Delayed Parity Update for Bridging the Gap between Replication and Erasure Coding in Server-based Storage

Takayuki Fukatani
Tokyo Institute of Technology
Hitachi Ltd.
Tokyo, Japan
fukatani@de.cs.titech.ac.jp

Hieu Hanh Le
Tokyo Institute of Technology
Tokyo, Japan
hanhlh@de.cs.titech.ac.jp

Haruo Yokota
Tokyo Institute of Technology
Tokyo, Japan
yokota@cs.titech.ac.jp

ABSTRACT

In recent years, there has been a demand for using inexpensive general-purpose server-based storage in a wide range of applications. To use server-based storage in legacy applications such as databases and virtual desktop infrastructure where random access is dominant, it is important to enable low data access latency. Because common server-based storage uses erasure coding (EC) to reduce the capacity overhead of redundant data, the degraded write response time of EC data becomes a performance issue. The prior studies improve the write response time of server-based storage by storing frequently accessed data with replication that enables a smaller response time than EC. However, the difference in response time between replication and EC causes unstable write response time and leads to the usability degradation of target applications. We propose dynamic redundancy control with delayed parity update, which asynchronizes parity updates of EC data to achieve stable write response time for server-based storage. The proposed method redirects the write request for EC data to the replicated differential data and achieves the equivalent write response time with the replication. Our evaluations confirmed that our proposed method reduces the 99-percentile write response time to one-eighth of the conventional method and enables the stable write response time in server-based storage.

1 INTRODUCTION

With the progress of cloud computing environments, there is a trend of migrating legacy applications such as database (DB) and virtual desktop infrastructure (VDI) to cloud computing environments [11, 15]. In cloud computing environments, a large number of commodity servers form a shared resource pool of computing and storage and dynamically allocate those resources to applications. The cloud computing environments enable cost reductions compared with legacy systems using dedicated servers and storage systems.

Server-based storage used in a cloud computing environment is also required to support those legacy applications. Server-based storage is an inexpensive and scalable storage system consisting of a large number of commodity servers aggregated over a network [20, 23]. Server-based storage achieves the high throughput and high reliability required in cloud computing environments by distributing data among many servers and making them redundant. However, the network distributed architecture of server-based storage tends to be disadvantageous in terms of random performance. Large network delay during read and write processing degrades the random performance of server-based storage. Traditionally, a legacy application uses dedicated storage appliances or directly attached storage (DAS) of the server where the application runs. The storage appliances and DAS have been improved in random performance to adapt to legacy application workloads where random access is dominant. By contrast, traditional server-based storage is unsuitable for legacy applications because of its low random performance.

Dynamic redundancy control (DRC) has been proposed as a method to improve the random access performance of server-based storage [5]. DRC alleviates the performance disadvantages of server-based storage by storing data in the same server as applications rather than distributing data over the network. In addition, DRC uses high-performance replication for frequently accessed data and capacity-efficient erasure coding (EC) for infrequently accessed data as a redundancy method. By dynamically controlling redundancy methods based on data access characteristics, DRC achieves high random performance while maintaining the capacity efficiency of conventional server-based storage.

However, DRC still has a performance issue of poor write response time for EC data. DRC enables a small write response time for replicated data. However, the write response time for EC data remains large because of the overhead of updating parity data. Because of the difference in write response times between replication and EC, the access response time for applications also becomes unstable. In large-scale systems, the unstable response time of storage access leads to usability degradation and becomes a problem [3].

In this paper, we propose dynamic redundancy control with delayed parity update (DRC-DPU) to improve the degraded write response time for EC data in DRC. DRC-DPU improves the write response time by performing parity update asynchronously with write request processing. DRC-DPU redirects write requests for EC data to replicated differential data so that it can merge the differential data to EC data later in the background. Furthermore, DRC-DPU consolidates parity updates for the same EC data to reduce the overhead of parity updates by delaying the differential data merge for frequently accessed data.

To confirm the effectiveness of the proposed method, we conducted performance evaluations using the synthesized workload. The evaluations showed that DRC-DPU reduces the 99-percentile write response time to one-eighth of DRC and enables the stable write response time in server-based storage. Furthermore, we evaluated the effectiveness of the proposed method in reducing the

number of parity updates and the metadata capacity overhead using the real-world workload. The evaluations showed that DRC-DPU reduces up to 20% parity updates and consumes less than 0.1% of total capacity for the metadata under the real-world workloads.

The main contributions of this paper are listed as follows.

- We propose a method to improve the degraded write response time of EC data in DRC by asynchronizing parity updates of EC data.
- We propose a method to reduce the EC overhead by consolidating parity updates for the same EC data.
- We demonstrate that our proposed method achieves a better write response time than conventional methods.
- We demonstrate the effectiveness of our proposed method in parity update reduction and metadata capacity overhead.

The remainder of this paper is structured as follows. In Section 2, we introduce the background knowledge of our study. In Section 3, we describe our motivation and approaches. In Section 4, we present the design of DRC-DPU. In Section 5, we present the evaluation results. In Section 6, we provide the related work of our proposal. In Section 7, we conclude the paper.

2 BACKGROUND

There is a trend of using server-based storage as back-end storage of legacy applications such as DB and VDI [11, 15, 27]. This section provides background knowledge of the use of server-based storage in legacy applications. Then, we discuss conventional approaches for improving the random performance of server-based storage.

2.1 Applying Server-based Storage to Legacy Applications

Server-based storage is a storage system that consists of a large number of general-purpose commodity servers. Server-based storage achieves high throughput and high reliability by distributing and redundantly storing data among servers. Server-based storage uses commodity servers, making it possible to build a salable storage system at a lower cost than conventional dedicated storage appliances. Server-based storage is becoming more common in the cloud computing environments and also in on-premise enterprise systems as a practical alternative to expensive storage appliances [21, 22].

With the proliferation of server-based storage, the use of server-based storage has also expanded from traditional cloud applications such as data analysis and archiving to legacy applications. As a result, server-based storage must support legacy applications.

Server-based storage usually stores data among servers over the network in a redundant and distributed manner. Parallel access between servers and service fail-over in the case of a server failure yields high throughput and fault tolerance on unreliable commodity servers. However, in terms of access response time, data distribution and redundancy over the network are disadvantageous to conventional storage appliances and DAS. Network communication and redundant processing prolong access response time and deteriorate random access performance.

Another issue with server-based storage is the decreased capacity efficiency because of data redundancy. Early server-based storage used three-way replication as a redundancy method for unreliable commodity servers [20]. Three-way replication requires three times the original capacity to hold two replicas of the same capacity as the original data. To improve capacity efficiency, many distributed storage implementations adopt EC [4, 9, 10]. EC constructs a code from an arbitrary number of data symbols and an arbitrary number of parity symbols. EC improves capacity efficiency by increasing the ratio of data to parity data [19].

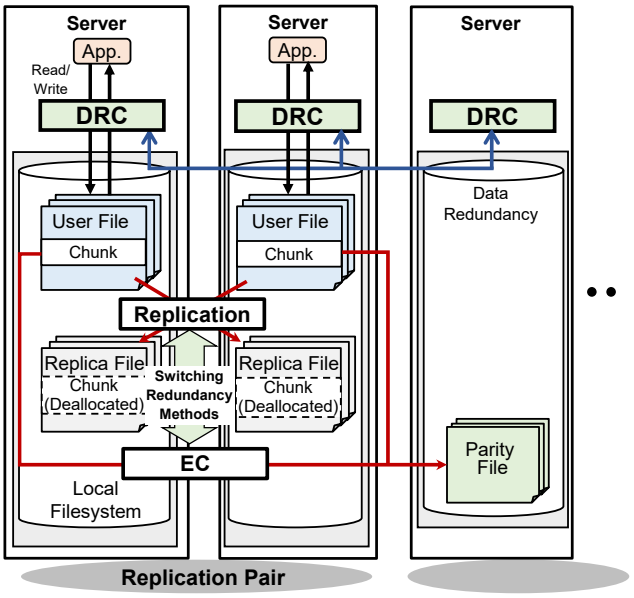


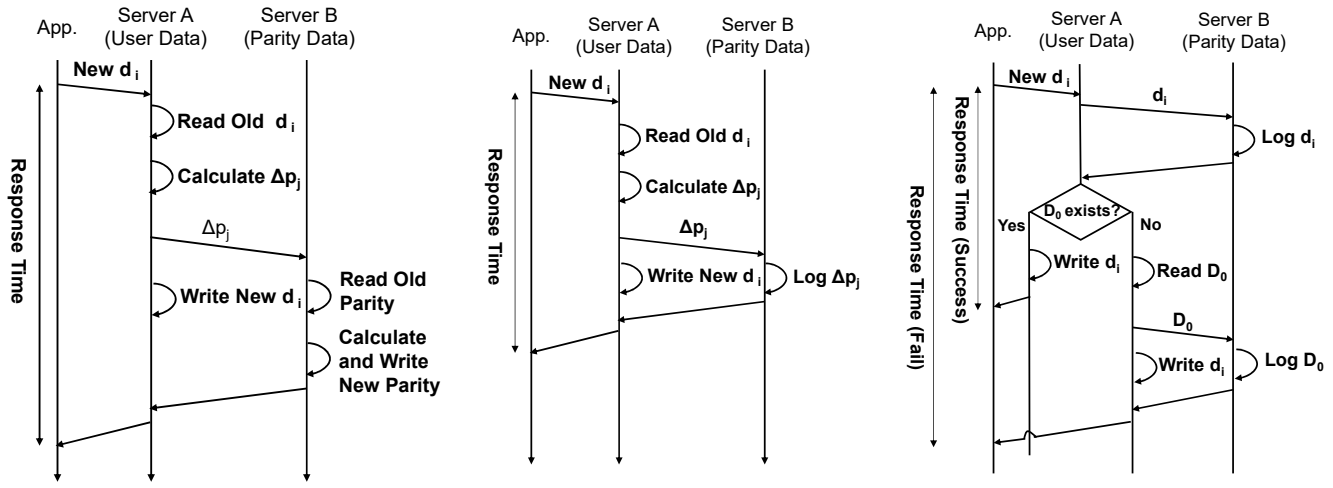
Figure 1: DRC Overview

Although EC enables high capacity efficiency, write performance degrades because parity updates become another performance issue. When a part of an EC stripe is updated, the parity data also need to be partially updated. Updating partial parity data requires reading the old data and the old parity from disks, calculating the new parity, and writing the new parity data to disks. This parity update process involves a considerably large amount of processing. Thus, the processing delay and processing load degrade the random write performance further.

The low random performance of server-based storage becomes a problem when using server-based storage in legacy applications where random access is dominant.

2.2 Conventional Approaches

2.2.1 Dynamic Redundancy Control. DRC has been proposed as a method to improve the random performance of server-based storage [5]. DRC is a redundancy control module that makes local data in each server redundant between servers. DRC places an application and its data on the same server and reduces the overhead of network communication. In addition, DRC also dynamically switches the data redundancy method between replication and EC, depending on the performance requirements and workload characteristics. DRC achieves high capacity efficiency while maintaining high performance. Figure 1 shows an overview diagram of DRC.



(a) Naïve Method (b) Parity Logging (c) Speculative Partial Write

Figure 2: EC Write Methods

In DRC, two or more servers form a replication pair and replicate local data to each other. DRC divides a file into 512 KB fixed-length chunks and switches the redundancy method of each chunk between replication and EC. DRC copies local files to one or more other servers for replicated data, whereas it also applies EC to chunks in the local files of multiple servers to reduce the amount of redundant data. DRC records the number of write accesses for each chunk and reduces the capacity of redundant data by encoding chunks with high update frequency. DRC keeps frequently accessed chunks as replicated data to mitigate the degradation of random write performance caused by EC.

Furthermore, DRC reduces network communication during metadata access by storing the metadata of the EC configuration in the local file system. DRC stores metadata of chunk and parity configuration in the custom file metadata, which is an extension of the local file system [6]. By storing metadata in the local file system, DRC avoids the metadata server becoming a performance bottleneck as in conventional methods [7, 25].

2.2.2 EC Write Optimizations. Several EC write optimizations have been proposed to improve the random write performance of EC. In the following, we describe the naïve method of EC write and explain two other prior methods: parity logging [2] and speculative partial write (SPW) [27].

Naïve method. In EC, parity data are calculated from original data and the coefficient matrix $A = [a_{ij}]_{m \times k}$ with the following equation [19].

$$(p_1, \cdots, p_k)^T = A(d_1, \cdots, d_m)^T \quad (1)$$

where p_n represents the n-th parity data symbol, and d_n represents the n-th data symbol.

When updating EC data, parity data of the same code must also be updated at the same time. If the new data do not cover the entire data of an EC stripe, the parity data cannot be calculated only from the new data. In that case, the delta of the parity data is calculated from the new data and the old data for linear codes such as Reed–Solomon codes as follows.

$$\Delta p_j = a_{ij} \times \Delta d_i \quad (2)$$

where Δp_j denotes the difference between the old p_j and the new p_j, Δd_i denotes the difference between the old d_i and the new d_i, and $a_{ij} \in A$. The new parity can be obtained by adding Δp_j to the old p_j.

Figure 2a shows the process flow of the naïve method. The figure shows the communication between the servers when the application performs a partial write to EC data stored in Server A. Server A, which receives the write request from the application, reads the old data from the disks and calculates Δp_j. Then, Server A sends Δp_j to Server B where the parity data are stored. Then, while the new data are written to the disks in Server A, the new parity data are calculated from the old parity data and Δp_j and written to the disks in Server B.

Parity Logging. Parity logging eliminates the need to read the old parity data from the disks in the naïve method by logging the delta data of old and new parity data [2]. The failure recovery process recovers parity data from the old parity data stored on the disks and the delta log of parity data. Parity logging reduces the overhead of parity updates by reducing the number of disk reads. Figure 2b shows the processing flow of parity logging.

As the figure shows, Δp_j is logged without reading the old parity on Server B. Parity logging avoids reading the old parity data from the disks and allows faster response time compared with the naïve method.

Speculative Partial Write. SPW eliminates the need to read old data from the disks in parity logging by logging new data instead of the delta of parity data. When a failure occurs, the failure recovery process uses Equation 2 to calculate the sequence of Δp_i from the

logged new data in order. The failure recovery process calculates the latest parity data by adding all Δp_i to the old parity data stored on the disks. If the first data are not logged on the parity server when new data are logged, SPW reads the old data from the disks and logs them with the new data. If the first data are logged in the parity server, SPW does not need to read the old data from disks, and the response time improves. Figure 2c shows the process flow of SPW.

Here, D_0 indicates the first data. SPW behaves differently depending on whether D_0 exists on Server B. If D_0 already exists on Server B, SPW handles it as a speculation success and does not read the old data from the disks. By contrast, if D_0 does not exist on Server B, SPW handles it as a speculation failure. SPW reads the old data as D_0 on Server A and sends it to Server B. In the case of a speculation success, reading old data becomes unnecessary on Server A, and response time improves. However, in the case of a speculation failure, another log write is required on Server B, and the response time degrades.

SPW is considered to be effective in real-world workloads where high write locality is expected.

2.3 Challenge

DRC improves the random access performance of server-based storage. DRC reduces the overhead of network communication by storing data in the local storage of servers. DRC also reduces the number of writes to EC data by encoding data with low-access frequencies. These improvements result in high throughput and lower average response time.

However, the write response time of EC data in DRC is considerably worse than replicated data because of the overhead of parity updates. DRC uses the naïve method for the parity update process; therefore, the write response time of EC data is much worse than that of replication. One possible solution is to use parity logging or SPW instead of the naïve method.

As a preliminary experiment, we evaluated the write response time of the conventional methods. In the preliminary experiment, we evaluated the difference in write response times of the naïve method, parity logging, SPW, and DRC compared with replication. For the experiment, we implemented parity logging and SPW to the DRC prototype [5]. We measured the cumulative distribution function of write response time in the same configuration as in Section 5.1.1. We used fio [1] for benchmarking. One thread issued 8 KB random writes to a 48 GB file. In DRC measurements, 20% of the data were encoded into EC, and therefore 20% of the accesses were issued to the EC data, and 80% of the accesses were issued to the replication data. Figure 3 shows the evaluation results.

The write response time of DRC is equivalent to replication for 80% of writes to replicated data and close to the naïve method for the remaining 20% of writes to EC data. Although parity logging and SPW show better response time than the naïve method, they are much worse than replication.

As shown, even with the prior EC write optimizations, the write response time of EC data is considerably degraded compared with replication. Even if DRC uses these conventional methods, the poor write response time of EC data remains an issue. There remains a challenge to improve the write response time for EC data to the same level as the write response time for replicated data.

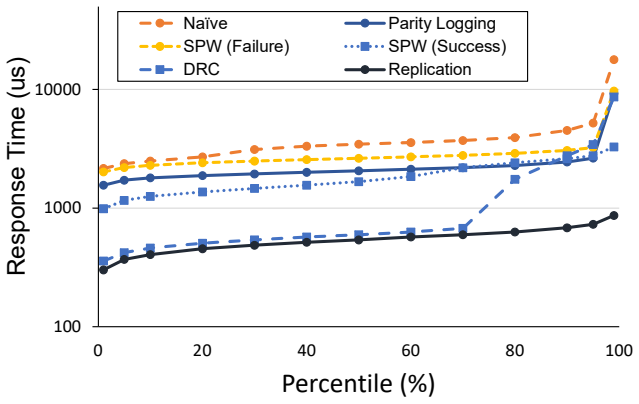


Figure 3: Write Response Time Comparison of Conventional Methods

3 MOTIVATION AND APPROACHES

We aim to solve the problem of the degraded write response time of DRC and enable stable storage access in server-based storage. The goal of this study is to make the write response time of EC data in DRC equivalent to replication.

We take the following two approaches to achieve the goal.

Redirecting EC Writes to Replicated Differential Data. The first approach introduces redirecting control of write requests for EC data to the replicated differential data. This approach enables the delayed parity update by asynchronously merging the difference data to the EC data. By responding immediately after updating the replicated differential data, this approach makes the write response time of EC data equivalent to that of replication. Furthermore, the proposed method controls the amount of EC data to compensate for the capacity increase caused by the differential data.

Parity Update Reduction Using Temporal Write Locality. The second approach uses the temporal write locality to reduce the number of parity updates in the EC write processing. This approach reduces the number of parity updates by consolidating parity updates to the same EC data by delaying the differential data merge for frequently accessed data. This approach reduces the overhead of parity updates and reduces the negative impact of the EC overhead on write request processing.

4 DYNAMIC REDUNDANCY CONTROL WITH DELAYED PARITY UPDATE

In this section, we propose DRC-DPU to improve the degraded write response time of EC data in DRC.

4.1 Overview

DRC-DPU improves the write response time of EC data by asynchronizing parity updates. Figure 4 shows an overview diagram of DRC-DPU.

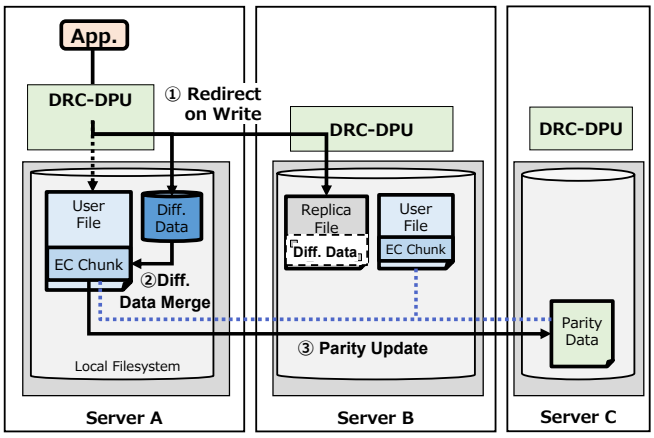


Figure 4: DRC-DPU Overview

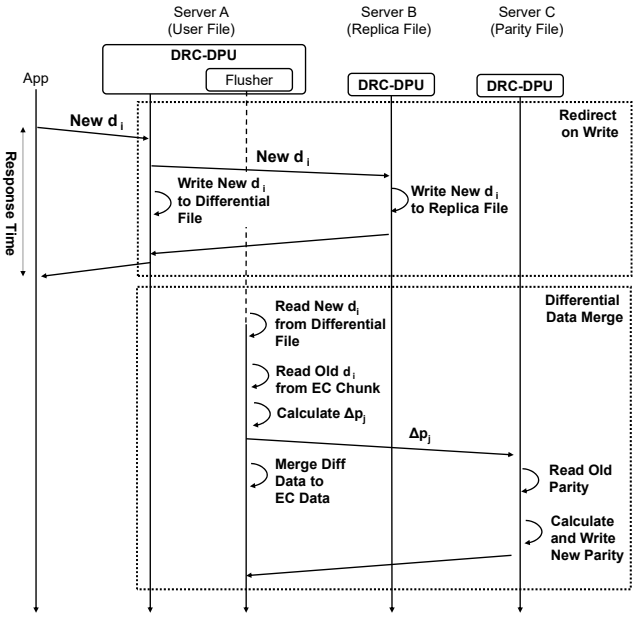


Figure 5: Flow of Delayed Parity Update

DRC-DPU is a stackable module on top of the local filesystem in the same way as DRC. One implementation of DRC-DPU is a filesystem-in-the-user-space (FUSE)-based module with the Linux filesystem such as ext4 or XFS [12, 13, 24]. It uses the standard Linux interface for communication with the Linux filesystem, enabling it to run on any commodity server that supports Linux and FUSE.

DRC-DPU adds controls called "redirect on write" and "differential data merge" to DRC to enable asynchronous parity update.

In redirect on write, DRC-DPU redirects write for an EC chunk to a special file called the differential file. DRC-DPU improves the write response time by responding to the application immediately after writing the new data to the differential file.

In differential data merge, DRC-DPU asynchronously merges the differential data in a differential file to the EC chunks. DRC-DPU retains a certain amount of differential data and consolidates multiple parity updates for the same EC data to reduce the number of parity updates. As with traditional write buffer techniques, differential data merge takes advantage of temporal write locality of the real-world workload [8, 18].

Figure 5 shows the entire processing flow of DRC-DPU.

In the following, we explain the details of the redirect on write and differential data merge.

4.2 Redirect on Write

DRC-DPU redirects a write request to an EC chunk to the differential file. The differential file is a file for storing differential data of EC chunks. The differential file is prepared for each user file that has EC chunks, and the entity is an empty regular file. DRC-DPU guarantees the reliability of the differential data by replicating it to a replica file on the replication pair. DRC-DPU uses a differential bitmap to record the existence of differential data for each EC chunk. Figure 6 shows the data layout of DRC-DPU.

DRC-DPU uses the custom file metadata [6], which is an extension of the Linux standard file metadata, to manage metadata of the differential data. The custom file metadata is additional file metadata of arbitrary size for user files. The custom file metadata is stored in a file called the metadata file. DRC-DPU stores the file handle of the differential file and the differential bitmap in the custom file metadata for reading and writing the differential data.

DRC-DPU subdivides a 512 KB chunk, which is the management unit of the DRC, into 4 KB pages and manages the differential data on a page basis. DRC-DPU writes the differential data of the EC chunk to the differential file with the same file offset and size. The differential bitmap is a bitmap that records the presence of differential data for each page of the chunk. The differential bitmap is a 128-bit bitmap with bits for each 4 KB page that makes up a 512 KB chunk. DRC-DPU turns on the corresponding bit of the differential bitmap when the differential page is added to the differential file.

In addition, DRC-DPU performs read–modify–write processing for writes smaller than the page size. When DRC-DPU receives a small write to a page that does not have the differential data, it complements the rest of the data in the page with the read–modify–write processing. In the read–modify–write processing, DRC-DPU reads the old data of the written page from the EC chunk, combines it with the new data, and writes it to the differential file as a differential page.

The read–modify–write processing causes a performance overhead. The number of the read–modify–write processing can be reduced using the smaller page size. However, the smaller page size increases the metadata capacity consumption of differential bitmaps. We evaluate the performance impact of the read–modify–write process and the trade-off between performance and capacity consumption on different page sizes in Section 5.

When DRC-DPU receives a read request for an EC chunk, it examines the differential bitmap and switches the read target. DRC-DPU reads data from the differential file when the differential bit is on and from the EC chunk when the differential bit is off. DRC-DPU merges the data in the EC chunk with the differential data and returns the merged data to the applications

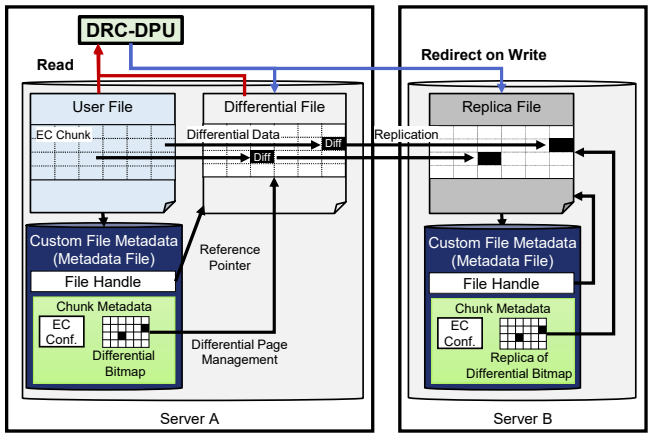


Figure 6: Data Layout in DRC-DPU

Furthermore, DRC-DPU replicates the differential data to the replication pair that contains the replica file during redirect on write. DRC-DPU sends the write request to the replication pair, and DRC-DPU on the replication pair updates the replica of the differential data and the differential bitmap. DRC-DPU on the replication pair writes the replica of the differential data into the replica chunk whose capacity is de-allocated when the EC chunk is encoded. DRC-DPU on the replication pair updates the replica of the differential bitmap at the same time as the replica of the differential data. In the event of a server failure, DRC-DPU decides whether to rebuild the EC chunk or read a replica of the differential data based on the replica of the differential bitmap.

4.3 Differential Data Merge

After responding to the write request, DRC-DPU performs differential data merge in the background to merge the differential data into EC chunks. DRC-DPU periodically starts an internal thread called Flusher in the background to perform differential data merge. Flusher copies the differential pages in the differential file to the EC chunk of the user file and then releases the capacity of the differential pages. DRC-DPU performs parity updates in the same way as a normal write to an EC chunk in DRC.

DRC-DPU uses the least recently used (LRU) list of differential pages to delay differential data merge for recently updated differential pages. DRC-DPU reduces the number of parity updates when the same differential page is updated multiple times.

DRC-DPU reserves a certain amount of capacity for storing differential pages. DRC-DPU delays differential data merge until the total capacity of the differential pages reaches the differential capacity ratio B. If the total capacity ratio of the differential pages exceeds B, DRC-DPU merges the differential pages with the oldest update to EC chunks.

To prevent the decrease in capacity efficiency because of differential pages, DRC-DPU increases the ratio of EC data and reserves the capacity for differential pages. DRC-DPU determines the capacity ratio of the EC data based on the following equation for the original EC target rate R.

Table 1: Experimental Environment

Items	Settings
Machine	HP Proliant DL 160 G6 x 3
CPU	Intel (R) Xeon E 5620 2.40 GHz, 4 core x 2
Memory	12GB
Disk	HP SATA SSD x 2,
Network Port	HP NC550SFP 10 GbE Server Adapter
OS	Ubuntu 16.04.4 with XFS
Benchmark tool	fio 2.2.10

$$ECRatio = R + B * r/(r - (m + k)/m) \quad (3)$$

where r is the replication factor, m is the number of data symbols, and k is the number of parity symbols. Based on the evaluation using the real-world workloads in Section 5.2, the default value for the differential capacity ratio B is set to 0.5%.

Per-page LRU list management enables finer-grained access monitoring compared with the conventional chunk-based redundancy control. DRC-DPU encodes only low-access frequency chunks into EC data. However, if access is concentrated on a small number of pages in a chunk, chunk-based monitoring might determine that the chunk is a low-access frequency even if the access frequency of the pages is high in page-based monitoring. DRC-DPU treats such frequently accessed pages in EC chunks as replicated differential pages. DRC-DPU reduces parity updates of these pages and the performance overhead of EC.

5 EVALUATION

In this section, we confirm the effectiveness of DRC-DPU. In Section 5.1, we evaluate the performance improvement of the proposed method using the synthesized workload. In Section 5.2, we evaluate the parity update reduction and metadata capacity overhead of the proposed method using the real-world workload.

5.1 Synthesized Workload Evaluation

5.1.1 Experimental Environment. Table 1 shows the experimental environment. We used three commodity servers and ran the prototype of proposed and conventional methods. We implemented parity logging, SPW, and DRC-DPU in the DRC prototype, which we developed in the prior study [5]. We ran the benchmarking tool *fio* on one of the servers [1]. As a redundancy method, we used two-way replication and 2D1P EC.

5.1.2 Experimental Results. We evaluated write response time and throughput under random write workload for the proposed method and the conventional methods. We evaluated the performance degradation from replication for the naïve method, parity logging, SPW, DRC, and DRC-DPU. For SPW, we evaluated both the success case and the failure case. In DRC and DRC-DPU measurements, 80% of the data were replicated, and 20% of the data were encoded into EC. Because *fio* has no access locality, 80% of the data accesses were issued to the replicated data and 20% to the EC data. As high write locality is expected in the real world, the larger amount of

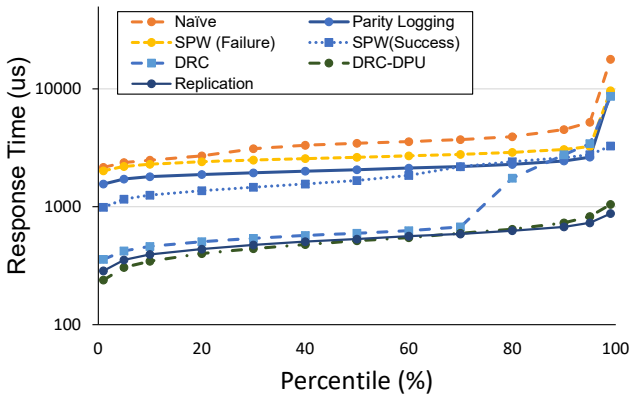


Figure 7: Comparison of Write Response Time (8 KB IO)

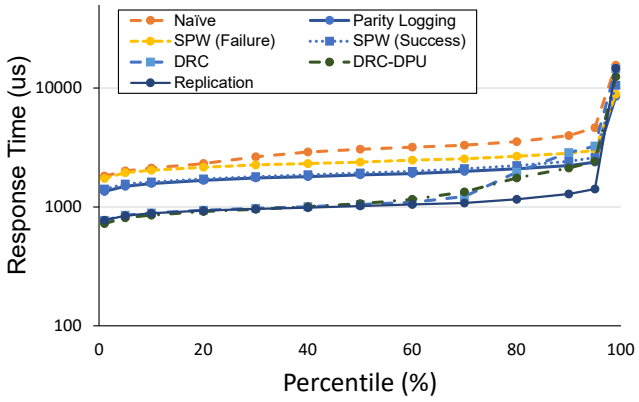


Figure 8: Comparison of Write Response Time (2 KB IO)

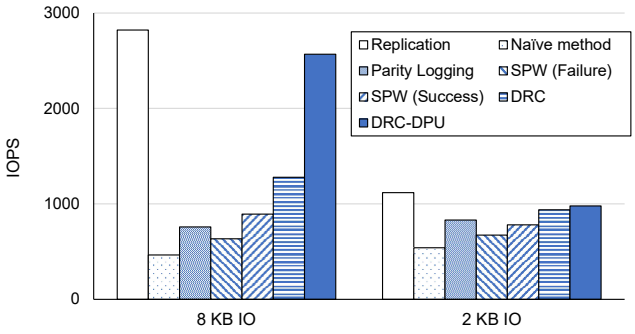


Figure 9: IOPS Comparison

data can be encoded for the same EC data access ratio in actual use cases [26]. To evaluate the performance impact of the read–modify–write processing, we evaluated 8 KB random write, which does not require the read–modify–write processing, and 2 KB random write, which requires the read–modify–write processing. One thread issued random writes with the target size to a 48 GB file. Figure 7–9 show the evaluation results.

Table 2: Write Access Characteristics in Evaluated Trace Periods

Trace	Number of Writes	Average Size	Number of Accessed Chunks
hm_0	348,074	7.79 KB	3,947
prn_1	401,281	14.8 KB	27,626
proj_1	8,458,027	13.7 KB	10,953
prxy_1	473,201	9.44 KB	36,920

Figure 7 shows that DRC-DPU achieves a nearly equivalent write response time to replication under an 8 KB random write workload. The write response time of DRC-DPU is almost the same as that of replication, even for the 20% capacity on the right side where EC data access occurs. As a result, DRC-DPU improved the 99-percentile response time from 8.6*ms* to 1.0*ms*, which is one-eighth of the conventional DRC. The response time of DRC-DPU is also superior to other conventional methods.

By contrast, Figure 8 shows that the write response time of DRC-DPU under 2 KB random write is degraded against replication because of the read–modify–write processing. However, even with the read–modify processing, DRC-DPU still shows better response time than DRC. For 99-percentile response time, all methods, including replication, show a large response time. This is presumably because of the performance characteristics of the local file system.

Figure 9 shows that the throughput degradation of DRC-DPU against replication is only 9% at 8 KB and 13% at 2 KB. In addition, DRC-DPU shows a higher throughput than other conventional methods.

5.2 Real-world Workload Evaluation

In this section, we evaluated the efficiency of the proposed method using the real-world workload. We used Microsoft Research Cambridge (MSR) traces, which is 1-week I/O traces of enterprise servers, for the evaluations [16]. We evaluated the reduction in the number of parity updates and the metadata capacity overhead of the proposed method.

5.2.1 Parity Update Reduction. To verify the effectiveness of differential data merge, we evaluated the reduction of the number of parity updates. We examined the change in the number of parity updates when the total capacity of the differential pages is changed.

We used the MSR traces with a large number of writes: hardware monitoring (hm_0), print server (prn_10), project files (proj_1), and firewall/web proxy (prxy_1). We assumed that the same accesses with the MSR traces were issued to virtual disk image files of virtual machines on a system using DRC-DPU. We used the first six days in the MSR traces to encode low-access frequency chunks into EC chunks. We simulated the change in the number of parity updates for the remaining day in the MSR traces. Table 2 shows the write access characteristics of these traces for the evaluated trace periods.

We also set the EC target rate R to 0.9 and used Equation 3 to determine the total amount of EC data. We examined the number of parity updates for the different capacity ratios B of differential

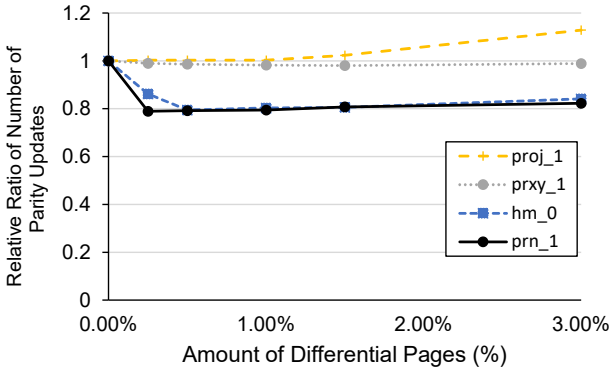


Figure 10: Parity Update Reduction in MSR Traces

pages to the total amount of chunks accessed during the trace period. Figure 10 shows the evaluation results.

DRC-DPU reduces parity updates by about 20% for hm_0 and prn_1. DRC-DPU largely reduces the number of parity updates at $B = 0.5\%$, and no significant effect is seen with the higher values of B. As there is a large write locality in these traces, the effect of differential data merge becomes reasonably large.

By contrast, there is no significant improvement in prxy_1 and proj_1. The reason for the poor results in prxy_1 and proj_1 is thought to be because of fewer writes to the same EC chunks. In addition, the larger amount of EC data to secure the capacity of the differential pages seems to increase the number of parity updates conversely.

These results show that the proposed method achieves a considerably large parity update reduction for workloads with write locality. For these workloads, 0.5% differential pages are considered sufficient.

5.2.2 Metadata Capacity Overhead. We evaluated the capacity overhead of the metadata used in DRC-DPU.

First, we evaluated the capacity consumption of the custom file metadata for DRC and DRC-DPU with different page sizes of differential data. We calculated the capacity consumption of the custom file metadata for an EC stripe when using 2D1P and 6D2P EC. Then, we investigated the ratio of metadata to the total capacity consumption. Tables 3 and 4 show the evaluation results. The tables show the capacity consumption of user data, parity data, metadata, and the metadata ratio to the total capacity consumption for each method. The size in parentheses indicates the page size.

The results show that the use of smaller page sizes increases the metadata capacity consumption. However, even if the page size is 1 KB, the metadata ratio in the capacity consumption is 0.097% in the 2D1P configuration and 0.056% in the 6D2P configuration. These results indicate that the impact of adding the custom file metadata on capacity efficiency is sufficiently small.

Next, we examined the impact of different page sizes on performance. In the evaluation, we examined the occurrence frequency of read–modify–write processing in the simulations using MSR trace as in the previous section. Figure 11 shows the evaluation result of the occurrence frequency of read–modify–write processing.

Table 3: Metadata Capacity Overhead of DRC-DPU (2D1P)

	User Data	Parity Data	Meta-data	Meta-data Ratio
DRC	1.0 MB	0.5 MB	0.38 KB	0.024%
DRC-DPU (1 KB)	1.0 MB	0.5 MB	1.5 KB	0.097%
DRC-DPU (2 KB)	1.0 MB	0.5 MB	0.95 KB	0.061%
DRC-DPU (4 KB)	1.0 MB	0.5 MB	0.66 KB	0.042%
DRC-DPU (8 KB)	1.0 MB	0.5 MB	0.52 KB	0.033%

Table 4: Metadata Capacity Overhead of DRC-DPU (6D2P)

	User Data	Parity Data	Meta-data	Meta-data Ratio
DRC	3.0 MB	1.0 MB	1.2 KB	0.030%
DRC-DPU (1 KB)	3.0 MB	1.0 MB	2.4 KB	0.057%
DRC-DPU (2 KB)	3.0 MB	1.0 MB	1.8 KB	0.043%
DRC-DPU (4 KB)	3.0 MB	1.0 MB	1.5 KB	0.036%
DRC-DPU (8 KB)	3.0 MB	1.0 MB	1.4 KB	0.033%

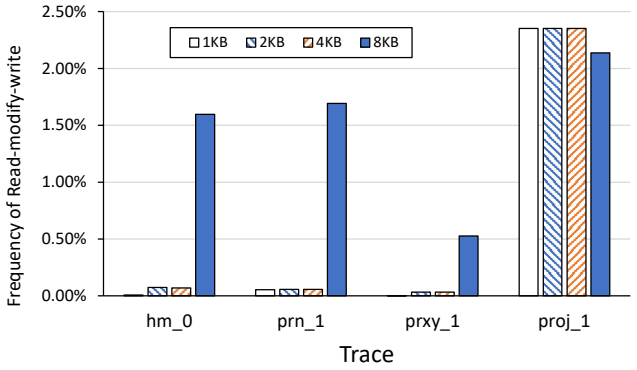


Figure 11: Evaluation of Number of Read–modify–write Processes

The evaluation results show that the occurrence frequency of read–modify–write processing increases as the page size increases. In particular, when the page size is set to 8 KB, the number of read–modify–write operations increases significantly for three out of four traces, while it stays almost the same in prj_1. In prj_1, all writes are not 4 KB aligned, and read–modify–write processing occurs regardless of page sizes, while most writes are 4 KB aligned in other traces. By contrast, if the page size is less than 4 KB, there are fewer differences in the number of read–modify–write processing.

These results indicate that the metadata capacity consumption of DRC-DPU is sufficiently small. The 4 KB page adopted in DRC-DPU is considered reasonable in terms of both metadata capacity consumption and performance.

6 RELATED WORK

There has been extensive research on write buffers using volatile memory [8, 18]. These studies have achieved improved performance using volatile memory as the write buffer. DRC-DPU can be seen as using the differential data as the write buffers, like the prior write buffer techniques. However, DRC-DPU stores the differential data in free space, while the prior write buffer techniques use the dedicated devices for the write buffer. DRC-DPU achieves performance improvement without additional devices using the free space for the write buffer.

Much research has been performed in the area of hierarchical storage control between different storage devices. Ceph provides a write-proxy function that temporarily writes the write data to the upper tier and updates the lower tier with a delay in a similar way as the proposed method [21]. However, Ceph has a fixed upper tier and lower tier capacity, making it difficult for Ceph to dynamically change tier capacity. DRC-DPU stores all data types in the same file system, and the capacity of each data type can be changed dynamically. DRC-DPU has an advantage over Ceph in terms of capacity efficiency because it stores all types of data in the same shared storage space.

File metadata extension has been studied in the field of network-attached storage to extend the capability of filesystems. Several studies use additional file metadata for managing referencing pointers between files to enable cloud backup, data migration, and data redundancy across servers [5, 14, 17]. Network-protocol-specific metadata is another application of file metadata extension to achieve higher protocol coverage of network protocol servers [6]. The proposed method extends file metadata to manage differential data to improve EC write performance. This study can be seen as another application of the file metadata extension.

7 CONCLUSION

In this paper, we propose DRC-DPU to improve the degraded write response time of DRC in server-based storage. DRC-DPU improves the write response time by asynchronizing the parity update during EC write processing. DRC-DPU improves the write response time of EC data to the same level as replication.

In the performance evaluation, we confirmed that the proposed method achieves almost the same write response time as replication, which is one-eighth of the 99% response time of the conventional method. In addition, the evaluation using the real-world workload confirmed that the proposed method achieves considerable parity update reduction and reasonable metadata capacity consumption.

REFERENCES

[1] 2021. Flexible IO Tester. https://github.com/axboe/fio

[2] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. 2014. Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 163–176.

[3] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[4] Andrew Fikes. 2010. Storage architecture and challenges. *Talk at the Google Faculty Summit* (2010).

[5] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. 2019. Lightweight dynamic redundancy control for server-based storage. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS 2019)*. IEEE, 353–369.

[6] Takayuki Fukatani, Atsushi Sutoh, and Takahiro Nakano. 2018. A method to adapt storage protocol stack using custom file metadata to commodity linux servers. *International Journal of Smart Computing and Artificial Intelligence (IJSCAI)* 2, 1 (2018), 23–42.

[7] Garth Gibson. 2010. DiskReduce v2.0 for HDFS. https://www.pdl.cmu.edu/DiskReduce/talks/2010-Jan-28-OpenCirrus-Gibson.pdf visited on 2021-05-29.

[8] Binny S Gill, Michael Ko, Biplob Debnath, and Wendy Belluomini. 2009. STOW: A Spatially and Temporally Optimized Write Caching Algorithm. In *2009 USENIX Annual Technical Conference (USENIX ATC '09)*. 29–42.

[9] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*. 15–26.

[10] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*. 20.

[11] Won Kim. 2009. Cloud computing: Today and tomorrow. *J. Object Technol.* 8, 1 (2009), 65–72.

[12] libfuse. 2021. libfuse. https://github.com/libfuse visited on 2021-05-29.

[13] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. 21–33.

[14] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. 2020. Practical quick file server migration. *ACM Transactions on Storage (TOS)* 16, 2 (2020), 1–30.

[15] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. 2014. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 257–273.

[16] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 1–23.

[17] Jun Nemoto, Atsushi Sutoh, and Masaaki Iwasaki. 2017. Directory-aware file system backup to object storage for Fast on-demand restore. *International Journal of Smart Computing and Artificial Intelligence (IJSCAI)* 1, 1 (2017), 1–19.

[18] Scott Peterson. 2009. Using Persistent Memory and RDMA for Ceph Client Write-back Caching. In *USENIX Annual Technical Conference*. 29–42.

[19] James S Plank. 2013. Erasure codes for storage systems: A brief primer. *Login: The USENIX Magzine* 38, 6 (2013), 44–50.

[20] Ghemawat Sanjay. 2003. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP 2003)*. ACM, 29–43.

[21] Vikhyat Umrao, Michael Hackett, and Karan Singh. 2017. *Ceph cookbook: practical recipes to design, implement, operate, and manage Ceph storage systems*. Packt Publishing Ltd.

[22] vmware. 2021. VMware vSAN 7 Technical Overview. https://core.vmware.com/resource/vsan-7-technology-overview visited on 2020-05-29.

[23] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)*. USENIX Association, 307–320.

[24] XFS.org. 2013. XFS.org. https://xfs.org/ visited on 2021-05-29.

[25] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A Pease. 2015. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*. 213–226.

[26] Yue Yang and Jianwen Zhu. 2016. Write skew and zipf distribution: evidence and implications. *ACM transactions on Storage (TOS)* 12, 4 (2016), 21.

[27] Yiming Zhang, Huiba Li, Shengyun Liu, Jiawei Xu, and Guangtao Xue. 2020. PBS: An efficient erasure-coded block storage system based on speculative partial writes. *ACM Transactions on Storage (TOS)* 16, 1 (2020), 1–25.