operations: a *read* requests a systematic data block and does not require additional content or computation, while a *degraded read* requests an unavailable data block which is decoded from the remaining blocks in the stripe. Note that in array codes, while coding equations refer to elements, failures affect blocks as a whole.

When a block is permanently unavailable due to reasons we discuss below, it is recovered in a similar manner to a degraded read operation, and its content is stored to allow future (non-degraded) reads. This process is called *recovery* or *rebuild*, and is usually applied to a large number of blocks, from different stripes, that were stored on a failed storage device. The *rebuilding ratio* is the fraction of elements required for recovery of a single block, out of the surviving elements in the stripe. For example, the rebuilding ratio of the code in Figure 1.1(a) is $\frac{k}{n-1}$. The theoretical lower bound on the rebuilding ratio for an (n,k) MDS code was shown to be $\frac{1}{r}$ [7]. Several recent studies show explicit codes that achieve this ratio for rebuilding data blocks, while the recovery of parity blocks required reading k whole surviving blocks [10, 28, 34]. We follow their convention throughout this paper and refer to the rebuilding ratio of data blocks. We discuss recent results for efficient recovery of parity blocks in Chapter 6.

2.2 Erasure Codes in Practice

Modern data centers typically consist of hundreds to tens of thousands of storage servers (*nodes*), each serving thousands of IOPS over 8Gbps to 40Gbps network links [26, 32]. Data is stored as *objects*, consisting of one or more stripes according to their size and system parameters. The data and parity blocks of each stripe are stored on separate storage nodes, and different objects are striped across different sets of nodes for load balancing.

In such complex settings, node failures are the norm. A *failed* node does not respond to periodical "heartbeat" probes, and is therefore incapable of serving read or write requests. It is customary to distinguish between *permanent* failures and *transient* ones. Permanent failures require the entire

node or its storage drives to be replaced, due to hardware faults or planned upgrades. A transient failure is a temporary one, caused, for example, by power or network outages, system reboot, or process restarts [11, 16].

Modern storage systems employ several approaches to ensure reliability when such failures occur. For instance, in Microsoft Azure, [3], every node belongs to a separate *fault domain* which has redundant networking and power. In addition, data resides in different data-centers in order to withstand large scale disasters.

In this context, data requested from a failed node must be served as a degraded read, and permanent node failures require recovery of all the data they were storing. Most systems apply a threshold, typically 15 minutes, after which a non-responsive node is considered permanently failed and recovery is triggered [11, 26].

The motivation to postpone recovery is the high cost it incurs: the surviving blocks in each stripe must be read from several nodes, and transferred to the healthy node responsible for decoding and storing the lost block. These reads interfere with the ability of the nodes to serve application requests by occupying precious network and storage bandwidth—recovery may occupy as much as 10%-20% of the data center's bandwidth [30]. Thus, recovery jobs are often scheduled as background processes with low priority and bandwidth limits. For example, in Ceph, the priority of client operations is set to 63, whereas that of the recovery operations is set to 10 [4]. In turn, it may take several hours to fully recover a failed node.

The tradeoff between recovery cost and storage overhead is straightforward: for a given reliability requirement, i.e., a fixed number of parity blocks (r), increasing the stripe size (k) reduces the storage overhead but increases the amount of data required for recovery, and thus increases recovery cost.[1]

Replication is one example of this tradeoff: where $k = 1$ and r is the number of replicas, recovery of a replicated node requires reading one of its replicas. This is clearly the minimum amount of data required, and is also exactly $\frac{1}{r}$ of the surviving blocks in the stripe. However, due to its high overhead, the use of replication is usually limited to hot data, while

[1] Increasing the stripe size while fixing the number of parity blocks also reduces the mean time to data loss (MTTDL). In the scope of this work we assume that n and k were chosen according to external considerations, and focus on the recovery performance of the system.

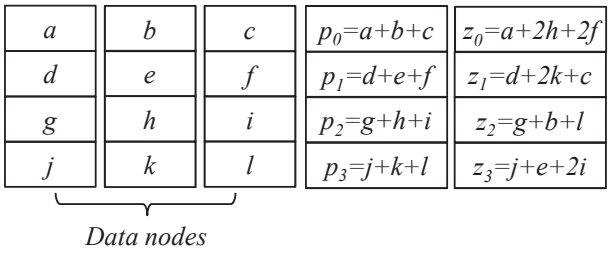


Figure 2.1: (5,3) Zigzag code construction. p_i is the simple parity element of row i, and z_i is the i'th element of the Zigzag parity.

most of the data center's stripes are encoded with $k > 1$. Typical values of (n,k) erasure codes are (14,10) in Facebook's data centers [26, 30], (16,12) in Windows Azure [16] and (9,6) in Google Colossus FS [39].

The number of parity nodes is determined accoding to the stripe's target $MTTF$ — mean time to failure which is the average time that a disk is expected to function before it fails.

2.3 Zigzag Codes

Zigzag is a family of systematic MDS array codes, which can be constructed for any (n,k) combination and achieve the optimal rebuilding ratio. The explicit construction of Zigzag codes, as well as the proofs of their correctness and optimality, are detailed in previous theoretical work [34]. We focus here on an intuitive explanation of their structure and properties relevant to their application to real systems.

An (n,k) Zigzag code with r parities, $r = n - k$, has $m = r^{k-1}$ rows. The first parity column (block) in any Zigzag code is simple parity. In other words, each element is a XOR of all the data elements in its row. The remaining $r-1$ columns are called *Zigzag parities*. Each element in a Zigzag parity is a linear combination of k data elements from different rows, one from each column. The coefficients in the coding equations ensure the MDS property, i.e., that up to r lost blocks can be recovered. The combination of dependency sets and coefficients ensures that the code is MDS and that the recovery of one lost data block requires exactly $\frac{m\times(n-1)}{r}$ elements ($\frac{m}{r}$ from each of the $n-1$ surviving nodes), which yields the optimal rebuilding ratio $\frac{1}{r}$. Each such combination is a Zigzag construction.

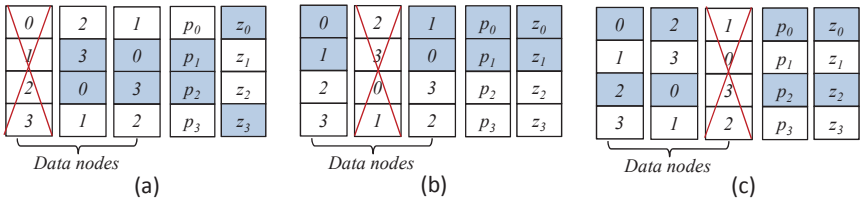


Figure 2.2: Recovery of a single failed node in the (5,3) code in Figure 2.1. Data elements are numbered according to the Zigzag parity whose dependency set they belong to. In each scenario, the failed node is crossed out, and the shaded elements are the ones read for its recovery.

Figure 1.1(b) shows an example of a (4,2) Zigzag code, where the coding equations for the first and second Zigzag parity elements are $a + 2d$ and $c + b$, respectively. When the first (leftmost) data block is lost, it can be recovered by reading the three shaded elements out of the six surviving ones. Figure 2.1 shows an example of a (5,3) Zigzag code and the coding equations for the simple parity elements (p_i) and Zigzag parity elements (z_i). We note that adding a data block to the stripe doubles the number of rows in the array. Figure 2.2 shows the recovery scenarios when each of the data blocks fails, using the equations of the shaded parity elements. They all require eight out of the sixteen surviving elements.

The construction in Figure 2.1 exemplifies the inherent limitation of Zigzag codes, and in fact, of array codes in general. While recovery of a data node requires a limited portion of the surviving elements, reading these elements entails nonsequential I/O access in the storage nodes. Although the I/O access is not completely random, and does not require additional arm movements, it does incur excessive rotational delay. Consider, for example, rebuilding a permanently failed node storing blocks from multiple objects according to the recovery scenario in Figure 2.2(c). If the size of each element is 4KB, the standard HDD sector size, reading half of the elements in each surviving node would require the same time as reading all the data in the node. As a matter of fact, this would also be the case for the recovery scenarios in Figures 2.2(a) and (b). As long as the requested contiguous elements do not fill an entire track, the resulting rotational delay would cancel the benefit from reducing the rebuilding ratio.

The problem of nonsequential access has been identified in several previous studies [17, 20, 25]. Since the element size equals the object size divided

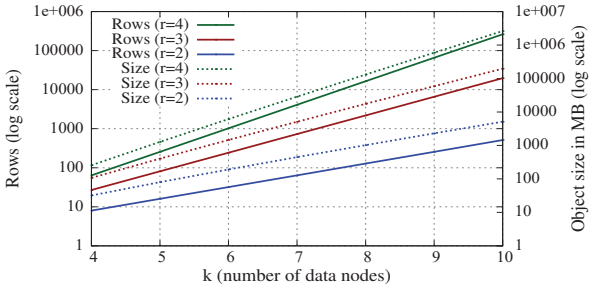


Figure 2.3: The number of rows (r^{k-1}) and the corresponding object sizes ($r^{k-1} \times k$) required for ensuring that elements are 1MB each. Both measures increase exponentially with k.

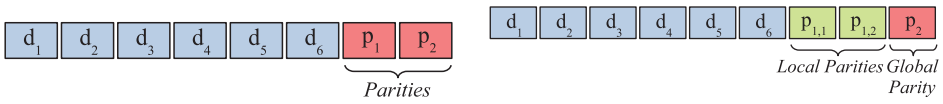
by the number of elements, the problem was addressed thus far by applying array codes to extremely large objects, where block sizes range from 64MB to 256MB [17, 25]. For smaller objects, the network bandwidth corresponds to the rebuilding ratio, but storage bandwidth experiences only a minor reduction, and sometimes even increases [20].

This problem is exacerbated in Zigzag codes, where the number of rows is exponential in the number of columns. Figure 2.3 plots the number of rows required for representative values of k and r, and the corresponding object size required for ensuring that elements are 1MB in size (an optimistic estimate of HDD track size). This is a frustrating example of the tension between theory and practice: a code that is optimal in theory leads to very high recovery costs in practice. Fortunately, several optimizations of Zigzag codes, which we discuss in the following section, enable us to resolve this tension and reduce recovery cost considerably.

2.4 Local Reconstruction Codes (LRC)

Local reconstruction codes add parity nodes to an array, so that one node can be recovered by accessing only a small subset of the nodes in the array. This is a popular approach to reducing recovery costs, examples of which include Product [9] and Pyramid codes [15], LRC, which is used in Windows Azure [16], HACFS [39], Xorbas [30], and optimal-LRC [33]. We now summarize their advantages and limitations.

These codes are non-MDS: each of the additional *local* parity nodes,



(a) A $(8, 6)$ Reed-Solomon (b) A $(8, 6)$ Basic pyramid code

Figure 2.4: A basic Pyramid code constructed from a $(8, 6)$ Reed-Solomon code. It has 6 data blocks $(d_1, \ldots, d_6)$, two local parities and a global parity. The local parity block $p_{1,1}$ is a linear combination of d_1, d_2, d_3 and the local parity block $p_{1,2}$ is a linear combination of d_4, d_5, d_6. In addition, p_2 is a global parity block, a linear combination of all data blocks. In case d_1 fails, we can recover it by reading 3 blocks: d_2, d_3 and $p_{1,1}$.

presented in Figure 2.4(b), are computed based on a subset of the data nodes, so they do not protect against arbitrary node failures. Thus, these codes introduce a trade-off between storage efficiency, which is optimal in Zigzag, and repair cost. Like ours, this trade-off can be viewed as bridging the gap between theoretical optimality and performance in practice.

On the other hand, these codes are scalar and read entire blocks during recovery. Thus, they exhibit better sequentiality than Zigzag in most setups. By accessing only a limited number of nodes, they limit the recovery load to a restricted portion of the system. At the same time, if the number of nodes participating in the recovery is too small, the reduced parallelism may increase recovery time. This is usually not a problem in large scale systems that stripe objects across hundreds of nodes. Indeed, locally repairable codes have been applied to very large scale systems. In other systems, the advantages of Zigzag may be more pronounced.

In addition to comparing Zigzag to Reed-Solomon codes, we add a detailed analysis of how Zigzag compares with *basic Pyramid codes*, from the Local Reconstruction Codes (LRC) family [15]. The basic Pyramid codes can be simply derived from any existing codes, and thus all known efficient encoding/decoding techniques directly apply [15].

In order to fully understand Section 5.3, we present a common notation of basic Pyramid codes, from the LRC family. Let an object consist of n blocks, where k blocks are data blocks, and the other $r = n - k$ blocks are redundant blocks. Use d_i $(i = 1, \ldots, k)$ to denote data blocks and p_j $(j = 1 \ldots, r)$ to denote the redundant blocks.

We construct an example which derives from a $(8, 6)$ MDS code, specifi-

cally Reed-Solomon. Let the 6 data blocks be separated into two equal size groups $S_1 = d_1, d_2, d_3$ and $S_2 = d_4, d_5, d_6$. A single redundant block, say c_1, splits into two, where the rest of the parity blocks stay unchanged, and are now called *global* redundant blocks. The global redundant blocks are computed from all data blocks. Then, a new redundant block is computed for group S_1, which is denoted as a *local* redundant block $p_{1,1}$. The computation is done as if computing p_1 in the original MDS code, except for setting all the data blocks in S_2 to 0. Similarly, local redundant block $p_{1,2}$ is computed for S_2. Clearly, local redundant blocks are only affected by data blocks in their corresponding groups, and *not* by other groups at all. This yields a $(8, 6)$ basic Pyramid code [15], which is described in Figure 2.4(b).

We use the basic Pyramid code construction because it can easily match each of our chosen Reed-Solomon configurations, by splitting a single parity block into two local parities, and since it can be easily configured in Ceph. We will further explain the trade-offs in choosing this construction in Section 5.3.

Chapter 3

Our Approach

3.1 Reducing the Number of Elements

Our optimizations for implementing Zigzag codes are designed to make recovery reads as sequential as possible. This is done primarily by reducing the number of elements in a stripe, which is equivalent to reducing the number of rows. Whenever possible, we also reduce the number of elements (or rows) that should be "skipped" between those elements that are required for recovery.

3.1.1 Code Duplication

An appealing property of Zigzag codes is that a $(k'+r,k')$ Zigzag code can be *duplicated* to form a $(k+r,k)$ array, so that $k = s \times k'$ for some duplication factor s. The number of rows in the duplicated code is $m = r^{k'-1}$, as in

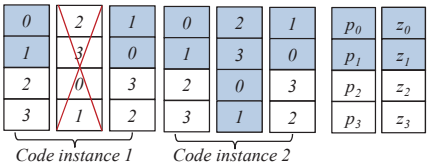


Figure 3.1: This (8,6) Zigzag construction is a 2-duplication of the code in Figure 2.1. Data elements are numbered according to the Zigzag parity whose dependency set they belong to. The shaded elements are those required for the recovery of the second data block. The fifth block is the twin of the failed one, and is thus read entirely during recovery.

the original $(k'+r,k')$ code. An *s-duplication* consists of s sets of k' data columns each. We refer to each such set as a *code instance*, where blocks (or elements) in the same position in different instances are *twins*, and belong to the same dependency sets. For example, Figure 3.1 shows the (8,6) code that is a 2-duplication of the (5,3) code in Figure 2.1. The fourth node is a twin of the first, the dependency set of p_0 is the entire first row, and the dependency set of z_0 are all the elements marked with 0. Note that only the element positions in the coding equations are duplicated, not the data itself.

The coefficients in a duplicated code are chosen from a larger finite field than those of the original code, to preserve its MDS property, as discussed in Section 3.1.4. Recovery of a single failed block in a duplicated code requires $\frac{1}{r}$ of every block, and the remaining elements in the $s-1$ twins of the failed block. For example, the shaded elements in Figure 3.1 are required for the recovery of the second block in the duplicated (8,6) code. Thus, the reduction in the number of rows comes at the cost of increased rebuilding ratio. In this example, recovery requires 16 of 28 surviving elements, corresponding to a rebuilding ratio of 0.57. The rebuilding ratio in the general case is $\frac{1}{r} + (\frac{r-1}{r})(\frac{s-1}{sk'+r-1})$.

This increase in the rebuilding ratio means that duplicated codes are not theoretically optimal. Thus, they have not received much attention since they were suggested in the original Zigzag work [34]. However, our evaluation shows that duplicated constructions reduce recovery cost in many configurations in which the original code is not applicable at all.

3.1.2 Virtual Nodes

An (n,k) erasure code can be implemented as an s-duplication of a smaller code only if s divides k. This limits the applicability of duplication and the available points on the tradeoff between the number of rows and the rebuilding ratio. In order to fully exploit the design space offered by duplication, we utilize the concept of *virtual nodes* [1]. A virtual node does not exist in the system and is treated by the encoding and decoding mechanisms as storing all zeroes. Whenever a computation requires an element in a virtual node, this element is replaced by a buffer full of zeroes. Figure 3.2 plots the number of rows resulting from implementing the same (n,k) code

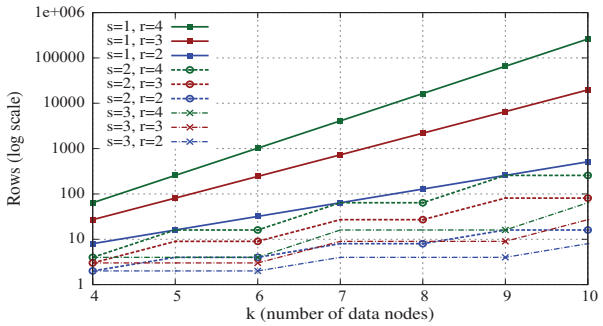


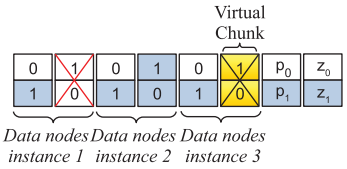
Figure 3.2: Number of rows of (n,k) Zigzag codes implemented as s-duplication, when $s \in \{1,2,3\}$. The marked data points correspond to constructions that do not require virtual nodes.

as an s-duplication for several values of s, with and without virtual nodes. For example, a (7,5) Zigzag array that would normally require 16 rows, can be implemented as the (8,6) construction in Figure 3.1, replacing the sixth data node with a virtual one. Interestingly, the average rebuilding ratio of an (n,k) code with virtual nodes is slightly (up to 7%) smaller than that of an (n,k) code where all the nodes exist, because the virtual nodes are never read. We discuss the applicability of code duplication and virtual nodes to other array codes in Chapter 6.

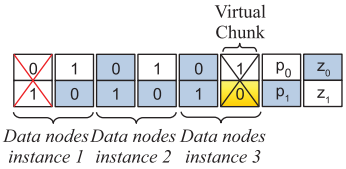
3.1.3 Effective Rebuilding Ratio

The theoretical rebuilding ratio takes into account only failures of data nodes, and assumes that all n nodes physically exist in the system. Thus, it does not accurately predict the minimal amount of data that must be read, on average, in order to recover one arbitrary failed node. We define the *effective rebuilding ratio* as the number of physical elements read as a fraction of the physical elements in the surviving nodes in the array. We denote the number of virtual nodes as v, the number of code blocks which store physical data as k (so that s divides $k+v$), and the number of blocks in each duplicated code instance as k', and present an expression to accurately predict the mentioned effective rebuilding ratio.

We consider the following block repair scenarios. In scenario (a), a parity node fails, and we need to read $k \cdot m$ elements to recover the failed node by recomputing the parity. The probability of this scenario is $\frac{r}{k+r}$.



(a) The last twin in this example is virtual.



(b) In this example the failed block has no last twins.

Figure 3.3: Recovery of a $(8, 6)$ Zigzag code with duplication factor $s = 3$. Notice that in figure (a) the last twin is not read while in figure (b) it is read.

In scenario (b), a data node with a virtual twin fails, and we need to read $(s-2)\cdot m$ elements from the twin blocks, consisting of all the twins but the failed and virtual ones, plus $\frac{m}{r}$ elements from the $(k+r-(s-1))$ remaining non virtual nodes which we haven't considered. There are $k+r-1$ remaining non-virtual nodes, and we already considered $s-2$ of them. The probability of this scenario is $\frac{v\cdot s}{k+r}$.

In scenario (c), a data node corresponding to a block with only physical twins fails, and we need to read $(s-1)\cdot m$ elements from all the twin blocks but the failed node, plus $\frac{m}{r}$ elements from the $(k+r-s)$ remaining non virtual nodes which we haven't considered. There are $k+r-1$ remaining non-virtual nodes, and we already considered $s-1$ of them. The probability of this scenario is $\frac{(k'-v)\cdot s}{k+r}$.

The effective rebuilding ratio, rr_e, considers only data disks, so we treat only scenarios (b) and (c). It is computed as follows. Figure 3.3a explains visually how the rebuilding ratio in (b)-type blocks is calculated, and Figure 3.3b explains the same for (c)-type blocks.

$$rr_e = \left[\underbrace{\frac{v\cdot s}{k}\cdot\frac{m}{r}\cdot(k+r-(s-1))}_{(b)} + \underbrace{\frac{(k'-v)\cdot s}{k}\cdot\frac{m}{r}\cdot(k+r-s)}_{(c)} \right] /_m \tag{3.1}$$

3.1.4 Recovery Coefficients

The general construction of Zigzag codes defines the dependency sets of the parity elements, but in some cases, does not specify the coefficients of the data elements in the coding equations. The optimality of Zigzag codes

relies on a proof of the existence of suitable coefficients, but their values are not given for all the combinations used in our evaluation. Specifically, the coefficients are known for $r = 2$ and any s, and for $r = 3$ with $s = 1$ [34]. In order to implement Zigzag with its optimizations we must obtain the proper coefficients. Coefficients are usually chosen from the smallest possible finite field in order to simplify and accelerate the encoding and recovery procedures [22].

A trivial approach to solving the issue for $r = 3$ is to duplicate the coefficient values together with the element positions in the dependency sets. However, in this case, each of the twins will be linearly dependent. This results in losing the MDS property. Therefore we have to choose the coefficients from a field larger than the one proposed for $r = 3$ and $s = 1$.

In order to compute the parity disks in the Zigzag code, we prefer to use a field of size 256, denoted $GF(2^8)$ since it fits well to commodity hardware instructions [22]. In 4 configurations: $\langle k = 8, r = 4, s = 2\rangle$, $\langle k = 10, r = 4, s = 3\rangle$, $\langle k = 10, r = 3, s = 2\rangle$, $\langle k = 6, r = 3, s = 1\rangle$, we use $GF(2^{16})$ instead, which will be further explained below. Tamo et al. proved that in Zigzag code with $r = 2$ and different duplication factors s, the field size from which we choose coefficients can be $s + 1$ which is small enough for all of our configurations. For $r = 3$, Tamo at al. provide coefficients for the basic construction, in which $s = 1$ [34].

For the rest of our configurations, we must exhaustively search all possible coefficient combinations and check whether they result in an MDS code. In other words, in order to ensure that the resulting construction allows recovery from a failure of any r nodes, the linear equation system that is solved in the decoding procedure must be linearly independent. This property was verified using a simple Matlab program, by exhaustively searching the coefficients space until we reach a valid set. In all but the 4 configurations mentioned in the last paragraph, we found valid coefficients over $GF(2^8)$. However, in the mentioned configurations, we found valid coefficients over $GF(2^{16})$. This doesn't appear to be a problem since the performance of calculations over $GF(2^{16})$ is approximately the same as over $GF(2^8)$ [22].

3.2 Making I/Os More Sequential

In order to quantify the sequentiality of the recovery I/Os, we define the *read distance* as the inclusive distance (in elements) between the first and last elements read in a block in order to recover the failed block. For example, in the $(6, 4)$ code in Figure 3.4(a), when the leftmost data node fails, the read distance is 7 in each of the data nodes and in the row parity. Our first objective is to minimize the *average read distance* — the average read distance when considering all possible single-node failures.

Our second objective is to minimize the number of I/O requests issued to the hard disk in each node. A reduction in the number of requests is expected to reduce recovery time by saving host overheads and by providing the disk scheduler with more reordering opportunities.

3.2.1 Optimal Dependency Sets

We optimize disk access by choosing, for each recovery scenario, to read the elements which result in the most sequential I/O pattern. For example, the shaded elements in Figure 2.2(a) can be read more efficiently than the non-shaded ones. We consider the *read distance* in each surviving data node, and choose the elements which results in the shortest distance for the recovery of the failed node (we disregard the distance of the Zigzag parity elements). Note that the distance may be the same for all parts (as in Figures 2.2(b) and 2.2(c)), and that the optimal part can be easily identified in advance for every recovery scenario.

3.2.2 Optimal Constructions

Zigzag codes enjoy another degree of freedom that does not apply to all erasure codes. They define certain constraints on the choice of coding dependencies and equations, but for every (n, k) pair there are many possible constructions that share the MDS and optimal recovery properties. Specifically, for $m > 4$, some choices of dependency sets result in partitions with shorter average read distances than others. For example, consider the (6,4) code in Figure 3.4 when the first data node fails. With the basic construction (a) the distance between the first and last elements read in each of the surviving data nodes is 7, while in the alternative construction (b) it is

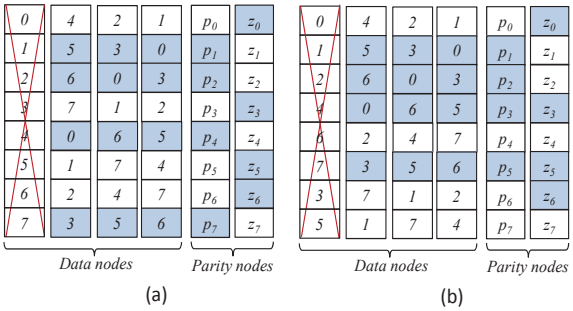


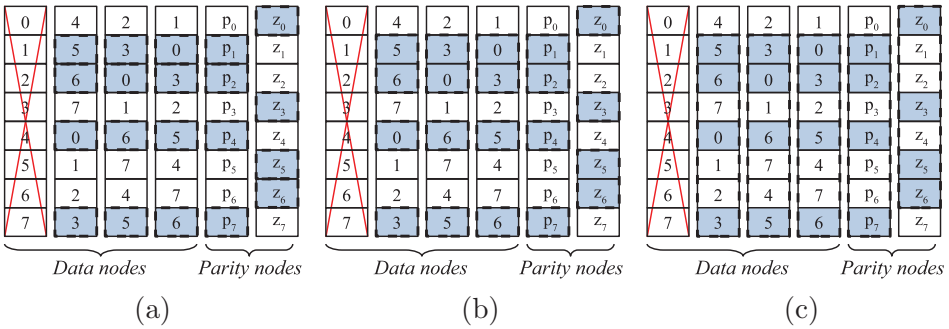
Figure 3.4: Basic (a) and alternative (b) constructions for a (6,4) Zigzag code. When the first node fails, the distance between the elements read in each data node is reduced from 7 in the basic construction to 5 in the alternative (optimal) one.

5. The alternative construction reduces the distance for all but one failure scenario. The optimal code construction in this context is the one with the shortest average distance during recovery. The basic construction (as it appears in the literature [34]) is optimal for $m \leq 4$. For $m > 4$, we find the optimal alternative construction as follows.

r	m	Basic	Optimal
2	8	6	5
3	9	5	3.67
4	16	9	5.67

Table 3.1: Basic/optimal distance.

Alternative constructions can be viewed as a permutation of the rows of the data and row-parity blocks in the array, although no data is actually shifted—they are implemented by adjusting the dependency sets. The optimal construction for a given (n,k) pair can be found by calculating the average distance of each permutation exhaustively. We were able to find the optimal constructions for eight, nine, and sixteen rows in a matter of seconds (see Table 3.1). For more rows, however, examining all possible constructions requires hundreds to thousands of compute hours. In addition, our initial search indicated that most of the optimal constructions improve over the basic ones, and there are simple thumb rules to predict that. We examine the benefit from optimal constructions in Section 5.2.7.



(a) (b) (c)

Figure 3.5: An example of naïve (a), conservative (b) and aggressive (c) I/O request approaches in a $(6, 2)$ Zigzag code

Optimal constructions only replace the order of rows in the code's generator matrix, therefore they don't affect the MDS property of Zigzag.

In our search for optimal constructions we do not attempt to optimize the read distance in the Zigzag parity blocks. Permuting these rows as well would have significantly increased the complexity of our implementation, and we expect the additional benefit to be negligible.

3.2.3 Aligning Elements to Sector Boundaries

When the data requested from the hard disk is not aligned to $4KB$-sector boundaries, it is aligned by the disk scheduler and additional data is read. This may result in a significant amount of excessive reads, and mask the benefit of the small rebuilding ratio [20]. In order to ensure that all our elements are aligned to sector boundaries, we increase the object size by padding. Padding may result in a significant increase in object size. For example, when the element size is originally $5KB$, with $k = 10, r = 3, s = 2$, padding increases the object size by 60%. This is a huge burden on disk and network storing a large amount of zeroes and may decrease performance significantly. Therefore, we treat the resulting "zero elements" as logical zeroes, not storing nor sending them over the network. In this way we avoid unnecessary disk reads, network transmissions and recovery time.

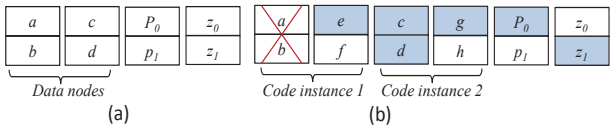


Figure 3.6: (a) Striping in block granularity using the (4,2) Zigzag construction in Figure 1.1(b). (b) Optimized block assignment in the (6,4) Zigzag construction that is a 2-duplication of the (4,2) code in (a).

3.2.4 Coalescing Consecutive Elements

The naïve approach for reading elements from disk is sending a single I/O request per element. In order to minimize the number of I/O requests issued to the hard disk, we coalesce consecutive elements into a single I/O request by using one of two approaches.

In the *conservative approach*, we read neighboring elements in a single I/O request. Each group of consecutive elements required is read by a separate I/O request. For example, in the $(6, 4)$ code in Figure 3.5(b), when the leftmost node fails, we issue 3 disk I/Os instead of the 4 issued by the naïve approach.

In the *aggressive approach*, we read all the required elements within a group in a single I/O request. This minimizes the number of I/O requests, but results in possibly reading more elements that are unnecessary for recovery. For example, in Figure 3.5 5 elements are required for the recovery of the leftmost node, but we aggressively read 7 elements in a single I/O request.

3.3 Degraded Reads and Update Penalty

Erasure codes allow a degree of freedom in the way an object's data can be split into elements. For example, a and b in Figure 1.1(b) are contiguous data elements that were striped in *element granularity* and assigned to nodes in round-robin order. In our design, we stripe objects in the granularity of blocks, rather than elements. This allows read requests that are smaller than an entire object to be served by the minimal number of nodes. Figure 3.6(a) shows the same data from Figure 1.1(b) striped in block granularity. This optimization can be applied to any array code.

In addition, we replace the standard round-robin layout and assign con-

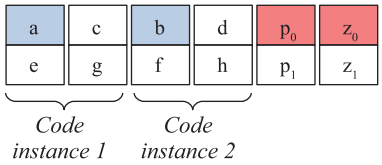


Figure 3.7: An example of a $(6, 2)$ Zigzag code with a layout optimized for low update penalty. Both elements a and b belong to the dependency sets of parity elements p_0 and z_0.

tiguous blocks to twin columns. In many cases, this reduces the number of extra elements required for completing a degraded read request, because the twins of an unavailable block are read anyway. Consider, for example, the (6,4) code in Figure 3.6(b), which is a 2-duplication of the code in Figure 3.6(a), and assume that the first node is unavailable. If the application requests the first two data blocks, elements c and d, which are part of the request, can be used "for free" in the recovery of elements a and b. While the benefit from data striping and assignment depends on the distribution of read request sizes, note that they never *increase* the number of required elements or nodes accessed during recovery.

An interesting property of the mentioned layout is that in addition to improving degraded reads, it can also improve the *update penalty* — the number of elements that are physically updated due to a logical update of a single element. For the following optimization we rely on the row-ordered elements layout described in Figure 3.7, and consider partial object sequential writes. As seen in the figure, updating a single element in an object costs r additional element updates, one for each parity. In the proposed layout, we can update s elements at the cost of one, since twin elements depend on the same parity elements. Therefore update penalty which appears to be a serious problem in array-codes is partially solved using the duplication optimization.

3.4 Recovery by Simple Parity

With hard disks as physical storage, increasing the amount of data read can increase efficiency if this data is read sequentially. Thus, there are two cases in which recovery by the simple parity elements is more efficient than

recovery by both simple and Zigzag parities. In the first case, the client reads and entire object. All surviving data blocks are read as part of the client's request, and the unavailable block can be recovered by additionally reading the entire simple parity block. This is more efficient than reading $\frac{1}{r}$ of each of the parity blocks since the recovery equations are over a binary field, on which the computation is faster.

In the second case, the client requests only c elements from the unavailable node, $c < m$. If $c \leq \frac{m}{r}$, recovery can be accomplished by reading c data rows and their simple parity elements, which results in less than the optimal rebuilding ratio. As a matter of fact, as long as c is strictly smaller than the distance of the optimal recovery partition, recovery by simple parity will result in more efficient I/O than recovery by simple and Zigzag parities.

Chapter 4

Implementation

4.1 Erasure Codes in Ceph

Ceph is a distributed storage system whose performance and scalability build on unreliable, intelligent *object storage devices (OSDs)*. A Ceph cluster is composed of one or more metadata servers (MDS), one or more monitors (MON), and up to hundreds of thousands of OSDs [36]. Ceph's backend storage is RADOS, which is responsible for object placement, failure detection, and recovery [38]. OSDs are organized into *pools*—the storage units exposed to the clients via a block device, file system, or object store interface.

4.1.1 Placement Group Roles

Within a pool, RADOS maps objects to *placement groups*—logical collections of objects that are replicated across the same set of OSDs. An (n,k) erasure-coded pool will have n OSDs in each placement group. Load balancing and parallel I/O access are achieved by defining several placement groups within each pool, so that each OSD belongs to several groups. RADOS uses CRUSH, a pseudo-random mapping function of objects to placement groups [37]. Within each placement group, the *primary* OSD is responsible for serving client read/write requests, possibly with the help of the other (*secondary*) OSDs in the group. When an OSD fails, the primary of its placement group is responsible for managing the recovery of objects stored on them in this group, and degraded reads of these objects. Degraded reads can also be initiated by the primary in case of an unresponsive OSD.

Each placement group can be viewed as an array of nodes. Thus, an OSD that belongs to several placement groups will likely be assigned the role of a different node in each. Specifically, an OSD will store data blocks of one placement group and parity blocks of another. Similarly, it will be the primary OSD (henceforth called the primary) in some placement groups, and a secondary OSD in others. For simplicity, from here on we refer to a single placement group in the description of our implementation.

In an erasure-coded pool, each object is handled as a stripe. A large object may be split into several stripes, but small objects are padded to form a complete stripe. The object size is set when the pool is created, with 4MB as the default. Ceph supports object sizes between 4KB and 2GB. Clients send write requests directly to the primary, which splits the object into k data blocks (*shards*), and calls the `encode` function to generate the parity blocks. It then uses CRUSH to distribute one block to each OSD, according to its position in the array. The write completes when the primary receives an acknowledgement from all the OSDs in the group.

Read requests are also sent directly to the primary. In normal operation, the primary fetches the data blocks requested by the client from the OSDs that store them. If one or more data nodes are unavailable, the primary executes a *degraded read* in order to recover the missing blocks. Specifically, if exactly one data node is unavailable, the primary requests the entire stripe from the surviving data nodes. When k blocks arrive at the primary, it recovers the lost data block by calling the `decode` function, and sends the requested data to the client.

4.1.2 Recovery Process

When a node is permanently unavailable, a new node must be assigned to the placement group. However, CRUSH instead replaces most of the nodes in the placement group, which leads to unexpected data movement. This, in turn, leads to increased disk reads and writes of data that needs to be recovered. We prevent these redundant data movements, by preventing CRUSH from recalculating the placement groups, forcing it to only replace the failed node[1].

Then, the placement-group enters recovery state. We consider two cases:

[1] This is done by setting the `chooseleaf_stable` parameter to true

If the original primary has failed, the new primary will read the required data for recovery and write the recovered data directly to its low level storage. However, if a secondary has failed, the primary fetches data from the surviving OSDs in order to recover the data that was stored on the failed OSD. Then, the primary decodes the missing data, stores it on its own disk and sends the data to the new secondary which replaced the failed one. Even though the data is stored on the primary's disk, it is pipelined directly to the destination OSD and does not incur additional reads.

CRUSH relies on a pseudo random function to assign objects to placement groups. When both the number of nodes and number of objects are sufficiently large, this results in a uniform distribution. However, in our evaluation, we observed significant differences between the number of recovered objects in each experiment. In order to minimize such influences, we set the number of placement groups to 512, which was the maximal possible for our setup.

4.1.3 Limitations

Ceph's erasure-coded pools are ideal for evaluating the performance implications of new erasure codes. They support a wide range of object sizes, with the encoding and decoding mechanisms on the critical path of both writes and degraded reads. However, erasure-coded pools in Ceph are restricted in that they do not allow updates. Partial object updates are implemented by splitting the object into $4KB$ encoded stripes, and ensuring an entire stripe is updated as a whole. Thus, they are unsuitable for the demonstration of a code's small update cost. We discuss the update cost of Zigzag codes in Section 3.3. Ceph also prevents partial degraded reads of erasure coded objects, and converts them to reads of entire objects—all the surviving data blocks are read, even if the decoding mechanism does not require them for recovery. This conversion is a fundamental design choice in Ceph, and thus we were unable to implement our optimizations described in Section 3.3 within the scope of this work.

4.2 Zigzag in Ceph

4.2.1 Code Functionality

When implementing Zigzag codes in Ceph, we must consider the basic encoding and decoding functionality required when a new code is added to a system that supports erasure-coded pools. We must also consider the structural changes required when adding an array code to a system that was designed for scalar codes.

We implement Zigzag encoding for a wide range of parameters, including $6 \leq k \leq 10$, $2 \leq r \leq 4$ and $1 \leq s \leq 5$. We implement and optimize decoding for the recovery from one failure, which is the focus of this work. For all the parameter combinations in our evaluation, we compute, in advance, the coefficients in the coding equations and the optimal dependency sets to use in the recovery of each failed node. When applicable, we choose the optimal construction from Table 3.1, i.e., the one that results in the shortest average distance during recovery. We assign contiguous blocks to twin nodes with Ceph's chunk mapping mechanism.

Some of our parameter combinations allow codes to be constructed with coefficients from a finite field as small as $GF(2^2)$, while the largest arrays require them to be chosen from $GF(2^8)$. The GFComplete library provides an efficient implementation that is similar in performance for $GF(2^4)$ and $GF(2^8)$ [22]. Thus, we simplify our implementation by using coefficients from $GF(2^8)$ for all constructions. This is also the default field size for Reed-Solomon in Ceph. For constructions with virtual nodes, we replace the coefficients of the virtual elements with zeroes in the coding equations they appear in. This effectively removes them from the dependency sets, so that they are never actually required for decoding.

4.2.2 Structural Changes

The main challenge in implementing Zigzag in Ceph is handling elements as sub-parts of blocks. CRUSH is repeatedly queried by both primary and secondary OSDs, in order to assign and discover the location of data and parity blocks. A näive implementation will split objects into elements, emulating a $k \times m$ scalar code, and allow CRUSH to determine their location. However, CRUSH will then treat each element as a block, and might place

elements of the same block on different OSDs, violating the MDS property of the code. In order to use CRUSH correctly, we are forced to distinguish between elements and blocks within the erasure code implementation itself, and to modify the interface between the primary and secondary OSDs as follows. We modify `encode` to split objects into an $m \times k$ array and generate $m \times r$ parity elements. The elements in each column are grouped into a block, so that CRUSH distributes the n blocks across the placement group, oblivious to their internal structure.

We ensure that elements are aligned to sector (4KB) boundaries, to allow efficient access during recovery. This is a modification of the original design that aligns blocks to the largest vector word size boundaries (128B-512B). Alignment is done by "rounding up" the stripe size according to the aligned element size, adding padding at the end of the stripe. The amount of padding depends on the number of elements and their non-aligned sizes, and can be considerably larger than that of the original block alignment. We logically treat the padded data as zeroes, which makes its effect on the evaluation negligible. The primary, which is responsible for encoding and decoding, keeps record of the elements that were added at the end of the stripe as a result of padding and contain only zeroes, and removes them from the dependency sets they belong to. Thus, they are never required for recovery, and never requested from the secondary OSDs.

Next, we modify `required_to_reconstruct`, the function that determines which blocks to fetch for degraded reads or recovery. In its original implementation, in case of a single failure, this function marks all available data blocks and the simple parity block as "required" in a special structure, `read_request_t`, which is sent to k OSDs. Each OSD identifies its own block in this structure, reads it, and sends it to the primary. This is the case even if the client requests a small portion of the unavailable block. We modify the `read_request_t` structure to include a bitmap of the object's elements, and modify the function to set the bits corresponding to those data and parity elements in the dependency sets chosen for recovery.

We modify the block-based interface between the primary and secondary OSDs to allow recovery by elements, rather than full blocks. The primary includes a bitmap of the object's elements, and sets the bits corresponding to those data and parity elements in the dependency sets chosen for recovery. Each OSD reads the required elements from its low level storage,

and aggregates them into a single response buffer. The primary maps the content of each response buffer to the required elements and then calls the `decode` function to recover the missing data.

Chapter 5

Evaluation

We use our implementation of Zigzag in Ceph to evaluate the practical benefit of the optimal rebuilding ratio. Our goal is to understand how our optimizations affect the encoding and decoding throughput of Zigzag codes, and how the tradeoff between element size and rebuilding ratio is reflected in the overall recovery cost. We focus on codes that do not incur additional overheads, and thus use Reed-Solomon, the most widely used MDS code, as our baseline. We discuss additional codes in Chapter 6.

5.1 Encoding and Decoding Throughput

Previous studies showed that the encoding and decoding efficiency have little to no effect on the overall recovery cost of large scale systems. In multiprocessor storage nodes, these computations are streamlined with the network transfers and I/O overheads required for distributing an object's blocks and storing them on (or fetching them from) durable storage [7, 16, 17, 19, 21]. To confirm these assumptions, we measure the in-memory encoding and decoding throughput with a standalone implementation of Zigzag that uses the Jerasure [23] and GFComplete [22] libraries. It also includes a Reed-Solomon implementation that is Ceph's default for erasure-coded pools. We run our tests on a dedicated server with two 8-core Intel® Xeon® 2.40GHz CPUs.

In each test, we write an object of random data and encode it while it is still in memory. We then measure the average time to decode the object

when one block is missing. We repeat the process for each of the blocks, separately, and average the results for all n blocks. Each experiment consists of a different set of parameters, for which we run five tests, each with a new randomly generated object, and report the average of the results.

We vary the stripe size ($6 \leq k \leq 10$), the number of parity nodes ($2 \leq r \leq 4$), and the duplication factor ($1 \leq s \leq 5$), and run an experiment for every valid parameter combination. We repeat each experiment with different object sizes (4MB-32MB), field sizes ($GF(2^8)$-$GF(2^{32})$), alignment values (512B-16KB), and optimized constructions. In relevant configurations (see Table 3.1), we repeat the experiment with basic and optimized constructions. With a total of almost 2000 experiments, we summarize our findings and omit the detailed results.

We find that only two parameters affect computation efficiency considerably. The first is the field size, which affects Reed-Solomon and Zigzag in a similar manner: their encoding throughput with $GF(2^{16})$ is up to 23% and 16% higher than with $GF(2^8)$ and $GF(2^{32})$, respectively. We believe the cause for this behavior is the implementation of the finite field multiplication in GFComplete. For consistency, we use Ceph's default field, $GF(2^8)$, in the rest of our evaluation.

The second parameter was the number of parity nodes, which affected the encoding throughput of Zigzag more than it did this to Reed-Solomon. The encoding throughput of Reed-Solomon was up to 96% higher than Zigzag's with $r = 2$, and up to 52% and 36% higher with $r = 3$ and $r = 4$, respectively. This is the result of the number of coefficients which equal 1 in the coding equations of Zigzag, which is considerably lower than those of the Reed-Solomon implementation in Ceph. *Jerasure* applies simple XOR operations in those cases, instead of the finite-field multiplication required in Zigzag. Decoding a single missing block is 3-6 times faster in Reed-Solomon, which uses the simple parity block and equations in this scenario. Nevertheless, the results from our evaluation in Ceph show that the CPU utilization is similar during both Zigzag and Reed-Solomon decoding, and is not the bottleneck of the recovery process.

We find that the optimized constructions, element alignment, stripe size, and virtual nodes do not affect encoding and decoding throughput. This is true also for code duplication, with one interesting exception. Zigzag with $r = 2$ and no duplication is 39%-108% faster than Zigzag with the same

parameter combination with duplication. In this basic Zigzag construction, 51% and 79% of the coefficients are 1, when k is 6 and 8, respectively. This eliminates a considerable amount of finite field multiplications, speeding up encoding and decoding considerably.

Encoding in Reed-Solomon is 4%-96% faster than in Zigzag, thanks to its highly optimized implementation in Ceph. This advantage is more pronounced with $r = 2$ (21%-96%faster) than with $r = 3$ (4%-52% faster) and $r = 4$ (5%-36% faster). Decoding a single missing block is 3-6 times faster in Reed-Solomon, which uses the simple parity block and equations in this scenario. However, if we force decoding with another parity, decoding is 33%-66% slower in Reed-Solomon. This is because Zigzag decodes half of the elements with simple parity equations.

Increasing the field size from $GF(2^8)$ to $GF(2^{16})$ in Zigzag increased encoding throughput by 8%-23%. However, increasing it to $GF(2^{32})$ *reduced* throughput by 1%-16%. Reed-Solomon experienced similar behavior. Decoding throughput, however, was not consistently affected by field size. We believe the cause for this behavior is the implementation of the finite field multiplication in GFComplete. We use Ceph's default field, $GF(2^8)$, in the rest of our evaluation.

The relative difference between encoding and decoding throughput of Reed-Solomon and Zigzag codes is high. However, on a single core, Zigzag encoding throughput is 60-230MB/sec, and its recovery throughput is 50-150MB/sec. In a multiprocessor storage node, these computations are streamlined with the network transfers and I/O overheads required for distributing an object's blocks and storing them on (or fetching them from) durable storage. We thus focus on the storage and network costs of recovery.

5.2 Cluster Recovery Cost

Our cluster consists of 10 servers, each equipped with two 8-core Intel® Xeon® 2.40GHz processors and two 0.5TB HDDs, connected by a 10Gb Ethernet switch. We use one of the servers to run the metadata server, monitor, a single OSD and an I/O workload generator, which we describe below. We set up two OSDs on each of the remaining servers, one per HDD, so that the total number of OSDs in each experiment is 19, 18 of which survive. In each experiment, we first populate the cluster with objects so

that each HDD stores a total of 10GB of data blocks, plus the parity blocks generated by the encoding mechanism.

To ensure that recovery data is read directly from the disk, we clear the caches of Ceph before every run of the experiment[1]. During the recovery process, every object is read at most once so that the caches do not interfere with our measurements.

Next, we kill one OSD daemon and remove this OSD from the cluster. This initializes the recovery process, which recovers the data and distributes it across the remaining OSDs. We repeat the entire experiment with Reed-Solomon and with Zigzag, using the configurations summarized in Table 5.1. Due to the random mapping of objects to placement groups in CRUSH, the number of objects that are recovered in each configuration may vary by up to 15%. We eliminate this variation from our results by calculating the amount of data read from disk and the amount of data transferred per GB of data recovered in each experiment. We use three different object sizes, 4,16, and 64MB, aligning all elements to sector (4KB) boundaries.

As we expected, encoding and write throughput are the same for Reed-Solomon and Zigzag. The CPU utilization is also the same, both during cluster population and during recovery. We thus focus on the storage and network costs of recovery.

5.2.1 Disk Reads and Rebuilding Ratio

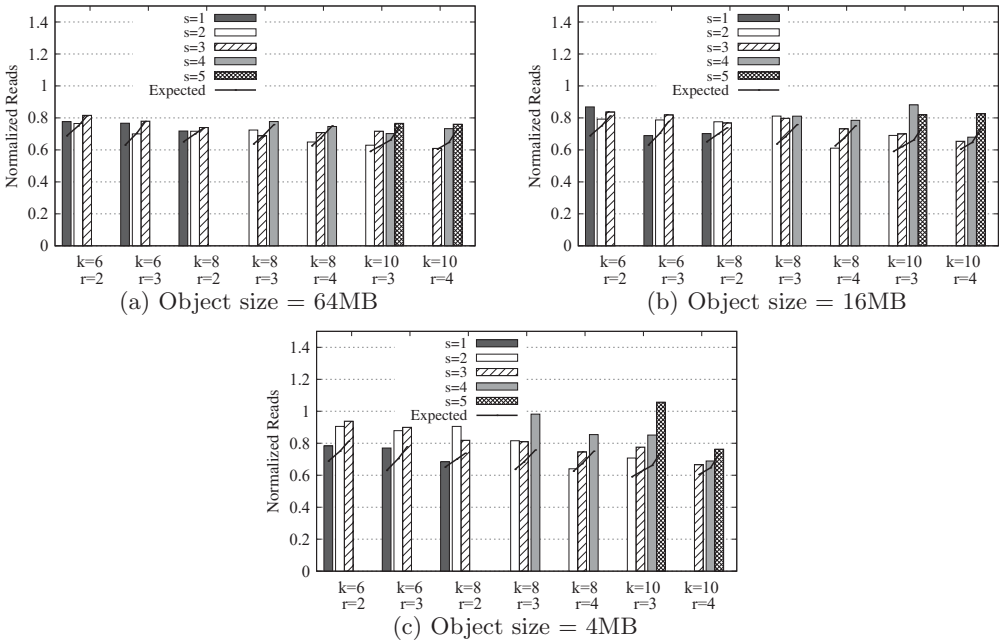
Figure 5.1 shows the amount of data read from disk per GB recovered by each of the Zigzag constructions, normalized to the amount of data read by Reed-Solomon with the same k,r, and object size. Our results show that Zigzag can reduce the amount of data read for all (k,r) configurations with at least one construction, and usually with all of them. With $64MB$ objects (Figure 5.1(a)), Zigzag reads as little as 63% and 61% of the data read by Reed-Solomon, when $k = 10$ and r is 3 and 4, respectively. The reduction in the amount of data read is 20%-30% in almost all the combinations we examined.

For each configuration, we also show the *expected* amount of reads, which

[1]We do this by applying two simple commands: `sudo echo 3 | sudo tee /proc/sys/vm/drop_caches && sudo sync`

k	r	s	m	Data elements	RR	Element size (KB)
6	2	1	32	192	0.5	21 / 85 / 341
		2	4	24	0.57	171 / 683 / 2730
		3	2	12	0.64	341 / 1365 / 5461
	3	1	243	1458	0.33	3 / 11 / 45
		2	9*	56	0.42	73 / 293 / 1170
		3	3	18	0.5	228 / 910 / 3641
8	2	1	128	1024	0.5	4 / 16 / 64
		2	8*	64	0.55	64 / 256 / 1024
		3	4	32	0.6	128 / 512 / 2048
	3	2	27	216	0.4	19 / 76 / 303
		3	9*	72	0.45	57 / 228 / 910
		4	3	24	0.53	171 / 682 / 2731
	4	2	64	512	0.32	8 / 32 / 128
		3	16*	128	0.37	32 / 128 / 512
		4	4	32	0.45	128 / 512 / 2048
10	3	2	81	810	0.39	5 / 20 / 81
		3	27	270	0.42	15 / 61 / 243
		4	9*	90	0.47	45 / 182 / 728
		5	3	30	0.56	137 / 546 / 2185
	4	3	64	640	0.34	6 / 26 / 102
		4	16*	160	0.39	26 / 102 / 410
		5	4	40	0.48	102 / 410 / 1638

Table 5.1: Zigzag constructions in our evaluation. m is the number of rows, RR is the rebuilding ratio of data disks, and starred configurations also include an optimal construction. We include the resulting element size (before alignment) for 4/16/64MB objects.



(a) Object size = 64MB

(b) Object size = 16MB

(c) Object size = 4MB

Figure 5.1: The amount of data read by Zigzag during recovery, normalized to the amount read by Reed-Solomon, using the conservative reads approach.

is the rebuilding ratio of Zigzag, including recovery of parity nodes, divided by the rebuilding ratio of Reed-Solomon. Our results show that with large objects ($64MB$), the amount to data read is very close to this expected amount. The difference between the amount read and the expected amount increases as object size decreases, as shown in Figures 5.1(b) and 5.1(c), for objects of size $16MB$ and $4MB$, respectively. The reason for this difference is the disk's readahead mechanism, which reads more sectors from the disk than are requested. This affects Zigzag, especially with small elements, while it does not affect Reed-Solomon which reads entire blocks.

It is interesting to note that while the optimal recovery cost ($\frac{1}{r}$) decreases as r increases, the reduction in recovery cost is similar for all r values in our evaluation. The reason is that in practice, larger values of r require larger duplication factors, which increase the rebuilding ratio.

Increasing the duplication factor, s, reduces the element size, but also increases the theoretical rebuilding ratio. Indeed, we observe that its optimal value is not necessarily the maximal one—when the elements are large enough to ensure efficient disk reads, increasing the duplication factor fur-

ther increases the amount of data read. This increase is usually small (up to 3%), with the exception of one setup: with $k = 6$ and $r = 2$, increasing s from 2 to 3 increases the element size from 2730 to 5461KB, and increases the amount of data read by 19%.

Reducing the object size reduces the element size, which reduces the sequentiality of recovery reads. Figures 5.1(b) and 5.1(c) show the normalized amounts of data read with objects of size $16MB$ and $4MB$, respectively. Thanks to our use of code-duplication, we are able to ensure that elements are large enough to always reduce disk reads considerably. With $16MB$ objects, Zigzag reads as little as 57%, 66% and 65% of the data read by Reed-Solomon, when $k = 8$ and r is 2, 3 and 4, respectively. With $4MB$ objects and the optimal duplication factor, Zigzag reads as little as 68%, 81% and 64% of the data read by Reed-Solomon, when $k = 8$ and r is 2, 3 and 4, respectively.

We note that the optimal duplication factor for the same k and r combination depends on the object size. This demonstrates the sensitivity of read efficiency to the interaction between element size and disk characteristics—namely, its track size and read-ahead parameters. Choosing the optimal duplication factor ensures minimal recovery I/O cost, but may require careful fine tuning. Fortunately, our results indicate that a simple rule of thumb, such as choosing the smallest s for which $m \leq 256$, achieves most of the cost savings with little risk of unwanted overheads.

5.2.2 Read-Ahead Mechanism

The evaluation of the disk reads in Figure 5.1(c) reveals a common pattern in which objects with element sizes which are larger than and are not aligned to $128KB$ boundaries, incur higher read overheads. We specifically refer to the two configurations: $\langle k = 10, r = 3, s = 5\rangle$ and $\langle k = 8, r = 3, s = 4\rangle$, in which the element sizes are $137KB$ and $171KB$, respectively. In these configurations, the amount of data read is 42% and 30% more than the optimum.

In contrary, in the configuration $\langle k = 8, r = 2, s = 3\rangle$, in which the element size is precisely $128KB$, we read only 11% more than the optimum. This phenomena can be explained by disk read-ahead units, which are normally $32KB$-$128KB$ in size [8].

5.2.3 Disk Writes

Ceph's *journal* is responsible for lowering the writes latency by writing the client's data to a contiguous journal file before writing to the randomly positioned destination. Since sequential writes on HDDs are faster, the journal decreases Ceph's latency. However, the journal causes some of the data to be written twice, firstly at the journal and then at the object's location on disk. To avoid measurements of the double writes we moved the journal file to a `tempfs`, i.e. a file-system which resides on memory. Doing so in real system settings puts Ceph at risk of losing data in case of failures, but is acceptable for measurement and evaluation purposes in our lab setting. We ensure that the journal's data is flushed to disk before we start our recovery experiment, so that all the recovery reads are taken into account when measuring the amount of data read from disk.

The amount of measured writes during recovery is approximately double the amount of recovered data. The main reason is that the primary OSD manages recovery, and therefore all data needed for recovery is sent to the primary OSD and written on its local disk to ensure consistency. Only then, it is sent to the destination node. This increases the amount of writes as long as the primary is not the destination node. We calculated the expected number of writes considering the number of objects recovered, and the number of objects with a failed primary. The measured writes are 7% to 30% more than the expected, due to additional mechanisms of Ceph such as commit logs which we didn't consider in our calculation. Figure 5.2 demonstrates the ratio between the measured writes with $64MB$ objects. The results for other object sizes were higher. For example, writes for $4MB$ using the conservative approach were 60% to 227% higher than expected. We noticed that more elements increased the difference between measured to expected writes. This strengthens our assumption that consistency mechanisms which store per-object data cause the excessive writes.

5.2.4 Network Transfers

Figure 5.3 shows the amount of data transmitted across the cluster per GB recovered with each object size and configuration. We expect this amount to be slightly higher than the amount of data read from disk due to the additional metadata that must be transferred with each node's elements.

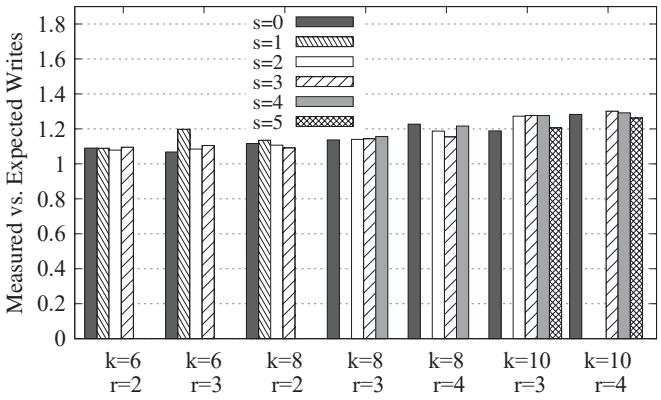


Figure 5.2: The ratio between measured writes to expected writes in 64MB objects using the conservative reads approach

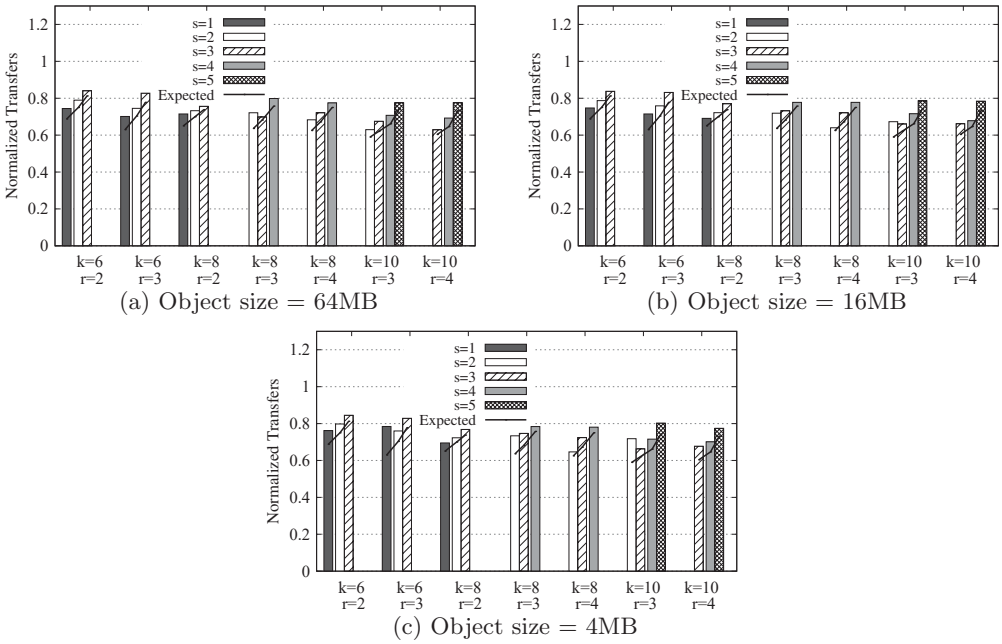
Indeed, in configurations where the elements are smaller than 64MB, the amount of data transferred is 3%-8% higher than the theoretical optimum.

However, In configurations with element sizes of approximately $64KB$-$512KB$, the amount of data transferred is smaller than the amount of data read from the disk, because it is not affected by the disk's read-ahead mechanism that reads unnecessary data.

5.2.5 Recovery Time

Figures 5.4 and 5.5 show the recovery time of Zigzag, normalized to that of Reed-Solomon, using conservative and aggressive approaches. With 64MB objects (Figure 5.4(a)), Zigzag completes the recovery in as little as 72% and 76% of the time required by Reed-Solomon, when $k = 6$ and r is 2 and 3, respectively. With objects of size 64MB and 16MB (Figure 5.4(b)), Zigzag can reduce the recovery time for all (k,r) configurations with at least one construction, and usually with all of them. However, with small objects (Figure 5.4(c)), there are many configurations in which the recovery time is close to, or even exceeds, that of Reed-solomon.

The reason is that Zigzag sends many small read requests to the disk, which must be performed synchronously and serialized due to Ceph's design. The number of requests is proportional to the number of elements (detailed



(a) Object size = 64MB

(b) Object size = 16MB

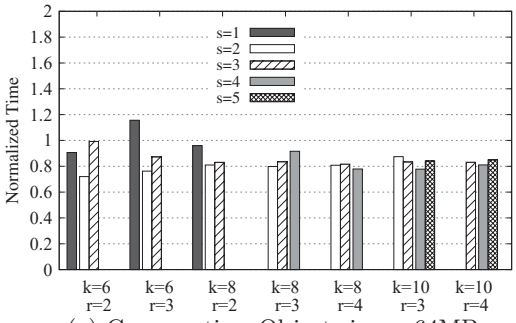
(c) Object size = 4MB

Figure 5.3: The amount of data transferred by Zigzag during recovery, normalized to the amount transferred by Reed-Solomon, using the conservative reads approach.

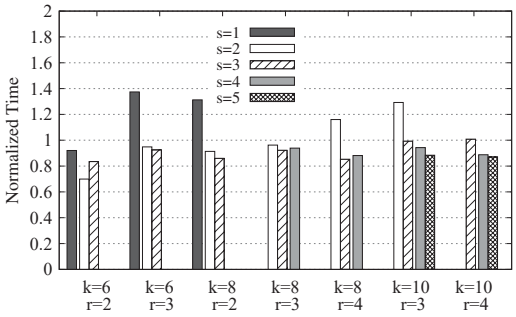
in Table 5.1). Although we coalesce requests to adjacent elements, the number of I/Os is still high, and their context-switching overhead is dominant when the amount of data read is small, even if much of it was prefetched by the disk. This problem motivated our aggressive approach for issuing disk accesses.

5.2.6 Aggressive Approach

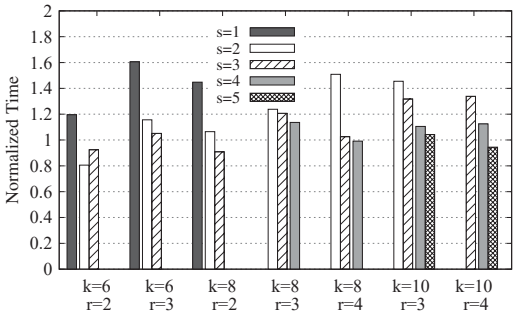
We repeated our experiments using the aggressive approach for coalescing elements. The recovery time measurements can be seen in Figure 5.5. Generally speaking, using the aggressive approach, larger elements result in a shorter recovery time. This is due to the lower amount of asynchronous operations required for recovery. When compared to the conservative approach, the aggressive approach has a shorter recovery time when the duplication factor s is large. However, with smaller duplication factors, the aggressive approach recovers much slower than the conservative approach. Smaller duplication factors result in more rows and therefore, using the aggressive



(a) Conservative, Object size = 64MB

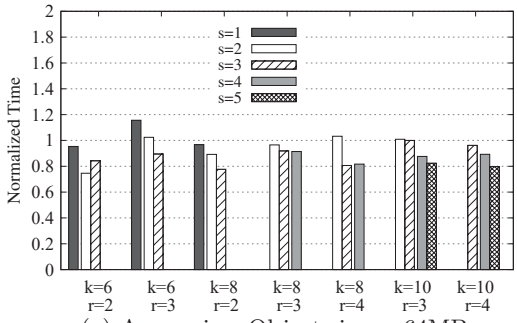


(b) Conservative, Object size = 16MB

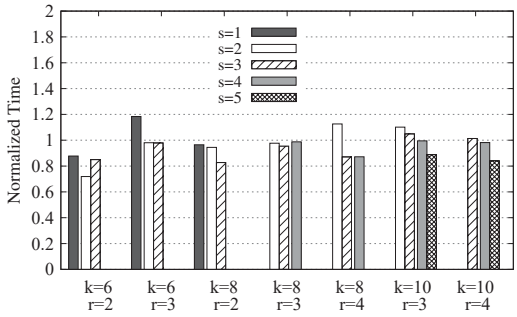


(c) Conservative, Object size = 4MB

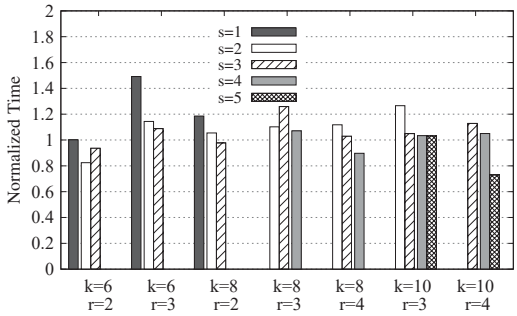
Figure 5.4: Recovery time of Zigzag, normalized to that of Reed-Solomon, using the Conservative approach.



(a) Aggressive, Object size = 64MB

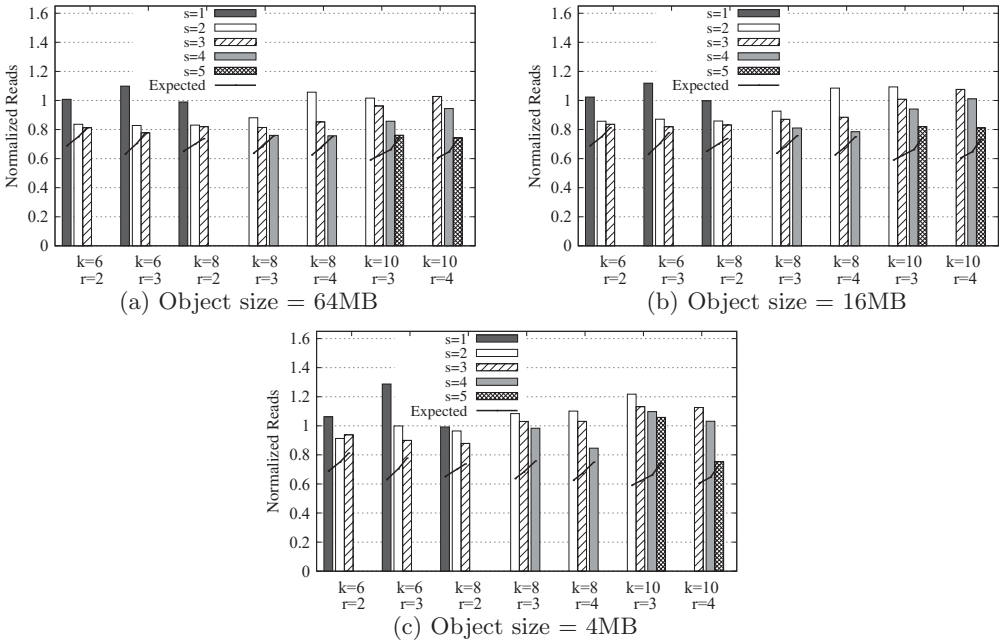


(b) Aggressive, Object size = 16MB



(c) Aggressive, Object size = 4MB

Figure 5.5: Recovery time of Zigzag, normalized to that of Reed-Solomon, using the Aggressive approach.



(a) Object size = 64MB

(b) Object size = 16MB

(c) Object size = 4MB

Figure 5.6: The amount of data read by Zigzag during recovery, normalized to the amount read by Reed-Solomon, using the **aggressive** reads approach.

approach, more elements which are not needed for recovery are read, increasing recovery time. However, when the number of rows is small, the aggressive approach results in fewer I/O requests and therefore the recovery is faster. For example, in the configuration $\langle k = 10, r = 4, s = 5\rangle$ with all object sizes, the aggressive approach results in a shorter recovery time.

Figure 5.6 shows the amount of data read using the aggressive approach. The same principle applies here: lower duplication factor causes the aggressive approach to read more. For example, when $\langle k = 8, r = 2, s = 1\rangle$, we read $98\% - 99\%$ of Reed-Solomon, in all object sizes. Using the conservative approach, the amount of reads varies between 68% to 71% with all object sizes. Using the aggressive approach, we read a number of elements precisely matching the *read distance* presented in Section 3.2. The difference in disk reads between aggressive to conservative approaches is equivalent to the difference between the read distance, and the number of elements needed for recovery. Clearly, this difference is larger when the number of rows is larger, therefore smaller duplication factors result in more reads.

r	2	3			4	
k	8	6	8	10	8	10
4MB	**22**	**10**	-1	-6	**1**	-3
16MB	-1	-8	-2	-8	**4**	**7**
64MB	-2	-6	-5	**1**	0	-4

(a) Conservative Approach

r	2	3			4	
k	8	6	8	10	8	10
4MB	-8	-10	-8	-9	-7	-11
16MB	-5	-9	-6	-10	-7	-11
64MB	**6**	-3	-5	-1	-1	0

(b) Aggressive Approach

Table 5.2: The difference (in percent) between the data read by the basic construction and the optimal one. Positive values (marked in bold) indicate that the optimal construction reads more data than the basic one.

5.2.7 Optimal Constructions

Using the aggressive approach, the optimal construction improves reads in all cases but one, as can be seen in Table 5.2(b). This is since the aggressive approach minimizes the number of separate reads. With a minimum value of separate reads, decreasing the average distance clearly reduces the number of measured reads. A single configuration acts differently: when $\langle k = 8, r = 2, s = 2\rangle$ with objects of size $64MB$, the optimal construction increases the amount of data read. The reason is that the optimal construction in this case enlarges a read of 4 sequential elements to 5 elements. The average read distance is indeed reduced, but the read-ahead mechanism might cause the optimal construction to read more.

The conservative approach, presented in Table 5.2(a), is usually preferable, because it reduces the amount of data read, compared to Reed-Solomon, more than the aggressive approach. The difference in the amount of data read by the optimal construction and the basic one was 1%-8% in almost all our configurations using the conservative approach, demonstrating their success in making I/Os more sequential. There was, however, one exception, the same one we observed in the aggressive approach. In the construction $\langle k = 8, r = 2, s = 2\rangle$, the amount of data read increased by 22%. The reason is that the optimal construction reduces the average distance of elements read, but might increase the number of I/O requests issued in some scenarios. This might increases the amount of data prefetched by the disk, as well as the overall recovery time. As observed in the aggressive approach, minimizing the number of separate reads improves the situation, but still the optimal construction does not improve in this specific configuration.

n	k	s	Reed-Solomon	Zigzag	Pyramid	Normalized Pyramid
8	6	1	6	4.12	3.33	3.75
9	6	1	6	3.78	3.6	4
10	8	1	8	5.2	4.36	4.8
11	8	2	8	5.09	4.67	5.09
12	8	2	8	5	4.92	5.33
13	10	2	10	5.9	5.71	6.15
14	10	3	10	6.04	6	6.43

Table 5.3: The expected amount of data read for reconstruction in case of a single failure, normalized to Reed-Solomon, for a (n, k) code of the specified type. We assume the lowest duplication factor used in our experiments in the Zigzag column. The minimal number of read blocks for a (n, k) configuration is marked in bold.

5.3 LRC Comparison

In order to provide a more thorough comparison of Zigzag to common code, we compare each (n, k) Reed-Solomon configuration with $r = n - k$, to a (n, k) basic Pyramid code configuration. We thus compare each Reed-Solomon configuration to an LRC configuration that protects against (at least) the same number of concurrent arbitrary failures, r. Namely, we replace one global parity with two local parities, increasing the storage overhead by $\frac{1}{k}$. Indeed, for the same reliability characteristics, Pyramid and Zigzag represent opposite points on the trade-off between storage overhead and data sequentiality. The storage overhead in Pyramid is larger than the overhead of Reed-Solomon and of Zigzag, as will be shown below.

The LRC configuration we consider requires an additional node in each placement group. As a result, each failed node will affect more placement groups and will require reconstruction of more missing blocks. Consider, for example, our evaluation setup with 19 nodes, 512 placement groups and 3040 objects. With $(n = 8, k = 6)$, we expect the failed node to store blocks from 1280 objects, since the probability that a placement group consists of the failed node is $\frac{8}{19}$. However, in the basic Pyramid construction, the stripe size is 9 due to the addition of an extra parity node. One failed node will store blocks from 1440 objects, since the probability that a placement group consists of the failed node is $\frac{9}{19}$ in this case. This is $\frac{9}{8}$ times the number of failed blocks in the corresponding Zigzag configuration.

In order to compare Pyramid to our Reed-Solomon and Zigzag results,

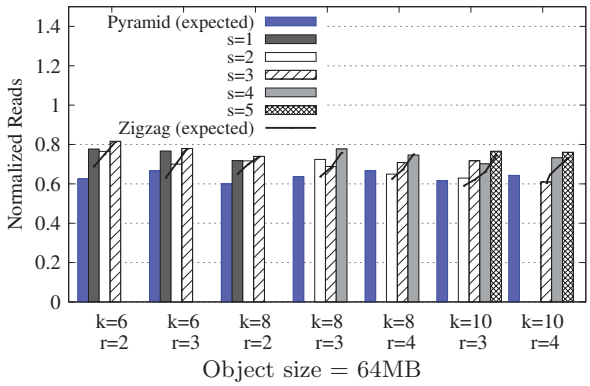


Figure 5.7: A theoretical evaluation of the expected ratio between the chosen Pyramid configuration reads in case of a failure, to Reed-Solomon. Next to it, we show the amount of data read by Zigzag during recovery, normalized to the amount read by Reed-Solomon, using the **conservative** request coalescing.

we first calculate the amount of data read per failed block in each code. We then multiply this amount in the case of LRC by $\frac{n+1}{n}$, to reflect the higher number of blocks that must be repaired for a single failure. Table 5.3 shows the results of these calculations for the configurations we examined. The normalized pyramid column reflects multiplying by the $\frac{n+1}{n}$ factor mentioned above. It is noticeable that the only configurations in which the Pyramid configuration reads less than Zigzag are $\langle n = 8, k = 6\rangle$ and $\langle n = 10, k = 8\rangle$. In the rest of the configurations, not only the storage overhead in Zigzag is lower, but also the amount of data required for recovery is lower than in basic Pyramid code.

We now compare the expected amount of data read by Pyramid code to our experimental measurements in Ceph. Figure 5.7 shows the amount of data read by Reed-Solomon and Zigzag with 64MB objects, as well as the amount of data we expect to read by the Pyramid configurations.

We observe that when $\langle k = 8, r = 4\rangle$ and $\langle k = 10, r = 4\rangle$, Zigzag is better than Pyramid both theoretically and practically. In both cases there are 4 parity nodes, which cause an extremely low theoretical bound on the rebuilding-ratio ($\frac{1}{r}$). In addition, when $\langle k = 6, r = 3\rangle, \langle k = 8, r = 3\rangle, \langle k = 10, r = 3\rangle$, Zigzag is theoretically better than Pyramid, but practically requires more reads. These configurations all have $r = 3$, therefore Zigzag has a low rebuilding ratio. On the other hand, according to Section 5.2.2,

these are the configurations in which the read-ahead mechanism induces the highest read overhead.

When $\langle k = 6, r = 2\rangle, \langle k = 8, r = 2\rangle$, Pyramid is better than Zigzag both theoretically and practically. In these configurations $r = 2$, and therefore the rebuilding ratio of Zigzag is higher than with larger r values. The extra reads caused by the read-ahead mechanism induce a non-negligible overhead, which causes Zigzag require more reads.

It is worth mentioning that in Pyramid code, we believe that the amount of data actually read will be a lot closer to the theoretical amount, thanks to reading full blocks. However, determining the effect on recovery time requires a full implementation in Ceph, which is outside the scope of this work.

Chapter 6

Discussion

6.1 Recovery of Parity Nodes

A recent study [40] described a general construction for *symmetric* codes, for which the rebuilding ratio of parity nodes equals that of data nodes, and is optimal ($\frac{1}{r}$). With their approach, we can construct *symmetric Zigzag*—an extension to Zigzag codes for which the rebuilding ratio is $\frac{1}{r}$ for all nodes. The structure of the dependency sets is similar to that in Zigzag, but the simple parity node is replaced with an additional Zigzag parity, to allow optimal recovery of the parity nodes. The improved rebuilding ratio comes at a cost: an (n,k) symmetric Zigzag code with $r = n - k$ parity nodes has $m = r^{n-1}$ rows, compared to r^{k-1} rows in Zigzag.

The optimizations described in Chapter 3 can be applied to symmetric Zigzag codes. However, they are likely to be less effective due to their large number of rows. Specifically, applying symmetric Zigzag in real system settings will require choosing a high duplication factor, s, which in turn in-

k	r	m	Zigzag (original)	Symmetric Zigzag
8	2	8	0.67 ($s = 2$)	0.67 ($s = 4$)
8	3	27	0.54 ($s = 2$)	0.8 ($s = 8$)
10	3	27	0.54 ($s = 3$)	0.83 ($s = 10$)
10	4	256	0.46 ($s = 2$)	0.77 ($s = 10$)

Table 6.1: Average rebuilding ratio (both data and parity nodes) of the original and symmetric Zigzag codes with the same number of rows (the smallest possible for the symmetric code).

creases the rebuilding ratio. Table 6.1 compares the average rebuilding ratio (of both data and parity nodes) of symmetric Zigzag and that of the original Zigzag for several (n,k) constructions with the same target number of rows. The average was calculated as $[k\times$(ratio of data blocks)$+r\times$(ratio of parity blocks)$]/n$. The resulting rebuilding ratio of the symmetric code is many times higher than that of the original one, which can also be constructed with considerably smaller numbers of rows. Thus, the applicability of the symmetric codes is restricted to systems that can either tolerate very small element sizes, or use very large objects.

6.2 Duplication of General Array Codes

Although duplication was originally proposed for Zigzag codes [34], it can be applied in a similar manner to array codes in general, with similar use of twin elements and nodes for recovery, and similar increase in rebuilding ratio. However, it requires finding the set of coefficients that will preserve the code's properties—most importantly, its being MDS. In some cases, the coefficients have to be chosen from a larger field than those of the original, duplicated code. This has no practical effect on Zigzag, but binary (XOR-based) codes such as Butterfly or EVENODD will not be binary in their duplicated version. The benefits from code duplication should therefore be evaluated in the context of the design goals of each specific code.

6.3 Small Update Cost

Most systems apply an append-only model in which blocks of an existing object cannot be updated once written to permanent storage [3, 16, 26]. If their data must be modified, the entire object is replaced. However, in the general case, client write requests may partially modify an encoded stripe's data, which requires the parity elements to be updated as well. The *small update cost* (or *update penalty*) is defined as the number of parity elements that must be updated when one data element is updated. This update requires reading each parity element's dependency set, though these sets need not be disjoint.

In Zigzag codes, each data element belongs to the dependency set of exactly r parity elements, which is the optimal lower bound. However, these

parity elements and the data elements in their dependency set are distributed across the array's rows, requiring nonsequential I/O access. This problem is common to all array codes, similar to the nonsequential I/O access during recovery. Due to the limitations in Ceph described above, we were unable to directly evaluate the update penalty of our optimized Zigzag constructions. We expect that our optimizations for reducing the number of rows will also reduce the update cost considerably.

6.4 Solid State Storage Nodes

While hard disks continue to dominate the storage landscape, solid state drives (SSD) store increasing portions of the world's data, and erasure-coded pools of SSDs are not uncommon [5]. SSDs have no mechanical moving part, enabling them to perform random and sequential I/O at comparable speeds. However, SSD performance is highly sensitive to write amplification (caused by out-of-place updates) and garbage collection overheads. Thus, although motivated by hard disks, reducing the number of rows and increasing element size to match the SSD page size is desirable when implementing Zigzag on nodes equipped with SSDs.

Chapter 7

Related Work

XOR based codes such as EVENODD [1], RDP [6], HoVer [13] and WEAVER [14] were designed to increase the efficiency of encoding and decoding procedures. However, recent studies show that with today's processors, finite field operations for fields up to $GF(2^{32})$ are highly efficient [22], and their computational overhead has little impact on overall recovery cost [7, 16, 17, 19, 21]. Thus, current research efforts focus instead on reducing network traffic and storage I/O costs.

One approach is to reduce recovery costs of existing codes. Examples include delaying recovery to amortize its costs [31] and storing parity on an additional non-volatile memory to reduce traffic and storage overheads [29]. Another algorithm [17] minimizes the number of elements read in each failure scenario in XOR based codes. It incurs nonsequential I/Os and achieves an average rebuilding ratio of $\frac{3}{4}$ (the optimal for EVENODD [35]) or higher.

Local reconstruction codes add parity nodes to an array, so that one node can be recovered accessing only a small subset of the nodes in the array. Examples include Product [9] and Pyramid codes [15], LRC, which is used in Windows Azure [16], HACFS [39], Xorbas [30], and optimal-LRC [33]. These codes are non-MDS: the dependency sets of the additional *local* parity nodes are a subset of the data nodes, so they do not protect against arbitrary node failures. Thus, these codes introduce a tradeoff between storage efficiency and repair speed. HACFS [39] builds on such codes to dynamically adjust the system's overall storage overhead and average recovery speed by migrating hot and cold data to arrays with more or fewer parity nodes, respectively. Another family of non-MDS codes addresses the

recovery of a failed block within an otherwise healthy node. SD [21] and STAIR codes [19] add parity blocks that allow efficient recovery of bad hard disk sectors or SSD blocks. In contrast to the above approaches, Zigzag codes are MDS, and reduce recovery costs without increasing storage costs.

Minimum storage regenerating (MSR) codes [7, 24] reduce network bandwidth but read the entire content of the surviving nodes. RBT [27] improves on product-matrix MSR codes [24] by reducing the amount of data read from some of the surviving nodes, and is applicable for arrays with $r = k$. In contrast to their approach, Zigzag codes minimize I/O as well as network costs and are applicable to a wide range of practical system settings. For example, in one of these code families, the data sent over network can be thought of as if it was optimally compressed to achieve the optimal ratio, however the data is not originally stored in this compressed manner. These codes are named minimum bandwidth regenerating (MBR) codes in [24].

RotatedRS [17] and Hitchhiker-XOR [25] reduce recovery I/O for a wider range of parameters. Based on Reed-Solomon, they are both array codes and rely on large elements in order to achieve real I/O savings, though their number of rows is small. Their rebuilding ratio varies according to the system parameters, but it is close to $\frac{1}{r}$ only in very limited settings. A recent study [12] achieves a rebuilding ratio of $\frac{3}{4}$ for unmodified Reed-Solomon codes and any r, but relies on the ability to read the surviving blocks in bit granularity.

A recent work that is closely related to ours implements Butterfly codes [10] in order to evaluate their real recovery cost [20]. Butterfly can be viewed as a modification of Zigzag, which is applicable only for $r = 2$. It requires only XOR operations, but some ($O(k^2)$) elements belong to the dependency sets of more than two parity elements. The authors' evaluation shows that large objects are essential for realizing the benefits of optimal rebuilding ratio, while for small objects the amount of data read and transferred is even greater than for Reed-Solomon. With our optimizations, Zigzag can reduce recovery costs for any r, and for small as well as large objects.

Chapter 8

Conclusions

Optimizing the recovery of a single failure in an erasure coded array has been the focus of numerous studies. However, as the theoretical optimizations continue to improve, their applicability to realistic system settings is limited. In this paper, we showed that by trading the theoretical optimality of recovery in Zigzag codes for I/O sequentiality, we can reduce recovery costs considerably in practical system settings. We do this for both large and small objects, and without increasing the storage overhead, which is the tradeoff adopted by most systems to date. Our evaluation results suggest additional system-level optimizations that are likely to reduce these costs even further, and are part of our future work. They also motivate further research to address the system-level implications with theoretical coding techniques.

Bibliography

[1] M. Blaum et al. "EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures". In: *21st Annual International Symposium on Computer Architecture (ISCA)*. 1994.

[2] E. Brewer et al. *Disks for Data Centers*. Tech. rep. Google, 2016.

[3] B. Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". In: *23rd ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[4] *Ceph Documentation - OSD Configuration Reference*. `http://docs.ceph.com/docs/jewel/rados/configuration/osd-config-ref/`. Accessed: 2017-04-03.

[5] J. Colgrove et al. "Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components". In: *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2015.

[6] P. Corbett et al. "Row-diagonal Parity for Double Disk Failure Correction". In: *3rd USENIX Conference on File and Storage Technologies (FAST)*. 2004.

[7] A. G. Dimakis et al. "Network Coding for Distributed Storage Systems". In: *IEEE Transactions on Information Theory* 56.9 (Sept. 2010), pp. 4539–4551.

[8] X. Ding et al. "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch". In: *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. ATC'07. Santa Clara, CA: USENIX Association, 2007, 20:1–20:14. ISBN: 999-8888-77-6. URL: `http://dl.acm.org/citation.cfm?id=1364385.1364405`.

[9] P. Elias. "Error-free Coding". In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (1954), pp. 29–37.

[10] E. En-Gad et al. "Repair-optimal MDS array codes over GF(2)". In: *IEEE International Symposium on Information Theory (ISIT)*. 2013.

[11] D. Ford et al. "Availability in Globally Distributed Storage Systems". In: *9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2010.

[12] V. Guruswami and M. Wootters. "Repairing Reed-solomon Codes". In: *48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 2016.

[13] J. L. Hafner. "HoVer Erasure Codes For Disk Arrays". In: *International Conference on Dependable Systems and Networks (DSN)*. 2006.

[14] J. L. Hafner. "WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems". In: *4th USENIX Conference on File and Storage Technologies (FAST)*. 2005.

[15] C. Huang, M. Chen, and J. Li. "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems". In: *Trans. Storage* 9.1 (Mar. 2013), 3:1–3:28.

[16] C. Huang et al. "Erasure Coding in Windows Azure Storage". In: *USENIX Annual Technical Conference (ATC)*. 2012.

[17] O. Khan et al. "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads". In: *10th Usenix Conference on File and Storage Technologies (FAST)*. 2012.

[18] J. Li and X. Tang. "Optimal Exact Repair Strategy for the Parity Nodes of the $(k+2, k)$ Zigzag Code". In: *IEEE Transactions on Information Theory* 62.9 (Sept. 2016), pp. 4848–4856.

[19] M. Li and P. C. Lee. "STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures". In: *Trans. Storage* 10.4 (Oct. 2014), 14:1–14:30.

[20] L. Pamies-Juarez et al. "Opening the Chrysalis: On the Real Repair Performance of MSR Codes". In: *14th Usenix Conference on File and Storage Technologies (FAST)*. 2016.

[21] J. S. Plank and M. Blaum. "Sector-Disk (SD) Erasure Codes for Mixed Failure Modes in RAID Systems". In: *Trans. Storage* 10.1 (Jan. 2014), 4:1–4:17. ISSN: 1553-3077.

[22] J. S. Plank, K. M. Greenan, and E. L. Miller. "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions". In: *11th USENIX Conference on File and Storage Technologies (FAST)*. 2013.

[23] J. S. Plank et al. "A Performance Evaluation and Examination of Open-source Erasure Coding Libraries for Storage". In: *7th Usenix Conference on File and Storage Technologies (FAST)*. 2009.

[24] 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". In: *IEEE Transactions on Information Theory* 57.8 (Aug. 2011), pp. 5227–5239.

[25] K. V. Rashmi et al. "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers". In: *ACM SIGCOMM*. 2014.

[26] K. V. Rashmi et al. "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster". In: *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2013.

[27] K. V. Rashmi et al. "Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth". In: *13th USENIX Conference on File and Storage Technologies (FAST)*. 2015.

[28] N. Raviv, N. Silberstein, and T. Etzion. "Constructions of high-rate minimum storage regenerating codes over small fields". In: *IEEE International Symposium on Information Theory (ISIT)*. 2016.

[29] E. Rosenfeld, N. Amit, and D. Tsafrir. "Using Disk Add-Ons to Withstand Simultaneous Disk Failures with Fewer Replicas". In: *7th Annual Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)* (2013).

[30] Maheswaran Sathiamoorthy et al. "XORing elephants: novel erasure codes for big data". In: *39th international conference on Very Large Data Bases (VLDB)*. 2013.

[31] M. Silberstein et al. "Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage". In: *International Conference on Systems and Storage (SYSTOR)*. 2014.

[32] A. Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. 2015.

[33] I. Tamo and A. Barg. "A Family of Optimal Locally Recoverable Codes". In: *IEEE Transactions on Information Theory* 60.8 (Aug. 2014), pp. 4661–4676.

[34] I. Tamo, Z. Wang, and J. Bruck. "Zigzag Codes: MDS Array Codes With Optimal Rebuilding". In: *IEEE Transactions on Information Theory* 59.3 (Mar. 2013), pp. 1597–1616.

[35] Zhiying Wang, Alexandros G Dimakis, and Jehoshua Bruck. "Rebuilding for array codes in distributed storage systems". In: *2010 IEEE Globecom Workshops*. 2010.

[36] S. A. Weil et al. "Ceph: A Scalable, High-performance Distributed File System". In: *7th Symposium on Operating Systems Design and Implementation (OSDI)*. 2006.

[37] S. A. Weil et al. "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data". In: *ACM/IEEE Conference on Supercomputing (SC)*. 2006.

[38] S. A. Weil et al. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters". In: *2nd International Workshop on Petascale Data Storage (PDSW): Held in Conjunction with Supercomputing*. 2007.

[39] M. Xia et al. "A Tale of Two Erasure Codes in HDFS". In: *13th USENIX Conference on File and Storage Technologies (FAST)*. 2015.

[40] M. Ye and A. Barg. "Explicit constructions of MDS array codes and RS codes with optimal repair bandwidth". In: *2016 IEEE International Symposium on Information Theory (ISIT)*. 2016.

מימוש קודי זיגזג במערכת אחסון מבוזרת

מתן לירם

מימוש קודי זיגזג במערכת אחסון מבוזרת

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

מתן לירם

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
אייר ה'תשע"ז חיפה מאי 2017

המחקר נעשה בהנחיית דר' גלה ידגר, פרופ' איתן יעקובי ופרופ' אסף שוסטר בפקולטה למדעי המחשב

ארצה להודות לגלה ידגר על השקעת הזמן העצומה שלה למען המחקר שלי, על מנת לעזור לי להתגבר על מכשולים, לכוון אותי לקבלת ההחלטות הנכונות ברגעי התלבטות ולעזור לי בהכנת תוכניות ניהול זמנים מפורטות וריאליות לאורך הדרך. כל הגשה והצגה שביצעתי, ואפילו הגשת התזה עצמה, קרו בזמן ובאיכות המירבית שיכלתי לספק, תודות לנסיון המחקר הרב וטכניקות התכנון המעולות שלה. בנוסף, ארצה להודות ליצחק תמו ואיתן יעקובי, על העזרה הרבה בהבנת החלק התאורטי של התזה. ללא עזרתם, לא היה ביכולתי לנתח את התוצאות בצורה כה מדויקת, ולדבר באופן כה שוטף ב'שפת קודי זיגזג'. ארצה להודות גם לעידו חכימי על העזרה הרבה במימוש גרסה מדהימה באיכותה של מקודד ומפענח קודי זיגזג, שחסכה לי שעות רבות שיכלו להתבזבז על מציאת באגים אילולא עזרה זו. בנוסף, ארצה להודות לאסף שוסטר על פניני החוכמה שסיפק לנו לגבי כיוונים למחקר, ועל כך שאפשר לנו לשאול את שאלות המחקר הנכונות ובאופן הטוב ביותר. מירב שנות הנסיון שלו במחקר אפשרו לו לתת לי תשובות מהירות לשאלות, שלאחר זמן רב התבררו כמדויקות להפליא.

ארצה להודות להורים שלי על תמיכתם הרבה בי לאורך תקופת המחקר, ועל כך שנתנו לי דחיפות מנטליות לאורך הדרך על מנת להמשיך ולחקור, ולא להתייאש גם בזמנים בהם המחקר לא התקדם במהירות הרצויה. הרצון הרב שלי להתקדם במחקר ולעמוד בלוחות הזמנים נבע רבות מהעידוד והתמיכה הרבה שלהם. ארצה להודות גם לשותפי למשרד לב יוחננוב, על הנעמת זמני במשרד, ועל סקרנותו הרבה בבעיות תאורטיות בהן נתקלתי לאורך המחקר. הסקרנות הרבה שלו והרעיונות היצירתיים שהעלה עזרו לי לפתור בעיות קשות ונתנו לי תנופה להמשך המחקר.

ככל הנראה, לא הייתי מצליח לסיים את המחקר ללא העזרה שהוזכרה לעיל. בשלבים מסוימים, היה נדמה שאני ממלא את תפקידי הקטן במפעל מחקר גדול ומשומן. אני מרגיש בר־מזל לבלות את השנתיים האחרונות עם קבוצה כה מקצועית, איכותית ומהנה.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תקציר

במרכזי אחסון נתונים מודרניים הכוללים מאות עד עשרות אלפי שרתים, נפילות שרתים הן עניין שבשגרה, ונפילות התלויות זו בזו הן שכיחות. כתוצאה מכך, מערכות אחסון גדולות מאורגנות עם דרגות גבוהות של יתירות: הן מאחסנות שלושה עותקים של כל אובייקט נתונים, או מוסיפות עד ארבעה שרתי יתירות בכל מערך שרתים. נפילות השרתים התכופות גוררות תהליך שחזור שיוצר עומס רב במערכת, החל מגישות עודפות לדיסק ותעבורה עודפת ברשת, וכלה בשעות פעילות של שרתים על עלויות צריכת האנרגיה והקירור הנלוות אליהן.

מחקר עדכני במרכזי האחסון של פייסבוק הראה שבמערכים המוגנים על ידי קודים לתיקון שגיאות, פעולות שחזור גוררות תעבורה עודפת בסדר גודל של 180TB. בעקבות זאת, מאמצי מחקר רבים מושקעים בניסיון להוריד את עלויות השחזור במערכות אלו. המחקר האמור הראה גם ש־98 אחוזים מן הנפילות כללו נפילה של שרת בודד. אכן, למרות שמערכות המוגנות על ידי קודים לתיקון שגיאות בנויות לשמור על המידע גם במקרה של מספר נפילות בו זמנית, מאמצים רבים מושקעים בשיפור הביצועים של שחזור במקרה של נפילה בודדת.

קודים לשחזור מקומי (LRC) מאפשרים שחזור בעזרת תת קבוצה קטנה של שרתים תקינים, אך מאופיינים בתקורה גבוהה של נפח אחסון. קודים אחרים מקטינים את כמות התעבורה הדרושה לשחזור, אך עדיין דורשים קריאה של רב הנתונים התקינים מן הדיסק הקשיח. מספר גישות נוספות מציעות להקטין את כמות הנתונים שיש לקרוא מכל שרת כאשר נעשה שימוש בקודים קיימים או בווריאציות עליהם. בשיטות אלו הגישות לדיסק מתבצעות בתבנית לא סדירה, ולכן מתאימות רק למערכות שבהן מאוחסנים אובייקטים גדולים במיוחד.

גישה מבטיחה מבוססת על קודים שממזערים את יחס השחזור, שהוא אחוז הנתונים שיש לקרוא מתוך כל הנתונים המאוחסנים בשרתים התקינים. קודים אלו משיגים את החסם התיאורטי התחתון על היחס הזה באמצעות בניות מפורשות. קודים אלו מאופיינים בפיצול פנימי רב: אובייקטים מקודדים מורכבים ממאות "יסודות". יחס השחזור הנמוך מתקבל באמצעות קריאה של תתי קבוצות של היסודות המאוחסנים בכל שרת תקין, אך היסודות בהכרח מאוחסנים בצורה לא סדרתית על גבי התקן האחסון.

דיסקים קשיחים מהווים את טכנולוגיית האחסון השלטת ברב מרכזי אחסון הנתונים, בעיקר בחלקים המוקצים לנתונים שניגשים אליהם בתדירות נמוכה ("קרים") ושמשתמשים בקודים לתיקון שגיאות כדי להבטיח את השרידות והזמינות שלהם. גישות לא סדרתיות כמו אלו המתוארות לעיל עלולות להיות הרסניות בהקשר של ביצועי דיסקים קשיחים. אפילו תבנית של "דילוגים", שבה נקראים רצפים של נתונים במקטעים ובמרווחים קבועים מורידה משמעותית את תפוקת המערכת. מחקר עדכני הדגים שהעלות של הגישות הלא סדרתיות עלול לבטל את הרווח מהקטנת יחס השחזור, ושבמקרים מסויימים הן עלולות להגדיל את כמות הנתונים שנקראת בפועל, כמו גם את הזמן הנדרש לשחזור.

במסגרת מחקרנו זה אנו מראים כיצד ניתן להקל על המתח בין התאוריה למעשה, ולהשיג יחס שחזור כמעט מיטבי במערכות אחסון אמיתיות. אנו עושים שימוש בקודי זיג־זג, המאופיינים ביחס שחזור ותקורת אחסון מיטביים, לצד גמישות מובנית המאפשרת לאזן בין יחס השחזור לבין סדרתיות הגישות. אנו מציגים סדרת שיפורים המאופשרים על ידי אותה גמישות, ומראים כיצד הגדלה מועטה ביחס השחזור התיאורטי מאפשרת הקטנה משמעותית של עלויות השחזור במערכת אמיתית. חלק מן השיפורים האמורים ניתנים להפעלה גם על קודים אחרים.

החדשנות של השיפורים שלנו נובעת מן המטרה שבבסיסם: בעוד שגישות קודמות כוונו להקטנת כמות הנתונים הדרושה לשחזור, השיפורים שלנו מתוכננים להגדיל את הסדרתיות של אותם נתונים, גם במחיר של קריאת נתונים רבים יותר. אנו משתמשים ב"שכפול קוד", טכניקה להגדלת מספר השרתים במערך ללא הגדלת הפיצול הפנימי שבו. אנו משלבים שכפול עם "שרתים וירטואליים": שרתים שאינם מאחסנים מידע (ועל כן לא קיימים במערכת) אך מגדילים את מרחב הפרמטרים הניתנים למימוש ומאפשרים למתכנני מערכות לבחור את הבניה המתאימה ביותר לצרכיהם. כמו כן, אנו מנצלים את התכנון הכללי של קודי זיג־זג על מנת לבחור את הבניות וסכמות השחזור עם גישות השחזור הסדרתיות ביותר. בנוסף, אנו מיישרים את יסודות האובייקטים לגבולות גזרה של 4KB על ידי ריפוד "לוגי" של סוף האובייקט באפסים שלא נשמרים בפועל על גבי הדיסק הקשיח. לבסוף, יסודות סמוכים נקראים מן הדיסק בבקשה בודדת על מנת להקטין את התקורה של חילופי ההקשר בשרת.

מימשנו את קוד זיג־זג יחד עם השיפורים שלנו ב־Ceph - מערכת אחסון מבוזרת בקוד פתוח - והשווינו אותם לקודי Reed־Solomon, שהם ברירת המחדל עבור קוד לתיקון שגיאות במערכת. הניסויים שלנו מראים שעל ידי הקטנת הפיצול הפנימי, קודי זיג־זג יכולים להקטין משמעותית את כמות הנתונים הנקראת מן הדיסק ונשלחת ברשת בזמן שחזור (ב־19 אחוזים עד 39 אחוזים), וכן את משך השחזור (עד 28 אחוזים). בניגוד לגישות קודמות, בעזרת קוד זיג־זג ניתן להשיג שיפור דומה עבור כל גדלי המערכים ומידות היתירות הנפוצים, וכן עבור אובייקטים קטנים.