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, and Helen H. W. Chan,
The Chinese University of Hong Kong

https://www.usenix.org/conference/fast14/technical-sessions/presentation/chan

This paper is included in the Proceedings of the
12th USENIX Conference on File and Storage Technologies (FAST '14).
February 17–20, 2014 • Santa Clara, CA USA
ISBN 978-1-931971-08-9

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
The Chinese University of Hong Kong
{*cwchan,qding,pclee,hwchan*}@*cse.cuhk.edu.hk*

Abstract

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 so as 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 testbed 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.

1 Introduction

Clustered storage systems are known to be susceptible to component failures [17]. 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 [37, 47]. For example, traditional 3-way replication used in GFS [17] and Azure [8] introduces 200% of redundancy overhead, while erasure coding can reduce the overhead to 33% and achieve higher availability [22]. Today's enterprise clustered storage systems [14, 22, 35, 39, 49] 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, 30], 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 [38]. 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 [17] and Azure [8] 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. In this paper, we make 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 implementation 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 perfor-

*The first two authors contributed equally to this work.

mance. 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× 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.

The rest of the paper proceeds as follows. In §2, we analyze the update behaviors in real-world traces. In §3, we introduce the background of erasure coding. In §4, we present different update schemes and describe our approach. In §5, we present the design of CodFS. In §6, we present testbed experimental results. In §7, we discuss related work. In §8, we conclude the paper.

2 Trace Analysis

We study two sets of real-world storage traces collected from large-scale storage server environments and characterize their update patterns. Motivated by the fact that enterprises are considering erasure coding as an alternative to RAID for fault-tolerant storage [40], we choose these traces to represent the workloads of enterprise storage clusters and study the applicability of erasure coding to such workloads. 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?*

2.1 Trace Description

MSR Cambridge traces. We use the public block-level I/O traces of a storage cluster released by Microsoft Research Cambridge [30]. 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 in various kinds of deployment including user home directories, project directories, source control, and media. Here, we choose 10 of the 36 volumes for our analysis. Each of the chosen volumes contains 800,000 to 4,000,000 write requests.

Harvard NFS traces. We also use a set of NFS traces (`DEAS03`) released by Harvard [13]. 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.

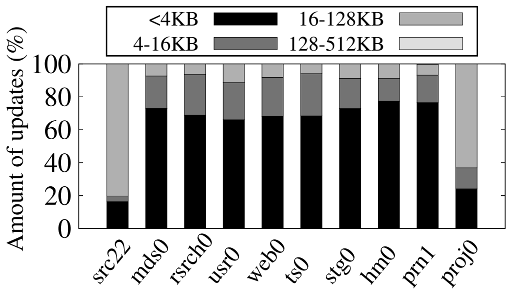


Figure 1: Distribution of update size in MSR Cambridge traces.

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 1: Properties of Harvard `DEAS03` NFS traces.

While the traces describe the workloads of a single NFS server, they have also been used in trace-driven analysis for clustered storage systems [1,20].

2.2 Key Observations

Updates are small. We study the update size, i.e., the number of bytes accessed by each update. Figure 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, all updates occurring in the traces are smaller than 512KB. Among the 10 traces, eight of them 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 1.

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 1 shows the results of the Harvard NFS traces. Among nearly 173 million write requests, more than 91% of them are updates. Table 2 shows the results of the MSR Cambridge traces. All the volumes show more than 90% of updates among all write requests, except for the print server volume `prn1`. We see limited relationship between the working set size (WSS) and the intensity of writes. For example, the project volume `proj0` has a small WSS, but it has much more writes than the source control volume `src22` that has a large WSS.

Update coverage varies. Although data updates are common in all traces, the coverage of updates varies. We measure the update coverage by studying the frac-

Volume	Workload Type	No. of Writes	WSS (GB)	Updated WSS(%)	Update Writes(%)
src22	Source control	805955	20.17	99.57	99.68
mds0	Media server	1067061	3.09	29.27	95.77
rsrch0	Research	1300030	0.36	69.53	97.41
usr0	Home directory	1333406	2.44	42.54	96.08
web0	Web/SQL server	1423458	7.26	37.25	96.23
ts0	Terminal server	1485042	0.91	49.84	95.65
stg0	Web staging	1722478	6.31	21.04	97.82
hm0	HW monitor	2575568	2.31	73.16	93.21
prn1	Print server	2769610	80.9	18.55	73.43
proj0	Project directory	3697143	3.16	56.67	98.89

Table 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.

tion of WSS that is updated at least once throughout the trace period. For example, from the MSR Cambridge traces in Table 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 Background: Erasure Coding

We provide the background details of an erasure-coded storage system considered in this work. We refer readers to the tutorial [33] for the essential details of erasure coding in the context of storage systems.

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* erasure coding, i.e., the original segment can be reconstructed from any k of the n data and parity chunks.

Each parity chunk can be in general encoded by computing a linear combination of the data chunks. Mathematically, for an (n,k)-code, let $\{\gamma_{ij}\}_{1\le i\le n-k, 1\le j\le k}$ 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 ($1 \le i \le n-k$) 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 $1 \le l \le k$) is updated to another data chunk D'_l. Then each new parity chunk P'_i ($1 \le i \le n-k$) 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 [33] 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.

4 Parity Updates

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.

We re-examine existing parity update schemes that fall into two classes: the RAID-based approaches and the delta-based approaches. We then propose a novel parity update approach that assigns a reserved space for keeping parity updates.

4.1 Existing Approaches

4.1.1 RAID-based Approaches

We describe three classical approaches of parity updates that are typically found in RAID systems [10,45].

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 [28]. 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 §3). 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 [41]. Second, logging introduces additional disk seeks to retrieve the updates, which often degrades sequential read and recovery performance [27]. 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 [15, 51], both approaches introduce significant network traffic since each update must transfer data or parity chunks between nodes for parity updates.

4.1.2 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 [9, 44]. A delta-based approach leverages the linearity of erasure coding described in §3. 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 [17] and Azure [8].

Parity-logging (PL). Parity-logging [24, 43] 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?*

4.2 Our Approach

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

4.2.1 An Illustrative Example

Figure 2 illustrates the differences of the delta-based approaches in §4.1.2 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

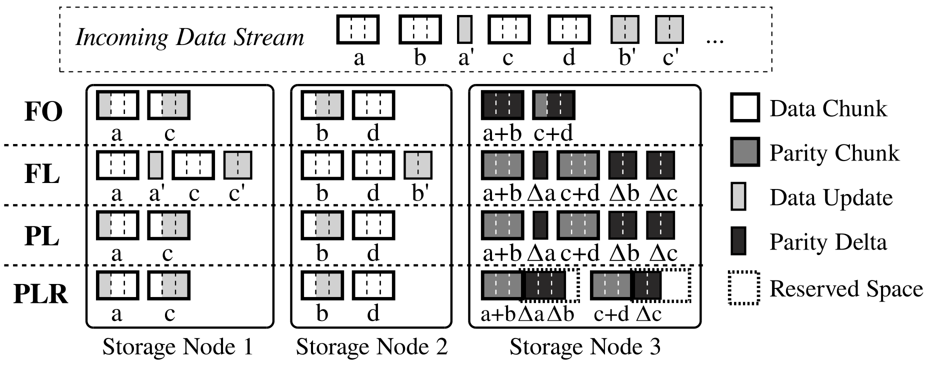


Figure 2: Illustration on different parity update schemes.

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 Δb 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 Δa and Δb are all placed in the contiguous reserved space following the parity chunk a+b.

4.2.2 Determining the 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 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.

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)
```

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 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 §6.3.2).

5 CodFS Design

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

5.1 Architecture

Figure 3 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.

5.2 Work Flow

CodFS performs erasure coding on the write path as illustrated in Figure 3. 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

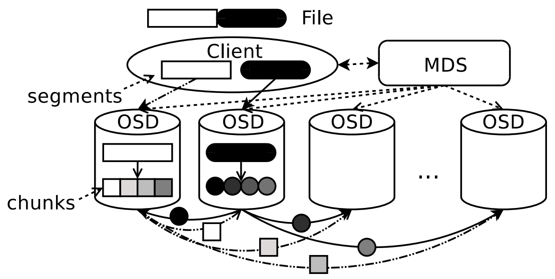


Figure 3: CodFS architecture.

$n - k$ parity chunks for some pre-specified parameters n and k. The primary OSD 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.

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., [25, 50]).

5.3 Issues

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

Consistency. CodFS provides close-to-open consistency [21], 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.

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.

Metadata Management. The MDS stores all metadata in a key-value database built on MongoDB [29]. CodFS can configure a backup MDS to serve the metadata operations in case the main MDS fails, similar to HDFS [5].

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.

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 §6.1). Smaller segments lead to more disk I/Os and degrade the write throughput, while larger segments cannot fully leverage the I/O parallelism across multiple OSDs.

5.4 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 [46], which manages a pool of threads that parallelize I/O and encoding operations, (2) Google Protocol Buffers [18], which serialize message communications between different entities, (3) Jerasure [32], which provides interfaces for efficient erasure coding implementation, and (4) FUSE [16], which provides a file system interface for clients.

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.

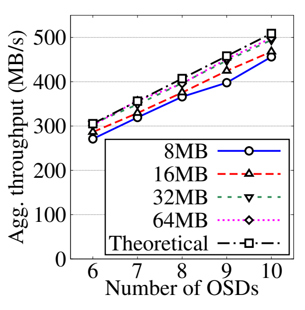
We adopt Linux Ext4 as the local filesystem of each OSD to support fast reserved space allocation. We preallocate the reserved space for each parity chunk using the Linux system call `fallocate`, which marks the allocated blocks as uninitialized. Shrinking of the reserved space is implemented by invoking `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.

6 Evaluation

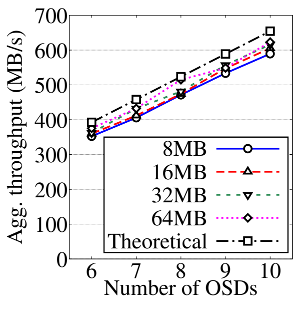
We evaluate different parity update schemes through our CodFS prototype. We deploy CodFS on a testbed with 22 nodes of commodity hardware configurations. Each node is a Linux machine running Ubuntu Server 12.04.2 with kernel version 3.5. The MDS and OSD nodes are each equipped with Intel Core i5-3570 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.

6.1 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

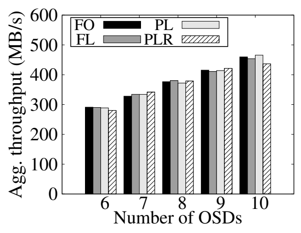


(a) Sequential write

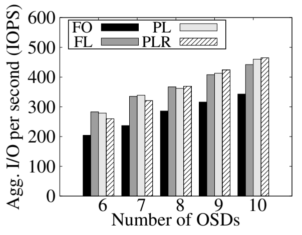


(b) Sequential read

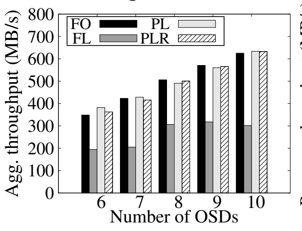
Figure 4: Aggregate read/write throughput of CodFS using the RDP code with $(n, k) = (6, 4)$.



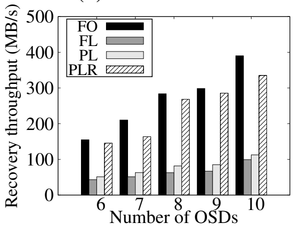
(a) Sequential write



(b) Random write



(c) Sequential read



(d) Recovery

Figure 5: Throughput of CodFS under different update schemes.

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, 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 §3.

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 $6 \le M \le 10$. We consider the RAID-6 RDP code [12] 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 4 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 also evaluate the throughput results of CodFS configured with the Reed-Solomon (RS) codes [34]. We observe that both RDP and RS codes have almost identical throughput, although RS codes have higher encoding overhead [32]. The reason is that CodFS masks the encoding overhead through parallelization. We do not present the results here in the interest of space.

6.2 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 [12] with $(n, k) = (6, 4)$, 16MB segment size, and the same cluster configuration as in §6.1. We measure the sequential write, random write, sequential read, and recovery performance of CodFS using IOzone [23]. For PLR, we use the baseline approach described in §4.2.2 and fix the size of reserved space to 4MB, which is equal to the chunk size in our configuration. We trigger a merge 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 [12] involves more updates to parity chunks than in RS codes (see [33] for illustration), and hence generates larger-size parity deltas. This triggers more frequent merge operations as the reserved space becomes full faster.

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

Table 3: Average non-contiguous fragments per chunk (F_{avg}) after random writes for synthetic workload.

6.2.1 Sequential Write Performance

Figure 5a 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.

6.2.2 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 §6.2.1. 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 5b 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.

6.2.3 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 3 shows the value of F_{avg} measured after random writes in §6.2.2. Both FO and PLR have $F_{avg} = 0$ as they either store updates and deltas in-place or 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 5c 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 §6.2.2.

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 3 where FL is the only scheme that contains non-contiguous fragments for data chunks.

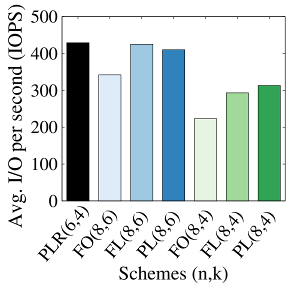
6.2.4 Recovery Performance

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

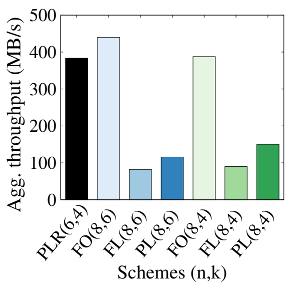
Figure 5d shows the measured recovery throughput for different update schemes. FO is the fastest in recovery and achieves substantial difference in recovery throughput (up to $4.5\times$) compared with FL due to the latter suffering from merging and disk seeking overhead for both data and parity chunks. 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.

6.2.5 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



(a) Random write



(b) Recovery

Figure 6: Throughput comparison under the same storage overhead using Cauchy RS codes with various (n, k).

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 [7] 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 6 shows the performance of random writes and recovery under the same synthetic workload described in §6.2.2. Results show that the $(8, 4)$ schemes perform significantly worse than the $(8, 6)$ schemes in random writes, since having more 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 §6.2.4, i.e., both FO and PLR give significantly higher recovery throughput than FL and PL.

6.2.6 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 3, 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.

6.3 Evaluation on Real-world Traces

Next, we evaluate CodFS by replaying the MSR Cambridge and Harvard NFS traces analyzed in §2.

6.3.1 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 [12] with $(n, k) = (6, 4)$ and 16MB segment size.

Update Performance. Figure 7 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 2) in general results in higher IOPS no matter which update scheme is used. For example, the `usr0` trace with 13.08% updated 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 §6.2.2 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 8 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

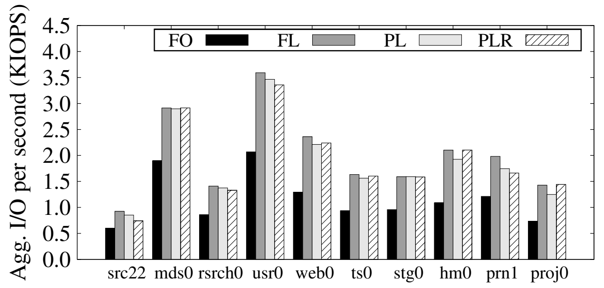


Figure 7: Number of write operations per second replaying the selected MSR Cambridge traces under different update schemes.

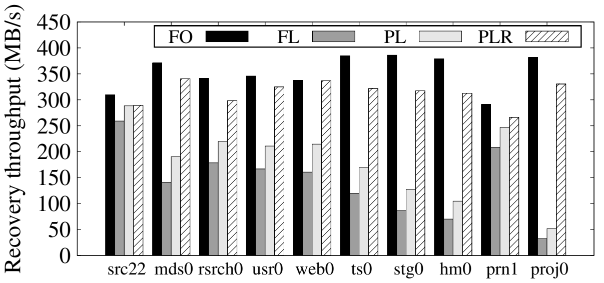


Figure 8: Recovery throughput in a double failure scenario after replaying the selected MSR Cambridge traces under different update schemes.

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.

6.3.2 Evaluation of Reserved Space

We evaluate our workload-aware approach in managing the reserved space size (see §4.2.2). 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 [7] 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* approach 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 §4.2.2 and set the smoothing factor $\alpha = 0.3$.

Simulation Results. Figure 9 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 10. 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 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.

6.3.3 Summary of Results

We show that PLR achieves efficient updates and recovery. It significantly improves the update through-

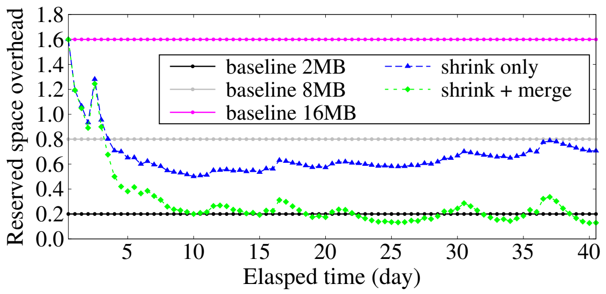


Figure 9: Reserved space overhead under different shrink strategies in the Harvard trace.

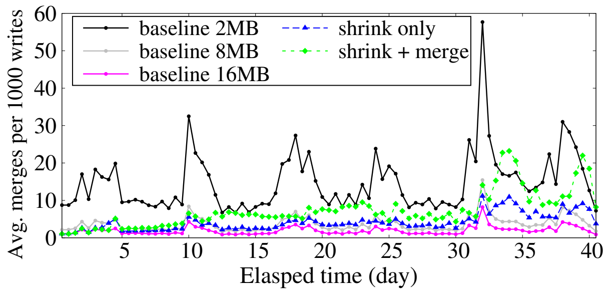


Figure 10: Average number of merges per 1000 writes under different shrink strategies in the Harvard trace.

put of FO and the recovery throughput of FL. We also evaluate 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.

7 Related Work

Quantitative analysis shows that erasure coding consumes less bandwidth and storage than replication with similar system durability [37, 47]. Several studies adopt erasure coding in distributed storage systems. OceanStore [26, 36] combines replication and erasure coding for wide-area network storage. TotalRecall [6] applies replication or erasure coding to different files dynamically according to the availability level predicted by the system. Ursa Minor [1] focuses on cluster storage and encodes files of heterogeneous types based on the failure models and access patterns. Panasas [49] performs client-side encoding on a per-file basis. TickerTAIP [9], PARAID [48] and Pergamum [44] offload the parity computation to the storage array. Azure [22] and Facebook [39] 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.

Log-structured File System (LFS) [38] first proposes to append updates sequentially to disk to improve write performance. Zebra [19] extends LFS for RAID-like distributed storage systems by striping logs across servers. Self-tuning LFS [27] exploits workload characteristics to improve I/O performance. Clustered storage systems, such as GFS [17] and Azure [8], also adopt the LFS design for the write-once read-many workload. The more recent work Gecko [42] uses a chained-log design to reduce disk I/O contention of LFS in RAID storage. 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. The above studies (including CodFS) focus on disk backends and commodity hardware, while the LFS design is also adopted in other types of emerging storage media, such as SSDs [3] and DRAM storage [31].

Parity logging [11, 43] has been proposed to mitigate the disk seek overhead in parity updates. It accumulates 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 [43]. 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.

8 Conclusions

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. To this end, we build CodFS, an erasure-coded clustered storage system that achieves efficient updates and recovery. We evaluate our CodFS prototype using both synthetic and real-world traces and show that PLR improves update and recovery performance over pure in-place and log-based updates. In future work, we plan to (1) evaluate other metrics (e.g., latency) of different parity update schemes, (2) evaluate the impact of the shrinking and merging operations on throughput and latency, and (3) explore a more robust design of reserved space management. The source code of CodFS is available for public-domain use on **http://ansrlab.cse.cuhk.edu.hk/software/codfs**.

Acknowledgments

We would like to thank our shepherd, Ethan L. Miller, and the anonymous reviewers for their valuable comments. This work was supported in part by grants AoE/E-02/08 and ECS CUHK419212 from the University Grants Committee of Hong Kong and ITS/250/11 from the ITF of HKSAR.

References

[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 Trans. on Storage*, 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] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

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

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

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

[20] 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.

[21] 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 Trans. Comput. Syst.*, 6:51–81, 1988.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] 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.

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

[34] 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.

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

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

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

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

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.