Enabling High-Performance Erasure-Coded Storage Systems via Workload-Aware Designs

CHAN, Chun Wing

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
June 2014

Abstract of thesis entitled:

Enabling High-Performance Erasure-Coded Storage Systems via Workload-Aware Designs

Submitted by CHAN, Chun Wing

for the degree of Master of Philosophy

at The Chinese University of Hong Kong in June 2014

Many modern storage systems adopt erasure coding to provide data availability guarantees with low redundancy. Log-based storage is often used to append new data rather than overwrite existing data to achieve high update efficiency, but introduces significant I/O overhead during recovery due to reassembling updates from data and parity chunks. We propose parity logging with reserved space, which comprises two key design features: (1) it takes a hybrid of in-place data updates and log-based parity updates to balance the costs of updates and recovery, and (2) it keeps parity updates in a reserved space next to the parity chunk to mitigate disk seeks. We further propose a workload-aware scheme to dynamically predict and adjust the reserved space size. We prototype an erasure-coded clustered storage system called CodFS, and conduct experiments on different update schemes under synthetic and real-world workloads. We show that our proposed update scheme achieves high update and recovery performance, which cannot be simultaneously achieved by pure in-place or log-based update schemes.

In addition to clustered storage, we also study the issues introduced by small random writes to SSD RAID. In the second part of this thesis, we propose TWEEN, a middleware application that aims toward write efficiency and endurance of SSD RAID. It comprises two key design features: (1) it combines the log-structured file system (LFS) and byte-addressable non-volatile RAM (NVRAM) to eliminate partial writes at both SSD and RAID levels, and (2) it mitigates the garbage collection overhead in LFS by grouping writes with similar update frequencies and absorbing LFS garbage collection writes in NVRAM.

香港中文大學
計算機科學及工程學系哲學碩士
陳雋永

很多現代存儲系統利用糾刪碼儲存少量冗餘數據，來提高系統的可用性。日誌結構檔案系統把數據更新順序地寫入硬盤，來提高更新效率。可是，當系統崩潰發生時，這種設計必須讀取分佈在不同地方的更新以重建數據塊和校驗塊，造成額外的讀寫成本。我們提出的 parity logging with reserved space 有兩項主要設計特色： (1) 結合 in-place 數據更新和 log-based 校驗更新，平衡更新和回復的效能。(2) 把校驗更新放在校驗塊旁的一個備用空間中，減少硬盤尋道的頻率。再者，我們提出一個按照系統負載預測和調整備用空間的大小的方案，並應用在我們開發的一個糾刪碼分佈式存儲系統 CodFS 上。利用 CodFS，我們測試了不同更新方案在各種真實負載下的表現，發現普通的 in-place 和 log-based 更新方案並不能像我們的方案般同時在更新和回復方面達到良好的性能。

除了分佈式存儲系統，隨機寫入也會對 SSD RAID 構成問題。在本論文的下半部分，我們提出了一個以提高寫入性能和 SSD 壽命爲目標的中間件 TWEEN。TWEEN 有兩項主要的設計特色： (1) 結合 log-structured 文件系統和非揮發性記憶體 (NVRAM)，避免在 SSD 層面和 RAID 層面進行局部寫入。(2) 組合更新頻率類近的寫入和利用 NVRAM 吸收垃圾回收引致的寫入，減少 LFS 層面中的垃圾回收開銷。

Acknowledgement

It is with immense gratitude that I acknowledge the consistent support and help of my supervisor, Prof. Patrick P. C. Lee. Without his mentoring, this thesis would not have been possible.

I would also like to thank my thesis committee members Prof. Irwin King, Prof. John C. S. Lui and Prof. Guangyan Zhang, who provided me with useful advice.

I wish to thank my final-year project supervisor, Dr. Tsz-Yeung Wong for introducing me to system research and his encouragement throughout my undergraduate and postgraduate studies.

I consider it an honor to work with Qian Ding and Hoi-Wan Chan, who collaborated seamlessly with me on both CodFS and TWEEN. I also share the credit of my work with Yan-Kit Li and Chuan Qin, who contributed to the early CodFS prototype.

I am indebted to many friends and colleagues, including Dun Ho, Ka-Chun Lam, Mingqiang Li, Jian Lin, Chun-Ho Ng, Chung-Pan Tang, Wai-Chung Tang, Kai-Yuan Tso, Min Xu and many others for their support and invaluable advice, and all the fun discussions during lunch and throughout the day.

Finally, I would like to take this opportunity to thank my family for all the love and support all these years.

List of Publications

Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage

Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, Helen H. W. Chan

Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14), Santa Calra, CA, February 2014

(Acceptance Rate: 24/133 = 18.0%)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we give an overview of our project and highlight our contributions.

1.1 Overview

1.1.1 CodFS

Clustered storage systems are known to be susceptible to component failures [20]. High data availability can be achieved by encoding data with redundancy using either replication or erasure coding. Erasure coding encodes original data chunks to generate new parity chunks, such that a subset of data and parity chunks can sufficiently recover all original data chunks. It is known that erasure coding introduces less overhead in storage and write bandwidth than replication under the same fault tolerance [56, 71]. For example, traditional 3-way replication used in GFS [20] and Azure [9] introduces 200% of redundancy overhead, while erasure coding can reduce the overhead to 33% and achieve higher availability [28]. Today's enterprise clustered storage systems [17, 28, 54, 58, 74] adopt erasure coding in production to reduce hardware footprints and maintenance costs.

For many real-world workloads in enterprise servers and network file systems [2, 46], data updates are dominant. There are two ways of performing updates: (1) *in-*

place updates, where the stored data is read, modified, and written with the new data, and (2) *log-based updates*, where updates are inserted to the end of an append-only log [57]. If updates are frequent, in-place updates introduce significant I/O overhead in erasure-coded storage since parity chunks also need to be updated to be consistent with the data changes. Existing clustered storage systems, such as GFS [20] and Azure [9] adopt log-based updates to reduce I/Os by sequentially appending updates. On the other hand, log-based updates introduce additional disk seeks to the update log during sequential reads. This in particular hurts recovery performance, since recovery makes large sequential reads to the data and parity chunks in the surviving nodes in order to reconstruct the lost data.

This raises an issue of choosing the appropriate update scheme for an erasure-coded clustered storage system to achieve efficient updates and recovery simultaneously. Our primary goal is to mitigate the network transfer and disk I/O overheads, both of which are potential bottlenecks in clustered storage systems. We propose and build an erasure-coded clustered storage system *CodFS*, which supports efficient updates and recovery.

CodFS is a joint work with Qian Ding, Helen H. W. Chan and Patrick P. C. Lee. Yan-kit Li and Chuan Qin also helped in the implementation of the early prototype.

1.1.2 TWEEN

To enhance the reliability of SSD systems beyond ECCs at the flash level [22, 23, 43], one option is to adopt parity-based RAID (*Redundant Array of Inexpensive Disks*) [48] (e.g., RAID-4, RAID-5, RAID-6) at the device level.

Nevertheless, when deploying parity-based SSD RAID (i.e., applying parity-based RAID to multiple SSDs), we must pay special attention to small random writes. It is known that small random writes incur high garbage collection overhead in SSDs and degrade the performance and endurance of SSDs [11, 34, 44]. Also, to maintain consistency between data and parity, each data write triggers parity updates, which not only incur extra I/Os that degrade write performance, but also further aggravate garbage

collection overhead in SSDs. The primary problem of small random writes is that they do not align the granularities of SSD garbage collection and RAID encoding operations. Such small random writes, which we term *partial writes*, incur high garbage collection overhead at the SSD level and high parity update overhead at the RAID level. Thus, for performance and endurance enhancements of SSD RAID, eliminating partial writes is a critical requirement.

To eliminate partial writes, we propose TWEEN, which batches updates in a log-structured file system (LFS) [57] until the aggregated write size aligns the granularities at both SSD and RAID levels. Our observation is that byte-addressable non-volatile RAM (NVRAM) is an emerging storage substrate [31, 36, 52, 69, 81], and can serve as non-volatile (persistent) cache for batching partial writes. This motivates us to realize a practical and deployable design that combines LFS and NVRAM to eliminate partial writes and hence enhance the performance and endurance of SSD RAID.

TWEEN is a joint work with Qian Ding, Helen H. W. Chan and Patrick P. C. Lee.

1.2 Contributions

This thesis mainly focuses on the CodFS project. We highlight the following contributions.

First, we provide a taxonomy of existing update schemes for erasure-coded clustered storage systems. To this end, we propose a novel update scheme called *parity logging with reserved space*, which uses a hybrid of in-place data updates and log-based parity updates. It mitigates the disk seeks of reading parity chunks by putting deltas of parity chunks in a reserved space that is allocated next to their parity chunks. We further propose a workload-aware reserved space management scheme that effectively predicts the size of reserved space and reclaims the unused reserved space.

Second, we build an erasure-coded clustered storage system CodFS, which targets the common update-dominant workloads and supports efficient updates and recovery. CodFS offloads client-side encoding computations to the storage cluster. Its implemen-

tation is extensible for different erasure coding and update schemes, and is deployable on commodity hardware.

Finally, we conduct testbed experiments using synthetic and real-world traces. We show that our CodFS prototype achieves network-bound read/write performance. Under real-world workloads, our proposed parity logging with reserved space gives a 63.1% speedup of update throughput over pure in-place updates and up to $10\times$ speedup of recovery throughput over pure log-based updates. Also, our workload-aware reserved space management effectively shrinks unused reserved space with limited reclaim overhead.

1.3 Organization

The rest of the thesis proceeds as follows.

- **Chapter 2**

  We introduce the background of erasure coding and describe the existing parity update approaches. We also review related work regarding various aspects of the project.

- **Chapter 3**

  We describe the motivation of our design by analyzing the update behavior of real-world traces.

- **Chapter 4**

  We propose a new parity update approach and address the design and implementation of CodFS.

- **Chapter 5**

  We present testbed experimental results.

- **Chapter 6**

  We introduce the goals and design of TWEEN, which improves write efficiency

and endurance of an SSD RAID using non-volatile cache.

- **Chapter 7**

  We summarize the thesis and describe future work.

□ **End of chapter.**

Chapter 2

Background and Related Work

In this chapter, we present essential background knowledge of erasure coding and parity update approaches, which are the two important components of our work. We also review related work on erasure-coded storage, log-structured file systems and parity logging.

2.1 Background

2.1.1 Erasure Coding

It has been known that failures in storage systems are common [20]. Components in a storage system can fail in numerous ways, including sector errors, bit flips and physical failures [51]. Erasure coding provides redundancy to prevent data loss in an event of component failure. Here, we provide the background details of the erasure codes considered in this work.

Replication

The simplest form of erasure code is replication, in which we make k copies for each byte in the data and distribute the copies across k devices. As a result, we can tolerate at most $k - 1$ failures as there is at least one surviving copy for each byte of

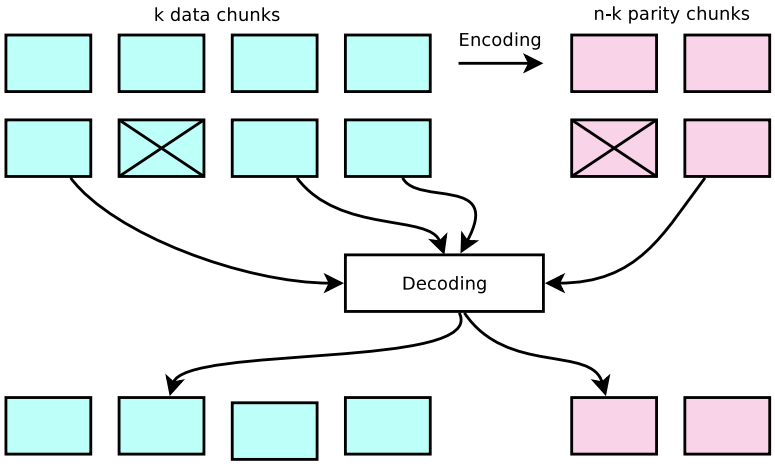


Figure 2.1: Illustration of the acts of encoding and decoding: an (n, k)-code encodes k data chunks into $n - k$ parity chunks. When up to $n - k$ chunks fail, their content can be recovered using the information in the remaining k chunks.

data.

Although replication is very simple, it introduces high storage overhead. For example, GFS [20] by default tolerates up to two simultaneous device failures, which requires a storage cost that is $3\times$ that of the original size of data.

Maximum Distance Separable Codes

Figure 2.1 illustrates the acts of encoding and decoding. We consider an erasure-coded storage cluster with M nodes (or servers). We divide data into *segments* and apply erasure coding independently on a per-segment basis. We denote an (n, k)-code as an erasure coding scheme defined by two parameters n and k, where $k < n$. An (n, k)-code divides a segment into k equal-size uncoded chunks called *data chunks*, and encodes the data chunks to form $n - k$ coded chunks called *parity chunks*. We assume $n < M$, and have the collection of n data and parity chunks distributed across n of the M nodes in the storage cluster. We consider *Maximum Distance Separable (MDS)* erasure codes, i.e., the original segment can be reconstructed from any k of the n data and parity chunks.

An MDS code provides optimal fault tolerance for a given storage overhead. For

example, using a $(8, 6)$-code can reduce the redundancy overhead to 33%, which is significantly less than 200% in three-way replication, without sacrificing the level of fault tolerance.

There are many variants of MDS codes, such as, Reed-Soloman (RS) codes [53], RDP [14], EVENODD [50] and Liberation [7]. These codes mainly differ in the ways of calculating the parity chunks.

Linearity Property of Erasure Codes

Each parity chunk can be in general encoded by computing a linear combination of the data chunks. Mathematically, for an (n, k)-code, let $\{\gamma_{ij}\}_{1\leq i\leq n-k, 1\leq j\leq k}$ be a set of encoding coefficients for encoding the k data chunks $\{D_1, D_2, \cdots, D_k\}$ into $n-k$ parity chunks $\{P_1, P_2, \cdots, P_{n-k}\}$. Then, each parity chunk P_i ($1 \leq i \leq n-k$) can be computed by: $P_i = \sum_{j=1}^{k} \gamma_{ij} D_j$, where all arithmetic operations are performed in the Galois Field over the coding units called *words*.

The linearity property of erasure coding provides an alternative to computing new parity chunks when some data chunks are updated. Suppose that a data chunk D_l (for some $1 \leq l \leq k$) is updated to another data chunk D'_l. Then each new parity chunk P'_i ($1 \leq i \leq n-k$) can be computed by:

$$P'_i = \sum_{j=1, j\neq l}^{k} \gamma_{ij} D_j + \gamma_{il} D'_l = P_i + \gamma_{il}(D'_l - D_l).$$

Thus, instead of summing over all data chunks, we compute new parity chunks based on the change of data chunks. The above computation can be further generalized when only part of a data chunk is updated, but a subtlety is that a data update may affect different parts of a parity chunk depending on the erasure code construction (see [51] for details). Suppose now that a word of D_l at offset o is updated, and the word of P_l at offset $\hat{o}$ needs to be updated accordingly (where o and $\hat{o}$ may differ). Then we can express:

$$P_i'(\hat{o}) \quad = \quad P_i(\hat{o}) + \gamma_{il}(D_l'(o) - D_l(o)),$$

where $P_i'(\hat{o})$ and $P_i(\hat{o})$ denote the words at offset $\hat{o}$ of the new parity chunk P_i' and old parity chunk P_i, respectively, and $D_l'(o)$ and $D_l(o)$ denote the words at offset o of the new data chunk D_l' and old data chunk D_l, respectively. In the following discussion, we leverage this linearity property in parity updates.

2.1.2 Parity Update Approaches

Handling updates in a replicated storage is relatively straight-forward, i.e., we propagate each update to all replicas. In addition, techniques including chain-replication [68] can be used to reduce the update overhead. On the other hand, updates in an erasure-coded storage is complicated by the need to maintain parity consistency, which often introduces network and disk I/O overhead. In this section, we re-examine existing parity update schemes that fall into two classes: the RAID-based approaches and the delta-based approaches.

RAID-based Approaches

We describe three classical approaches of parity updates that are typically found in RAID systems [12, 66].

Full-segment writes. A full-segment write (or full-stripe write) updates all data and parity chunks in a segment. It is used in a large sequential write where the write size is a multiple of segment size. To make a full-segment write work for small updates, one way is to pack several updates into a large piece until a full segment can be written in a single operation [42]. Full-segment writes do not need to read the old data or parity chunks, and hence achieve the best update performance.

Reconstruct writes. A reconstruct write first reads all the chunks from the segment that are not involved in the update. Then it computes the new parity chunks using the read chunks and the new chunks to be written, and writes all data and parity chunks.

Read-modify writes. A read-modify write leverages the linearity of erasure coding for parity updates (see Section 2.1.1). It first reads the old data chunk to be updated and all the old parity chunks in the segment. It then computes the change between the old and new data chunks, and applies the change to each of the parity chunks. Finally, it writes the new data chunk and all new parity chunks to their respective locations.

Discussion. Full-segment writes can be implemented through a log-based design to support small updates, but logging has two limitations. First, we need an efficient garbage collection mechanism to reclaim space by removing stale chunks, and this often hinders update performance [61]. Second, logging introduces additional disk seeks to retrieve the updates, which often degrades sequential read and recovery performance [41]. On the other hand, both reconstruct writes and read-modify writes are traditionally designed for a single host deployment. Although some recent studies implement read-modify writes in a distributed setting [18, 78], both approaches introduce significant network traffic since each update must transfer data or parity chunks between nodes for parity updates.

Delta-based Approaches

Another class of parity updates, called the *delta-based approaches*, eliminates redundant network traffic by only transferring a *parity delta* which is of the same size as the modified data range [10, 65]. A delta-based approach leverages the linearity of erasure coding described in Section 2.1.1. It first reads the range of the data chunk to be modified and computes the delta, which is the change between old and new data at the modified range of the data chunk, for each parity chunk. It then sends the modified data range and the parity deltas computed to the data node and all other parity nodes

for updates, respectively. Instead of transferring the entire data and parity chunks as in read-modify writes, transferring the modified data range and parity deltas reduces the network traffic and is suitable for clustered storage. In the following, we describe some delta-based approaches proposed in the literature.

Full-overwrite (FO). Full-overwrite [4] applies in-place updates to both data and parity chunks. It merges the old data and parity chunks directly at specific offsets with the modified data range and parity deltas, respectively. Note that merging each parity delta requires an additional disk read of old parity chunk at the specific offset to compute the new parity content to be written.

Full-logging (FL). Full-logging saves the disk read overhead of parity chunks by appending all data and parity updates. That is, after the modified data range and parity deltas are respectively sent to the corresponding data and parity nodes, the storage nodes create logs to store the updates. The logs will be merged with the original chunks when the chunks are read subsequently. FL is used in enterprise clustered storage systems such as GFS [20] and Azure [9].

Parity-logging (PL). Parity-logging [30, 64] can be regarded as a hybrid of FO and FL. It saves the disk read overhead of parity chunks and additionally avoids merging overhead on data chunks introduced in FL. Since data chunks are more likely to be read than parity chunks, merging logs in data chunks can significantly degrade read performance. Hence, in PL, the original data chunk is overwritten in-place with the modified data range, while the parity deltas are logged at the parity nodes.

Discussion. Although the delta-based approaches reduce network traffic, they are not explicitly designed to reduce disk I/O. Both FL and PL introduce disk fragmentation and require efficient garbage collection. The fragmentations often hamper further accesses of those chunks with logs. Meanwhile, FO introduces additional disk reads for the old parity chunks on the update path, compared with FL and PL. Hence, to take a

step further, we want to address the question: *Can we reduce the disk I/O on both the update path and further accesses?* We describe our novel parity update approach in Section 4.1.

2.2 Related Work

In this section, we review related work on erasure coding, erasure-coded storage, log-structured file systems and parity logging.

2.2.1 Erasure-coded Storage

Traditionally, clustered storage systems adopt replication to provide data redundancy. GFS [20] first proposes to store three replicas for data, which is adopted by subsequent designs including Ceph [73] and Hadoop [63].

Quantitative analysis shows that erasure coding consumes less bandwidth and storage than replication with similar system durability [56, 71]. Under the same storage overhead, it is shown that the *mean time to failure* (MTTF) for storage systems utilizing erasure coding can be orders of magnitude higher than those using replication [71].

Therefore, several studies adopt erasure coding in clustered storage systems. OceanStore [35,55] combines replication and erasure coding for wide-area network storage. It uses a Byzantine-fault tolerant model to provide strong consistency among replicas.

TotalRecall [6] applies replication or erasure coding to different files dynamically according to the availability level predicted by the system. In addition to the level of redundancy, TotalRecall switches between (1) *eager repair*, in which the level of redundancy is maintained by immediate repair action after a failure, and (2) *lazy repair*, in which the additional redundancy is temporarily used to delay repair action.

Ursa Minor [1] focuses on cluster storage and encodes files of heterogeneous types based on the failure models and access patterns. It adapts to changing workload pattern by supporting online re-encoding of data. In addition, it supports versioning for its objects.

Panasas [74] performs client-side encoding on a per-file basis. It is shown by extensive experiments that by moving the computation to clients, the parity computation power of the system scales up as the number of clients increases. In addition, this design can be extended with *end-to-end integrity* by computing and verifying checksum at both ends of the write paths.

Unlike Panasas, some designs such as TickerTAIP [10], PARAID [72] and Pergamum [65] offload the parity computation to the storage array. TickerTAIP utilizes a group of *intelligent* worker nodes, which are nodes with coding ability, connected in a small-area network. This is similar to the concept of *object storage devices* (OSDs) in modern context. PARAID is designed for a single-host RAID. It achieves energy-savings using a skewed stripping pattern and spinning down unused disks during light loads. Pergamum also focuses on energy optimization. It adds non-volatile memory (NVRAM) to each node in order to defer metadata updates and small writes to powered off disks.

R-Admad [37] enhances the reliability of a deduplication system by adopting erasure coding to store deduplicated data. It speeds up recovery by distributing the repair workload among the storage nodes.

Zhang *et al.* [80] modify HDFS with inline erasure coding and evaluate different workloads atop erasure-coded data.

CAROM [39] uses replication for caching in its erasure coded distributed storage system so as to benefit high throughput from replicated cache and low storage overhead from erasure-coded data. It redirects all writes to a primary replica to ensure cache consistency.

Azure [28] and Facebook [58] propose efficient erasure coding schemes to speed up degraded reads.

We complement the above studies by improving update efficiency and recovery performance in erasure-coded clustered storage.

2.2.2 Log-structured File System

Log-structured File System (LFS) [57] first proposes to append updates sequentially to disk to improve write performance. LFS buffers all updates and metadata in an in-memory segment, and flushes the segment to disk using a large sequential write.

Zebra [25] extends LFS for RAID-like distributed storage systems by striping logs across servers. Zebra maintains an append-only log for each client and mitigates parity update overhead by computing parity for the log instead of the files.

AutoRAID [75] uses a workload-aware approach to store active data using replication (RAID-1) and inactive data using erasure coding (RAID-5). In the RAID-5 layer, it batches updates in a log to mitigate read-modify write overhead in RAID.

Several studies focus on designing more efficient garbage collection algorithms for LFS. Self-tuning LFS [41] exploits workload characteristics to improve I/O performance. It speeds up garbage collection in LFS by dynamically choosing between two cleaning algorithms, (1) traditional cleaning, which combines live data from several partially empty segments to form a full segment and (2) hole-plugging, which copies live data to holes found in other segments. BSD-LFS [60, 61] simplifies LFS by moving the cleaner from kernel space to user space. It also allows multiple cleaners with different cleaning policies to execute in parallel.

Clustered storage systems, GFS [20] and Azure [9], also adopt the LFS design for the write-once read-many workload. The more recent work Gecko [62] uses a chained-log design to reduce disk I/O contention of LFS in RAID storage.

The LFS design is also adopted in other types of emerging storage media, such as SSDs [3] and DRAM storage [47]. RAMCloud [47] keeps all data permanently on DRAM and exploits scalability in recovery by distributing data randomly over a large number of nodes.

CodFS handles updates differently from LFS, in which it performs in-place updates to data and log-based updates to parity chunks. It also allocates reserve space for parity logging to further mitigate disk seeks (see Chapter 4). On the other hand, TWEEN

exploits the LFS design to improve write performance and endurance of SSD RAID (see Chapter 6).

2.2.3 Parity Logging

Parity logging [13,64] has been proposed to mitigate the disk seek overhead in parity updates caused by small writes. The main idea is to delay the read of the old parity and the write of the new parity by storing the difference between the old and new parity (or parity updates) temporarily in a log. It appends parity updates for each parity region in a log and flushes updates to the parity region when the log is full. The parity and log regions can be distributed across all disks [64]. On the other hand, CodFS reserves log space next to each parity chunk so as to reduce disk seeks due to frequent small writes. It extends the prior parity logging approaches by allowing future shrinking of the reserved space based on the workload.

□ **End of chapter.**

Chapter 3

Motivation

Our work is based on the assumption that updates are common in real-world workload. We back our claim by studying two sets of real-world storage traces collected from large-scale storage server environments and characterizing their update patterns. Motivated by the fact that enterprises are considering erasure coding as an alternative to RAID for fault-tolerant storage [59], we choose these traces to represent the workloads of enterprise storage clusters. We want to answer three questions: (1) *What is the average size of each update?* (2) *How common do data updates happen?* (3) *Are updates focused on some particular chunks?*

3.1 Trace Description

3.1.1 MSR Cambridge Traces

We use the public block-level I/O traces of a storage cluster released by Microsoft Research Cambridge [46]. The traces are captured on 36 volumes of 179 disks located in 13 servers. They are composed of I/O requests, each specifying the timestamp, the server name, the disk number, the read/write type, the starting logical block address, the number of bytes transferred, and the response time. The whole traces span a one-week period starting from 5PM GMT on 22nd February 2007, and account for the workloads

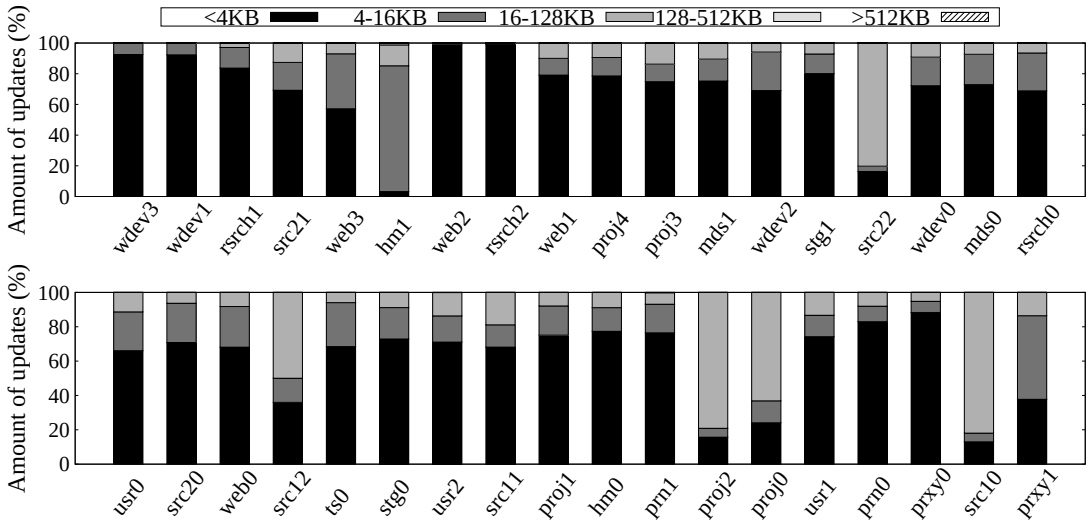


Figure 3.1: Distribution of update size in MSR Cambridge traces.

in various kinds of deployment including user home directories, project directories, source control, and media. Each of the chosen volumes contains 800,000 to 4,000,000 write requests.

3.1.2 Harvard NFS Traces

We also use a set of NFS traces (DEAS03) released by Harvard [16]. The traces capture NFS requests and responses of a NetApp file server that contains a mix of workloads including email, research, and development. The whole traces cover a 41-day period from 29th January 2003 to 10th March 2003. Each NFS request in the traces contains the timestamp, source and destination IP addresses, and the RPC function. Depending on the RPC function, the request may contain optional fields such as file handler, file offset and length. While the traces describe the workloads of a single NFS server, they have also been used in trace-driven analysis for clustered storage systems [1, 26].

No. of Writes	172702071
WSS (GB)	174.73
Updated WSS (%)	68.39
Update Writes (%)	91.56
No. of Accessed Files	2039724
Updated Files (%)	12.10
Avg. Update Size Per Request (KB)	10.58

Table 3.1: Properties of Harvard DEAS03 NFS traces.

3.2 Update Behavior in Real-world Workload

3.2.1 Updates Are Small

We study the update size, i.e., the number of bytes accessed by each update. Figure 3.1 shows the average update size ranges of the MSR Cambridge traces. We see that the updates are generally small in size. Although different traces show different update size compositions, almost all updates occurring in the traces are smaller than 512KB. Among the 36 traces, 28 have more than 60% of updates smaller than 4KB. Similarly, the Harvard NFS traces comprise small updates, with average size of only 10.58KB, as shown in Table 3.1.

3.2.2 Updates Are Common

Unsurprisingly, updates are common in both storage traces. We analyze the write requests in the traces and classify them into two types: *first-write*, i.e., the address is first accessed, and *update*, i.e., the address is re-accessed. Table 3.1 shows the results of the Harvard NFS traces. Among nearly 173 million write requests, more than 91% of them are updates. Table 3.2 shows the results of the MSR Cambridge traces. All the volumes show more than 60% of updates among all write requests, except for two Web/SQL volumes (web2 and web3), a research volume (rsrch1) and a project volume (proj3). We see limited relationship between the working set size (WSS) and the intensity of writes. For example, the volumes with tiny WSS, such as those from firewall/web proxy machines, can have much more writes than those with large WSS,

such as the project volumes.

3.2.3 Update Coverage Varies

Although data updates are common in all traces, the coverage of updates varies. We measure the update coverage by studying the fraction of WSS that is updated at least once throughout the trace period. For example, from the MSR Cambridge traces in Table 3.2, the `src22` trace shows a 99.57% of updated WSS, while updates in the `mds0` trace only cover 29.27% of WSS. In other words, updates in the `src22` trace span across a large number of locations in the working set, while updates in the `mds0` trace are focused on a smaller set of locations. The variation in update coverage implies the need of a dynamic mechanism to improve update efficiency.

3.3 Summary

The above analysis shows that there are workloads in which the majority of writes are small updates, which motivates us to build a clustered storage that provides update efficiency. As we would like to maintain the storage efficiency provided by erasure-coding, we need to devise an update mechanism that mitigates parity maintenance overhead.

Also, the variation in update coverage implies the need of a dynamic mechanism to improve update efficiency as caching alone may not be sufficient to achieve superior update performance.

□ **End of chapter.**

	Volume	Workload Type	No. of Writes	WSS (GB)	Updated WSS (%)	Update Writes (%)
	wdev3	Test web server	671	<0.01	3.33	82.74
	wdev1	Test web server	1055	<0.01	3.52	72.71
	rsrch1	Research projects	13738	0.11	10.42	32.27
	src21	Source control	14104	19.2	4.8	67.44
	web3	Web/SQL server	21330	0.59	15.63	1.9
	hm1	HW monitoring	28415	0.2	52.58	85.61
	web2	Web/SQL server	38963	65.96	0.7	10.1
	rsrch2	Research projects	71223	0.59	74.73	74.27
	web1	Web/SQL server	73833	3.71	3.9	86.54
	proj4	Project directory	95865	123.34	2.06	82.8
	proj3	Project directory	116341	5.83	13.12	30.76
	mds1	Media server	116676	84.23	2.38	63.07
	wdev2	Test web server	181077	0.08	99.64	94.4
	stg1	Web staging	796452	79.69	3.43	93.42
*	src22	Source control	805955	20.17	99.57	99.68
	wdev0	Test web server	913732	0.52	63.74	95.33
*	mds0	Media server	1067061	3.09	29.27	95.77
*	rsrch0	Research projects	1300030	0.36	69.53	97.41
*	usr0	Home directory	1333406	2.44	42.54	96.08
	src20	Source control	1381085	0.71	62.2	95.35
*	web0	Web/SQL server	1423458	7.26	37.25	96.23
	src12	Source control	1423694	1.97	47.84	98.59
*	ts0	Terminal server	1485042	0.91	49.84	95.65
*	stg0	Web staging	1722478	6.31	21.04	97.82
	usr2	Home directory	1994612	378.59	9.26	75.88
	src11	Source control	2170271	119.61	23.22	88.08
	proj1	Project directory	2496935	693.88	6.41	72.38
*	hm0	HW monitoring	2575568	2.31	73.16	93.21
*	prn1	Print server	2769610	80.9	18.55	73.43
	proj2	Project directory	3624878	409.54	55.3	96.66
*	proj0	Project directory	3697143	3.16	56.67	98.89
	usr1	Home directory	3857714	655.52	15.07	67.41
	prn0	Print server	4983406	14.8	55.49	73.21
	prxy0	Web proxy	12135444	0.88	81.35	98.74
	src10	Source control	16302998	120.46	96.65	99.86
	prxy1	Web proxy	58224504	13.68	36.03	98.25

* selected for evaluation

Table 3.2: Properties of MSR Cambridge traces: (1) number of writes shows the total number of write requests; (2) working set size refers to the size of unique data accessed in the trace; (3) percentage of updated working set size refers to the fraction of data in the working set that is updated at least once; and (4) percentage of update writes refers to the fraction of writes that update existing data.

Chapter 4

CodFS Design

In this chapter, we propose a new parity update approach which reduces the disk seek overhead of PL presented in Section 2.1.2. We then move to discuss the design and implementation of CodFS, which is an erasure-coded clustered storage that supports efficient update and recovery.

4.1 Parity Update Approach

Data updates in erasure-coded clustered storage systems introduce performance overhead, since they also need to update parity chunks for consistency. We consider a deployment environment where network transfer and disk I/O are performance bottlenecks. Our goal is to design a parity update scheme that effectively mitigates both network transfer overhead and number of disk seeks.

In this section, we propose a new delta-based approach called **parity-logging with reserved space (PLR)**, which further mitigates fragmentation and reduces the disk seek overhead of PL in storing parity deltas. The main idea is that the storage nodes reserve additional storage space next to each parity chunk for keeping parity deltas. This ensures that each parity chunk and its parity deltas can be sequentially retrieved. While the idea is simple, the challenging issues are to determine (1) the appropriate amount of reserved space to be allocated when a parity chunk is first stored and (2) the

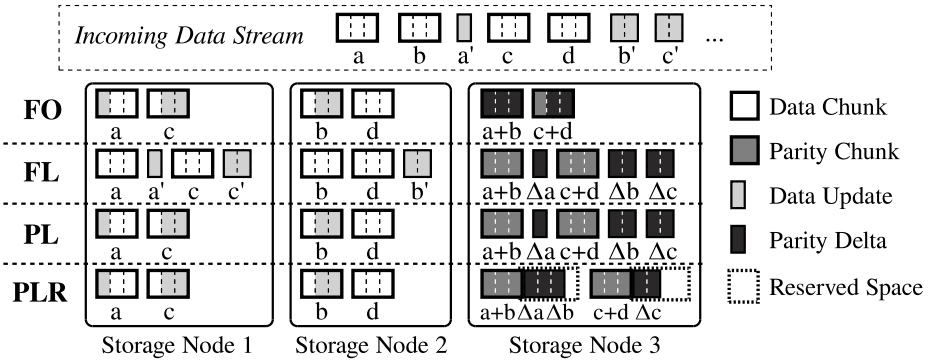


Figure 4.1: Illustration on different parity update schemes.

appropriate time when unused reserved space can be reclaimed to reduce the storage overhead.

4.1.1 An Illustrative Example

Figure 4.1 illustrates the differences of the delta-based approaches in Section 2.1.2 and PLR, using a (3,2)-code as an example. The incoming data stream describes the sequence of operations: (1) write the first segment with data chunks `a` and `b`, (2) update part of `a` with `a'`, (3) write a new segment with data chunks `c` and `d`, and finally (4) update parts of `b` and `c` with `b'` and `c'`, respectively. We see that FO performs overwrites for both data updates and parity deltas; FL appends both data updates and parity deltas according to the incoming order; PL performs overwrites for data updates and appends parity deltas; and PLR appends parity deltas in reserved space.

Consider now that we read the up-to-date chunk `b`. FL incurs a disk seek to the update `b'` when rebuilding chunk `b`, as `b` and `b'` are in discontinuous physical locations on disk. Similarly, PL also incurs a disk seek to the parity delta Δ`b` when reconstructing the parity chunk `a+b`. On the other hand, PLR incurs no disk seek when reading the parity chunk `a+b` since its parity deltas Δ`a` and Δ`b` are all placed in the contiguous reserved space following the parity chunk `a+b`.

4.1.2 Determination of Reserved Space Size

Finding the appropriate reserved space size is challenging. If the space is too large, then it wastes storage space. On the other hand, if the space is too small, then it cannot keep all parity deltas.

A baseline approach is to use a fixed reserved space size for each parity chunk, where the size is assumed to be large enough to fit all parity deltas. Note that this baseline approach can introduce significant storage overhead, since different segments may have different update patterns. For example, from the Harvard NFS traces shown in Table 3.1, although 91.56% of write requests are updates, only around 12% of files are actually involved. This uneven distribution implies that fixing a large, constant size of reserved space can imply unnecessary space wastage.

For some workloads, the baseline approach may reserve insufficient space to hold all deltas for a parity chunk. There are two alternatives to handle extra deltas, either logging them elsewhere like PL, or *merging* existing deltas with the parity chunk to reclaim the reserved space. We adopt the merge alternative since it preserves the property of no fragmentation in PLR.

To this end, we propose a workload-aware reserved space management scheme that dynamically adjusts and predicts the reserved space size. The scheme has three main parts: (1) predicting the reserved space size of each parity chunk using the measured workload pattern for the next time interval, (2) shrinking the reserved space and releasing unused reserved space back to the system, and (3) merging parity deltas in the reserved space to each parity chunk. To avoid introducing small unusable holes of reclaimed space after shrinking, we require that both the reserved space size and the shrinking size be of multiples of the chunk size. This ensures that an entire data or parity chunk can be stored in the reclaimed space.

Algorithm 1 describes the basic framework of our workload-aware reserved space management. Initially, we set a default reserved space size that is sufficiently large to hold all parity deltas. Shrinking and prediction are then executed periodically on

Algorithm 1: Workload-aware Reserved Space Management

```
1 reserved ←DEFAULT_SIZE
2 while true do
3     sleep(period)
4     foreach chunk in parityChunkSet do
5         utility ← getUtility(chunk)
6         size ← computeShrinkSize(utility)
7         doShrink(size, chunk)
8         doMerge(chunk)
```

each storage node. Let $\mathcal{S}$ be the set of parity chunks in a node. For every time interval t and each parity chunk $p \in \mathcal{S}$, let $r_t(p)$ be the reserved space size and $u_t(p)$ be the reserved space utility. Intuitively, $u_t(p)$ represents the fraction of reserved space being used. We measure $u_t(p)$ at the end of each time interval t using exponential weighted moving average in `getUtility`:

$$u_t(p) = \alpha \frac{use(p)}{r_t(p)} + (1 - \alpha) u_{t-1}(p),$$

where $use(p)$ returns the reserved space size being used during the time interval, $r_t(p)$ is the current reserved space size for chunk p, and α is the smoothing factor. According to the utility, we decide the unnecessary space size $c(p)$ that can be reclaimed for the parity chunk p in `computeShrinkSize`. Here, we aggressively shrink all unused space $c(p)$ and round it down to be a multiple of the chunk size:

$$c(p) = \left\lfloor \frac{(1 - u_t(p)) r_t(p)}{ChunkSize} \right\rfloor \times ChunkSize.$$

The `doShrink` function attempts to shrink the size $c(p)$ from the current reserved space $r_t(p)$. Thus, the reserved space $r_{t+1}(p)$ for p at time interval $t + 1$ is:

$$r_{t+1}(p) = r_t(p) - c(p).$$

If a chunk has no more reserved space after shrinking (i.e., $r_{t+1}(p) = 0$), any subsequent

update requests to this chunk are applied in-place as in FO.

Finally, the `doMerge` function merges the deltas in the reserved space to the parity chunk p after shrinking and resets *use(p)* to zero. Hence we free the parity chunk from carrying any deltas to the next time interval, which could further reduce the reserved space size. The merge operations performed here are off the update path and have limited impact on the overall system performance.

The above workload-aware design of reserved space management is simple and can be replaced by a more advanced design. Nevertheless, we find that this simple heuristic works well enough under real-world workloads (see Section 5.3.2).

4.2 Architecture

We design CodFS, an erasure-coded clustered storage system that implements the aforementioned delta-based update schemes to support efficient updates and recovery.

Figure 4.2 shows the CodFS architecture. The *metadata server (MDS)* stores and manages all file metadata, while multiple *object storage devices (OSDs)* perform coding operations and store the data and parity chunks. The MDS also plays a monitor role, such that it keeps track of the health status of the OSDs and triggers recovery when some OSDs fail. A CodFS client can access the storage cluster through a file system interface.

4.3 Work Flow

4.3.1 Normal Operations

CodFS performs erasure coding on the write path as illustrated in Figure 4.2. To write a file, the client first splits the file into segments, and requests the MDS to store the metadata and identify the *primary OSD* for each segment. The client then sends each segment to its primary OSD, which encodes the segment into k data chunks and $n - k$ parity chunks for some pre-specified parameters n and k. The primary OSD

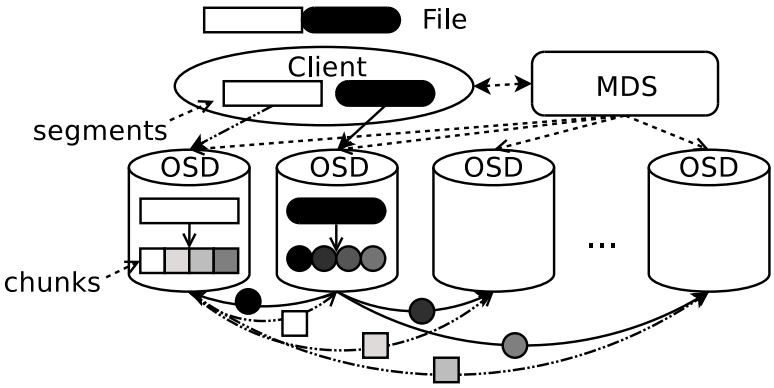


Figure 4.2: CodFS architecture.

stores a data chunk locally, and distributes the remaining $n-1$ chunks to other OSDs called the *secondary OSDs* for the segment. The identities of the secondary OSDs are assigned by the MDS to keep the entire cluster load-balanced. Both primary and secondary OSDs are defined in a logical sense, such that each physical OSD can act as a primary OSD for some segments and a secondary OSD for others.

To read a segment, the client first queries MDS for the primary OSD. It then issues a read request to the primary OSD, which collects one data chunk locally and $k-1$ data chunks from other secondary OSDs and returns the original segment to the client. In the normal state where no failure occurs, the primary OSD only needs the k data chunks of the segment for rebuilding.

CodFS adopts the delta-based approach for data updates. To update a segment, the client sends the modified data with the corresponding offsets to the segment's primary OSD, which first splits the update into sub-updates according to the offsets, such that each sub-update targets a single data chunk. The primary OSD then sends each sub-update to the OSD storing the targeted data chunk. Upon receiving a sub-update for a data chunk, an OSD computes the parity deltas and distributes them to the parity destinations. Finally, both the updates and parity deltas are saved according to the chosen update scheme.

4.3.2 Degraded Operations

CodFS switches to degraded mode when some OSDs fail (assuming the number of failed OSDs is tolerable). The primary OSD coordinates the degraded operations for its responsible segments. If the primary OSD of a segment fails, CodFS promotes another surviving secondary OSD of the segment to be the primary OSD. CodFS supports degraded reads and recovery. To issue a degraded read to a segment, the primary OSD follows the same read path as the normal case, except that it collects both data and parity chunks of the segment. It then decodes the collected chunks and returns the original segment. If an OSD failure is deemed permanent, CodFS can recover the lost chunks on a new OSD. That is, for each segment with lost chunks, the corresponding primary OSD first reconstructs the segment as in degraded reads, and then writes the lost chunk to the new OSD. Our current implementation of degraded reads and recovery uses the standard approach that reads k chunks for reconstruction, and it works for any number of failed OSDs no more than $n - k$. Nevertheless, our design is also compatible with efficient recovery approaches that read less data under single failures (e.g., [33, 77]).

4.4 Implementation Issues

We address several implementation issues in CodFS and justify our design choices.

4.4.1 Consistency

CodFS provides close-to-open consistency [27], which offers the same level of consistency as most Network File Systems (NFS) clients. Any open request to a segment always returns the version following the previous close request. CodFS directs all reads and writes of a segment through the corresponding primary OSD, which uses a lock-based approach to serialize the requests of all clients. This simplifies consistency implementation.

4.4.2 Offloading

CodFS offloads the encoding and reconstruction operations from clients. Client-side encoding generates more write traffic since the client needs to transmit parity chunks. Using the primary OSD design limits the fan-outs of clients and the traffic between the clients and the storage cluster. In addition, CodFS splits each file into segments, which are handled by different primary OSDs in parallel. Hence, the computational power of a single OSD will not become a bottleneck on the write path. Also, within each OSD, CodFS uses multi-threading to pipeline and parallelize the I/O and encoding operations, so as to mitigate the overhead in encoding computations.

4.4.3 Metadata Management

The MDS stores all metadata in a key-value database built on MongoDB [45]. CodFS can configure a backup MDS to serve the metadata operations in case the main MDS fails, similar to HDFS [5].

4.4.4 Caching

CodFS adopts simple caching techniques to boost the entire system performance. Each CodFS client is equipped with an LRU cache for segments so that frequent updates of a single segment can be batched and sent to the primary OSD. The LRU cache also favors frequent reads of a single segment, to avoid fetching the segment from the storage cluster in each read. We do not consider specific write mitigation techniques (e.g., lazy write-back and compression) or advanced caches (e.g., distributed caching or SSDs), although our system can be extended with such approaches.

4.4.5 Segment Size

CodFS supports flexible segment size from 16MB to 64MB and sets the default at 16MB. This size is chosen to fully utilize both the network bandwidth and disk throughput, as shown in our experiments (see Section 5.1). Smaller segments lead to more disk

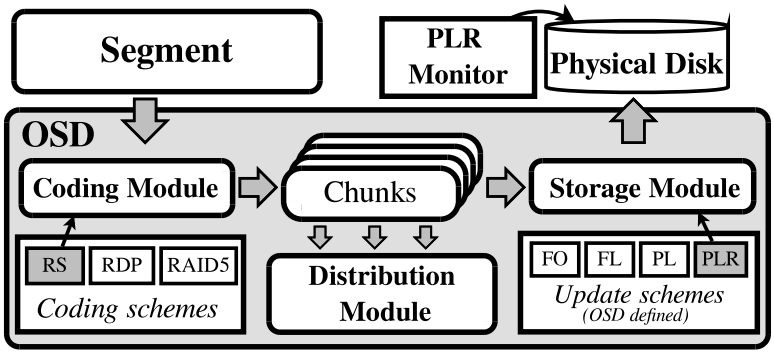


Figure 4.3: Internals of an OSD.

I/Os and degrade the write throughput, while larger segments cannot fully leverage the I/O parallelism across multiple OSDs.

4.5 Implementation Details

We design CodFS based on commodity configurations. We implement all the components including the client and the storage backend in C++ on Linux. CodFS leverages several third-party libraries for high-performance operations, including: (1) Threadpool [67], which manages a pool of threads that parallelize I/O and encoding operations, (2) Google Protocol Buffers [21], which serialize message communications between different entities, (3) Jerasure [49], which provides interfaces for efficient erasure coding implementation, and (4) FUSE [19], which provides a file system interface for clients.

Figure 4.3 illustrates the OSD design. We design the OSD via a modular approach. The *Coding Module* of each OSD provides a standard interface for implementation of different coding schemes. One can readily extend CodFS to support new coding schemes. The *Storage Module* inside each OSD acts as an abstract layer between the physical disk and the OSD process. We store chunk updates and parity deltas according to the update scheme configured in the *Storage Module*. By default, CodFS uses the PLR scheme. Each OSD is equipped with a *Monitor Module* to perform garbage collection in FL and PL and reserved space shrinking and prediction in PLR.

We adopt Linux Ext4 as the local filesystem of each OSD to support fast reserved space allocation. We pre-allocate the reserved space for each parity chunk using the Linux system call `fallocate`, which marks the allocated blocks as uninitialized. Shrinking of the reserved space is implemented by invoking `fallocate` with the `FALLOC_FL_PUNCH_HOLE` flag. Since we allocate or shrink the reserved space as a multiple of chunk size, we avoid creating unusable holes in the file system.

□ **End of chapter.**

Chapter 5

Evaluation

We evaluate different parity update schemes through our CodFS prototype. We deploy CodFS on a testbed with 22 nodes of commodity hardware configurations. Each node is a Linux machine running Ubuntu Server 12.04.2 with kernel version 3.5. The MDS and OSD nodes are each equipped with Intel Core i5-3570 3.4GHz CPU, 8GB RAM and two Seagate ST1000DM003 7200RPM 1TB SATA harddisk. For each OSD, the first harddisk is used as the OS disk while the entire second disk is used for storing chunks. The client nodes are equipped with Intel Core 2 Duo 6420 2.13GHz CPU, 2GB RAM and a Seagate ST3160815AS 7200RPM 160GB SATA harddisk. Each node has a Gigabit Ethernet card installed and all nodes are connected via a Gigabit full-duplex switch.

5.1 Baseline Performance

We derive the achievable aggregate read/write throughput of CodFS and analyze its best possible performance. Suppose that the encoding overhead can be entirely masked by our parallel design. If our CodFS prototype can effectively mitigate encoding overhead and evenly distribute the operations among OSDs, then it should achieve the theoretical throughput.

We define the notation as follows. Let M be the total number of OSDs in the system,

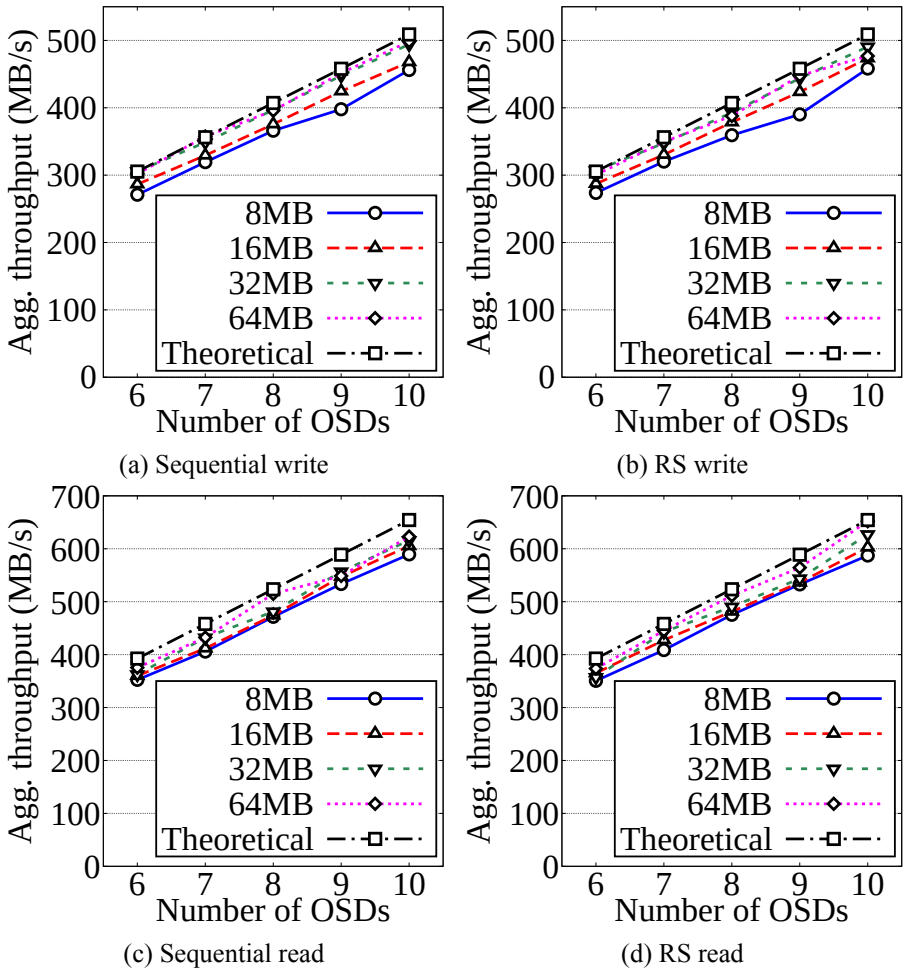


Figure 5.1: Aggregate read/write throughput of CodFS using the RDP code with $(n, k) = (6, 4)$.

and let B_{in} and B_{out} be the available inbound and outbound bandwidths (in network or disk) of each OSD, respectively. Each encoding scheme can be described by the parameters n and k, following the same definitions in Section 2.1.1.

We derive the effective aggregate write throughput (denoted by T_{write}). Each primary OSD, after encoding a segment, stores one chunk locally and distributes $n - 1$ chunks to other secondary OSDs. This introduces an additional $(n-1)/k$ times of segment traffic among the OSDs. Similarly, for the effective aggregate read throughput (denoted by T_{read}), each primary OSD collects $(k - 1)$ chunks for each read segment from the secondary OSDs. It introduces an additional $(k - 1)/k$ times of segment traffic. Thus, T_{write} and T_{read} are given by:

$$T_{write} = \frac{M \times B_{in}}{1 + \frac{n-1}{k}}, \qquad T_{read} = \frac{M \times B_{out}}{1 + \frac{k-1}{k}}.$$

We evaluate the aggregate read/write throughput of CodFS, and compare the experimental results with our theoretical results. We first conduct measurements on our testbed and find that the effective disk and network bandwidths of each node are 144MB/s and 114.5MB/s, respectively. Thus, the nodes are network-bound, and we set $B_{in} = B_{out} = 114.5$MB/s in our model. We configure CodFS with one node as the MDS and M nodes as OSDs, where $6 \leq M \leq 10$. We consider the RS codes and RDP codes with $(n, k) = (6, 4)$. The coded chunks are distributed over the M OSDs. We have 10 other nodes in the testbed as clients that transfer streams of segments simultaneously.

Figure 5.1 shows the aggregate read/write throughput of CodFS versus the number of OSDs for different segment sizes from 8MB to 64MB. We see that the throughput results match closely with the theoretical results, and the throughput scales linearly with the number of OSDs. For example, when $M = 10$ OSDs are used, CodFS achieves read and write throughput of at least 580MB/s and 450MB/s, respectively.

We observe that both RDP and RS codes have almost identical throughput, although RS codes have higher encoding overhead [49]. The reason is that CodFS masks the encoding overhead through parallelization.

5.2 Evaluation on Synthetic Workload

We now evaluate the four delta-based parity update schemes (i.e., FO, FL, PL, and PLR) using our CodFS prototype under a synthetic workload. Unless otherwise stated, we use the RDP code [14] with $(n, k) = (6, 4)$, 16MB segment size, and the same cluster configuration as in Section 5.1. We measure the sequential write, random write, sequential read, and recovery performance of CodFS using IOzone [29]. For PLR, we use the baseline approach described in Section 4.1.2 and fix the size of reserved space to 4MB, which is equal to the chunk size in our configuration. We trigger a merge

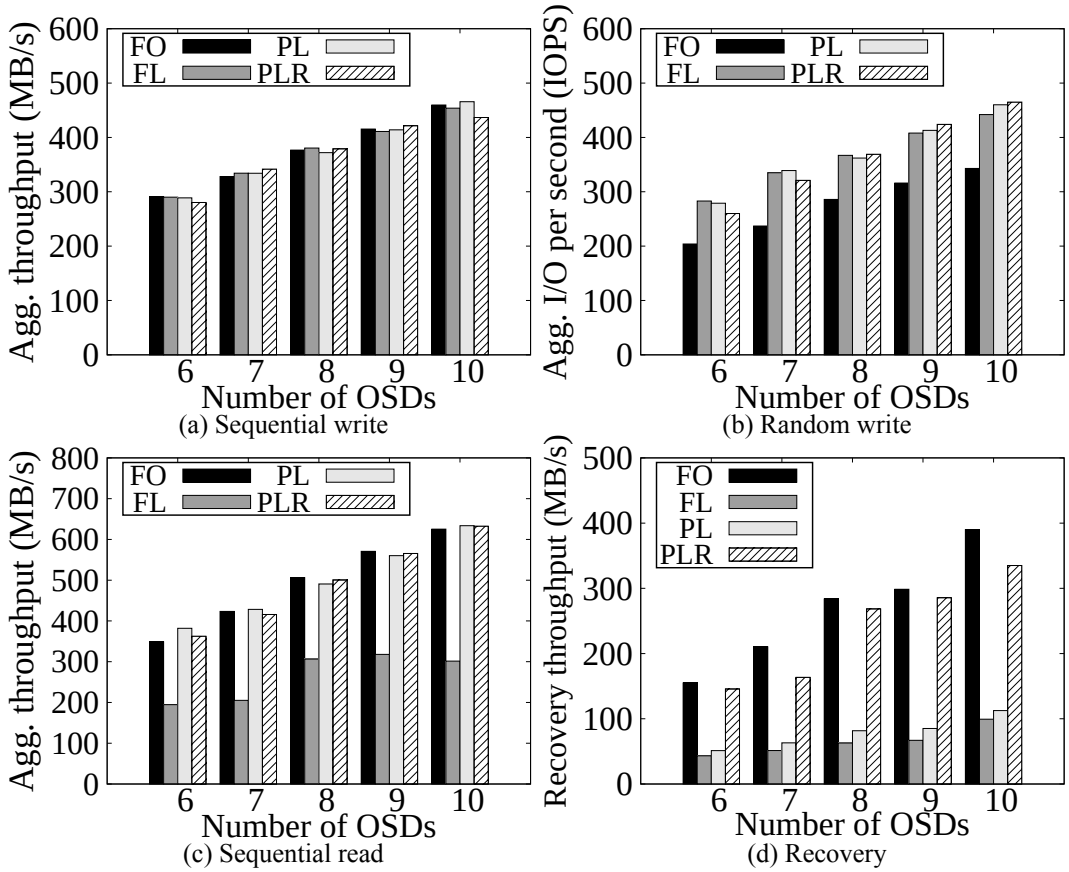


Figure 5.2: Throughput of CodFS under different update schemes.

operation to reclaim the reserved space when it becomes full. Before running each test, we format the chunk partition of each OSD to restore the OSD to a clean state, and drop the buffer cache in all OSDs to ensure that any difference in performance is attributed to the update schemes.

We note that an update to a data chunk in RDP [14] involves more updates to parity chunks than in RS codes (see [51] for illustration), and hence generates larger-size parity deltas. This triggers more frequent merge operations as the reserved space becomes full faster.

5.2.1 Sequential Write Performance

Figure 5.2a shows the aggregate sequential write throughput of CodFS under different update schemes, in which all clients simultaneously write 2GB of segments to

the storage cluster. As expected, there is only negligible difference in sequential write throughput among the four update schemes as the experiment only writes new data.

5.2.2 Random Write Performance

We use IOzone to simulate intensive small updates, in which we issue uniform random writes with 128KB record length to all segments uploaded in Section 5.2.1. In total, we generate 16MB of updates for each segment, which is four times of the reserved space size in PLR. Thus, PLR performs at least four merge operations per parity chunk (more merges are needed if the coding scheme triggers the updates of multiple parts of a parity chunk for each data update). Figure 5.2b shows the numbers of I/Os per second (IOPS) of the four update schemes. Results show that FO performs the worst among the four, with at least 21.0% fewer IOPS than the other three schemes. This indicates that updating both the data and parity chunks in-place incurs extra disk seeks and parity read overhead, thereby significantly degrading update efficiency. The other three schemes give similar update performance with less than 4.1% difference in IOPS.

5.2.3 Sequential Read Performance

Sequential read and recovery performance are affected by disk fragmentation in data and parity chunks. To measure fragmentation, we define a metric F_{avg} as the *average number of non-contiguous fragments per chunk* that are read from disk to rebuild the up-to-date chunk. Empirically, F_{avg} is found by reading the physical block addresses of each chunk in the underlying file system of the OSDs using the `filefrag -v` command which is available in the `e2fsprogs` utility. For each chunk, we obtain the number of non-contiguous fragments by analyzing its list of physical block addresses and lengths. We then take the average over the chunks in all OSDs.

Table 5.1 shows the value of F_{avg} measured after random writes in Section 5.2.2. Both FO and PLR have $F_{avg} = 0$ as they either store updates and deltas in-place or

		FO	FL	PL	PLR
Synthetic	Data	0	29.41	0	0
	Parity	0	117.66	117.66	0

Table 5.1: Average non-contiguous fragments per chunk (F_{avg}) after random writes for synthetic workload.

in a contiguous space next to their parity chunks. FL is the only scheme that contains non-contiguous fragments for data chunks, and it has $F_{avg} = 29.41$ in the synthetic benchmark. Logging parity deltas introduces higher level of disk fragmentation. On average, both FL and PL produce 117.66 non-contiguous fragments per parity chunk in the synthetic benchmark. We see that F_{avg} of parity chunks is about $4\times$ that of data chunks. This conforms to our RDP configuration with $(n, k) = (6, 4)$ since each segment consists of four data chunks and modifying each of them once will introduce a total of four parity deltas to each parity chunk.

Figure 5.2c shows a scenario which we execute a sequential read after intensive random writes. We measure the aggregate sequential read throughput under different update schemes. In this experiment, all clients simultaneously read the segments after performing the updates described in Section 5.2.2.

Since CodFS only reads data chunks when there are no node failures, no performance difference in sequential read is observed for FO, PL and PLR. However, the sequential read performance of FL drops by half when compared with the other three schemes. This degradation is due to the combined effect of disk seeking and merging overhead for data chunk updates. The result also agrees with the measured level of disk fragmentation shown in Table 5.1 where FL is the only scheme that contains non-contiguous fragments for data chunks.

5.2.4 Recovery Performance

We evaluate the recovery performance of CodFS under a double failure scenario, and compare the results among different update schemes. We trigger the recovery pro-

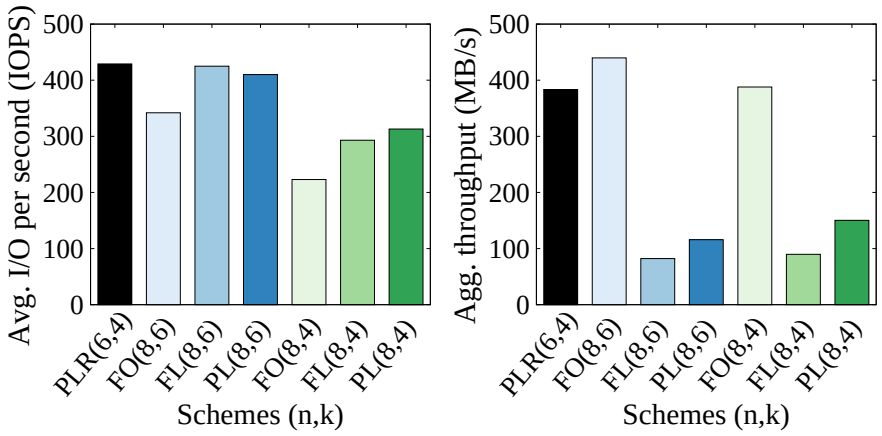
cedure by sending SIGKILL to the CodFS process in two of the OSDs. We measure the time between sending the kill signal and receiving the acknowledgement from the MDS reporting all data from the failed OSDs are reconstructed and redistributed among the available OSDs.

Figure 5.2d shows the measured recovery throughput for different update schemes. FO is the fastest in recovery and achieves substantial difference in recovery throughput (up to $4.5\times$) compared with FL due to the latter suffering from merging and disk seeking overhead for both data and parity chunks. By keeping data chunks updates in-place, PL achieves a modest increase in recovery throughput compared with FL. We also see the benefits of PLR for keeping delta updates next to their parity chunks. PLR gains a $3\times$ improvement on average in recovery throughput when compared with PL.

5.2.5 Reserved Space versus Update Efficiency

We thus far evaluate the parity update schemes under the same coding parameters (n, k). Since PLR trades storage space for update efficiency, we also compare PLR with other schemes that use the reserved space for storage. Here, we set the reserved space size to be equal to the chunk size in PLR with $(n, k) = (6, 4)$. This implies that a size of two extra chunks is reserved per segment. For FO, FL, and PL, we substitute the reserved space with either two data chunks or two parity chunks. We realize the substitutions with erasure coding using two coding parameters: $(n, k) = (8, 6)$ and $(n, k) = (8, 4)$, which in essence store two additional data chunks, and two additional parity chunks over $(n, k) = (6, 4)$, respectively. Since RDP requires $n - k = 2$, we choose the Cauchy RS code [8] as the coding scheme. We also fix the chunk size to be 4MB, so we ensure that each coded segment in all 7 configurations takes 32MB of storage including data, parity, and reserved space.

Figure 5.3 shows the performance of random writes and recovery under the same synthetic workload described in Section 5.2.2. Results show that the $(8, 4)$ schemes perform significantly worse than the $(8, 6)$ schemes in random writes, since having more



(a) Random write (b) Recovery

Figure 5.3: Throughput comparison under the same storage overhead using Cauchy RS codes with various (n, k).

parity chunks implies more parity updates. Also, we see that FO $(8, 6)$ is slower than PLR $(6, 4)$ by at least 20% in terms of IOPS, indicating that allocating more data chunks does not necessarily boost update performance. Results of recovery agree with those in Section 5.2.4, i.e., both FO and PLR give significantly higher recovery throughput than FL and PL.

5.2.6 Summary of Results

We make the following observations from our synthetic evaluation. First, although our configuration has twice as many data chunks as parity chunks, updating data chunks in-place in PL does not help much in recovery throughput. This implies that the time spent on reading and rebuilding parity chunks dominates the recovery performance. Second, as shown in Table 5.1, both FO and PLR do not produce disk seeks. Thus, we can attribute the difference in recovery throughput between FO and PLR solely to the merging overhead for parity updates. We see that PLR incurs less than 9.2% in recovery throughput on average compared with FO. We regard this as a reasonable trade-off since recovery itself is a less common operation than random writes.

5.3 Evaluation on Real-world Traces

Next, we evaluate CodFS by replaying the MSR Cambridge and Harvard NFS traces analyzed in Section 3.

5.3.1 MSR Cambridge Traces

To limit the experiment duration, we choose 10 of the 36 volumes for evaluating the update and recovery performance. We choose the traces with the number of write requests between $800,000$ and $4,000,000$. Also, to demonstrate that our design does not confine to a specific workload, the traces we select for evaluation all come from different types of servers.

We first pre-process the traces as follows. We adjust the offset of each request accordingly so that the offset maps to the correct location of a chunk. We ensure that the locality of requests to the chunks is preserved. If there are consecutive requests made to a sequence of blocks, they will be combined into one request to preserve the sequential property during replay.

We configure CodFS to use 10 OSDs and split the trace evenly to distribute replay workload among 10 clients. We first write the segments that cover the whole working set size of the trace. Each client then replays the trace by writing to the corresponding offset of the preallocated segments. We use RDP [14] with $(n, k) = (6, 4)$ and 16MB segment size.

Update Performance

Figure 5.4 shows the aggregate number of writes replayed per second. To perform a stress test, we ignore the original timestamps in the traces and replay the operations as fast as possible. First, we observe that traces with a smaller coverage (as indicated by the percentage of updated WSS in Table 3.2) in general results in higher IOPS no matter which update scheme is used. For example, the `usr0` trace with 13.08% updated

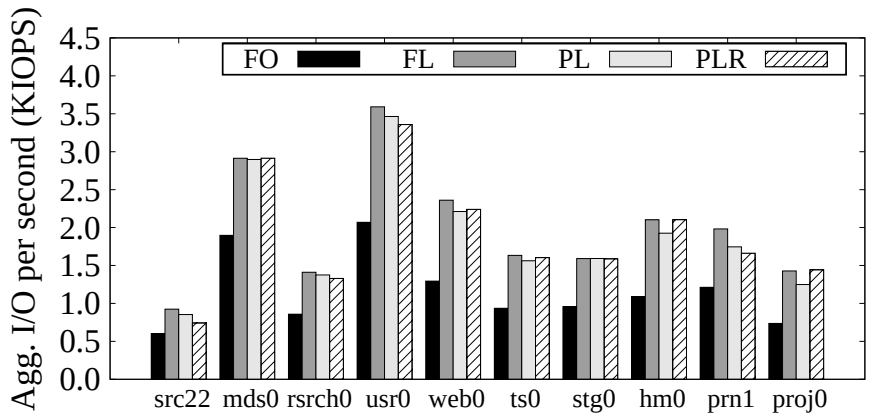


Figure 5.4: Number of writes per second replaying the selected MSR Cambridge traces under different update schemes.

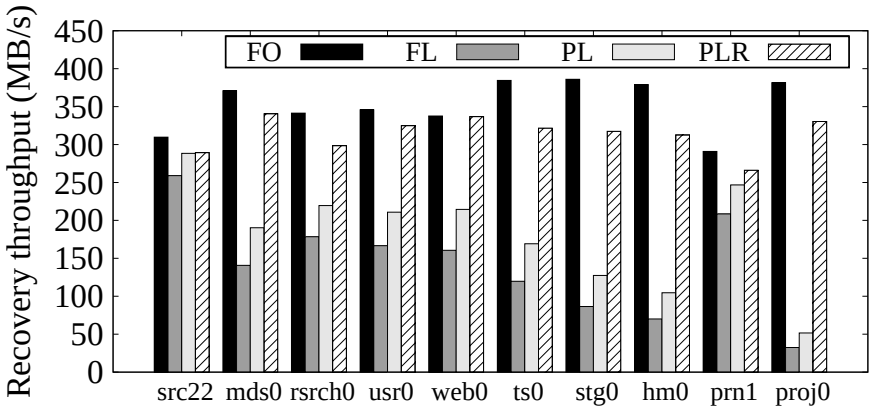


Figure 5.5: Recovery throughput in a double failure scenario after replaying the selected MSR Cambridge traces under different update schemes.

WSS shows more than $3\times$ update performance when compared with the `src22` trace with 99.57% updated WSS. This is due to a more effective client LRU cache when the updates are focused on a small set of chunks. The cache performs write coalescing and reduces the number of round-trips between clients and OSDs. Second, we see that the four schemes exhibit similar behaviour across traces. FL, PL and PLR show comparable update performance. This leads us to the same implication as in Section 5.2.2 that the dominant factor influencing update performance is the overhead in parity updates. Therefore, the three schemes that use a log-based design for parity chunks all perform significantly better than FO. On average, PLR is 63.1% faster than FO.

Recovery Performance

Figure 5.5 shows the recovery throughput in a two-node failure scenario. We see that in all traces, FL and PL are slower than FO and PLR in recovery. Also, PLR outperforms FL and PL more significantly in traces where there is a large number of writes and F_{avg} is high. For example, the measured F_{avg} for the `proj0` trace is 45.66 and 182.6 for data and parity chunks, respectively, and PLR achieves a remarkable $10\times$ speedup in recovery throughput over FL. On the other hand, PLR performs the worst in the `src22` trace, where F_{avg} is only 0.73 and 2.82 for data and parity chunks, respectively. Nevertheless, it still manages to give an 11.7% speedup over FL.

5.3.2 Evaluation of Reserved Space

We evaluate our workload-aware approach in managing the reserved space size (see Section 4.1.2). We use the Harvard NFS traces, whose 41-day span provides long enough duration to examine the effectiveness of shrinking, merging, and prediction. We calculate the *reserved space storage overhead* using the following equation, which is defined as the additional storage space allocated by the reserved space compared with the original working set size without any reserved space:

$$\Gamma = \frac{\sum ReservedSpaceSize}{\sum (DataSize + ParitySize)}.$$

A low Γ means that the reserved space is small compared with the total size of all data and parity chunks.

Using the above metric, we evaluate our workload-aware framework used in PLR by simulating the Harvard NFS traces. We set the segment size to 16MB and use the Cauchy RS code [8] with $(n, k) = (10, 8)$. Here, we compare our workload-aware approach with three baseline approaches, in which we fix the reserved space size to 2MB, 8MB, and 16MB without any adjustment.

We consider two variants of our workload-aware approach. The *shrink+merge* ap-

proach executes the shrinking operation at 00:00 and 12:00 on each day, followed by a merge operation on each chunk. The *shrink only* approach is identical to the *shrink +merge* approach in shrinking, but does not perform any merge operation after shrinking (i.e., it does not free the space occupied by the parity deltas). On the first day, we initialize the reserved space to 16MB. We follow the framework described in Section 4.1.2 and set the smoothing factor $\alpha = 0.3$.

Simulation Results

Figure 5.6 shows the value of Γ under the three different approaches by simulating the 41-day Harvard traces. The 2MB, 8MB, and 16MB baseline approaches give $\Gamma = 0.2$, 0.8, and 1.6, respectively, throughout the entire trace since they never shrink the reserved space. The values of Γ for both workload-aware variants drop quickly in the first week of trace and then gradually stabilize. At the end of the trace, the *shrink only* approach has Γ of about 0.36. With merging, the *shrink+merge* approach further reduces Γ to 0.12. Γ is lower than that of the 2MB baseline, as around 13% of parity chunks end up with zero reserved space size.

Aggressive shrinking may increase the number of merge operations. We examine such an impact by showing the average number of merges per 1000 writes in Figure 5.7. A lower value implies lower write latency since fewer writes are stalled by merge operations. We make a few key observations from this figure. First, the 16MB baseline gives the best results among all strategies, since it keeps the largest reserved space than other baselines and workload-aware approaches throughout the whole period. On the contrary, using a fixed reserve space that is too small increases the number of merges significantly. This effect is shown by the 2MB baseline. Second, the performance of the workload-aware approaches matches closely with the 8MB and 16MB baseline approaches most of the time. Day 30-40 is an exception in which the two workload-aware approaches perform significantly more merges than the 16MB baseline approach. This reflects the penalty of inaccurate prediction when the reserved space is not large enough

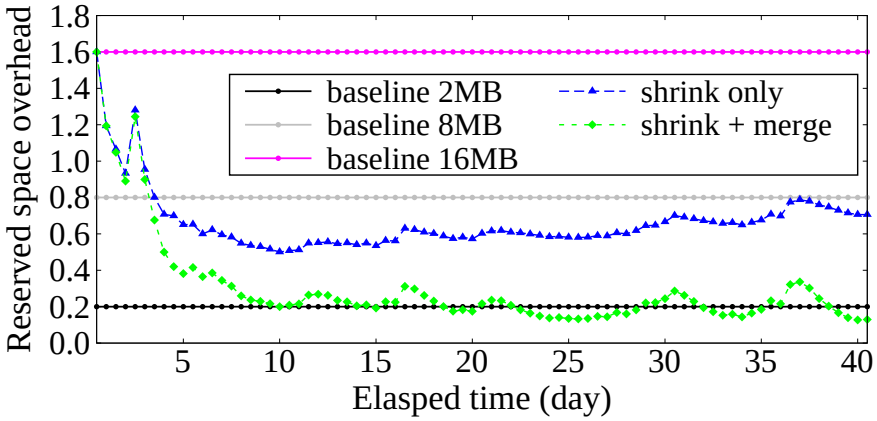


Figure 5.6: Reserved space overhead under different shrink strategies in the Harvard trace.

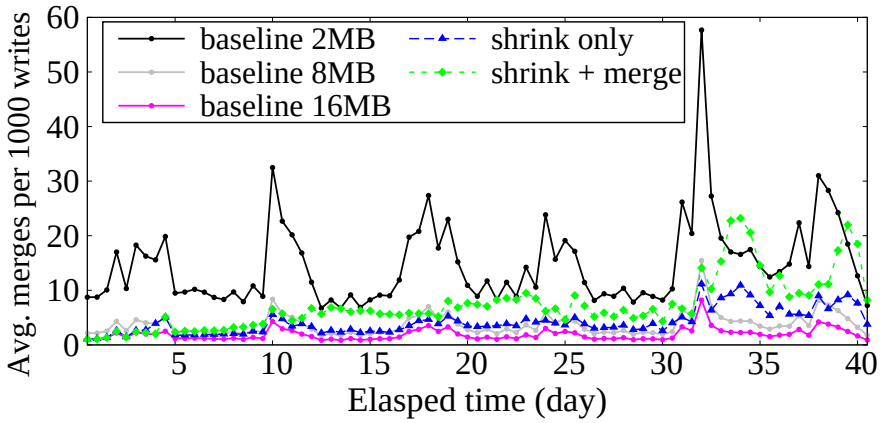


Figure 5.7: Average number of merges per 1000 writes under different shrink strategies in the Harvard trace.

to handle the sudden bursts in updates. Third, although the *shrink+merge* approach has a lower reserved space storage overhead, it incurs more penalty than the *shrink only* approach in case of a misprediction. However, we observe that on average less than 1% of writes are stalled by a merge operation regardless of which approach is used (recall that the merge is performed every 1000 writes). Thus, we expect that there is very little impact of merging on the performance in PLR.

5.3.3 Summary of Results

We show that PLR achieves efficient updates and recovery. It significantly improves the update throughput of FO and the recovery throughput of FL. We also eval-

uate our workload-aware approach on reserved space management. We show that the *shrink+merge* approach can reduce the reserved space storage overhead by more than half compared to the 16MB baseline approach, with slight merging penalty to reclaim space.

□ **End of chapter.**

Chapter 6

Toward Write Efficiency and Endurance in SSD RAID

We thus far focus on tackling small random writes in an erasure-coded clustered storage. We now introduce another project called TWEEN, which tackles small random writes in SSD RAID. In this chapter, we give an overview of the partial-write problem in SSD RAID and describe the design of TWEEN.

6.1 Partial Writes in SSD RAID

Small random writes are detrimental to both SSD performance [11,34,44] and RAID performance [64]. They exhibit in many real-world workloads including those in online transaction-processing (OLTP) systems [76], desktops [24] and enterprise servers [32]. Also, modern databases and critical applications tend to call `fsync/sync` frequently to force *synchronous writes*, which cause writes to be relatively small and random [24].

This section considers two specific types of small random writes that can happen in an SSD RAID, namely *partial-block writes* and *partial-stripe writes*, whose request sizes are too small and do not align with the granularities of the SSD and RAID operations, respectively. We collectively call them *partial writes*, which incur performance overhead to SSD RAID as described below. Figure 6.1 shows the SSD RAID layout

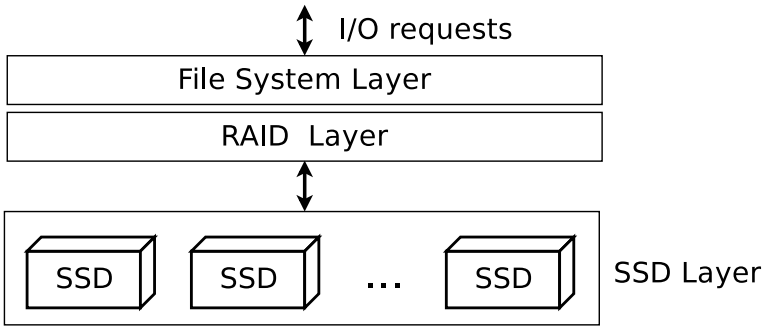


Figure 6.1: The SSD RAID layout considered in TWEEN.

that we consider in TWEEN.

6.1.1 Partial-Block Writes

In the SSD layer, partial-block writes are the writes that update only a portion of pages in an SSD block. Because of out-of-place writes in SSDs [3], partial-block writes eventually cause SSD blocks to contain a mix of live and stale pages, leading to internal fragmentation [11]. Such internal fragmentation increases the write amplification overhead of garbage collection, because each time garbage collection needs to copy many live pages from the chosen block to a free block. If the number of stale pages in a block is small, then live pages of multiple blocks need to be relocated so as to reclaim a block that is full of clean pages. Note that the large write amplification overhead is seen regardless of the underlying address mapping schemes in FTL [44].

6.1.2 Partial-Stripe Writes

Partial-stripe writes are the writes whose request sizes are smaller than the size of a stripe in RAID. Depending on the write size, RAID handles a partial-stripe write based on one of the two approaches [12]: (1) *read-modify write* and (2) *reconstruct write* (see Section 2.1.2). Both reconstruct writes and read-modify writes are inferior in performance as they incur extra reads before writes. Instead, it is more desirable to issue *full-stripe writes*, which incur no read by updating all chunks in a stripe and are the most efficient type of writes in RAID [12].

6.2 TWEEN Design

We propose TWEEN, a middleware application that aims toward write efficiency and endurance in SSD RAID. TWEEN comprises two key technologies. It couples LFS and NVRAM, by batching partial writes in NVRAM buffers and issuing only large writes to SSD RAID. Also, it mitigates the garbage collection overhead in LFS by grouping writes with similar file access frequencies and absorbing garbage collection writes in NVRAM. In this section, we elaborate the design details of TWEEN and justify how this design addresses the partial-write problem in SSD RAID.

6.2.1 Key Technologies

We elaborate two key technologies of TWEEN which help improve write efficiency and endurance of SSD RAID: (1) eliminating partial writes and (2) mitigating the LFS garbage collection overhead.

Eliminating Partial Writes

The core design of TWEEN is the log-structured file system (LFS) [57] coupled with byte-addressable non-volatile RAM (NVRAM) buffering. TWEEN follows LFS [57] by batching incoming small writes sequentially and flushing them to SSD RAID when the aggregated write size aligns the unit sizes of RAID encoding and SSD garbage collection.

TWEEN batches partial writes in buffers. In this work, we choose NVRAM-based buffering, and take advantage of the low-latency and persistence properties of NVRAM. NVRAM has emerged to be a potential candidate for substituting DRAM in computer systems due to its lower energy consumption and persistence properties [36, 81]. We consider byte-addressable NVRAM such as PCM (phase-change memory), FRAM (ferro-electric RAM), STT-RAM (spin-transfer torque RAM) and NVDIMM (non-volatile DIMM). While these variants differ in capacity, performance and endurance

[31], they all offer better performance and endurance than flash. For example, the write latency for PCM is 180 ns, as opposed to 200 μs for NAND flash [31]. Also, PCM and STT-RAM can sustain more than 10^7 and 10^{15} rewrites per cell, respectively [31, 81]. NVDIMM has recently emerged as a new type of NVRAM which provides the same speed and endurance as DRAM, and can be seamlessly integrated in existing DDR3 architecture [70]. Previous studies have extensively studied the use of NVRAM in storage systems to improve performance and reliability [31, 36, 52, 69, 81].

TWEEN keeps multiple NVRAM buffers, each of which corresponds to a hotness group (see Section 6.2.1). It associates each NVRAM buffer with a *segment*, which is the aggregate of multiple partial writes. Each segment has the same size as a RAID stripe and is the minimum flush unit in TWEEN. TWEEN only issues full-stripe writes to SSD RAID so as to avoid reading old data and parity chunks as seen in partial-stripe writes. This reduces the parity update overhead. Also, by setting the RAID chunk size to be a multiple of the SSD block size, TWEEN simultaneously eliminates partial-block writes in each SSD.

Compared to typical hardware RAID controllers (e.g., [15,38]) which treat NVRAM merely as a write-back cache, TWEEN introduces NVRAM as an extra tier of storage for keeping partially written segments. The persistence nature of NVRAM allows TWEEN to efficiently handle `fsync/sync` commands issued by upper-layer applications and avoid issuing partial writes to storage.

TWEEN uses only a small amount of NVRAM. Recall that each NVRAM buffer is configured to have the same size as a RAID stripe. Our default configuration of TWEEN uses the 8 MB stripe size and four NVRAM buffers, accounting for 32 MB of NVRAM in total. Note that the configured NVRAM capacity is independent of the total amount of data written to SSD RAID. In addition, TWEEN stores all metadata in NVRAM for simplicity and performance. We find the metadata only occupies a limited amount of NVRAM in the worst case and under real-world workloads.

TWEEN offloads garbage collection from SSDs to the LFS layer by always writing

in units of blocks (i.e., avoiding partial-block writes). However, the garbage collection overhead still exists in TWEEN , as it is now moved to the LFS layer. The LFS garbage collection is known to hamper I/O performance by (1) introducing copying overhead [61] and (2) causing I/O contentions [62]. We address this issue in the following discussion.

Mitigating LFS Garbage Collection Overhead

TWEEN eliminates partial writes using the LFS design, and hence it needs to address garbage collection in the LFS layer. Here, we propose two workload-aware approaches to mitigate the LFS garbage collection overhead.

First, TWEEN exploits workload characteristics to reduce copying overhead of garbage collection. It has been known that writes in real-world workloads are non-uniform and have different update frequencies [44, 79]. Thus, TWEEN continuously monitors the update frequency of each file and assigns a *hotness* value to each file. We pack writes with similar hotness in the same segment using a simple clustering algorithm (see Section 6.2.2). The intuition is that hot segments are frequently updated and hence likely contains stale data, while cold segments are likely occupied with live data. TWEEN greedily garbage-collects hot segments with most stale data to minimize the amount of data reclaimed.

Second, when TWEEN reclaims live data during LFS garbage collection, it returns the reclaimed data back to NVRAM buffers instead of writing the data directly to SSD RAID. This prevents LFS garbage collection itself from issuing partial writes to SSD RAID and contending with normal user writes. Also, this provides a feedback mechanism to re-assess the hotness of the reclaimed live data after LFS garbage collection, which may have different update frequencies. Such *iterative* hotness grouping can improve the accuracy of workload characterization.

The number of hotness groups affects the clustering accuracy, which is directly related to the garbage collection performance in TWEEN. Using too many hotness groups

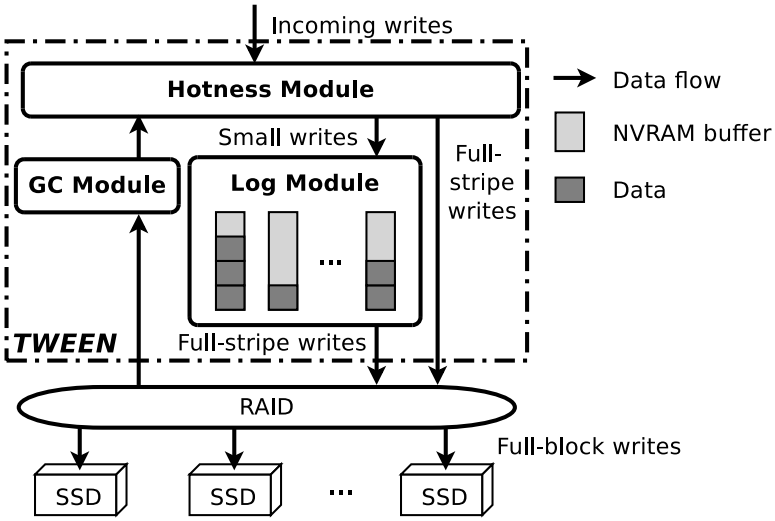


Figure 6.2: TWEEN architecture.

may lead to *over-fitting*, in which noises in samples lower the clustering accuracy; on the other hand, using too few groups may also affect clustering accuracy due to coarse-grain classification. Choosing the right number of hotness groups is non-trivial and highly dependent on the target workload.

6.2.2 Architecture

We now describe the internal architecture of TWEEN and explain how it realizes the aforementioned technologies. TWEEN focuses on optimizing both write and garbage collection flows. It uses the same read flow as the conventional LFS, and we omit the details here.

Figure 6.2 shows the TWEEN architecture, which consists of three major modules. The *hotness module* manages system metadata, monitors the update frequency of each file, and assigns a hotness group to each write based on the file's hotness value. The *log module* handles the write flow. It maintains an NVRAM buffer for each hotness group. Each buffer batches writes until a segment is full and can be flushed to the SSD RAID as a full-stripe write. Recall that the chunk size is set to be a multiple of SSD block size. Each full-stripe write results in full-block writes to the SSDs. The *garbage*

collection (GC) module reclaims space from stale segments by reading segments from the SSD RAID and sending live data back to the hotness module. In the following, we describe the design details of each module.

Hotness Module

The hotness module keeps track of each file's hotness value, denoted by H_f^t of each file f at time t. The hotness value captures the update frequency of a file over time. We calculate H_f^t using an exponential moving average approach:

$$H_f^t = \alpha \frac{1}{t - \hat{t}} + (1 - \alpha) H_f^{\hat{t}}$$

where $\hat{t}$ denotes the last modified time of file f and α is a smoothing factor.

To determine the hotness group of a particular file, the hotness module performs data clustering based on the hotness values of all files and selects one cluster which is closest to the file in question as its hotness group. As classification is performed for every write, the hotness module adopts the efficient sequential K-means clustering algorithm [40] to minimize computational overhead.

Algorithm 2 describes the hotness classification mechanism. Here, the user specifies the number of hotness groups N. During startup, we make initial random guesses for the cluster centers. Next, for each classification query, we find the closest center using the `diff(x,y)` function, which computes the absolute difference between a query H_f^t and a center C_i for the i-th cluster (where $1 \leq i \leq N$). Upon selecting a center C_i, we update the center value with an aging factor β and return the query answer.

The hotness value as well as other file system metadata are kept in NVRAM for performance and reliability reasons, as both types of data are frequently accessed and updated.

Algorithm 2: Hotness clustering and classification

```
N ←HOTNESS_GROUP
Make initial guesses for the cluster centers C_1, C_2, ..., C_N
while true do
    Acquire the next classification query, H_f^t
    if diff(C_i, H_f^t) is smallest among all centers then
        C_i ← C_i + β(H_f^t − C_i)
        Classify H_f^t as hotness group i
```

Log Module

The log module maintains N NVRAM buffers for logging writes, where N is the number of hotness groups. Each buffer holds a segment for each hotness group, and the segment size is set to be the same as the RAID stripe size. The NVRAM buffers also act as non-volatile cache and serve read requests for data that is not yet flushed to the underlying SSD RAID.

Each incoming write is first handled by the hotness module, which updates the file hotness and assigns a hotness group for the write. If the write size is larger than a stripe, the hotness module can bypass logging by issuing full-stripe writes to SSD RAID until the remaining data is insufficient to fill a stripe. Next, the log module receives the data from the hotness module after classification and appends the write to the corresponding NVRAM buffer. If the buffer is full, the log module seals and flushes the segment to SSD RAID by issuing a full-stripe write. It then clears the buffer and associates it with a new segment. Finally, the log module acknowledges the hotness module on metadata updates.

Garbage Collection (GC) Module

The GC module cleans stale segments to reclaim space. We design the GC module to be aware of the file update frequency by coupling it with the hotness module. In each garbage collection operation, the GC module reads the entire garbage-collected segment and reclaims live data within. It then sends the remaining live data of each

file back to the hotness module, which re-classifies and directs data to the log module. It frees the address of the reclaimed segment by updating the corresponding metadata entry.

TWEEN supports two garbage collection modes: (1) *periodic* and (2) *active* [57]. Periodic mode reclaims space from stale data during system idle time in the background, while active mode reclaims space when the total storage space including both live and stale data exceeds some threshold. For more write-intensive workloads, active mode is preferred over periodic mode as the idle time between requests is short. In our evaluation, we choose one of the two modes depending on the workloads we consider.

6.2.3 Implementation Details

We build TWEEN entirely as a user-space application, and hence it is compatible with off-the-shelf RAID (including its hardware and software RAID variants) and SSDs. We implement the TWEEN prototype in C++ on Linux. The TWEEN core implements all the software modules described in Section 6.2.2. It has around 2400 lines of code in total.

Our TWEEN prototype exports the file system interface as a client API that allows upper-layer applications to access the underlying SSD RAID through TWEEN. The API supports the essential file system operations for our performance evaluation, such as creating, reading, and writing files. Since TWEEN supports the basic file system functionalities, we can compare it with state-of-the-art file systems. Nevertheless, our current prototype does not support all file system features such as permissions and security. We pose the full file system implementation as future work.

6.3 Summary

We propose TWEEN, a middleware application designed to enhance the write efficiency and endurance of SSD RAID. TWEEN has several design goals: (1) write efficiency and endurance, (2) metadata efficiency, (3) data availability and (4) compat-

ibility with off-the-shelf solutions. TWEEN utilizes the emerging NVRAM technology as non-volatile cache. It combines the log-structured file system (LFS) and NVRAM to batch incoming partial writes and flush them to SSD RAID as full-stripe writes. It also exploits workload characteristics to reduce LFS garbage collection overhead. TWEEN is a user-space application and works seamlessly with off-the-shelf SSD RAID solutions. Our design speeds up random writes, lowers SSD garbage collection overhead, and extends SSD lifetime.

□ **End of chapter.**

Chapter 7

Conclusions

7.1 Summary

In CodFS, our key contribution is the parity logging with reserved space (PLR) scheme, which keeps parity updates next to the parity chunk to mitigate disk seeks. We also propose a workload-aware scheme to predict and adjust the reserved space size. We build the CodFS prototype, which is an erasure-coded clustered storage system that achieves efficient updates and recovery. We evaluate CodFS using both synthetic and real-world traces and show that PLR improves update and recovery performance over pure in-place and log-based updates.

We also propose TWEEN, a user-space middleware application designed to enhance the write efficiency and endurance of SSD RAID. TWEEN utilizes the emerging NVRAM technology as non-volatile cache and works seamlessly with off-the-shelf SSD RAID solutions. It combines the log-structured file system (LFS) and NVRAM to batch incoming partial writes and flush them to SSD RAID as full-stripe writes. It also exploits workload characteristics to reduce LFS garbage collection overhead.

7.2 Future Work

7.2.1 Applying CodFS on Other Storage Architectures

Currently, CodFS optimizes update performance on disk-based clustered storage with commodity hardware configurations. We may extend CodFS to other storage architectures, such as those utilizing DRAM for storage [47], or those using a combination of NVRAM and SSDs [52]. Although the seek time may not be as significant in these types of storage media, we believe CodFS can still improve performance by reducing fragmentation in the system. However, an important limitation for these types of storage media is the high storage cost. We pose the issue of further reducing the reserved space overhead as future work.

7.2.2 Improving Reserved Space Management Designs

In our work, CodFS only uses a simple workload-aware design for reserved space management. Although we show that this heuristic works reasonably well in evaluations, we plan to explore more advanced designs. For example, instead of shrinking the reserved space of all segments daily, it is beneficial to periodically shrink some of the popular segments during idle time. To choose the right segments to shrink, we may utilize existing victim selection policies in LFS, such as cost-benefit [57] and cost-hotness [44].

7.2.3 Deploying TWEEN on Single-SSD Systems

Currently, TWEEN focuses on SSD RAID deployments, which may limit its applications to enterprise environments. However, since our system decouples the RAID layer from the file system layer, it is also possible to adopt TWEEN on a single-SSD setup. We believe that by batching small random writes in a non-volatile buffer and aligning the granularity of the SSD garbage collection, our design can also benefit the write performance and endurance in a desktop environment using a single SSD. We

pose the evaluation of TWEEN on single-SSD deployments under desktop workloads as future work.

□ **End of chapter.**

Bibliography

[1] M. Abd-El-Malek, W. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. Sambasivan, et al. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of USENIX FAST*, Dec 2005.

[2] I. F. Adams, M. W. Storer, and E. L. Miller. Analysis of Workload Behavior in Scientific and Historical Long-Term Data Repositories. *ACM TOS*, 8:6:1–6:27, 2012.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC*, Jun 2008.

[4] M. K. Aguilera and R. Janakiraman. Using Erasure Codes Efficiently for Storage in a Distributed System. In *Proc. of IEEE DSN*, Jun 2005.

[5] Apache. HDFS Architecture Guide. `http://hadoop.apache.org/docs/stable1/hdfs_design.html`.

[6] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of USENIX NSDI*, Oct 2004.

[7] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE TOC*, 44(2): 192–202, Feb. 1995.

[8] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based Erasure-resilient Coding Scheme. Technical report, International Computer Science Institute, Berkeley, USA, 1995.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

[10] P. Cao, S. B. Lin, S. Venkataraman, and J. Wilkes. The TickerTAIP Parallel RAID Architecture. *ACM TOCS*, 12:236–269, 1994.

[11] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proc. of ACM SIGMETRICS*, 2009.

[12] P. M. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *Proc. of ACM SIGMETRICS*, 1995.

[13] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, Jun 1994.

[14] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proc. of USENIX FAST*, Mar 2004.

[15] Dell. PowerEdge RAID Controller Cards. `https://www.dell.com/downloads/global/products/pvaul/en/perc-technical-guidebook.pdf`.

[16] D. J. Ellard. *Trace-based Analyses and Optimizations for Network Storage Servers*. PhD thesis, Cambridge, MA, USA, 2004. AAI3131831.

[17] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[18] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A Decentralized Algorithm for Erasure-Coded Virtual Disks. In *Proc. of IEEE DSN*, Jun 2004.

[19] FUSE. Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[20] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

[21] Google. Google Protocol Buffers. `https://code.google.com/p/protobuf/`.

[22] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proc. of IEEE/ACM MICRO*, 2009.

[23] L. M. Grupp, J. D. Davis, and S. Swanson. The Bleak Future of NAND Flash Memory. In *Proc. of USENIX FAST*, 2012.

[24] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proc. of ACM SOSP*, 2011.

[25] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. *ACM TOCS*, 13:274–310, 1995.

[26] J. Hendricks, R. R. Sambasivan, S. Sinnamohideen, and G. R. Ganger. Improving Small File Performance in Object-based Storage. Technical Report CMU-PDL-06-104, Carnegie Mellon University, May 2006.

[27] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM TOCS*, 6:51–81, 1988.

[28] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, Jun 2012.

[29] IOzone. IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[30] C. Jin, D. Feng, H. Jiang, and L. Tian. RAID6L: A Log-assisted RAID6 Storage Architecture with Improved Write Performance. In *Proc. of IEEE MSST*, 2011.

[31] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM TOS*, 6:3:1–3:25, 2010.

[32] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *Proc. of IEEE Int. Symp. on Workload Characterization (IISWC)*, 2008.

[33] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, Feb 2012.

[34] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proc. of USENIX FAST*, 2008.

[35] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ACM ASPLOS-IX*, Nov 2000.

[36] E. Lee, H. Bahn, and S. H. Noh. A Unified Buffer Cache Architecture That Subsumes Journaling Functionality via Nonvolatile Memory. *ACM TOS*, 10(1):1:1–1:17, Jan. 2014.

[37] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang. R-ADMAD: High Reliability Provision for Large-Scale De-duplication Archival Storage Systems. In *Proc. of ICS*, June 2009.

[38] LSI. MegaRAID. `http://www.lsi.com/downloads/Public/MegaRAID%20SAS/41450-04_RevA.pdf`.

[39] Y. Ma, T. Nandagopal, K. Puttaswamy, and S. Banerjee. An Ensemble of Replication and Erasure Codes for Cloud File Systems. In *Proc. of IEEE INFOCOM*, Apr 2013.

[40] J. Macqueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proc. of Berkeley Symp. on Mathematical Statistics and Probability*, 1967.

[41] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proc. of ACM SOSP*, Oct 1997.

[42] J. Menon. A Performance Comparison of RAID-5 and Log-structured Arrays. In *Proc. of 4th IEEE International Symposium on High Performance Distributed Computing (HDPC)*, 1995.

[43] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit Error Rate in NAND Flash Memories. In *IEEE Int. Reliability Physics Symp.*, 2008.

[44] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proc. of USENIX FAST*, 2012.

[45] MongoDB, Inc. MongoDB. `http://www.mongodb.org/`.

[46] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical Power Management for Enterprise Storage. *ACM TOS*, 4:10:1–10:23, 2008.

[47] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, Oct 2011.

[48] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of ACM SIGMOD*, 1988.

[49] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, Feb 2009.

[50] J. S. Plank. The RAID-6 Liberation Codes. In *Proc. of USENIX FAST*, 2008.

[51] J. S. Plank and C. Huang. Erasure Codes for Storage Systems: A Brief Primer. *;login: the Usenix magazine*, 38(6):44–50, Dec 2013.

[52] S. Qiu and A. Reddy. NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD. In *Proc. of IEEE MSST*, 2013.

[53] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, Jun 1960.

[54] J. K. Resch and J. S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proc. of USENIX FAST*, Feb 2011.

[55] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore Prototype. In *Proc. of USENIX FAST*, Mar 2003.

[56] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Proc. of IPTPS*, Feb 2005.

[57] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM TOCS*, 10:26–52, 1992.

[58] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proc. of the VLDB Endowment*, Aug 2013.

[59] SearchStorage. RAID Alternatives: Will Erasure Codes Rule? `http://searchstorage.techtarget.com/tip/RAID-alternatives-Will-erasure-codes-rule`.

[60] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An Implementation of a Log-structured File System for UNIX. In *Proc. of USENIX Winter Conference*, 1993.

[61] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of USENIX 1995 Technical Conference (TCON)*, 1995.

[62] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious Disk Arrays for Cloud Storage. In *Proc. of USENIX FAST*, Feb 2013.

[63] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[64] D. Stodolsky, G. Gibson, and M. Holland. Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA)*, May 1993.

[65] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-based Archival Storage. In *Proc. of USENIX FAST*, Feb 2008.

[66] A. Thomasian. Reconstruct Versus Read-modify Writes in RAID. *Inf. Process. Lett.*, 93(4):163–168, Feb 2005.

[67] Threadpool. `http://threadpool.sf.net/`.

[68] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proc. of USENIX OSDI*, 2004.

[69] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proc. of USENIX FAST*, 2011.

[70] Viking Technology. ArxCis-NV: The Non-Volatile DIMM. `http://www.vikingtechnology.com/uploads/arxcis_productbrief.pdf`.

[71] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[72] C. Weddle, M. Oldham, J. Qian, and A. i Andy Wang. PARAID: A Gear-Shifting Power-Aware RAID. In *Proc. of USENIX FAST*, Feb 2007.

[73] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proc. of USENIX OSDI*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[74] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of USENIX FAST*, Feb 2008.

[75] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proc. of ACM SOSP*, 1995.

[76] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proc. of USENIX ATEC*, 2002.

[77] L. Xiang, Y. Xu, J. C. Lui, and Q. Chang. Optimal Recovery of Single Disk Failure in RDP Code Storage Systems. In *Proc. of ACM SIGMETRICS*, Jun 2010.

[78] F. Zhang, J. Huang, and C. Xie. Two Efficient Partial-Updating Schemes for Erasure-Coded Storage Clusters. In *Proc. of IEEE Seventh International Conference on Networking, Architecture, and Storage (NAS)*, Jun 2012.

[79] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming Up Storage-level Caches with Bonfire. In *Proc. of USENIX FAST*, 2013.

[80] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does Erasure Coding Have a Role to Play in my Data Center? Technical Report MSR-TR-2010-52, Microsoft Research, May 2010.

[81] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proc. of ACM ISCA*, 2009.