# 論文 / 著書情報
# Article / Book Information

| | |
|---|---|
| 題目(和文) | 拡張ファイルメタデータを用いたサーバベースストレージの高機能・高性能化に関する研究 |
| Title(English) | A Study on Applications of Custom File Metadata for High-functional and High-performance Server-based Storage |
| 著者(和文) | 深谷崇元 |
| Author(English) | Takayuki Fukatani |
| 出典(和文) | 学位:博士（工学）,<br>学位授与機関:東京工業大学,<br>報告番号:甲第12106号,<br>授与年月日:2021年9月24日,<br>学位の種別:課程博士,<br>審査員:横田 治夫,宮崎 純,渡部 卓雄,吉瀬 謙二,金子 晴彦 |
| Citation(English) | Degree:Doctor (Engineering),<br>Conferring organization: Tokyo Institute of Technology,<br>Report number:甲第12106号,<br>Conferred date:2021/9/24,<br>Degree Type:Course doctor,<br>Examiner:,,,, |
| 学位種別(和文) | 博士論文 |
| Type(English) | Doctoral Thesis |

# A Study on Applications of Custom File Metadata for High-functional and High-performance Server-based Storage

by

Takayuki Fukatani

Submitted to the Department of Computer Science, School of Computing in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at the TOKYO INSTITUTE OF TECHNOLOGY July 2021.

**Thesis Committee:**

Prof. Jun Miyazaki

Prof. Takuo Watanabe

Prof. Kenji Kise

Assoc. Prof. Haruhiko Kaneko

Prof. Haruo Yokota, Chair

**Abstract**

In recent years, the demand for large-scale storage systems has been increasing due to the progress of infrastructure system consolidation in the cloud and on-premise environments. To offer large-scale storage at a low cost, server-based storage using commodity servers is increasingly being used as a promising replacement for dedicated storage appliances. Server-based storage aggregates a large number of inexpensive commodity servers over a network and provides redundancy among the servers to achieve scalable and reliable storage. Server-based storage, which offers low-cost large-scale storage, is becoming more widely used in both cloud and enterprise on-premise environments.

As server-based storage becomes popular, its target applications are also expanding. Early server-based storage was intended for limited uses such as data analysis and archiving in cloud environments. The early server-based storage introduced simple functionality and workload-specific architecture for limited applications to achieve high scalability. Today, the use of server-based storage has spread to legacy applications such as databases and virtual desktop infrastructure. Legacy applications require high functional coverage of the standard network protocols and high random access performance for storage systems. To meet the demands of these expanding applications, there is a need to improve the functionality and performance of server-based storage.

This doctoral dissertation proposes methods for improving the functionality and performance of server-based storage. All these proposed methods use the custom file metadata, a new metadata framework to offer the extended file metadata capabilities on the commodity operating system (OS) running on server-based storage. This doctoral dissertation proposes three applications of the custom file metadata as shown below.

First, this doctoral dissertation proposes the portable metadata module (PMM) to improve the functionality of server-based storage by using the custom file metadata. PMM is a software module for using the custom file metadata on the commodity OS. PMM allows the protocol stack of the storage appliances to manage the protocol-specific metadata on the commodity OS. PMM uses new user-space metadata management and journal management to achieve the flexibility and reliability of the custom file metadata. PMM enables the protocol stack of the

storage appliances to run on the commodity servers, and improves functional coverage of network protocols for server-based storage. I incorporated PMM into the protocol stack of a commercial storage appliance and made the protocol stack runnable on commodity servers. The evaluations confirmed that PMM improves the functional coverage of the network protocol.

Secondly, this doctoral dissertation proposes dynamic redundancy control with adaptive encoding (DRC-AE) to improve the performance of server-based storage by using the custom file metadata. DRC-AE stores metadata of redundancy configuration in the custom file metadata and enables redundancy control of local data across servers. DRC-AE improves the performance of server-based storage by placing an application and its data on the same server. DRC-AE also performs the adaptive encoding, which dynamically changes the data redundancy method according to performance requirements and workloads. DRC-AE uses the adaptive encoding to achieve high capacity efficiency while maintaining high performance. The evaluations confirmed that DRC-AE achieves higher random performance than conventional distributed storage.

Thirdly, this doctoral dissertation proposes dynamic redundancy control with delayed parity update (DRC-DPU) as another performance improvement method using the custom file metadata for server-based storage. DRC-DPU manages differential data for erasure-coded (EC) data by storing metadata of the differential data in the custom file metadata. DRC-DPU asynchronizes slow parity updates by redirecting writes for EC data to replicated differential data. DRC-DPU solves the problem of the degraded write response time of EC data in DRC-AE. The evaluations confirmed that DRC-DPU improves the degradation of the write response time of EC data in DRC-AE.

Finally, this doctoral dissertation discusses the overall effectiveness of the three proposed methods on expanding the target applications of server-based storage. I evaluated the effectiveness of the proposed methods in terms of performance, functionality, and reliability for the major target applications of network storage. I confirmed that the proposed methods dramatically improve the performance of server-based storage in the target applications while meeting the functional and reliability requirements. I also compared the server-based storage with the proposed methods to the conventional storage appliances. The evaluation revealed the performance advantages of server-based storage with the proposed methods and the future challenges in reliability.

The works in this doctoral dissertation contribute directly to designing server-based storage systems that support a wider range of target applications in large-scale infrastructures.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In recent years, the demand for large-scale storage systems has been increasing due to the progress of infrastructure system consolidation in the cloud and on-premise environments [BHRM18, Kim09]. Large-scale storage systems are required to provide stable services to a large number of applications and users. To enable stable services, storage systems are required to provide high performance, high functionality, and high reliability.

Traditionally, the storage appliances have been widely used to meet these requirements [Van21, Inc21, Net21]. The storage appliances use specialized software and hardware for high performance, high functionality, and high reliability. However, development for the specialized software and hardware makes storage appliances expensive. As systems become larger and larger, the high cost of storage appliances has been a problem.

Server-based storage using inexpensive commodity servers is spreading as an alternative to the storage appliances [Mat12, BHRM18]. Server-based storage achieves high reliability and high throughput by configuring distributed storage among a large number of commodity servers. Many cloud and enterprise systems have used server-based storage as inexpensive scalable storage.

However, the demands for high functionality and high performance in large-scale storage systems limit the use of server-based storage. Server-based storage usually consists of open-source software (OSS) protocol stacks and distributed storage running on commodity servers. OSS protocol stacks have a disadvantage over storage appliances in terms of functional coverage of network protocols. In addition, the performance of distributed storage tends to be an issue due to the network communication overhead.

This doctoral dissertation proposes methods of increasing the functionality and performance of server-based storage to expand its target applications. The proposed methods achieve

high functionality and high performance of server-based storage by utilizing a method named the custom file metadata, which extends the file metadata of commodity operating systems (OSs).

## 1.1 Conventional Approaches

This section outlines the conventional approaches to improve the functionality and performance of server-based storage. The details of the conventional approaches are explained in the respective chapters.

### 1.1.1 Storage Protocol Stack for Server-based Storage

Server-based storage uses OSS protocol stacks to support the standard network storage protocols such as the server message block (SMB) and the network file system (NFS). OSS protocol stacks are protocol stack implementations developed by the OSS community. OSS protocol stacks have been developed for SMB and NFS, respectively [DLL07, sam21].

The protocol functional coverage of OSS protocol stacks is basically lower than the storage appliances. In particular, the functional coverage of SMB used by clients of the non-POSIX compliant Windows OS [Cor21d] is much worse than the storage appliances. SMB requires protocol-specific metadata such as security descriptors, file attributes, and metadata required for each SMB function. On the other hand, OSS protocol stacks have been implemented as user-space applications for POSIX-compliant commodity OSs. POSIX-compliant commodity OSs do not support the protocol-specific metadata, and cannot provide functions using the protocol-specific metadata as is. To increase the functional coverage of network protocols in server-based storage, the OSS protocol stacks need to complement the gap between network protocol-specific metadata and metadata of the commodity OS.

RichACL has been proposed as an implementation of SMB-compliant access control list (ACL) for OSS protocol stacks [KGB10]. RichACL supports SMB-compatible ACL by storing SMB-format ACLs in the standard file metadata of the commodity OSs. However, the ACL-specific metadata implementation is not applicable to other protocol-specific metadata [KGB10]. In addition, due to the capacity limitation and capacity efficiency of file metadata in the commodity OSs, RichACL has a limited number of ACLs and larger capacity consumption.

Thus, the OSS protocol stacks used in conventional server-based storage have low functional coverage of network protocols due to the limitations of POSIX-compliant commodity OSs. As a result, OSS protocol stacks have a functional disadvantage over the storage appli-

ances, which use specialized software and hardware to achieve high-functional coverage of network protocols.

### 1.1.2 Distributed Storage for Server-based Storage

Server-based storage configures distributed storage among many commodity servers to achieve high reliability and high throughput [WBM+06, Bor07]. Distributed storage provides data redundancy among low-reliability commodity servers, and enables service continuity in the event of server failure. Also, by using capacity efficient erasure coding (EC), distributed storage mitigates the capacity consumption due to data redundancy [HSX+12]. Distributed storage also achieves high throughput by consolidating a large number of commodity servers.

In contrast to high sequential performance, distributed storage suffers from low random performance. Network-based distributed storage is associated with large network communication latency and thus has low random performance. Low random performance becomes a problem when supporting legacy applications such as database (DB) and the virtual desktop infrastructure (VDI) where random access is dominant.

A variety of performance improvements have been proposed for server-based storage. DiskReduce and Hadoop adaptively-coded distributed file system (HACFS) mitigate the performance degradation due to inter-server data redundancy in Hadoop Distributed File System (HDFS) by changing the data redundancy methods based on workloads [Gib10, XSBP15]. In addition, write performance improvements such as parity logging [CDLC14] and speculative parity write [ZLL+20] also reduce the performance overhead of EC.

However, even with these performance improvements, the random performance of distributed storage is considered as a disadvantage over conventional storage appliances.

## 1.2 Approach Overview

The purpose of this doctoral dissertation is to expand the target applications of server-based storage by increasing its functionality and performance. The goals of this doctoral dissertation are as follows.

The first goal of this doctoral dissertation is to enable the execution of the protocol stack of the storage appliances on the commodity servers. I aim to improve the functionality of server-based storage by making the high-functional protocol stack runnable on the commodity servers.

The second goal of this doctoral dissertation is to achieve high performance by enabling

redundancy control of local data among servers. I aim to improve the performance of server-based storage by making local data redundant between servers and placing an application and its data on the same server.

The final goal of this doctoral dissertation is to provide stable performance in server-based storage by improving the write process of EC data. I aim to solve the write performance degradation of EC data and stabilize the write response time.

To achieve these three goals, I introduce a new file metadata framework, the custom file metadata. The custom file metadata offers the extended file metadata capabilities on the commodity OS by utilizing the existing commodyty OS capabilities. I aim to achieve these goals through the following three approaches using the custom file metadata.

The first approach uses the custom file metadata to enable the protocol stack of the storage appliances to run on the commodity OS running on the commodity servers. This approach stores network protocol-specific metadata in the custom file metadata to enable the protocol stack of the storage appliances to run on the commodity OS. This approach makes server-based storage more functional with a specialized protocol stack that has high-functional coverage of network protocols.

The second approach uses the custom file metadata to introduce a new redundancy control of local data among servers. This approach stores the metadata of the redundancy configuration between servers in the custom file metadata and uses it to control the redundancy of local data between servers. This approach improves data access performance in service-based storage by placing an application and its data on the same server.

The third approach uses the custom file metadata to improve the write response time of the erasure-coded data (EC data). This approach stores the metadata of the differential data in the custom file metadata and asynchronizes the parity update of EC data. This approach stabilizes the write response time of server-based storage by making the parity update asynchronous with the write request processing.

These approaches improve functionality and performance and server-based storage and expand its target applications.

## 1.3 Contributions

This section presents the three technical contributions that make up this doctoral dissertation.

**Portable Storage Procotol Stack** First, this doctoral dissertation proposes the portable metadata module (PMM) to realize a portable storage protocol stack using the custom file metadata. I clarify the limitations of the commodity OS in server-based storage, and investigate the design of the custom file metadata that enables the extension of the commodity OS. Then, I propose PMM, a software module for using the custom file metadata on the commodity OS. The custom file metadata allows applications to add their file metadata to user files. PMM provides the flexibility and reliability of the custom file metadata by introducing new user-space metadata management and journal management. PMM enables the protocol stacks of the storage appliances to run on the commodity OS. The evaluations show that the protocol stack of a commercial storage appliance using the custom file metadata improves the network protocol coverage of server-based storage.

This proposal enables the server-based storage to be highly functional.

**Dynamic Redundancy Control with Adaptive Encoding** Next, this doctoral dissertation proposes dynamic redundancy control with adaptive encoding (DRC-AE), which is a redundancy control using the custom file metadata. DRC-AE stores redundancy configuration metadata in the custom file metadata to enable redundancy of local data across servers. By storing data on the same server as the application, DRC-AE achieves higher random performance compared to the conventional distributed storage. DRC-AE also performs the adaptive encoding, which dynamically changes the data redundancy method according to the performance requirements given by a user and workloads. DRC-AE uses the adaptive encoding to achieve high capacity efficiency while maintaining high performance. The evaluations show that DRC-AE achieves higher random performance than conventional distributed storage.

This proposal enables high-performance of server-based storage.

**Dynamic Redundancy Control with Delayed Parity Update** Finally, this doctoral dissertation proposes dynamic redundancy control with delayed parity update (DRC-DPU), which incorporates an EC write optimization using the custom file metadata into DRC-AE. DRC-DPU stores the metadata of differential data in the custom file metadata to enable the asynchronous update of the parity data. DRC-DPU delays slow parity updates by keeping write data to EC data as replicated differential data. DRC-DPU improves the degraded write response time of EC data in DRC-AE. The evaluations confirm that DRC-DPU improves the write response time of EC data and achieves stable response performance.

This proposal enables further performance enhancement of server-based storage.

## 1.4 Dissertation Organization

This doctoral dissertation consists of seven chapters (see Figure 1.1). The next chapter discusses the background knowledge relating to the solutions in this doctoral dissertation. Chapter 3 proposes PMM that improves the functional coverage of server-based storage. Chapter 4 presents DRC-AE for high-performance and high capacity efficiency for server-based storage. Chapter 5 introduces DRC-DPU to improve write response time for EC data access, which is a weak point of DRC-AE. Chapter 6 discusses the overall effectiveness of our proposals. Finally, Chapter 7 summarizes the doctoral dissertation, and discusses the future research direction.

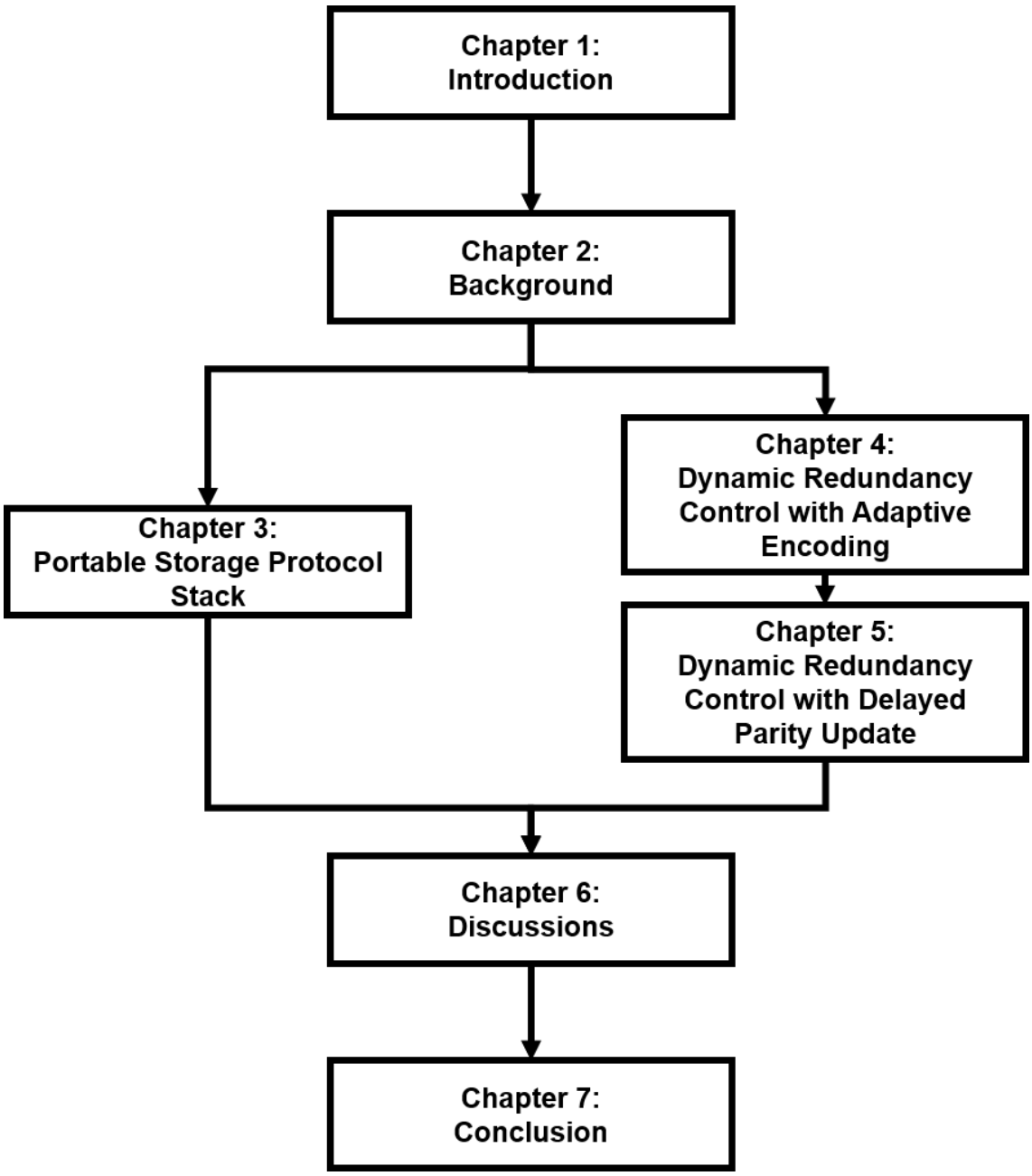



Figure 1.1: Thesis Layout

# Chapter 2

# Background

This chapter describes the trend in large-scale storage systems as background knowledge for this doctoral dissertation. Then, I give an overview of the storage appliances and server-based storage, which have been used in large-scale storage systems.

## 2.1 Trend of Large-scale Storage Systems

Large-scale storage systems are required to provide high performance, functionality, and reliability as the foundation of information technology (IT) infrastructures [SS12]. The storage vendors respond to the ever-increasing user demands by continuously developing storage systems. The storage vendors have provided enterprise storage systems for IT infrastructures within companies and organizations for many years. In recent years, with the progress of infrastructure consolidation, storage systems have become even larger in scale.

The storage appliances have long been used as storage systems for enterprise applications [Van21, Inc21, Net21]. The storage appliances have been popular since the 1990s. Many on-premise infrastructure systems use commercial products such as NetApp Filer [HLM94]. High-performance and reliable storage appliances meet the high requirements of IT infrastructures.

However, the cost of expensive storage appliances tends to be a problem for large-scale storage systems. The storage appliances use specialized software and hardware to achieve high reliability, high performance, and high functionality. The development of specialized software and hardware increases the cost of the storage appliances. If a large number of expensive storage appliances are used in the IT infrastructures, the system cost increases significantly. There is an increasing demand for cost reduction as well as performance, functionality, and reliability.

Furthermore, in consolidated infrastructures, applications with less performance and functional requirements, such as data analysis and archiving, are also often used. For these applications, the high performance and high functionality of storage appliances become excessive.

Low-cost commodity server-based storage has become a practical alternative to the storage appliances [Mat12, BHRM18]. Server-based storage consolidates commodity servers through software control to form inexpensive and scalable storage systems. Server-based storage is increasingly being used in both cloud and on-premise environments to build large-scale storage systems at a low cost.

However, server-based storage has disadvantages in terms of functionality and performance compared to the storage appliances. OSS protocol stacks of server-based storage have lower functional coverage of network protocols than the specialized protocol stacks of the storage appliances. In addition, the network-based distributed storage of server-based storage has a disadvantage in random performance due to the network communication overheads. The lack of functionality and performance of server-based storage becomes a problem in applying server-based storage to applications with high-functionality and high-performance requirements.

To expand the target applications of server-based storage, it is necessary to improve functionality and performance.

## 2.2 Storage Appliance

The storage appliance is a scale-up storage system that uses specialized hardware and software. The storage appliances have been widely used in enterprise storage systems [Van21, Inc21, Net21]. Figure 2.1 shows the overview of the storage appliance.

The storage appliances using specialized software and hardware provide high reliability, high performance, and high functionality for enterprise applications. The storage vendors develop proprietary software and hardware to meet increasingly complex protocol specifications and user requirements. The storage vendors also respond to the recent increase in the scale of storage systems by coordinated control of multiple appliances. On the other hand, the development of such storage appliances requires a large amount of development cost and increases the cost of the storage appliances.

The following describes the specialized software and hardware used in the storage appliances. Then, I explain the overview of an storage appliance product as a case study.

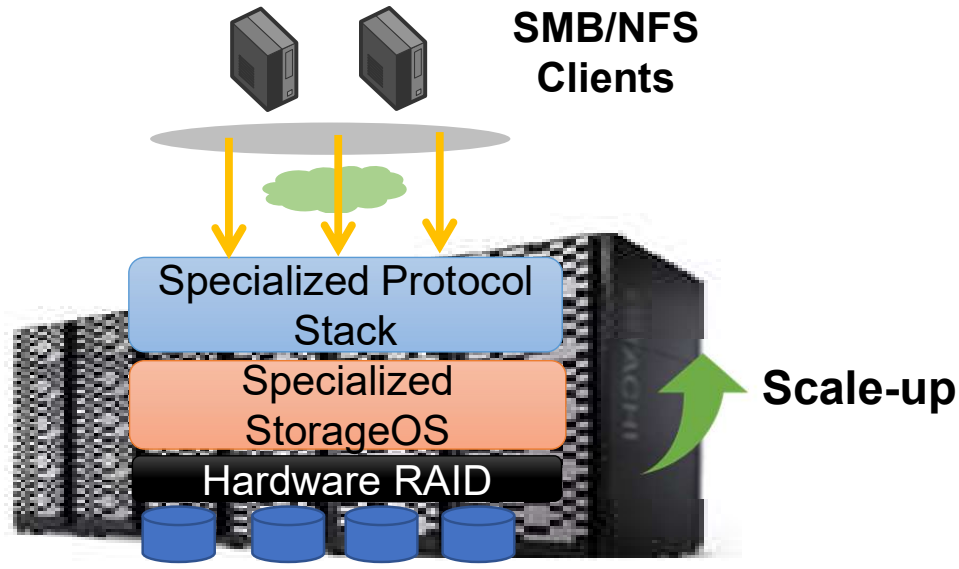



Figure 2.1: Storage Appliance

### 2.2.1 Specialized Software for Storage Appliance

The specialized software for storage appliances provides high functionality with its own protocol stacks and storage OSs [DVS12]. The storage appliances support standard specifications of network storage protocols such as SMB and NFS by using the specialized software [Cor21b, Cor21c, For95, HN15]. The high functional coverage of the specialized software allows clients using the storage appliances to benefit from the latest SMB and NFS specifications.

The specialized software has been developed over the years to keep up with the evolution of network protocol specifications. Early network storage protocols such as SMB1.0 and NFSv3, which are designed for file sharing in small networks, only specify simple file access such as read, write, and file open. Subsequently, SMB2.0/3.0 and NFSv4 are extended to support large-scale systems and the internet environment. These enhancements include security enhancements, availability enhancements, and support for multiple sites.

The specialized software is also able to support mixed SMB and NFS environments [AHB+99, Bor98]. SMB and NFS have different semantics in some functions such as ACL and the operation lock. To absorb such semantic differences, the specialized software handles the interconversion between protocols. The specialized software enables mixed SMB and NFS access to the same file.

The specialized software has developed its own storage OSs and protocol stacks to comply with the evolving specifications of network protocols. The storage vendors modify their storage

OSs and protocol stacks to keep up with the protocol specification changes, and offer protocol-compatible metadata and access interfaces. The custom-made specialized software eliminates the constraints imposed by existing software and achieves high-functional coverage by using protocol-compatible implementations.

### 2.2.2 Specialized Hardware for Storage Appliance

The storage appliances use specialized hardware such as application specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs), and non-volatile random access memory (NVRAM) [SS12]. The storage appliances offload the lower layer processing such as network communication and redundant arrays of inexpensive disks (RAID) control to the specialized hardware to achieve high throughput and short response time. Some storage vendors also use the specialized hardware for higher layers such as file systems and network protocol stacks to achieve even higher performance [BWB+04].

The storage appliances have redundancy in all components, including the specialized hardware. The storage appliances achieve high reliability by eliminating a single point of failure in the hardware from the entire appliance. In the event of a component failure within the appliance, the remaining redundant components continue processing to ensure service continuity.

The storage appliances also form a high availability (HA) cluster with multiple appliances to handle node failures. Even in the event of a node failure, the service can be taken over by another node in the HA cluster.

In this way, the storage appliances achieve high performance through the use of specialized hardware and high reliability through hardware redundancy.

### 2.2.3 Case Study: High-performnace Network Attached Storage

In this section, I introduce High-performance Network Attached Storage (HNAS) as a case study of the specialized storage appliance [BWB+04]. HNAS is an enterprise NAS product that achieves high performance with an FPGA-based filesystem. HNAS is mainly used in large-scale systems such as the infrastructure of banks and public institutions. HNAS achieves high throughput and low latency by utilizing the filesystem implemented in dedicated FPGA boards. Figure 2.4 shows the overview of the HNAS architecture.

HNAS constitutes an HA cluster whose nodes share a back-end disk array. Each node of the HNAS consists of the data processing board with FPGAs and the control board with CPU. The data processing board implements low-level processing such as network, file system, and

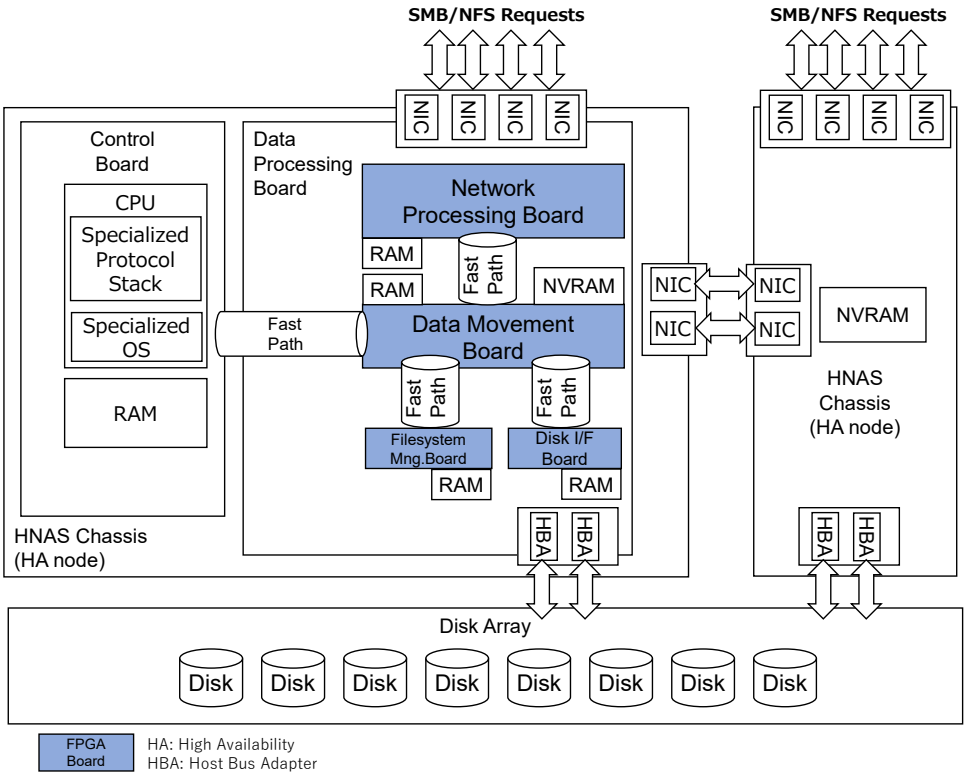



Figure 2.2: HNAS Architecutre Overview

disk access as dedicated FPGAs and achieves high processing performance compared to CPU processing. On the other hand, the control board handles complex processing such as security functions and quota through the dedicated software implementation on the CPU. HNAS achieves both high functionality and high performance by combining FPGAs for the low-level data processing and dedicated software running on CPU for complex processing. HNAS also achieves high reliability by directly connecting the nodes via ethernet cables and storing write data in each other's NVRAM for redundant protection. As shown above, HNAS uses dedicated hardware to achieve high performance and high reliability, while the dedicated hardware increases the system cost.

## 2.3 Server-based Storage

Server-based storage is a storage system that consists of many inexpensive commodity servers [Mat12]. Because of the low cost of commodity servers, the cost of a server-based storage system tends to be considerably lower than the storage appliances. Server-based storage initially

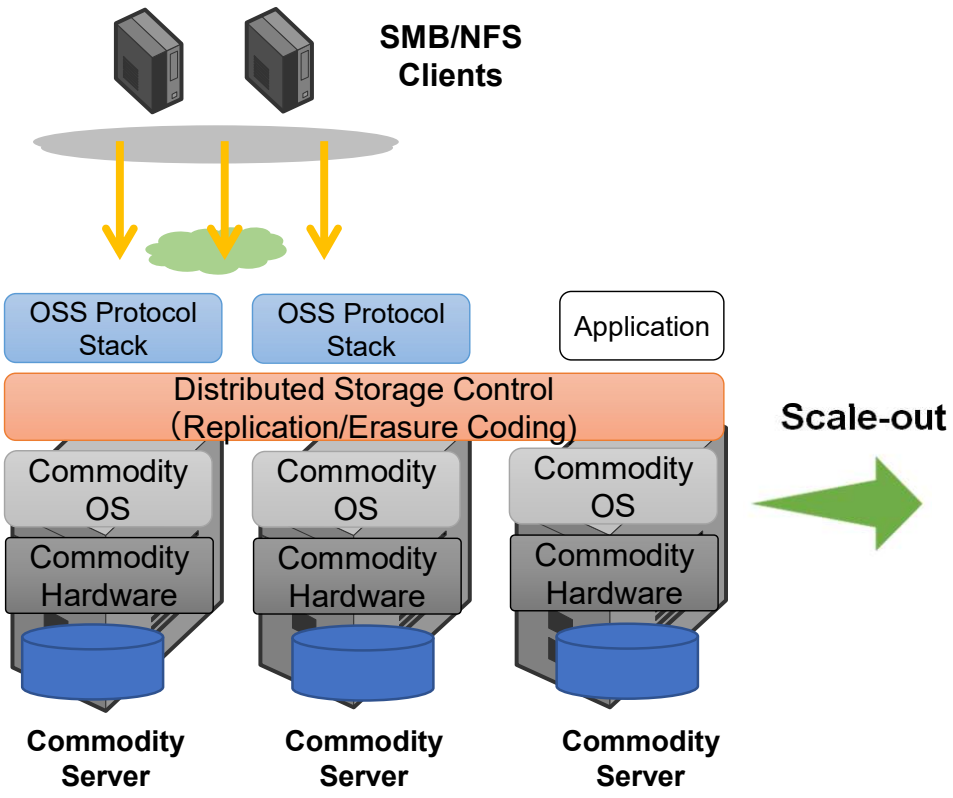



Figure 2.3: Server-based Storage

evolved as back-end storage of cloud computing environments, and soon became widespread in on-premise storage infrastructures. Figure 2.3 shows the overview diagram of server-based storage.

### 2.3.1 OSS Software for Server-based Storage

Server-based storage uses commodity OSs such as Linux to support a variety of commodity servers. Server-based storage basically uses the latest version of the commodity OSs without modifications to keep up with the latest hardware. Typical server-based storage consists of a storage system with OSS protocol stacks and distributed storage running on the commodity OS.

**OSS Protocol Stack** The server-based storage supports network protocols such as SMB and NFS with OSS protocol stacks. The OSS protocol stacks are protocol stack implementations developed by the OSS community. Samba [sam21] and NFS-ganesha [DLL07] are prominent OSS protocol stack implementations for SMB and NFS, respectively.

The OSS protocol stacks run on standard POSIX-compatible commodity OSs [IG18]. The constraints of the application programming interface (API) and support functions of the commodity OSs make it difficult for the OSS protocol stacks to fully support network protocol functions. As a result, the functional coverage of OSS protocol stacks tends to be lower than that of the storage appliances. In particular, the OSS protocol stacks are disadvantageous in SMB, which is a non-POSIX compliant protocol for Windows OS, and in SMB/NFS mixed environments.

**Distributed Storage** Typical server-based storage systems use scale-out distributed storage to meet the requirements of high reliability and high scalability [San03, HSX+12, Bor07, Kim09]. The basic idea of the use of distributed storage for server-based storage came from early implementations of cloud storage. Google file system (GFS), prominent cloud storage, achieves high sequential access throughput by aggregating low-performance commodity servers [San03]. As GFS stores data among a large number of unreliable commodity servers, the high frequency of server failures had become a problem. GFS initially introduced three-way replication, which stores two replicas of user data on other servers, to achieve high reliability on unreliable commodity servers. Then, EC was applied to reduce the amount of replica data in three-way replication that triples the storage consumption [Fik10]. The design of GFS has had a significant impact on other cloud storage systems [HSX+12] and scale-out distributed storage implementations such as Hadoop distributed file system (HDFS) and CephFS [Bor07, WBM+06].

In distributed storage, network communication and data redundancy between servers become a performance overhead in terms of response time. Therefore, although distributed storage has high sequential performance, it has a disadvantage over the storage appliances in terms of random access performance.

### 2.3.2 Commodity Hardware for Server-based Storage

Server-based storage uses commodity servers for hardware. Commodity servers, which can reduce costs through mass production, are less expensive than specialized hardware. These low-cost commodity servers make server-based storage less expensive than the storage appliances. In addition, the hardware vendors tend to provide early support of the latest hardware components such as the network interface card (NIC) or the disks for the commodity servers. Therefore, server-based storage can be equipped with the latest hardware earlier than the traditional storage appliances.

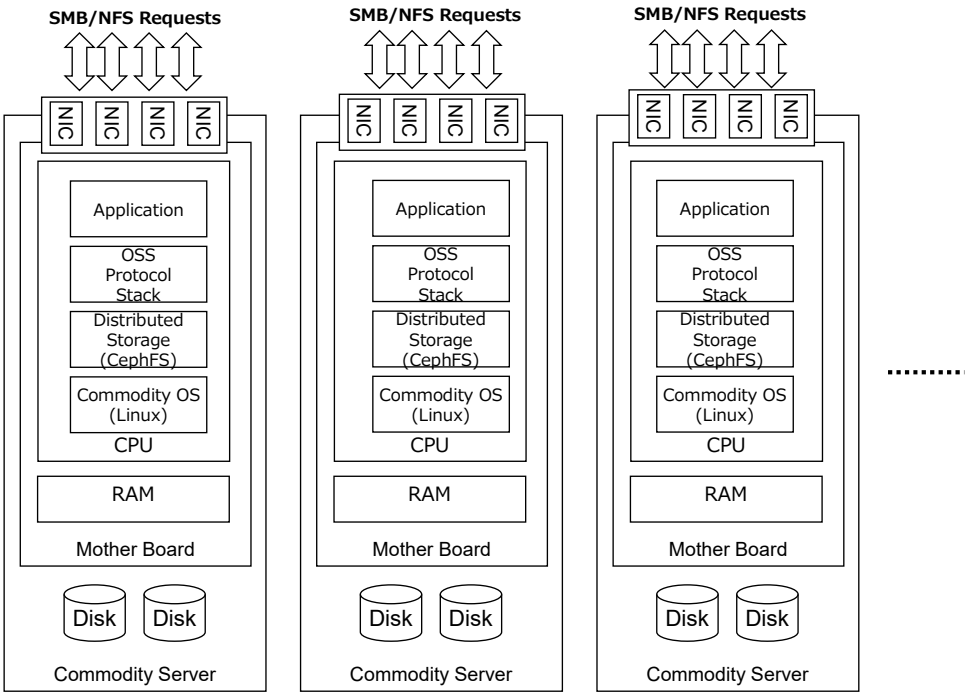



Figure 2.4: CephFS-based Server-based Storage Architecture Overview

Commodity servers basically have no hardware redundancy like in storage appliances, and many components such as CPU and memory become a single point of failure. For this reason, server-based storage incorporates software redundancy control between servers, and multiple servers form a cluster to make the servers redundant. If a component of a server fails, server-based storage continues its service using another server. Server-based storage complements the reliability of the commodity server by providing hardware redundancy on a per-server basis.

### 2.3.3 Case Study: Server-based Storage using CephFS

In this section, I introduce a server-based storage implementation using CephFS as a case study of the server-based storage [UHS17]. Figure 2.4 shows the overview diagram of a typical implementation of server-based storage using CephFS.

In this example, CephFS consists of distributed storage with many commodity servers. Each server is composed of commodity hardware; CPU, RAM, disks, and NICs. With the recent development of commodity hardware, the latest commodity hardware such as multi-core CPU, large capacity memory, NVMe SSD and 100 GbE NIC enable high performance.

CephFS distributes data among many servers and makes them redundant. CephFS achieves

high scalability and enables a throughput proportional to the number of servers. With data redundancy, in the event of a server failure, the service can be continued on the remaining servers. The OSS protocol stacks run on each server and handles SMB/NFS requests. In addition, applications run on each server and access the distributed storage directly.

# Chapter 3

# Portable Storage Protocol Stack Using Custom File Metadata

This chapter proposes a method using the custom file metadata to make server-based storage more functional. First, I clarify the limitations of the file metadata of the commodity OS in server-based storage. Then, I discuss the design of the custom file metadata that enables arbitrary file metadata extensions in the commodity OS. Then, I propose PMM, which is the first application of the custom file metadata.

For many years, storage vendors have developed specialized storage appliances to satisfy user demands for performance, functionality, and reliability [WBYS99, ISI21]. Conventionally, software-based protocol stacks handle client requests such as NFS, SMB, and iSCSI on those appliances [Cor21b, Cor21c, For95, HN15, BHH+04]. The vendors have made large effort to incorporate the frequent updates in the protocol specifications into their protocol stacks.

The conventional storage vendors have struggled to adapt their protocol stacks to the commodity servers. With the increasing interest in the software-defined storage [CYS+14, MMP+12], users have demanded cheaper commodity servers for storage hardware for lightweight workloads. Some storage vendors use the hypervisor-based virtual machine to achieve the transition [ISI21]. However, this approach is not applicable for the protocol stacks which depend on the specialized hardware.

Many storage appliances use their original filesystems to store user files and protocol-specific file metadata [WBYS99, ISI21]. These original filesystems are often implemented in the specialized hardware or software. The protocol-specific file metadata in those filesystems enable the protocol stacks to offer protocol functions in a protocol compliant manner. However, the dependency on the original filesystems makes the protocol stacks unavailable on

other filesystems.

The use of the standardized POSIX filesystem interface removes the dependency on the specialized hardware or software from the protocol stacks [IG18]. The POSIX filesystem interface is used in a wide range of OSS filesystems. Once the protocol stacks support the POSIX filesystem interface, the protocol stacks become available on the commodity servers on which OSS filesystems are deployable.

However, on the POSIX-based filesystems, the protocol stacks cannot offer protocol functions which rely on the protocol-specific file metadata. The POSIX-based filesystems store the file metadata in the POSIX compliant format which do not cover protocol-specific metadata. The protocol functional coverage of the protocol stacks largely decreases on Linux servers because of the lack of the protocol-specific file metadata.

This chapter proposes PMM which enables the capability of the protocol-specific metadata on OSS Linux filesystems [BC05]. PMM introduces a new metadata framework named the custom file metadata to accommodate the protocol-specific file metadata in Linux filesystems. PMM utilizes POSIX-based Linux API to store the custom file metadata in Linux filesystems so that the protocol stacks can use the custom file metadata on Linux servers.

PMM consists of two components: the metadata management which stores the custom file metadata in Linux filesystems, and the journal management which ensures data integrity of the custom file metadata [SWZ05]. These modules complement the differences between the protocol-specific file metadata and the POSIX file metadata.

I applied PMM to the protocol stack of a proprietary storage appliance named HNAS. PMM allows the HNAS protocol stack to offer protocol functions which rely on the protocol-specific file metadata on Linux servers.

The evaluations showed that PMM increases the functional coverage of the HNAS protocol stack from 75.0% to 96.7% on Linux servers while suppressing the performance degradation to at most 8% in the typical file server workload. I confirmed that HNAS protocol stack with PMM achieves competitive file server performance with other OSS protocol stack implementations. Also, I confirmed that the journal management of PMM ensures data integrity of the custom file metadata by offering fast and scalable recovery processing upon a system failure.

This chapter is an extended version of our prior paper [FLY19].

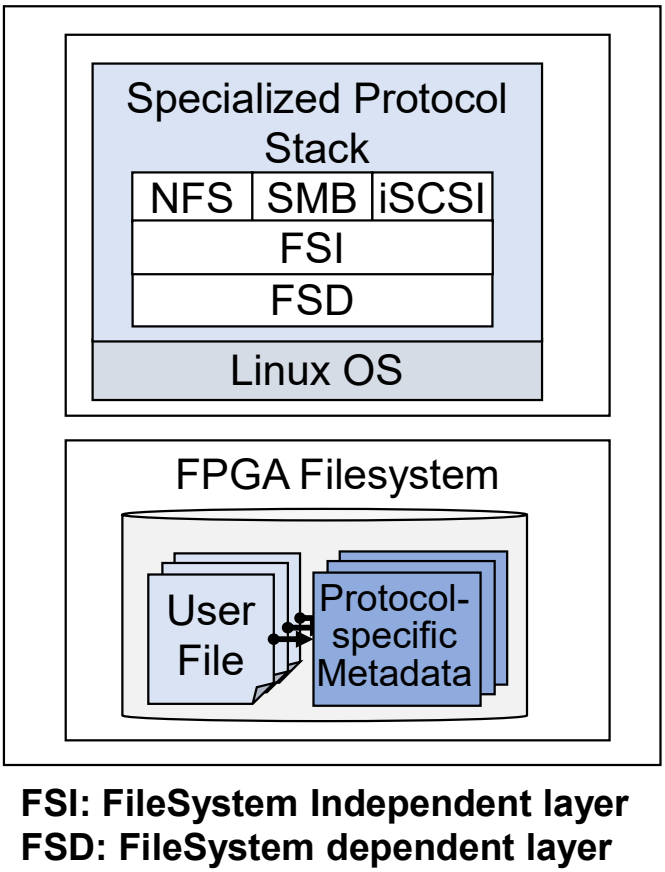



Figure 3.1: HNAS Overview

## 3.1 Conventional Approaches

### 3.1.1 Specialized Protocol Stack

Many proprietary protocol stacks use specialized hardware or software to store the protocol-specific file metadata to comply with the protocol specifications. These hardware or software usually have the original interface and functionality.

In the case of HNAS, which is the target storage appliance of this study, uses an original FPGA-based filesystem [BWB+04]. Likewise many other storage appliances, HNAS uses the original filesystem to store the protocol-specific file metadata in a protocol compliant format. The software-based protocol stacks handle client requests that involve advanced file processing such as the security descriptor, the named stream, quota, and the virus scan [RSI12]. HNAS protocol stack uses the protocol-specific file metadata to provide these capabilities. Figure 3.1 shows the overview of the HNAS architecture.

The HNAS protocol stack has a layered architecture which separates hardware dependent modules from other modules. The architecture consists of the protocol layer, the FileSystem Independent (FSI) layer, and the FileSystem Dependent (FSD) layer. The protocol layer accommodates NFS, SMB, and the internet small computer system interface (iSCSI) modules which interpret client requests. Underneath the protocol layer, the FSI layer offers the aforementioned advanced filesystem functions. The FSD layer encapsulates the FPGA filesystem implementation from the upper layers.

Although a large part of the HNAS protocol stack is independent of the FPGA filesystem, the protocol stack runs only on the HNAS appliance because of the dependency on the FPGA filesystem in the FSD layer.

### 3.1.2 OSS Protocol Stack

The OSS protocol stacks run on standard POSIX-compatible commodity OSs [IG18]. The constraints of the API and supported functions of the commodity OSs make it difficult for OSS protocol stacks to support network protocol functions fully . In particular, the OSS protocol stacks are disadvantageous in SMB, a non-POSIX compliant protocol for Windows OS, and SMB/NFS mixed environments.

RichACL has been proposed as an implementation of SMB-compliant ACL for OSS protocol stacks. RichACL uses the extended attributes of Linux filesystems to store RichACL as metadata of user files and directories. Figure 3.2 shows the overview of RichACL.

Rich ACL uses a security descriptor format compatible with the NT File System (NTFS) of Windows OS [RSI12]. A filesystem or a protocol stack reads RichACL and processes the ACL in an SMB-compatible manner. RichACL uses standard Linux filesystem functions to provide SMB-compatible ACL control for users.

However, RichACL has limitations in its applicability and scalability. RichACL has the following problems.

- The use of RichACL is limited to ACL processing and cannot be applied to other protocol-specific file metadata
- Due to the 64KB capacity limit of the extended attributes, the number of RichACLs per file is limited.
- Rich ACL cannot be shared among multiple files or directories. RichACL consumes metadata capacity according to the number of directories/files when subdirectories or files inherit ACLs.

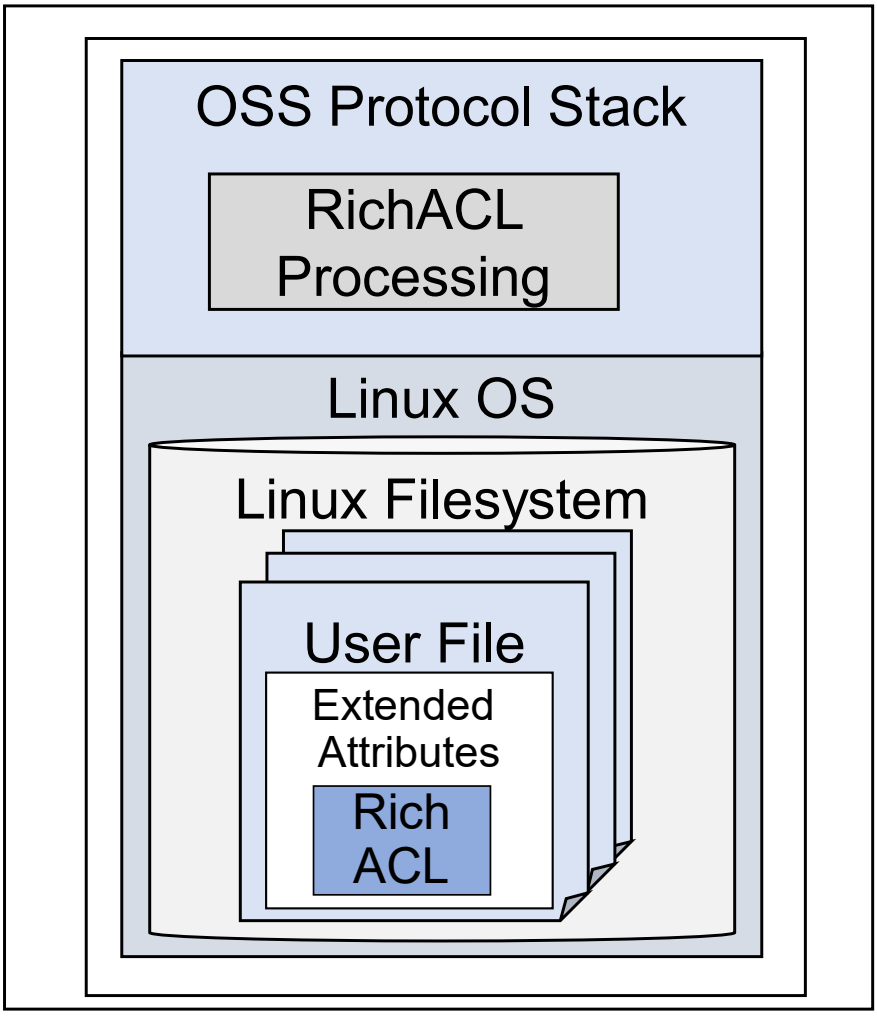



Figure 3.2: RichACL Overview

## 3.2 Motivation

Despite the large development effort, the proprietary protocol stacks are available only on the storage appliances. The dependency on the original filesystems makes the protocol stacks unavailable on other platforms. If the protocol stacks run with alternative filesystems, users can choose a wider range of configurations based on workload requirements.

Use of the POSIX filesystem interface removes the dependency on the specialized appliance from the protocol stacks. The POSIX filesystem interface is widely used in OSS filesystems such as Ext4/XFS [XFS13, MCB+07]. Once a protocol stack supports the POSIX filesystem interface, it becomes available on commodity Linux servers on which the OSS filesystem is commonly used.

However, to be compliant with the protocol specifications, the protocol stacks need the protocol-specific file metadata which are originally stored in the original filesystems. Without the protocol-specific file metadata, the protocol stacks cannot offer the protocol functions which require the protocol-specific file metadata.

The chapter aims to adapt the protocol stacks of the storage appliances to commodity Linux servers. I aim to introduce the new metadata framework named the custom file metadata to accommodate the protocol-specific file metadata in Linux filesystems. I develop PMM which enables the capability of the custom file metadata on Linux servers. I use the POSIX-based Linux API in PMM so that the protocol stacks can use the custom file metadata on Linux servers.

PMM aims to enable the protocol stacks to offer the protocol functions which rely on the protocol-specific file metadata on Linux servers. Along with the higher protocol functional coverage, PMM aims to offer reasonable access performance and data integrity assurance. The performance target is to achieve competitive performance against other OSS protocol stack implementations in the typical file server workload. Also, PMM aims to enable fast recovery processing of data integrity of the custom file metadata upon a system failure.

## 3.3 Protocol Metadata Module

### 3.3.1 Design Goals

PMM has to complement the differences between the POSIX-based Linux file metadata and the protocol-specific file metadata to comply with the protocol specifications. PMM stores the user data and the file metadata in Linux filesystems. However, Linux filesystems do not support the protocol-specific file metadata.

The differences between the Linux file metadata and the protocol-specific file metadata appear in metadata types and data integrity assurance. The protocol-specific file metadata are compliant with the NFS and SMB protocol specifications whereas the Linux file metadata are compliant with the POSIX standard. Although a large part of the NFS requirements overlaps the POSIX standard, SMB requires different types of the file metadata from the POSIX standard because of the different OS environments. Also, the protocol specifications require data integrity assurance in per client request basis whereas POSIX ensures data integrity in per system call basis [BC05].

Table 3.1 shows the differences between the Linux file metadata and the protocol-specific file metadata.

| Items | | Linux File Metadata | HNAS File Metadata |
|---|---|---|---|
| Metadata | File attributes | POSIX / Extended Attributes | SMB/NFS compliant |
| | Security | POSIX permission/ACL | SMB/NFS compliant |
| | Named stream | None | SMB/NFS compliant |
| | Quota database | Filesystem function | Protocol stack function |
| Data integrity assuranse | | Per system-call | Per SMB/NFS request |

Table 3.1: Linux File Metadata and Protocol-specific File Metadata

As for the metadata types, the Linux file metadata uses necessary file attributes and security metadata for UNIX applications. Also, modern Linux file systems provide the extended attributes to add small file attributes to user files [XFS13, MCB+07]. Most Linux filesystems use original quota databases to manage filesystem usage.

On the other hand, the protocol-specific file metadata comply with the SMB and NFS specifications [Cor21b, Cor21c, For95, HN15]. The protocol-specific file metadata use the additional set of file attributes such as the file creation time and the archive attribute. The security metadata contain the security descriptors of multiple owner files [RSI12]. The named stream stores named user data with arbitrary size. The quota database stores the quota entries given by the protocol stack.

For the data integrity assurance, modern Linux filesystems use the journaling systems to ensure the data integrity. These journaling systems guarantee the data integrity per system call basis, so it does not ensure the data integrity among multiple system calls [PCA+14]. And therefore, if a system failure happens between user file update and update of the file metadata stored in the extended attributes or system files, the data integrity between the user file and the file metadata is not guaranteed. Contrary to Linux metadata, the protocol-specific file metadata is required to guarantee the atomicity of all updates during single client request processing.

The above differences cause less protocol functional coverage and the lack of data integrity assurance when the protocol stacks using the custom file metadata run on Linux filesystems. PMM needs to complement these differences.

### 3.3.2 PMM Architecture

PMM consists of the metadata management and the journal management. The metadata management stores the custom file metadata in Linux file systems to enable the capability of the

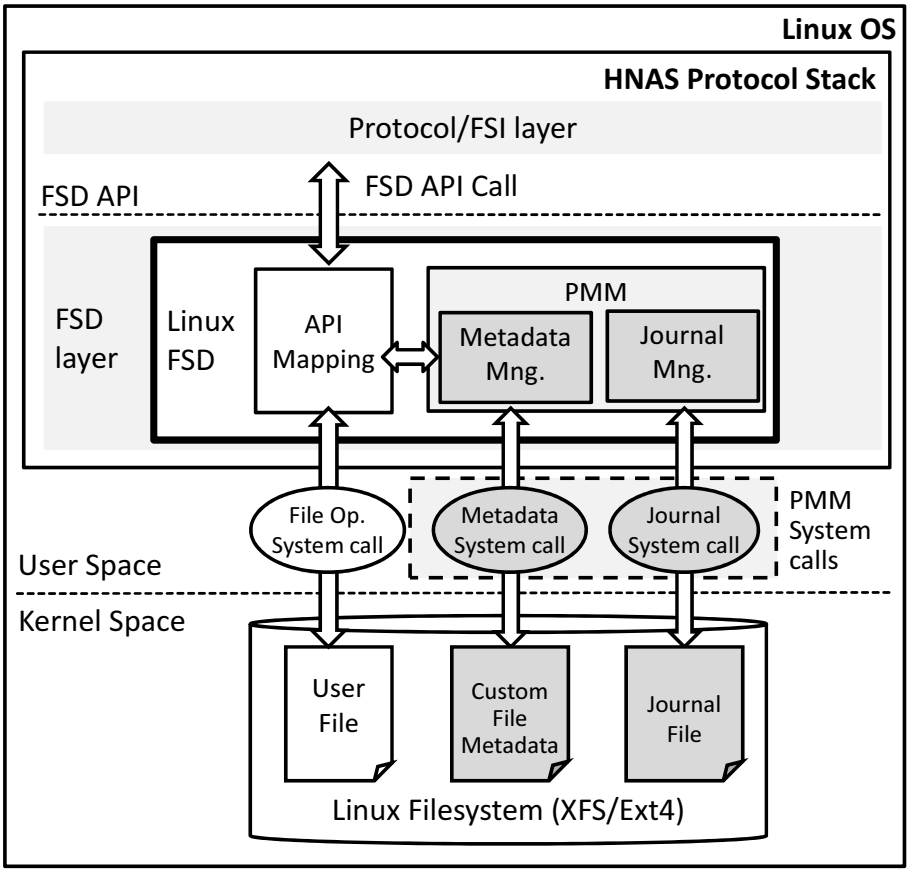



Figure 3.3: PMM Overview

protocol-specific file metadata on Linux servers. The journal management records the journal logs in the journal file to ensure the data integrity between user files and the custom file metadata.

I implemented PMM in the HNAS protocol stack. PMM works as an internal module in the new FSD module named Linux FSD. Linux FSD supports XFS and Ext4. An interface called FSD API clearly separates the FSD layer from the FSI layer in the HNAS protocol stack. The compliance with FSD API in Linux FSD enables the protocol layer and the FSI layer to work on Linux servers. Figure 3.3 shows the PMM overview.

Linux FSD maps an FSD API call to system calls for file operations. The file operations in FSD API are mostly compatible with system calls including open, read, write, and so on [BC05]. And therefore, the API mapping between FSD API and system calls for the file operations is straightforward.

Along with the file operations, PMM issues system calls for the metadata management and

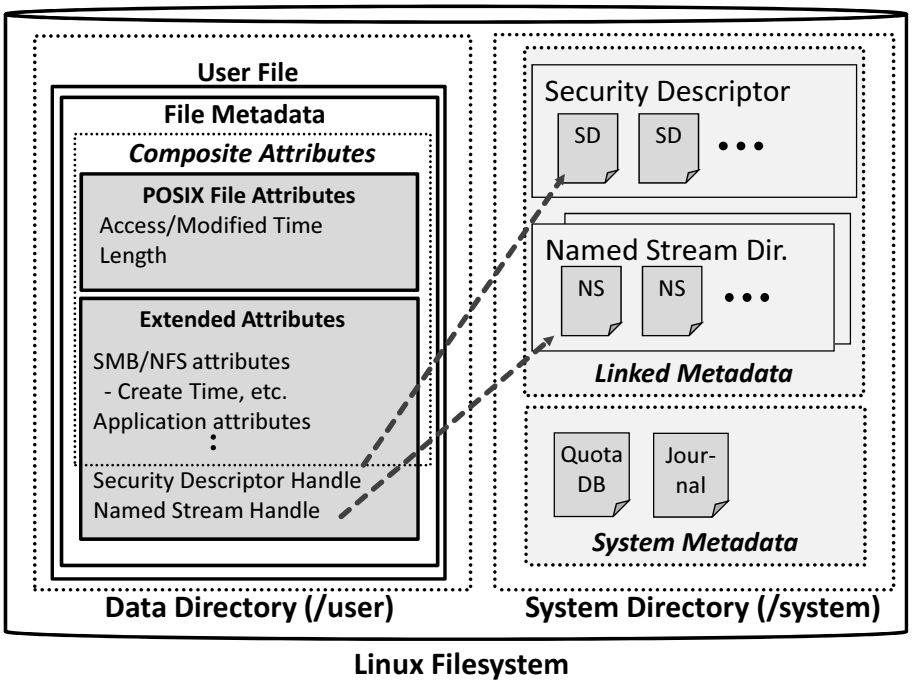



Figure 3.4: PMM Metadata Layout in Linux Filesystems

the journal management. Contrary to the file operations, the metadata management and the journal management are the original functions of PMM. I describe the details in the following sub-sections.

### 3.3.3 Metadata Management

The implementation of the metadata management consists of the namespace management, the extended attributes, and the open_by_handle system call [Cor10a]. PMM divides the namespace of Linux filesystems to the system directory and the data directory. PMM uses the extended attributes to store file attributes with fixed size and file handles of metadata files which contain metadata with arbitrary size. PMM uses the file handles and the open_by_handle system call to link the metadata files to one or more user files. These data structures enable the custom file metadata with arbitrary size or multiple-owner files in Linux filesystems. Figure 3.4 shows the metadata layout on Linux filesystems.

PMM offers three types of file metadata: the composite attributes, the linked metadata, and the system metadata. I explain these file metadata below.

**Composite attributes** PMM uses the composite attributes to offer pre-defined file attributes. The composite attributes contain SMB/NFS compatible file attributes and application attributes used in the HNAS protocol stack.

The storage of the composite attributes consists of the extended attributes and the POSIX file attributes of user files. PMM uses the extended attributes to store most part of the file attributes of the protocol-specific file metadata. On the other hand, PMM uses a small part of the POSIX file attributes and converts the POSIX file attributes to the compatible format with the custom file metadata. These POSIX file attributes include the access time, the modified time, and the file length which are updated during write system calls. The use of the POSIX file attributes eliminates the disk access to the extended attributes in the write request processing and improves write performance. I evaluate the performance improvement in the next section.

**Linked metadata** The linked metadata store the custom file metadata which has arbitrary size and multiple-owner files. PMM uses the linked metadata to store the named stream without the size limitation. And also, the capability of the multiple-owner files reduces the storage capacity of duplicated security descriptors likewise Windows NTFS [RSI12].

PMM creates metadata files or directories in the system directory for each linked metadata. PMM records the file handles of the linked metadata files and directories to the extended attributes of owner files. PMM uses the file handles and the open_by_handle system call to access the linked metadata. PMM manages the reference count of the linked metadata in the extended attributes of the linked metadata file. PMM deletes the linked metadata file when the reference count becomes zero to avoid that a linked metadata file becomes an orphan. In addition, PMM creates a metadata directory when the first named stream of a user file is created. PMM stores named streams of the user file to the same directory.

**System metadata** The system metadata contain the filesystem metadata such as the quota database and the journal log which is described in the next subsection. The system metadata is system files with pre-defined pathnames in the system directory. PMM provides an access interface of the system metadata files to the FSI layer through FSD API. The FSI layer stores arbitrary contents to the system metadata.

### 3.3.4 Journal Management

The journal management of PMM ensures data integrity between user files and the custom file metadata. PMM introduces user-space journaling to extend the scope of data integrity

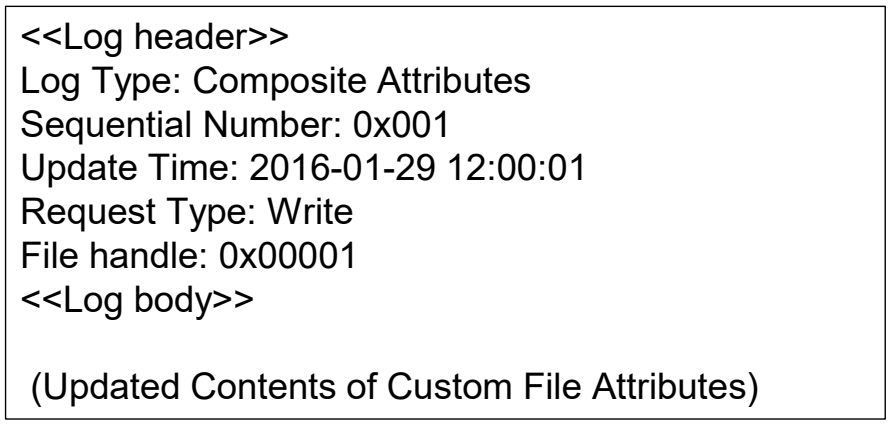



Figure 3.5: Example of Journal Log

assurance from per system call basis to per client request basis.

The custom file metadata could be inconsistent with user files without the journal management. PMM issues system calls to update user files and the custom file metadata separately. Without any additional protections, the user files and the custom file metadata could be inconsistent when a system failure happens between these system calls. These inconsistent states include the inconsistent file attributes, the inconsistent security descriptor or named stream handles, and the inconsistent linked metadata reference count. These inconsistent states could cause invalid file metadata, security violation, orphan metadata files, or unexpected system behaviors.

PMM offers journaling and replay processes to solve the above inconsistent states. I describe each process below.

#### Journaling

PMM manages a system file named the journal file to record all metadata updates as journal logs. PMM uses several types of journal logs corresponding to the above inconsistent states. A journal log consists of the log header and the updated metadata. A log header contains the necessary information for the replay process such as the log type, the sequential number, the update time, the file handle of a target user file, and request type. Figure 3.5 shows an example of the journal logs.

For each client request, PMM flushes journal logs to the journal file before issuing any system calls which update the target user files. An internal service thread, which processes client requests, adds the journal logs to a per-thread log buffer on memory. PMM merges

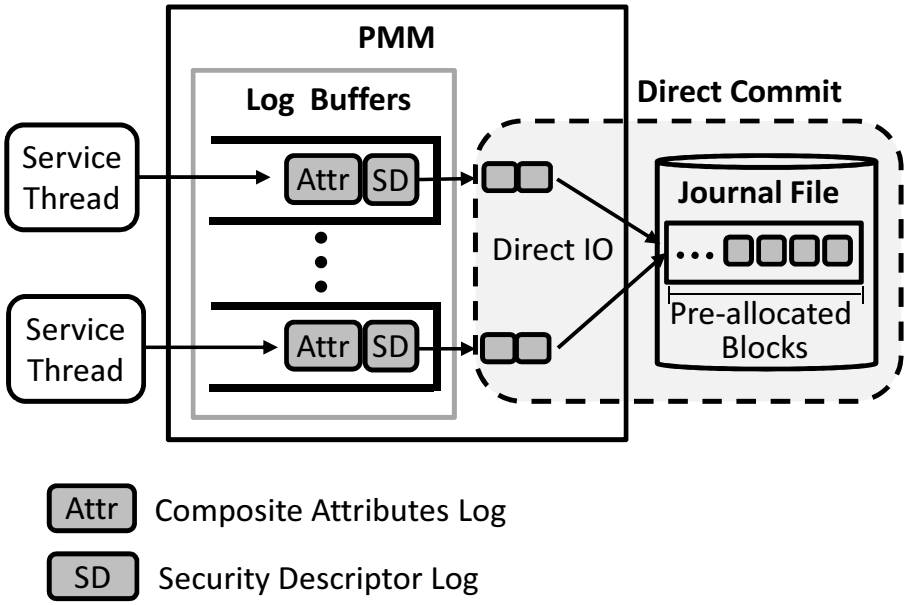



Figure 3.6: Journaling in PMM

the journal logs to a single disk write and flushes them to the journal file. PMM periodically restarts the journal file every several seconds so that the journal file does not consume a large disk capacity. Figure 3.6 shows the overview of the journaling processing.

PMM uses an optimization named the direct commit for reducing the journaling overhead. User-space journaling is known to cause a large delay due to the kernel journaling during synchronous journal file updates [LLS+15]. PMM preallocates the disk blocks of the journal file to avoid the delay. Also, PMM uses Direct IO to update the journal file to reduce the page cache overhead in the kernel space [BC05]. PMM uses pre-defined journal file size which is calculated from the journal log size and the required system throughput. If the total size of journal logs on the journal file exceeds the pre-defined file size, PMM appends the extra log items to the end of the journal file and truncates the journal file to the pre-defined size when PMM restart the journal file.

**Replay**

When a system failure happens, PMM starts the replay process in the next system reboot. In the replay process, PMM reads the journal logs on the journal file while the log headers contain the continuous sequential numbers.

For each journal log, PMM checks the existence and the POSIX modified time of the target user file given in the file handle field of the log header. PMM uses the open_by_handle system

| Spec. | Function Name | OSS | HNAS PS | | Spec. | Function Name | OSS | HNAS PS | | Spec. | Function Name | OSS | HNAS PS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | w/oPMM | wPMM | | | | w/oPMM | wPMM | | | | w/oPMM | wPMM |
| SMB1 | SMB command set | Yes | Yes | Yes | SMB2 | Message signing(SHA-256) | Yes | Yes | Yes | NFS3 | NLM/NSM | Yes | Yes | Yes |
| | User authentication | Yes | Yes | Yes | | Asynchronous operations | Yes | Yes | Yes | | portmap | Yes | Yes | Yes |
| | Remote administration | Yes | Yes | Yes | | Credit for flow control | Yes | Yes | Yes | | mountd | Yes | Yes | Yes |
| | Remote procedure call | Yes | Yes | Yes | SMB 2.1 | File lease | Yes | Yes | Yes | | rquota | Yes | No | Yes |
| | Named pipe | Yes | Yes | Yes | | Resilient file handles | No | Yes | Yes | NFS4 | NFSv4 command set | Yes | Yes | Yes |
| | Security descriptor | Yes | No | Yes | | Reauthenticate | Yes | Yes | Yes | | Mandatory auth. | Yes | Yes | Yes |
| | Oplock | Yes | Yes | Yes | | Dialect negotiation | Yes | Yes | Yes | | UCS file name | No | Yes | Yes |
| | File change notificatio | Yes | Yes | Yes | SMB3 | Transparent failover | Yes | No | Yes | | NFSv4 ACLs | No | No | Yes |
| | Distributed file system | Yes | Yes | Yes | | Multi-channel | Yes | Yes | Yes | | File delegation | Yes | Yes | Yes |
| | SMB file attributes | Yes | No | Yes | | Witness | No | Yes | Yes | | Compounded request | Yes | Yes | Yes |
| | Unicode file name | Yes | Yes | Yes | | Signing(AES-CMAC) | Yes | Yes | Yes | | Named attribute | No | No | Yes |
| | Long file name | No | No | No | SMB Other | Remote shared virtual disk | No | Yes | Yes | | quota | No | No | Yes |
| | 8.3 name | No | No | Yes | | Volume shadow copy | No | No | No | | Pseudo-fs | Yes | Yes | Yes |
| | Case insensitive FN | Yes | No | Yes | | Access-based enum. | Yes | Yes | Yes | NFS/ SMB | Mixed mode security | No | No | Yes |
| | Named stream | No | No | Yes | | Auditing event log | Yes | Yes | Yes | | Inter-protocol file lock | No | Yes | Yes |
| | Singning(MD5) | Yes | Yes | Yes | | Virus scan | Yes | No | Yes | iSCSI | SCSI access | Yes | Yes | Yes |
| SMB2 | SMB2 command set | Yes | Yes | Yes | NFS3 | NFSv3 command set | Yes | Yes | Yes | | Persistent reservation | Yes | Yes | Yes |
| | Durable open | Yes | Yes | Yes | | Unix permissions | Yes | Yes | Yes | | Multi-path | Yes | Yes | Yes |
| | Symbolic link | Yes | No | Yes | | DES encrypted auth. | Yes | Yes | Yes | | Path load balancing | Yes | Yes | Yes |
| | Compounded request | Yes | Yes | Yes | | Kerberos | Yes | Yes | Yes | | iSNS | Yes | Yes | Yes |
| Functional coverage: OSS 47/60(78%), HNAS PS without PMM 45/60 (75.0%), HNAS PS with PMM 58/60 (96.7%) | | | | | | | | | | | | | | |

Table 3.2: Protocol Functional Coverage Evaluation

call to find the target user file. If the target user file is created, deleted, or modified after the update time in the log header, PMM reflects the updated metadata to the target metadata based on the log type. Otherwise, PMM discards the journal log because the system calls which update the target user files do not take effect at the time of the system failure.

## 3.4 Evaluations

### 3.4.1 Protocol functional coverage

I evaluated the protocol functional coverage of the HNAS protocol stack on Linux servers. I listed up 60 HNAS protocol functions from SMB, NFS, and iSCSI specifications [Cor21b, Cor21c, For95, HN15, BHH+04]. The SMB includes not only SMB specifications but also other specifications used in the SMB environment such as the volume shadow copy (VSS). Also, NFS and iSCSI include other relating specifications

I evaluated the protocol functional coverage of the HNAS protocol stack without and with PMM (HNAS PS w/oPMM and wPMM) on Linux OS. I also evaluated the functional coverage of the conventional OSS protocol stacks (samba and NFS-ganesha). Table 3.2 shows the evaluation result.

PMM improves the functional coverage from 75.0% to 96.7%. PMM improves the HNAS protocol stack to achieve higher functional coverage than 78% of the conventional OSS proto-

| Item | | Description |
|---|---|---|
| Hardware (Server/ Client) | CPU | Intel Xeon E5-2620 (6 cores, 2.0 GHz) |
| | RAM | 12GB |
| | Storage | Seagate Constellation.2 ST9500620NS SATA 6Gb/s 500GB 7200 rpm |
| | NIC | Broadcom NetXtreme II BCM57810 10 Gigabit Ethernet |
| Software (Server) | OS | Debian Linux Wheezy (kernel 3.16.0-0) |
| | Filesystem | XFS (inode size 512 byte) |
| | NFS server | HNAS Protocol Stack, linex kernel server, NFS-ganesha |
| Software (Client) | OS | Debian Linux Jessie (kernel 3.16.0-4) |
| | Benchmark | filebench v 1.4.9.1 |

Table 3.3: Experiment Configuration

col stacks. The security descriptor enables SMB security descriptor, NFSv4 ACL, and Mixed mode security. The named stream enables SMB named stream, SMB2 symbolic link, SMB3 transparent failover, and NFSv4 named attribute. The quota database enables NFSv3 rquota and NFSv4 quota. The composite attributes enable other improvements.

PMM does not cover SMB long file name and Volume shadow copy because of the lack of corresponding functions in Linux filesystems. PMM is supposed to be capable of these features if Linux filesystems or other modules provide the correspondence functions.

### 3.4.2 Access Performance

#### Measurement Environment

I used a commodity server Dell R610 as an NFS server in the following evaluations. I used another server with the same configuration as an NFSv3 client. The client accesses the server via 10 Gbps network. I used a SATA disk on the server so that I can evaluate the impact of the metadata access increase of PMM in the disk bottleneck environment. Table 3.3 shows the experiment configuration.

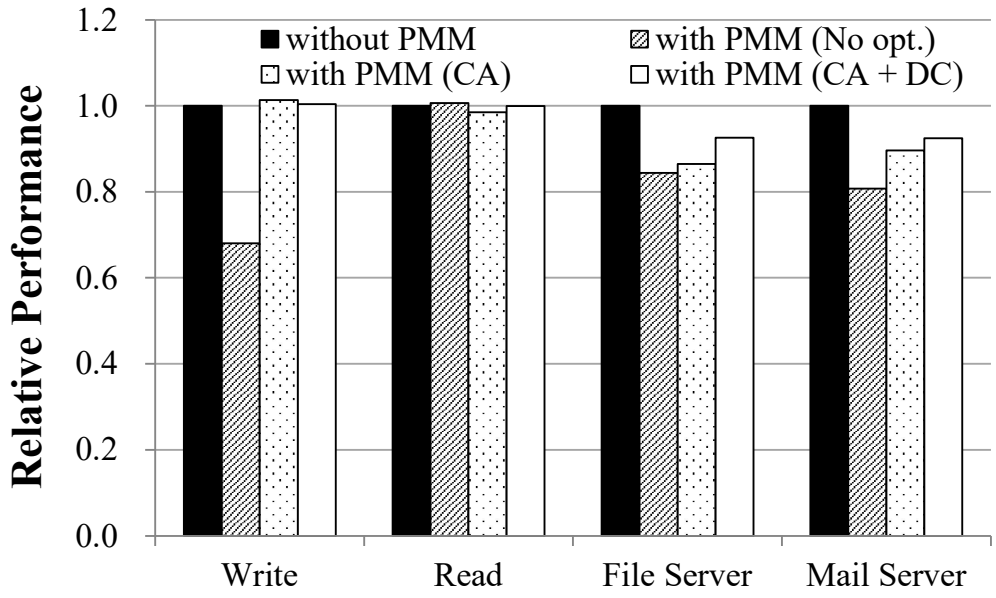



Figure 3.7: NFS Performance with and without Optimizations

**Measurement Results**

The measurements used filebench [TZS16] to measure the performance of the typical file server performances including read, write, file server, and mail server performance. The measurements used the fivestreamwrite and fivestreamread workloads of filebench to measure the read and write performance. The measurements used the fileserver workload for the file server performance and the varmail workload for the mail server performance. The fileserver workload consists of create, write, open, append, read, close, delete, and stat to middle-size files. The varmail workload consists of create, write, fsync, read, open, and close to small files. In the measurements, the total file-set size is set to at least two times larger than the size of server and client memory to avoid cache effects.

First, I evaluated the performance overhead of PMM. I evaluated the performance of the HNAS protocol stack without PMM and with PMM. In the evaluations with PMM, I measured the performance without any optimizations (no opt.), with the composite attributes optimization (CA), and with the direct commit (CA + DC). Figure 3.7 shows the measurement results.

The optimizations reduce the performance degradation of PMM from 32% to 0% in the write performance, 16% to 7% in the file server performance, and 20% to 8% in the mail server performance. For the read performance, no overhead was confirmed because the workload

does not involve PMM overhead.

As shown in Table 3.4, the decrease in the number of disk writes for a single filebench operation explains the reason for these performance improvements. In the write workload, the composite attributes optimization reduces the number of disk writes by over 75%. In the write workload, filebench issues 1MB write system calls which are split into 64 KB NFS write requests by the NFS client module. If the composite attribute optimization is not applied, PMM issues a disk write for updating the extended attributes in each 64 KB NFS write request. The composite attributes optimization eliminates these disk accesses during the NFS write request processing. In addition, 64KB NFS write requests are sequentially processed and merged into a single 128 KB disk write by Linux OS [BC05]. Theoretically, the composite attributes optimization decreases the number of disk writes from 24 to 8 for a single 1 MB write operation.

Table 3.4: Number of Disk Writes per Filebench Operation

| | Write | File server | Mail server |
|---|---|---|---|
| without PMM | 8.51 | 0.82 | 0.91 |
| with PMM (No opt.) | 25.14 | 1.29 | 1.46 |
| with PMM (CA) | 8.16 | 1.05 | 1.13 |
| with PMM (CA + DC) | 8.30 | 1.02 | 1.08 |

Also, the direct commit reduces the number of disk writes in the file server workload and the mail server workload by 2% and 5% respectively by eliminating the overhead in the kernel journaling and the page cache management.

Secondary, I evaluated the performance of the HNAS protocol stack (Proposal) against NFS-ganesha (ganesha) [DLL07] and the kernel nfs daemon (knfsd). As we can see in Figure 3.8, the HNAS protocol stack achieves about 10% and 20% better performance than OSSs in the write performance and the read performance respectively.

The HNAS protocol stack issues disk IOs to a single file as continual as possible to make the disk IOs more sequential. This optimization is supposed to contribute the better read and write performance than OSSs.

Even with the overhead of PMM, the HNAS protocol stack achieves equivalent performance with OSSs in the file server evaluation. As the file server workload uses 128 KB files on average, the better write and read performance are supposed to contribute to this result.

On the other hand, the HNAS protocol stack shows 20% lower performance in the mail server evaluation which uses 16 KB file size. This result suggests that the HNAS protocol

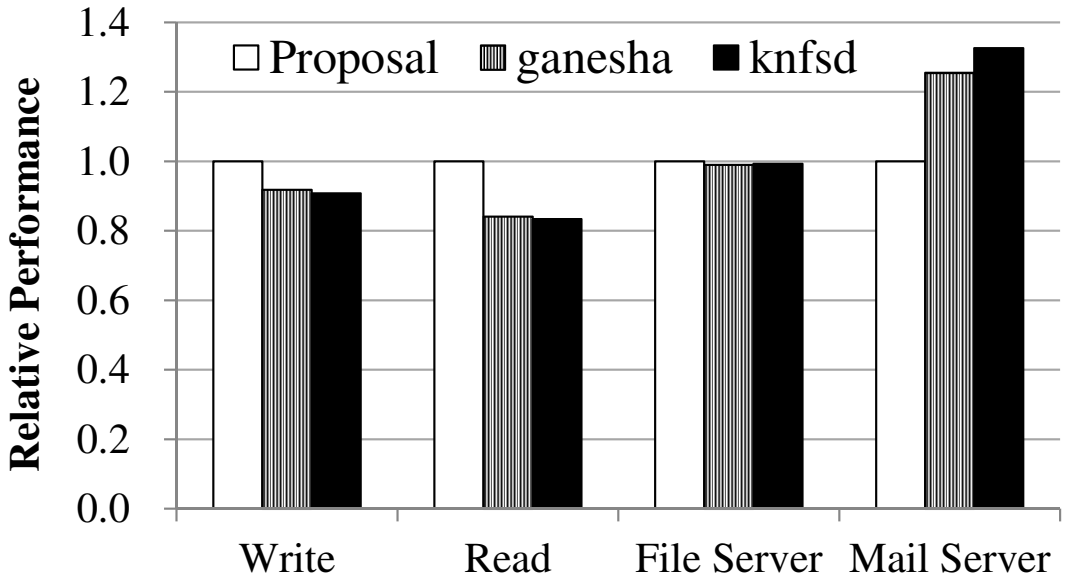



Figure 3.8: Performance Comparison with OSS Implementations

stack has lower performance than OSSs in the metadata-intensive workload because of the PMM overhead and its original performance characteristics.

As write, read, and file server workloads are the typical file server workloads, it can be said that PMM achieves the competitive performance against OSSs in the HNAS protocol stack.

### 3.4.3 Recovery Performance

I evaluated the recovery performance of the PMM journal management upon a system failure. In case of a system failure, PMM recovers the data integrity of the custom metadata when the HNAS protocol stack mounts the failed filesystem. I evaluated the amount of time to mount the failed filesystem with and without the PMM journal management. Also, I evaluated the performance of the exhaustive check which is used in the conventional filesystem utilities like fsck [SWZ05]. The exhaustive check scans all files in the failed filesystem and recovers the consistency of metadata if necessary.

In the evaluations, I intentionally caused system crashes while issuing file creation and deletion requests to the HNAS protocol stack. The evaluations were carried out with the different numbers of stored files to see the scalability of the recovery methods.

Figure 3.9 shows the measurement results.

The PMM journal management shows a similar mounting time with the mounting time without any recovery processing. The mounting time is only about 2 seconds when filesystem

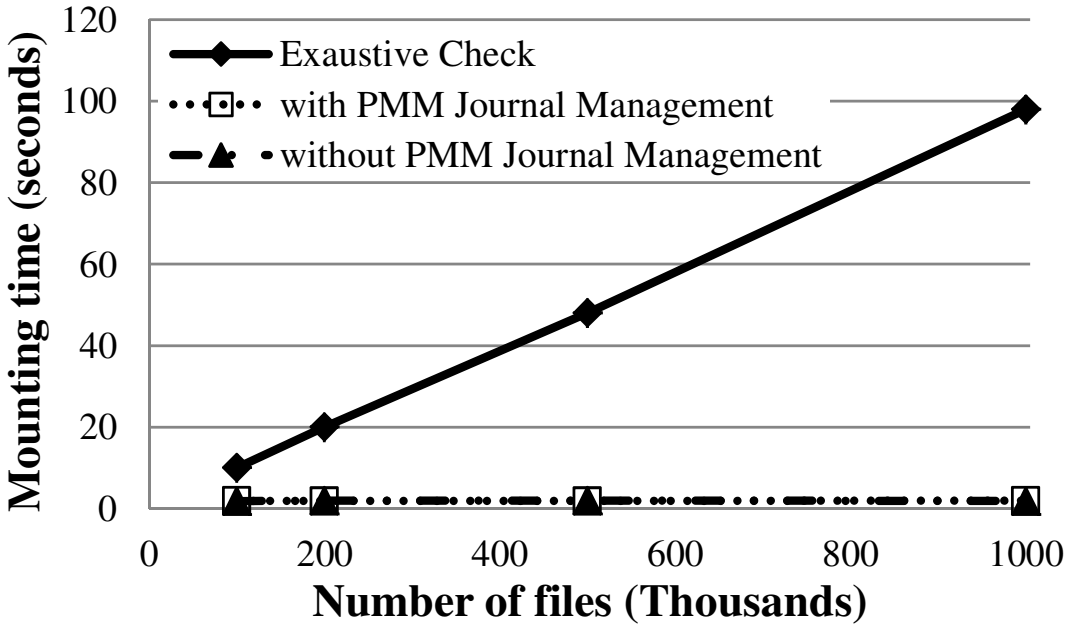



Figure 3.9: PMM Recovery Performance

contains one million files. The recovery process of PMM examines a small number of journal logs regardless of the total number of files in the filesystem. Therefore, the recovery time remains constant and short even if a filesystem contains a large number of files.

On the other hand, the mounting time of the exhaustive check increases in accordance with the number of files. The mounting time reaches nearly 100 seconds at one million files.

These results show that the journal management of PMM offers fast and scalable recovery upon a system failure.

## 3.5 Related Work

The OSS community has developed protocol stack implementations such as the kernel nfs daemon, NFS-ganesha, and samba [DLL07, sam21]. Some of these implementations use the extended attributes and system files to store the protocol compliant file attributes or the named stream in a similar way with the proposal. However, these OSSs do not ensure the data integrity of these protocol-specific file metadata upon system failures whereas PMM does. RichACL stores the security descriptor to the extended attributes of each user file in Linux filesystems likewise PMM [KGB10]. However, this approach imposes larger storage consumption of security descriptors than the proposed multiple-owner metadata approach. And also, PMM allows the reuse of the proprietary protocol stacks with multiple protocol support and third vendor ap-

plication supports. These capabilities enable advanced protocol functions such as mixed mode security, the inter-protocol file lock, and the certified proprietary virus scan support whereas OSSs do not support these functions.

Much research in recent years has focused on porting software of purpose-built appliances to commodity servers. Burtsev et al. proposed an efficient inter-virtual machine (VM) communication for software of purpose-built NAS appliances running on VMs [BSR+09]. Also, NetApp released the VM implementation of its NAS appliance to deploy the virtual appliance to the software-defined storage [Pas16]. These studies focus on the protocol stacks using software-based filesystem in the VM environment, and therefore their methods are not applicable to the protocol stack using the hardware-based filesystem like HNAS.

The data consistency guarantee of filesystems has been studied for a long time [BKL+16, PCA+14]. As for data integrity assurance in the kernel space, Verma et al. showed that kernel modifications allow user-space applications to ensure data integrity of application data [VMP+15]. However, these kernel changes are available only in their filesystems. PMM is applicable for Linux filesystems which are widely used. As for user-space data integrity assurance, many works have been done in the database research [www21, ZTH+14, TCW16]. This study focuses on the application of the user-space data integrity assurance to the network storage protocol stack.

Recently, Linux API is attracting widespread interest [AAG+16, TJAP16]. I think this study shows an application of Linux API to store the protocol compliant file metadata in Linux filesystems.

## 3.6 Chapter Summary

This chapter proposes a method to adapt storage protocol stacks using the custom file metadata to commodity Linux servers. I introduced a new metadata framework named the custom file metadata to accommodate the protocol-specific file metadata in Linux filesystems. I developed a new metadata management module named PMM which enables the protocol stack to use the custom file metadata on Linux servers. I implemented PMM in the protocol stack of a storage appliance, HNAS, to prove the concept of PMM.

PMM utilizes Linux APIs such as the namespace management, the extended attributes, and the open_by_handle system call to store the custom file metadata in Linux filesystems. PMM enables protocol stacks to achieve higher protocol functional coverage by using the custom file metadata on Linux servers

PMM consists of the metadata management and the journal management. These new modules complement the differences between the HNAS file metadata and POSIX file metadata. Also, I introduced performance optimizations to reduce the performance overhead of PMM.

The evaluations showed that PMM improves the coverage of the HNAS protocol functions from 75.0% to 96.2% on Linux servers. With PMM, the HNAS protocol stack achieves higher functional coverage than the OSS protocol stack. The proposed performance optimizations reduce the performance degradation of PMM to up to 8% in the typical file server workloads. As a result, the HNAS protocol stack with PMM shows the competitive performance against OSS protocol stack implementations. Also, we confirmed that the PMM journal management enables fast and scalable recovery processing for the data integrity assurance of the custom file metadata.

PMM allows the proprietary protocol stacks to achieve high protocol functional coverage while offering reasonable access performance and ensuring data integrity on commodity Linux servers.

# Chapter 4

# Dynamic Redundancy Control with Adaptive Encoding

This chapter proposes a performance improvement method using the custom file metadata for server-based storage. This chapter applies the cusom file metadata shown in Chapter 3 to establish high-performance redundancy control between servers. This chapter proposes DRC-AE which is the second application of the custom file metadata.

In recent years, commodity-server-based storage has been widely used as a low-cost alternative to dedicated-storage appliances [Mat12, vmw21]. Because server-based storage uses inexpensive commodity servers, hardware cost becomes much lower than that of a custom-made storage appliance. Initially, server-based storage had been used for cloud computing environments that require large capacity and high access throughput for aggregated computing resources. Currently, server-based storage is also applied to on-premises computing environments for data analytics and consolidated in-house systems [Kim09]. A typical server-based storage system has adapted scale-out distributed storage to achieve high reliability and high scalability required for these environments [San03].

Because of the high failure rate of typical commodity servers, data redundancy across multiple servers is a necessary requirement for a server-based storage system [BHRM18]. In general, components such as memory or disks in a commodity server are not redundantly configured, whereas redundant configurations are common in storage appliances. Failure of a device in a commodity server directly causes service unavailability or even loss of data if there is no redundancy associated with the failed device's data. To avoid such unacceptable events, conventional server-based systems use data redundancy methods such as three-way replication or EC [WBM+06, Bor07, BBP12].

Although data redundancy improves the reliability of server-based storage, it results in a considerably increased cost for the extra storage capacity. If data are replicated to other servers, the capacity requirement linearly increases with the amount of replicated data. This capacity requirement significantly increases both the number of servers and the overall system cost.

Much research in recent years has focused on EC as an approach to reduce the amount of redundant data [Nar16]. With EC, data are encoded with an arbitrary number of data symbols and parity symbols, therefore increasing the number of data symbols enables higher capacity efficiency. By contrast, the parity calculation required for EC involves much more network, CPU, and disk-access overhead in overwrite operations and read operations under a server failure than those in simple replication. To mitigate the performance overhead, methods for combining replication and EC have been investigated in previous work [FTXG11, HSX+12, XSBP15, CAA+17]. These methods switch the redundancy method of user data between replication and EC according to the application workload.

However, these approaches require distributing data over multiple servers for data redundancy; therefore, the performance overhead of the distribution of data remains a concern. Distributing data requires a storage system to process a large amount of network traffic and data placement management, as seen in today's distributed storage implementations [LJH+17, AWK+19]. In particular, the performance impact becomes substantial if the system is applied to random-intensive applications such as DB systems or VDI. These applications had commonly been used in stand-alone legacy systems. Because the legacy systems are moving to the cloud computing environments and on-premises infrastructures for reducing system costs [BCX+12], there has been increasing demand for a server-based storage system to be used for these random-intensive applications [CSGK11, LPGM08, spe17, XSG+15]. As a result, the performance concerns have become major issues for applying a server-based storage system to these applications.

In addition, the advent of high-performance disks such as nonvolatile memory express solid-state drives (NVMe SSDs) and storage-class memory (SCM) has increased the performance impact of distributing data [XSG+15, IYZ+19]. The high performance of these devices enables ultra-low latency and high access throughput for an application. If network communication is involved in data-access processing, it causes higher access latency and less access throughput. To utilize fully the performance of these devices, it is required for an application to directly access these devices installed on the same server.

Most conventional methods use dedicated metadata servers to manage data locations and the redundancy method of user data [FTXG11, XSBP15]. Although the use of metadata servers

enables combining replication and EC to reduce redundancy data, it requires remote metadata access for an application to access data. The performance overhead of remote metadata access deteriorates access performance even if an application can directly access its data in local storage.

This chapter proposes a new redundancy control method called DRC-AE for a server-based storage system. DRC-AE adopts a local-filesystem-based approach that stores user data in the local filesystem while adding data redundancy to the user data. DRC-AE eliminates the need for remote metadata access by storing the metadata in the local filesystem. The local filesystem-based approach enables DRC-AE to achieve high access performance under random-intensive workloads. In addition, DRC-AE uses fine-grained adaptive encoding to support random-intensive workloads. The adaptive encoding accurately identifies cold data by monitoring data-access frequency at a fine granularity. The adaptive encoding controls the amount of EC data according to workloads to maximize capacity efficiency against performance requirements. DRC-AE enables a server-based storage system to achieve both high performance and high capacity efficiency under random-intensive workloads.

To investigate the effectiveness of my proposal, I evaluated the access performance of DRC-AE under data-intensive workloads and metadata-intensive workloads in comparison with CephFS, a widely used file-based distributed storage [UHS17]. These investigations show that DRC-AE achieves higher performance except for sequential write performance in comparison with CephFS in a high-performance disk configuration. Furthermore, I confirmed that DRC-AE can achieve higher capacity efficiency than conventional methods while maintaining high performance.

The main contributions of this chapter are as follows:

- this chapter proposes a local filesystem-based dynamic redundancy control for server-based storage that achieves high access performance in a high-performance disk configuration
- this chapter proposes using the fine-grained adaptive encoding to improve capacity efficiency while maintaining high performance under random-intensive workloads
- this chapter demonstrates that the proposed method achieves higher access performance than CephFS except for sequential write in a high-performance disk configuration
- this chapter demonstrates that the proposed method achieves higher capacity efficiency than the conventional methods under random-intensive workloads while enabling a tunable trade-off between performance and capacity efficiency

This chapter is an extended version of our prior paper [FLY21b].

## 4.1 Conventional Approaches

This section provides an overview of server-based storage and conventional approaches for dynamic redundancy control. Then, I discuss where challenges exist for the conventional approaches.

### 4.1.1 Server-based Storage

Server-based storage is a storage system that consists of many inexpensive commodity servers. Because of the low-cost of commodity servers, the cost of a server-based storage system is generally lower than a traditional dedicated-storage appliance. Server-based storage initially evolved as back-end storage of cloud computing environments, and soon became widespread in on-premises storage infrastructures. Typical server-based storage systems use scale-out distributed storage to meet the requirements of high reliability and high scalability for these large-scale systems [San03, HSX+12, Bor07, Kim09].

The basic idea of server-based storage came from early implementations of cloud storage. GFS, prominent cloud storage, achieved high sequential access throughput by aggregating low-performance commodity servers [San03]. GFS specializes in workloads of cloud computing environments such as streaming read and append-only write to large files to achieve higher scalability. GFS offers a limited set of operations that is not compatible with the standard portable operating system interface (POSIX) [IG18] to avoid reducing scalability for supporting complex POSIX operations. As GFS stores data among a large number of unreliable commodity servers, the high frequency of server failures had become a problem. GFS initially introduced three-way replication, which stores two replicas of user data on other servers, to achieve high reliability on unreliable commodity servers. Then, EC was applied to reduce the amount of replica data in three-way replication that triples the storage consumption [Fik10]. The design of GFS has had a significant impact on other cloud storage systems [HSX+12] and scale-out distributed storage implementations such as HDFS and CephFS [Bor07, WBM+06].

With the proliferation of server-based storage, a wider range of applications are now using server-based storage systems. These applications include legacy POSIX-based applications [CDLC14, ZLL+20, MNE+14]. The open source software community and storage vendors have provided POSIX-compatible distributed storage such as CephFS and proprietary distributed storage systems for these applications [UHS17, CAA+17, vmw21]. These storage

implementations support the standard POSIX operations so that legacy applications can run on the storage systems without modifications.

Among the expansion of applications, server-based storage has been increasingly used as back-end storage for legacy server applications such as DB and VDI. Currently, many legacy systems are moving to cloud computing environments and consolidated on-premise infrastructures to reduce system costs. This movement makes more legacy server applications to use server-based storage systems. Because random access is dominant in these applications [XSG+15, spe17, LKM+17], the performance requirements for a server-based storage system have been broadened from the original sequential-intensive workload of the early cloud storage systems.

The advents of NVMe SSDs and SCM have also changed performance requirements for a server-based storage system. While GFS used hard disk drives (HDDs) in its original paper, the performance of the latest commodity NVMe SSD enables thousands of times better throughput and a hundred times better latency compared with an HDD. Furthermore, SCM enables much higher disk performance than an NVMe SSD [IYZ+19]. Because of the high performance of NVMe SSDs and SCM, there have been expectations for much higher performance for a system using high-performance disks.

To meet the expectations for high performance, there is a trend toward system configurations specialized for data locality to utilize high-performance disks. Major cloud service providers have offered storage-optimized VM instances that store data into local NVMe SSDs so that an application can access the local data directly [Ser20, Mic20, XZC+20]. The storage-optimized VM instances can achieve much lower access latency and higher throughput than VMs using conventional cloud storage. For on-premises environments, the enterprise storage vendor NetApp offers a proprietary storage appliance that uses local SCM as an upper storage tier of external enterprise storage so that an application can access local SCM without involving remote data access [Sha20].

### 4.1.2 Data Redundancy Control in Server-based Storage

EC is a promising method to reduce the capacity consumption of replica data even though it entails performance overheads of the parity calculation [Pla13]. EC enables much less storage consumption of parity data than replica data. For example, when EC with six data symbols and two parity symbols is used, capacity consumption of parity data is one-third of the original user data, whereas three-way replication requires twice the capacity of the original data for replica data. By contrast, the performance overheads of overwrite operations and read operations under

a server failure become critical issues when EC is used for workloads with these operations.

Data redundancy control has been investigated in previous studies to utilize the high capacity efficiency of EC without introducing large performance degradation. Data redundancy control switches the redundancy method of user data between replication and EC according to workloads. For example, DiskReduce [FTXG11] and HACFS [XSBP15], which are implementations of HDFS-based data redundancy control, store a newly created file with three-way replication and encode the file with EC if there are no more updates to the file. In addition to DiskReduce and HACFS, some cloud storage implementations adopt data redundancy control [HSX+12, KBP+12]. CephFS also applies a variant of data redundancy control that uses its cache tiering function [UHS17].

A simple method to realize data redundancy control is to divide storage capacity into a storage pool for replicated data and a storage pool for EC data and migrate data between them. CephFS and AzureStorage use this approach [UHS17, HSX+12]. Although this approach is simple and applicable for systems that need different storage control for replicated and EC data, it has limitations. The overhead of data migration between pools becomes considerably high and introduces a negative impact on access performance. Moreover, this approach cannot dynamically change the capacity ratio of replicated and EC data. Therefore, the capacity of replicated data easily becomes excessive or insufficient for a target workload in terms of performance.

The HDFS-based prior studies illustrate another approach that eliminates the need for data migration between the replication pool and the EC pool [FTXG11, XSBP15]. DiskReduce and HACFS enable data redundancy control by replacing replica data of user data with parity data. This approach uses metadata servers to manage data placement and data redundancy of user data. The metadata servers periodically select replicated user files with no more updates from multiple servers and generate parity data for them. After storing the parity data to the storage pool, the metadata servers remove the replica data of the encoded user files to reduce capacity consumption. This approach avoids the data migration cost and enables flexible control of the capacity ratio of replicated and EC data.

### 4.1.3 Challenges

With the growing use of server-based storage, there is a need for a server-based storage system to provide high performance for more workloads than just conventional sequential access. Scale-out distributed storage has an advantage in scalable sequential access throughput, but the network access overhead tends to be negative for other workloads. A large amount of network

traffic and data placement management overhead entails greater access latency and higher performance requirements for a CPU. In particular, for a system using high-performance disks, network bandwidth or CPU performance easily becomes the performance bottleneck. Especially for random-intensive applications, the performance impact tends to be substantial.

For random-intensive workloads, data locality requires more focus to achieve higher performance. If an application uses local data on the same server, it can fully utilize the disk performance without involving remote network access. In a system with high-performance NVMe SSDs or SCM, the performance gain of local data access becomes significant.

However, the requirements for random-intensive workloads and data locality introduce other challenges to data redundancy control, as described below.

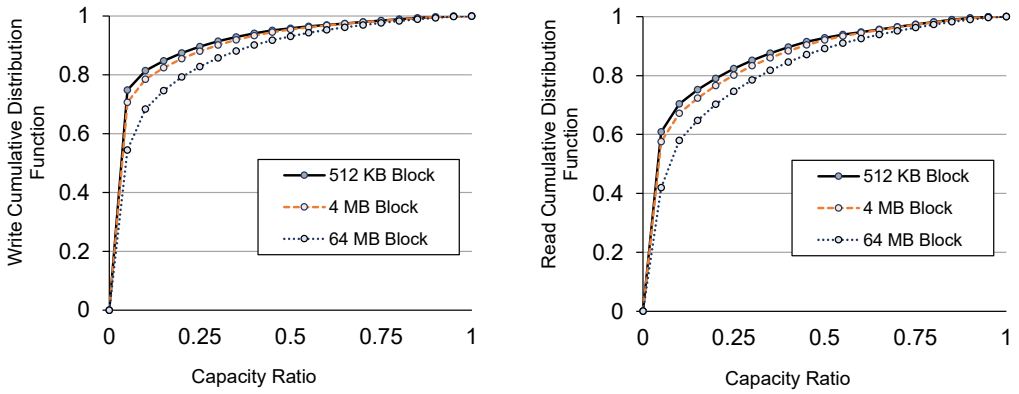
**Metadata access overhead** The HDFS-based conventional data redundancy control implementations use centralized metadata servers to control data placement and data redundancy of user data. The use of metadata servers enables switching the redundancy method of local user data without migrating the data to other servers. This approach can preserve the data locality of user data because the encoded data stays in the same server after encoding.

However, when using the metadata servers for redundancy control, the performance overhead of remote metadata access becomes a concern. When the metadata servers manage data placement and data redundancy, an application needs to ask the data location to the metadata servers for each data access [Gib10]. The overhead of remote metadata access becomes substantial under random-intensive workloads because of the higher requirement for access latency of these workloads. In addition, the metadata servers could be the performance bottleneck when the amount of metadata access increases.

These performance overheads of metadata access could undermine the performance advantage of local data access.

**Efficient EC data selection for random-intensive workloads** In data redundancy control, an efficient algorithm to select target data for encoding is essential to achieve high capacity efficiency. This EC data selection algorithm needs to match the characteristics of a target workload. For random-intensive applications such as DB or VDI, the EC data selection algorithm is required to adapt to the workload characteristics of these applications.

One of the main characteristics of these workloads is the high percentage of overwrite operations. Zhang et al. reported a high percentage of overwrite operations in Microsoft Research (MSR) traces, which are 1-week block IO traces of legacy enterprise servers [ZLL+20,




(a) Write IO Distribution

(b) Read IO Distribution

Figure 4.1: Cumulative Distribution Functions of Read and Write Access with Different Chunk Sizes for MSR traces

NDR08]. In MSR traces, the percentage of overwrite operations exceeds 50% in more than 60% of traces and about 40% of traces contain 65% to 85% overwrite operations.

Another characteristic of these workloads is small-size access with access-skew. The access sizes of random-intensive applications are generally smaller than the block sizes of scale-out distributed storage implementations. The average block sizes of DB and VDI are usually several KBs to several tens of KBs, whereas HDFS uses 64 MB blocks and CephFS uses 4 MB blocks by default [spe17, LKM+17]. If the block size is larger than the access size, it is difficult for an EC data selection algorithm to distinguish cold data from other data accurately. I analyzed MSR traces to investigate the impact of the large block sizes on the amount of distinguishable cold data. Figure 4.1 shows the cumulative distribution functions of the number of read operations and write operations against capacity with different block sizes. The analysis investigated the average cumulative distribution functions of all MSR traces for 64 MB blocks of HDFS, 4 MB blocks of CephFS, and 512 KB for the finer-grained blocks. The results show apparent Zipf-like distributions that indicate the existence of access-skew in both read and write operations [YZ16]. We can see that a larger block size requires a larger amount of capacity for the same number of operations. With 64 MB blocks, 90% of write accesses occur in 40% of capacity and 90% of read accesses occur in 55% of capacity, whereas with 512 KB blocks, these accesses occur in 25% and 40% of capacity, respectively.

The HDFS-based conventional methods could cause low-capacity efficiency for random-intensive workloads because of the different characteristics of these workloads from HDFS. These methods assume that data are immutable in HDFS. DiskReduce encodes data blocks

within a file once the file becomes immutable, whereas HACFS encodes blocks within a file with old modified times. These approaches make data remain replicated while overwrites to the file continue even if the overwrite frequency is sufficiently low. These approaches are thought to be inefficient for random-intensive workloads, which contain a high percentage of overwrite operations. In addition, the HDFS-based conventional methods use 64 MB blocks for the encoding unit. This large block size of the conventional methods decreases the amount of distinguishable cold data under random-intensive workloads, as shown in Figure 4.1. The indistinguishable cold data remain replicated and continue consuming storage capacity. Although the block size of HDFS is a tunable parameter, a smaller block size considerably increases the processing load and memory usage of the metadata servers.

Furthermore, for workloads with overwrite operations, the EC data selection algorithm needs to prevent an unacceptable level of performance degradation because of the EC performance overhead. For random-intensive workloads, the performance degradation of overwrite operations is a crucial problem. If the amount of overwrite operations to EC data increases without a limit, performance could decrease to an unacceptable level.

The low-capacity efficiency and the possible unacceptable performance degradation make the conventional approaches of data redundancy control undesirable for random-intensive workloads.

## 4.2 Motivation

This chapter aims to establish a new lightweight data redundancy control for a server-based storage system with random-intensive workloads. The new method achieves high performance and high capacity efficiency for these workloads. To achieve this aim, this chapter adopts the following two approaches.

**Lightweight Dynamic Redundancy Control Using Local Filesystem** Although the HDFS-based conventional methods improve capacity efficiency, its performance overhead of remote metadata access is inevitable. The overhead causes significant performance decreases in access performance even if an application can access local data stored in the same server directly.

Therefore, this chapter aims to develop a new redundancy control method that utilizes both replication and EC capability while achieving data locality. This approach uses a local filesystem to store user data and metadata to reduce network communication in data-access processing. Because the common local filesystem does not offer data redundancy functions, I

introduce a new redundancy control layer on top of the local filesystem to add both replication and EC capability. The new redundancy layer copies local data to one or more other servers for replicated data, whereas it also applies EC to the local data of multiple servers to reduce the amount of redundant data. The new redundancy layer in each server stores configuration information for EC in the additional metadata of a local file; therefore, no remote metadata access is necessary during data-access processing. The proposed method enables a lightweight dynamic redundancy control for random-intensive workloads.

**Fine-grained Adaptive Encoding** The parity calculation associated with EC requires a considerable amount of resource consumption on network, CPU, and disk access for overwrite operations and read operations under a server failure. Although the performance degradation of these operations to EC data is inevitable because of these resource consumptions, it is possible to mitigate the performance degradation by reducing the operations to the encoded data. The HDFS-based conventional methods avoid overwrite operations to EC data by limiting its encoding target to immutable data that are not updated anymore. However, this approach does not apply to random-intensive workloads because small-size random access with overwrite operations and access-skew is dominant in these workloads.

Therefore, I introduce a fine-grained adaptive encoding method to address the differences in the target workloads. The method switches the redundancy method of user data between replication and EC according to workload, as also performed by the conventional methods. I use a finer-grained chunk size and chunk-based access monitoring to identify cold data accurately. As the redundancy control layer is on top of the local filesystem, it can use the arbitrary chunk size regardless of the block size of the local filesystem. In addition, the access monitoring enables the method to encode more chunks with less access frequency if the frequency is sufficiently low. The method enables identifying a larger amount of cold data for encoding and achieves higher capacity efficiency by using fine-grained chunk size and access-frequency monitoring.

Furthermore, the method adaptively controls the amount of EC data according to the workload to meet the performance requirements by using the monitoring data. The method enables the maximized amount of EC data while maintaining the required level of performance.

## 4.3 Dynamic Redundancy Control with Adaptive Encoding (DRC-AE)

This section proposes a new redundancy control method named dynamic redundancy control with adaptive encoding (DRC-AE).

### 4.3.1 Overview

DRC-AE is a data redundant layer on top of the POSIX-based local filesystem [IG18]. DRC-AE changes the redundancy method used for user data between replication and EC according to dynamically changing workload. DRC-AE runs on each server and communicates with the other DRC-AEs running on other servers for redundancy control among multiple servers. An overview of DRC-AE is shown in Figure 4.2.

DRC-AE adopts local filesystem-based redundancy control (LFRC) for data redundancy. DRC-AE stores user data and file metadata in the local filesystem while adding redundant data to the user data. In a system using DRC-AE, a replication pair is formed with two or more servers. DRC-AE on each server replicates local user files to the other servers in the replication pair. DRC-AEs communicate with each other beyond the replication pairs for constructing erasure codes to reduce the amount of redundant data. When DRC-AE updates user data in the local filesystem, it also synchronously updates replica data or parity data through DRC-AE on the other servers. To add the capabilities of replication and EC to the standard POSIX-based filesystem, I use a technique called custom file metadata [FSN18]. I describe the details of LFRC in Section 4.3.3.

DRC-AE uses a new encoding control called adaptive encoding, which switches the redundancy method of user data between replication and EC. Adaptive encoding is a fine-grained encoding method in which the file is divided into fixed-sized chunks. Once a chunk is encoded and parity data are created, DRC-AE deallocates replica data of the chunk by using the *fallocate* interface [Cor10b] to reduce the amount of redundant data. The adaptive encoding preferably selects chunks with less access as the target of EC to minimize performance degradation because of EC. In addition, the adaptive encoding controls the amount of the encoded data to limit the performance degradation because of EC. The adaptive encoding allows users to set a trade-off between performance and capacity efficiency through setting encoding parameters. I describe the details of the adaptive encoding in Section 4.3.4.

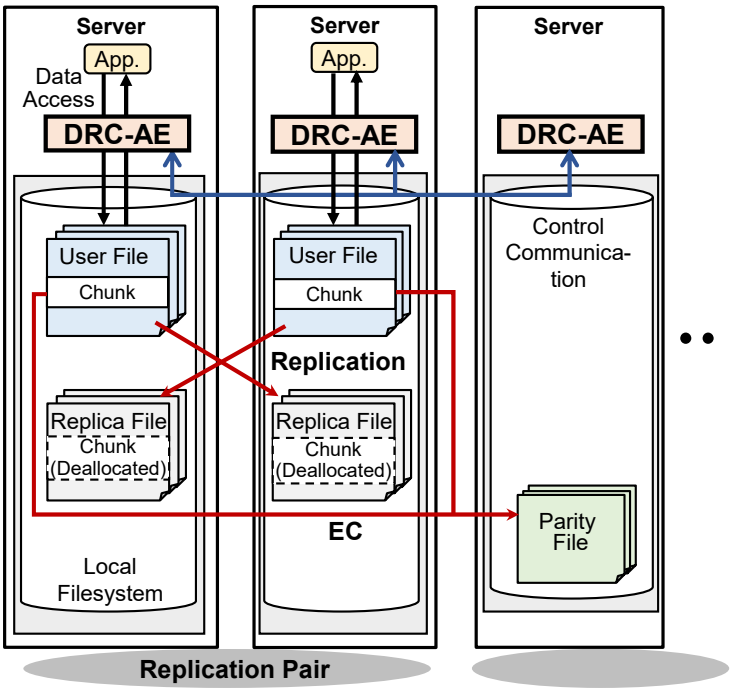



Figure 4.2: DRC-AE Overview

### 4.3.2 Architecture

DRC-AE is a stackable module on top of a local filesystem. One implementation of DRC-AE is a filesystem-in-the-user-space (FUSE)-based module with the Linux filesystem such as ext4 or XFS [lib21, XFS13, MCB+07]. 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. The use of FUSE makes DRC-AE more portable, in the same way *ceph-fuse* is more portable than the kernel client in CephFS [PBM+17]. An architectural diagram of DRC-AE is shown in Figure 4.3.

DRC-AE comprises two modules, namely, the DRC filesystem (FS) that processes file operations and the DRC Manager that performs encoding processing. DRC FS is in charge of LFRC, while DRC Manager conducts adaptive encoding. Both modules cooperate with other DRC-AEs running on other servers.

DRC FS comprises a request handler that handles user-data access from the FS interface and an FS agent that handles redundant-data access from other servers. In the FUSE-based implementation, the request handler is implemented as the FUSE daemon, whereas the FS interface is implemented as the FUSE kernel driver. When the request handler receives file access requests from the FS interface, it forwards the requests to the local filesystem. The

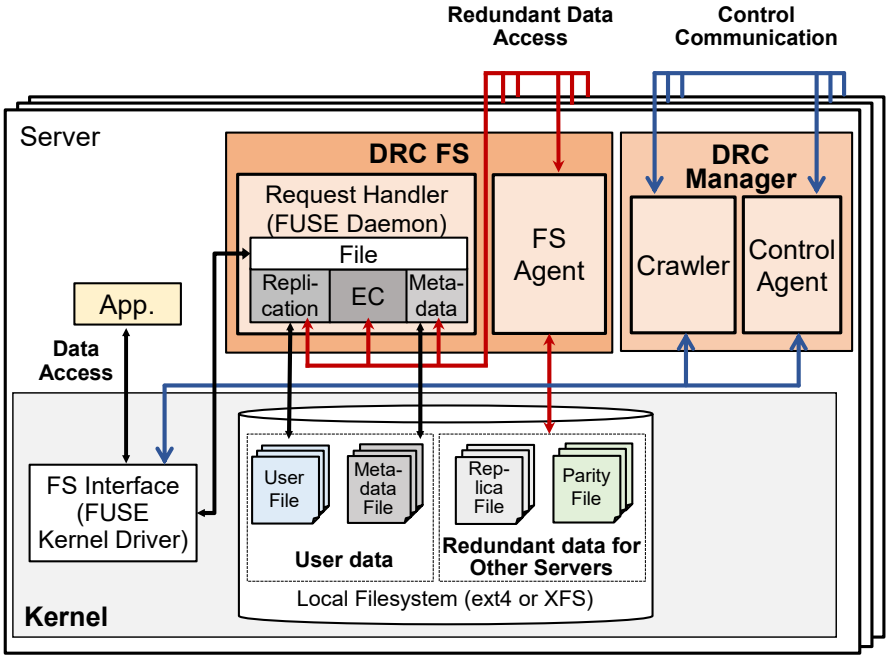



Figure 4.3: DRC-AE Architecture

request handler then updates the replica files or parity files on other servers by cooperating with the FS agents on these other servers. DRC FS uses the Reed–Solomon code for EC, but any EC methods with the linearity property can be used instead of the Reed–Solomon code [Pla13]. In addition, DRC FS records the EC configuration of all erasure-coded chunks in the custom file metadata, as described in the next section.

DRC Manager comprises a crawler that regularly collects monitoring data from DRC FS via the *ioctl* interface [BC05] and a control agent that processes control requests from other servers. The crawler conducts the adaptive encoding in cooperation with other servers. In addition, the crawler is used to rebuild data at the time of a server failure.

### 4.3.3 Local Filesystem-based Redundancy Control (LFRC)

DRC-AE adopts LFRC to avoid the performance overhead of distributing data among multiple servers for data redundancy. To implement data redundancy on local filesystems, DRC-AE adds the capability of replication and EC to the POSIX-based filesystem.

To enable LFRC, DRC-AE uses the original data layout, data-access processing, and rebuild processing, as described in the following sections.

**Data Layout**

DRC-AE stores user data and file metadata to files in underlying local filesystems. DRC-AE manages four types of files: a user file, a replica file, a parity file, and a metadata file. A user file is a normal file of the POSIX-based filesystem. An application can access a user file in the same way as a conventional local file. Each user file has a replica file that contains replica data of the user file in the replication pair. In addition, DRC-AE stores parity data and additional file metadata to a parity file and a metadata file, respectively. Figure 4.4 shows the data layout of DRC-AE.

To enable fine-grained control between replication and EC, DRC-AE logically splits user files into fixed-size chunks. When DRC-AE changes user data between replication and EC, it selects the same number of replicated chunks with data symbols and calculates the same number of parity data with parity symbols. The chunks are selected from different servers so that a server failure could involve only a single chunk in the same erasure code. After DRC-AE calculates parity data, it stores them in parity files in different servers and deallocates the corresponding replica data.

DRC-AE uses 512 KB for the chunk size. This chunk size limits the candidates of EC to files larger than 512 KB. It is well known that most of the total data capacity tends to be occupied by large files [WDQ+12, MB12]. Consequently, most of the capacity-saving effects occur even if the targeted files are limited to those files. By contrast, the number of small files accounts for a large percentage of the total number of files. If DRC-AE were to manage all of these small files, the overhead in performance and capacity consumption of file metadata would increase substantially. Inote also that several types of small files, such as temporary files or lock files, involve high-frequency access.

DRC-AE stores additional file metadata as the custom file metadata. The custom file metadata accommodate relationships between chunks and parity files in the same erasure code among multiple servers. From the perspective of data layout, the custom file metadata are stored either in the extended attributes [XFS13, MCB+07] of the local filesystems or in the metadata file, depending on the metadata size. Because the extended attributes have a size limitation, the metadata file is used for file metadata with no fixed-size limit.

There are two types of custom file metadata: chunk metadata and parity metadata. A user file and a replica file have the same chunk metadata that contains reference pointers to parity files for each encoded chunk. A parity file and its empty replica file have the parity metadata that contain reference pointers to chunks in the same erasure code. Because the size of the

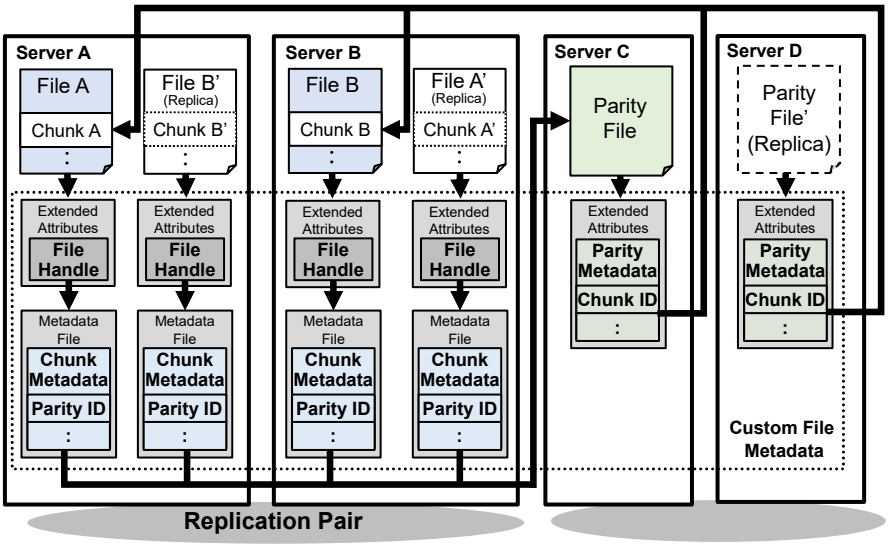



Figure 4.4: DRC-AE Data Layout

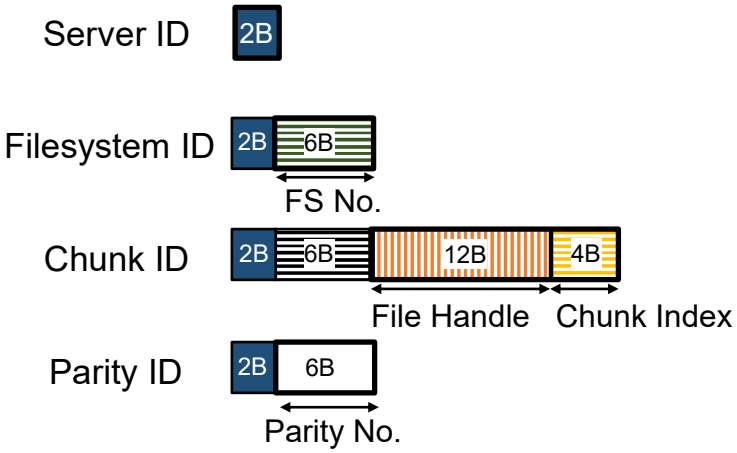



Figure 4.5: Global ID

chunk metadata is linear-proportional to the number of encoded chunks in a user file, DRC-AE uses the metadata file for the chunk metadata. By contrast, DRC-AE uses the extended attributes for the parity metadata because the number of reference pointers from a parity file is limited by the number of data symbols. Because these custom file metadata maintain the circular relationships among multiple servers, DRC-AE can identify all chunks and parity files in an erasure code from the custom file metadata.

To maintain the reference pointers among multiple servers, it is necessary to maintain system-wide identifiers. For that purpose, DRC-AE uses system-wide identifiers named the global IDs. Figure 4.5 shows the layout of the global IDs.

There are four IDs used in DRC-AE. The server ID is a 2-byte ID that uniquely identifies the server in the system. When adding a server to the system, DRC-AE on the new server communicates with existing servers and assigns a unique server ID to the new server. The filesystem ID is an 8-byte ID combining the server ID and a unique 6-byte internal identifier for a server-named filesystem number (the filesystem No.). The chunk ID is a 24-byte ID combining the filesystem ID, file handle for the file containing the chunk, and index of the chunk within the file. The index can be dynamically calculated from the offset of the chunk from the start of the file. DRC-AE uses the file handle to open the file for access to the chunk data [Cor10a]. The parity ID is another 8-byte ID combining the server ID and a 6-byte parity number (Parity No.), which is an internal identifier for a server. The parity ID contains the name of the parity file and is used for access to the parity file via the parity ID.

DRC-AE shares the mapping between the server IDs and the hostnames of all servers. It uses the mapping to access a server that contains the target object. When the FS agent or the control agent in the target server receives access from other servers, it interprets the global ID and identifies the file handle or name of the parity file. The agents then use the file handle or name of the parity file to access the target object.

Note that the ID space of the global IDs excluding the server ID is an exclusive ID space starting with the server ID. By using a prefixed server ID, it is possible to generate a global ID without communicating with other servers, reducing the complexity of the ID generation and in-network communications between DRC-AEs.

To enable data rebuild under a server failure, the custom file metadata of user files are also replicated to the replication pair. Because the parity data can be recalculated from other chunks, only the custom file metadata of a parity file is replicated to the replication pair without copying the parity data itself. I investigate these rebuild processes in Section 4.3.3.

DRC-AE requires storing and reading the custom file metadata; therefore, there are capacity and performance overheads of the metadata management. I evaluate the overheads of the metadata management in Section 4.4.4.

#### Data-access Processing

DRC-AE offers data-access interfaces for locally stored user files to applications. Although read access to a user file only involves local file read operation, write access requires updating replica data or parity data in different servers in addition to updating the local user file.

Regarding replicated data, updating replica files is straightforward. When DRC-AE receives a file update request, it forwards the request to DRC-AE on the replication pair while

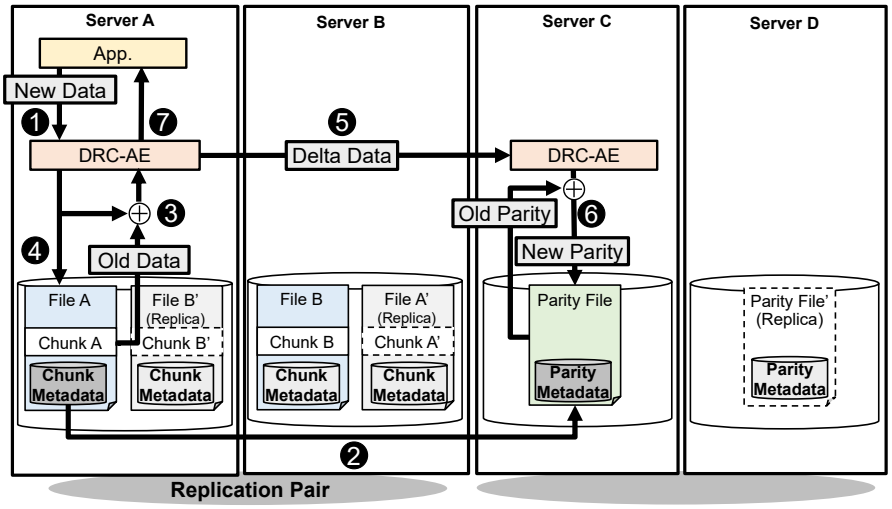



Figure 4.6: Overwrite Operation to EC Data

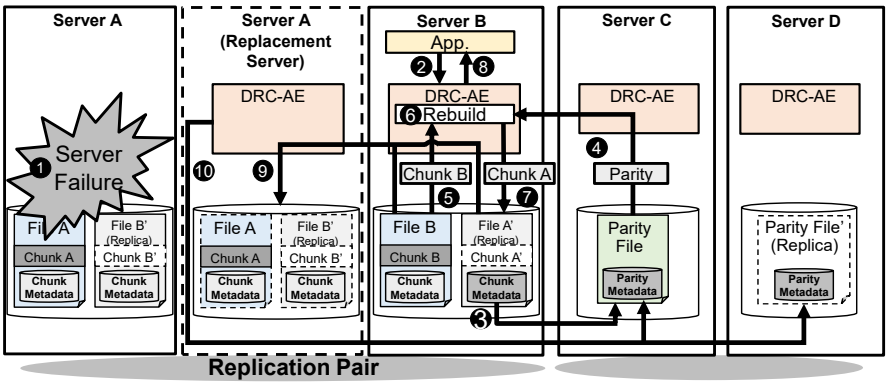
updating the user file in the local filesystem. DRC-AE on the replication pair updates the replica file when it receives the forwarded request. DRC-AE executes these update operations to the user file and replica files in parallel to reduce access latency.

By contrast, updating parity data needs to access the parity servers that store parity data of the target user file. To enable updating these redundant data, DRC-AE uses the chunk metadata of the user file to identify the locations of its parity files.

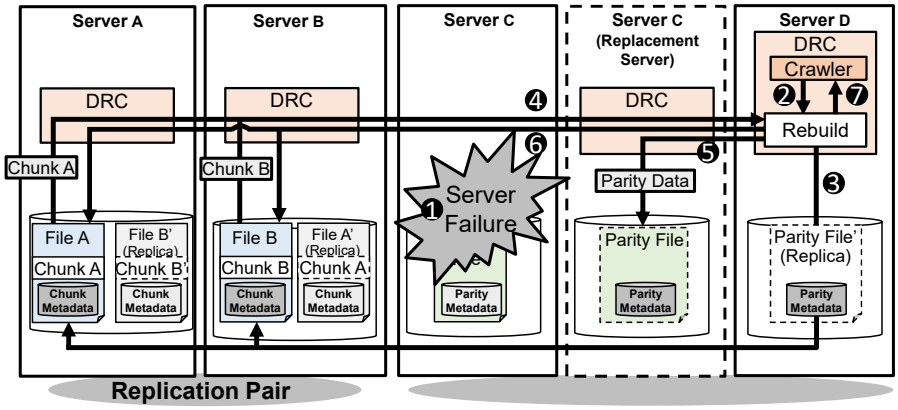
I now describe how DRC-AE uses the custom file metadata in overwrite operations in terms of the scenario shown in Figure 4.6. In Figure 4.6, Servers A and B form a replication pair, while Servers C and D form another replication pair. Servers A and B store File A, File B, and their corresponding replica files (File A' and File B', respectively). Chunk A in File A and Chunk B in File B form an erasure code, with the parity file being stored in Server C. The parity metadata of the parity file is replicated to Server D.

When DRC-AE in Server A receives a write request, it uses the chunk metadata of File A to identify the parity file in Server C. DRC-AE then communicates to Server C to obtain the lock of the parity file (Steps 1 and 2). DRC-AE then calculates the delta data between the new and old data of Chunk A (Step 3). Then, DRC-AE writes new data to Chunk A while sending the delta data to Server C (Steps 4 and 5).

DRC-AE in Server C then calculates the new parity from the given delta data and the old parity, updates the parity file, and responds to Server A (Step 6). Finally, DRC-AE in Server A releases the lock of the parity file and replies to the application (Step 7).




(a) EC Data Rebuild




(b) Parity Rebuild

Figure 4.7: Rebuild Processing

### Rebuild Processing

When a server failure occurs, DRC-AE recovers lost data and data redundancy from other remaining chunks and parity data. As for replicated data, the lost data can be rebuilt from the replica files by copying the replica data to a replacement server. By contrast, EC data require rebuilding the lost chunks from remaining chunks and parity data that form the same erasure code. In addition, lost parity data also need to be rebuilt to recover the redundancy of user data.

Similar to the EC overwrite operation, the custom file metadata enables DRC-AE to recover the lost chunk and the parity file. Once the lost chunk and the parity file are rebuilt, DRC-AE can recover the data redundancy in the same manner as the replicated data.

I now describe how DRC-AE uses the custom file metadata for data rebuild and parity rebuild.

**EC Data Rebuild** In the event of a server failure, an application moves to the replication pair and continues its service using the replica files. When DRC-AE receives the first read operation or write operation to a lost EC chunk in the replica files, it rebuilds the lost EC chunk from other remaining chunks and parity data. In addition, DRC-AE on the replication pair wakes up the crawler to issue read requests to the lost EC chunks to start the data rebuild so that the lost data do not remain unrecovered.

DRC-AE uses the chunk metadata of the replica file and the parity metadata of the parity file to rebuild the lost EC chunk. DRC-AE obtains the global IDs of the remaining chunks and parity files from these custom file metadata. Then, DRC-AE rebuilds the lost EC chunk from these remaining data. In the event of a server failure, data redundancy is temporarily reduced until the failed server recovers. DRC-AE restores original redundancy by rebuilding data to the replacement server or the recovered server from the replication pair after the server recovery. In addition, with three-way replication or EC with two or more parity, DRC-AE updates the rest of replica data or parity data during update operations under a server failure.

Figure 4.7a shows the flow of data rebuild in the same configuration with Figure 4.6. Under a failure of Server A, when DRC-AE in Server B receives a read or write request to Chunk A' from an application, it obtains the parity ID in the chunk metadata of File A' (Steps 1 to 3). Then, DRC-AE in Server B obtains the parity data and chunk IDs from the parity file in Server C (Step 4). DRC-AE in Server B then reads Chunk B from the local filesystem (Step 5) and rebuilds Chunk A from the parity data and Chunk B (Steps 6). Then, DRC-AE in Server B writes the rebuilt data to Chunk A' in File A' so that it can avoid data rebuild in the subsequent access to Chunk A' (Steps 7). Then, DRC-AE in Server B replies to the application (Step 8). After Server A is replaced, DRC-AE in the replaced Server A rebuilds File A and the replica of File B (i.e., File B') from Server B (Step 9). DRC-AE excludes the erasure-coded Chunk B from rebuilding so that the replica of Chunk B (i.e., Chunk B') does not increase capacity consumption. Then, DRC-AE deallocates Chunk A', which temporarily stores rebuilt data in Server B. Note that if the Chunk A' is not accessed by Step 9, the Chunk A' is not rebuilt in Server B. In that case, DRC-AE in Sever B rebuilds Chunk A and sends the rebuilt data to the replaced Server A without writing the rebuilt data to Chunk A'. Finally, DRC-AE in the replaced Server A instructs Server C and Server D to update the chunk ID in the parity metadata to the rebuilt Chunk A (Step 10).

**Parity Rebuild** Similar to data rebuild, parity files also need to be rebuilt to recover the data redundancy of user data. Because an application does not directly access the parity files,

DRC-AE uses the crawler to start the parity rebuild.

DRC-AE uses the replicated parity metadata of the lost parity file to recalculate the parity data. DRC-AE uses the chunk IDs in the replicated parity metadata to access the remaining chunks. Then, DRC-AE rebuilds the parity data from these chunks.

Figure 4.7b shows the flow of the parity rebuild. When Server C is replaced after a server failure, the crawler on Server D, which is the replication pair of Server C, starts the parity rebuild (Steps 1 and 2). Then, DRC-AE in Server D uses the replicated parity metadata of the lost parity file to identify the chunks of the same erasure code (Step 3). Then, DRC-AE in Server D obtains Chunk A and Chunk B from Server A and Server B, respectively (Step 4), and rebuilds the parity data in the replaced Server C (Step 5). Then, DRC-AE in Server D updates the parity ID of the chunk metadata in File A and File B to the rebuilt parity file (Step 6). Finally, DRC-AE in Server D replies to the crawler (Step 7).

### 4.3.4 Adaptive Encoding

The adaptive encoding is a chunk-based redundancy control that changes the redundancy method of a chunk according to monitoring data. DRC-AE adopts the adapting encoding to achieve less performance decrease for the same amount of EC data while controlling the amount of EC data to avoid unacceptable performance degradation.

The core logic of the adaptive encoding consists of two components: the EC priority calculation and the EC rate control. The purpose of the EC priority calculation is to minimize performance degradation because of EC for the same amount of encoded data. By contrast, EC rate control limits the amount of the encoded data so that the ratio of EC data accesses is lower than a given threshold. I describe the details of these components in the following sections.

#### EC Priority Calculation

DRC-AE calculates the EC priority of each chunk to determine which chunk is prioritized in the encoding process. The EC priority indicates the smallness of the performance impact of encoding the chunk. DRC-AE selects chunks with higher EC priority in the encoding process to minimize the performance impact.

DRC-AE uses the monitoring data of chunks for the EC priority calculation. DRC-AE monitors the number of write and read operations on chunks for a certain period. I describe how a user sets the monitor interval in Section 4.3.4.

DRC-AE uses the monitoring data to calculate the load and the priority of chunks, as shown

in Equations (4.1) and (4.2).

$$load(x) = IO_{write}(x) + \alpha IO_{read}(x) \qquad (x \in \mathbb{C}) \tag{4.1}$$

$$priority(x) = \frac{1}{1 + load(x)} \qquad (x \in \mathbb{C}) \tag{4.2}$$

Here, the value of $load(x)$ and $priority(x)$ indicate the load and the priority of a given chunk $x$ in a set of chunks $\mathbb{C}$ that represents all chunks in a system. $IO_{write}(x)$ and $IO_{read}(x)$ refer to the number of write operations and read operations of chunk $x$, respectively. The $\alpha$ indicates the read weight, which is a tunable parameter to set a weight on read operations in the EC priority calculation.

Because access performance decrease in the normal condition occurs only in write operations, DRC-AE uses $\alpha$ as zero by default. This setting makes DRC-AE ignore read operations in the EC priority calculation so that it achieves optimal performance and capacity efficiency in the normal condition. However, under server failure, read performance degradation also occurs because of the data rebuild processing. DRC-AE allows a user to set $\alpha$ depending on the performance requirement for a failure state, as I describe in Section 4.3.4.

By using Equations (4.1) and (4.2), DRC-AE prioritizes chunks with less load as its encoding target. This behavior enables stable performance as long as chunks with lower load remain unencoded. If a filesystem contains a large amount of cold data, DRC-AE preferably encodes the cold data without any significant decrease in performance.

### EC Rate Control

DRC-AE uses the EC rate control that limits the amount of encoded data so that the access ratio to the encoded chunks is also limited. The EC rate control uses Equation (4.3) to limit the amount of encoded data.

$$\frac{\sum_{x \in \mathbb{E}} load(x)}{\sum_{x \in \mathbb{C}} load(x)} < L \tag{4.3}$$

Here, $\mathbb{E}$ means the set of encoded chunks and $L$ means the threshold of the ratio of a total load of encoded chunks against all chunks. Hereafter, I call $L$ the load limit.

If a user sets the load limit $L$ as less than 1, DRC-AE stops the encoding process of remaining chunks once the left-hand side of Equation (4.3) reaches the load limit $L$. Otherwise, DRC-AE keeps encoding until the capacity ratio of encoded chunks reaches the EC target rate, $R$, which is also given by a user. In this way, a user can limit both performance degradation and

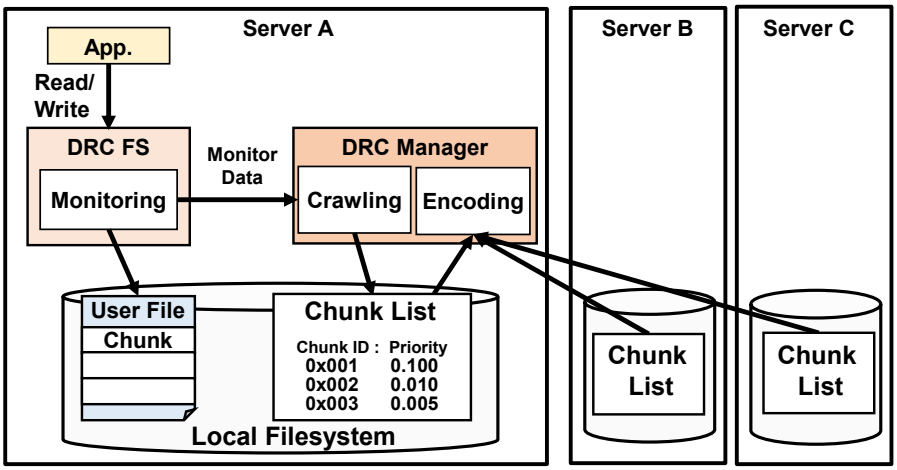



Figure 4.8: Encoding Processing

capacity by setting $L$ and $R$. The expected capacity gain for a given $R$ can be calculated from Equation (4.4).

$$CapacityEfficiency = \frac{1}{1 + m \times (1 - R \times \frac{k-1}{k})} \tag{4.4}$$

Here, $k$ means the number of data symbols, and $m$ means the number of parity symbols.

### Encoding Processing

The adaptive encoding comprises three processes: monitoring, crawling, and encoding. Figure 4.8 shows the encoding processing. I describe each process in the following paragraphs.

**Monitoring** DRC-AE monitors the number of read and write operations per chunk while processing read and write operations in DRC FS. DRC FS stores 16-byte monitoring data in memory for each chunk. It offers the monitoring data to other modules through a dedicated *ioctl* interface [BC05].

The monitoring function requires memory resources for the monitoring data. The total memory usage may reach 32 MB for a used capacity of 1 TB. I consider this memory requirement as sufficiently small because of the ongoing increase in the memory capacity of commodity servers. As for the performance, I evaluate the performance impact of the monitoring in Section 4.4.4

**Crawling** DRC-AE periodically wakes the crawler up and evaluates the load and priority of chunks in the local filesystem. The crawler runs on each server independently. It records chunk

IDs and the results of its evaluation in a list called the chunk list. The crawler uses Equations (4.1) and (4.2) to evaluate the load and priority of a chunk.

DRC-AE stores the chunk list to the local disk as a temporary file. DRC-AE records the 24-byte chunk ID and the 8-byte priority value for each chunk; therefore, 32 bytes are required for a chunk. The size of the chunk list may reach 64 MB for a used capacity of 1 TB. I consider this disk requirement as sufficiently small for the used capacity.

**Encoding** DRC-AE selects a server as the coordinator, coordinating the encoding among all servers. DRC-AE uses a leader selection implementation such as Apache ZooKeeper [Apa21] to select the coordinator. When the coordinator finishes its crawling operation, its DRC-AE collects the chunk lists from other servers and starts the encoding process.

The coordinator selects the chunks with the least EC priorities from the different servers and selects other servers to store the parity files. After having made these selections, the coordinator sends a command to one of the parity file owners to perform the encoding. The server that receives the instruction then conducts the encoding. After the encoding, the server deallocates the replica data of the encoded chunks by communicating with other servers. The coordinator repeats the encoding process as long as the capacity ratio of EC data is less than the EC target rate, $R$, and Equation 4.3 is satisfied.

When the coordinator finds a replicated chunk that has a higher priority than an EC chunk, it sends a command to the owner node of the EC chunk to swap these chunks. DRC-AE on the owner node recreates replica data of the low-priority EC chunk on the replication pair. Then, the DRC-AE updates the parity data and the custom file metadata of the parity files and the files with the chunks to replace the low-priority EC chunk with the high priority replicated chunk.

DRC-AE switches back low-priority EC chunks to replicated chunks in the same way when the total load of the EC chunks becomes larger than the limit of the EC rate control.

### Parameter Setting

DRC-AE provides a user with tunable parameters to adjust the redundancy control to satisfy the capacity and performance requirements. There are four parameters, the target EC rate, $R$, the load limit, $L$, the read weight, $\alpha$, and the monitoring interval.

A user decides the value of $L$ and $R$ based on the performance and capacity requirements. If there is enough free space, a user should set $R$ as 0 so that no data are encoded and no performance decrease occurs. If there is less free capacity, a user should enable the encoding process by setting $R$. If performance degradation is not allowed, a user should set $L$ as a small

value and set $R$ to 1. In this case, $L$ becomes a tighter limit than $R$; therefore, the encoding process stops when the load ratio of encoded chunks reaches $L$. As a result, performance degradation is limited to a small degree, whereas the effect of capacity saving is also limited. By contrast, if the capacity is a higher priority than performance, a user should set $L$ as 1 and decide the value of $R$ according to the required capacity.

If performance degradation in a failure case is not allowable, a user should set $\alpha$ as 1 to avoid the performance degradation under a server failure at the cost of performance and capacity efficiency in the normal condition. I evaluate the trade-off between performance and capacity efficiency in terms of the value of $\alpha$ in Section 4.4.3. A user can reduce the switching cost between replication and EC by setting a sufficiently long monitoring interval such as one day or one week. Switching the redundancy method of a chunk between replication and EC requires a considerable amount of data transfer and processing cost for the parity calculation. Therefore, the monitoring interval should be sufficiently long to suppress the switching cost because of temporary changes in the access pattern. On the other hand, with the longer monitoring interval, replicated chunks remain for a longer period after the data become cold data. As a result, a user needs to wait for the monitoring interval to increase free capacity by encoding the cold data. In addition, the longer monitoring interval also makes EC chunks remain longer even when their access frequency increases. Then, access performance to the encoded data remains low for the monitoring interval. If a user needs a faster response for these cases, the monitoring interval should be set to a shorter period.

## 4.4 Evaluation

To investigate the effectiveness of the proposed method, I developed and evaluated a FUSE-based prototype version of DRC-AE. I developed a C++ implementation for DRC FS and a Python implementation for the DRC Manager. The total size of these implementations was about 14,000 lines of code. I used *jerasure* [PSS07] for parity calculation and data rebuild.

In Section 4.4.2, I evaluated the access performance of DRC-AE in comparison with CephFS, which is one of the most widely used POSIX-based scale-out distributed storage. I evaluated the access performance under data-intensive workloads (Section 4.4.2) and the access performance under the metadata-intensive workloads (Section 4.4.2). In Section 4.4.3, I investigated how the adaptive encoding affects performance against capacity efficiency. I used synthetic workloads (Section 4.4.3) and real-world workloads (Section 4.4.3) for these evaluations. I also evaluated the adaptive encoding in comparison with HDFS-based conventional methods

Table 4.1: Experimental Environment

| Items | Settings |
|---|---|
| Machine | SYSTEM WORKS POWER MASTER Server S5204 x 8 |
| CPU / Server | Intel Xeon Silver 4215 CPU 2.50 GHz (8 cores) x 2 |
| Memory / Server | 192 GB (Available memory: 24 GB) |
| Disk / Server | SSD disk configuration : SATA SSD WDS500G2B0A x 2,<br>NVMe SSD configuration : Intel Optane SSD 900p x 2 |
| Network Interface Card / Server | Mellanox MCX516A-CDT (10GbE/100GbE) |
| OS | Ubuntu 18.04 LTS (Linux kernel 4.15) |
| Storage software | DRC-AE prototype with XFS<br>Ceph version 15.2.4-1 Octopus |
| Benchmark tool | filebench1.5-alpha3 |

for real-world workloads. In Section 4.4.4, I investigated the overhead of the metadata management in DRC. In Section 4.4.5, I investigated the rebuild performance of DRC-AE in comparison with CephFS.

### 4.4.1 Experimental Environment

Table 4.1 describes the experimental environment. For the experiments, I used eight commodity servers. I ran *filebench* [TZS16] as a benchmark tool on one of the servers. For storage software, I investigated the DRC-AE prototype using two-way replication, three-way replication, and EC (6D2P). I used three-way replication and EC (6D2P) in the CephFS evaluation. I excluded two-way replication from CephFS configurations because CephFS, which distributes data across all servers, needs higher failure tolerance than DRC-AE with two-way replication, which consists of only two servers.

I investigated both an SSD configuration and NVMe SSD disk configurations using Intel 3D Xpoint memory technology [cor21a]. To match a typical server configuration, I limited the bandwidth of NIC from 100 Gbps to 10 Gbps except for experiments with the large network bandwidth. I make notes about the use of the 100GbE NIC configuration when I use the configuration. To avoid the cache effect in the experiments, I limited the available memory size to 24 GB by allocating 168 GB out of 192 GB memory to a dummy program. The dataset

size was double the available memory size. Each measurement was run three times, and the average performance was evaluated. For the CephFS client, I investigated Ceph kernel client rather than *ceph-fuse* as Ceph kernel client showed higher performance than *ceph-fuse*.

### 4.4.2 Access Performance

#### Data-intensive Workloads

I investigated the performance of DRC-AE for the data-intensive workloads in comparison with CephFS. I used the sequential read and write, the random read and write, and the online-transaction-processing (OLTP) workloads in *filebench*. For the sequential and random workloads, five threads accessed five large files separately with an IO size of 1 MB for the sequential workloads and 8 KB for the random workloads. To reduce the impact of the cache effect, I used file open with the O_DIRECT flag for the random workloads. The OLTP workload simulated a real-world OLTP with a single log-write thread, 10 concurrent DB-write threads, and 200 concurrent DB-read threads. The average IO size of the OLTP workload was 2 KB, and the number of DB files was set as 200. For the OLTP workload, I also investigated CPU requirements because the CPU tends to be the performance bottleneck for OLTP workloads. I also evaluated the performance of the OLTP workload in the event of a server failure.

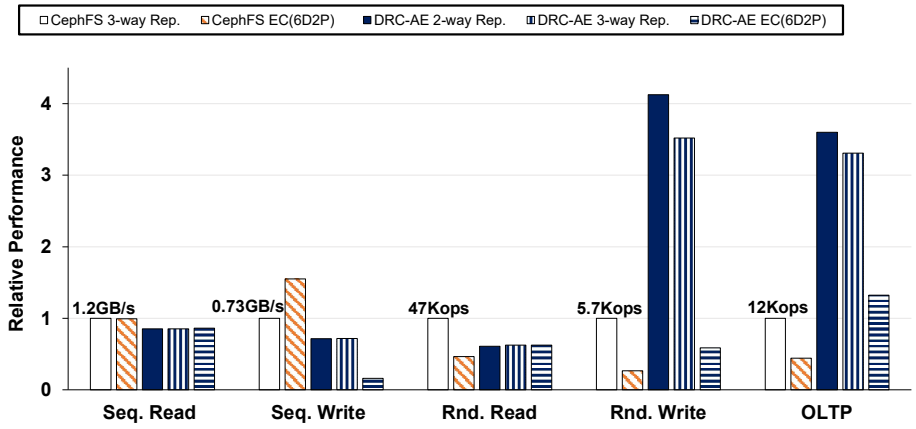
Figure 4.9 shows the performance comparison between DRC-AE and CephFS. For DRC-AE, I used configurations for which all chunks were two-way or three-way replicated and a configuration for which all chunks were converted into (6D+2P) EC. The figures show the relative performances of DRC-AE ("DRC-AE 2-way Rep.", "DRC-AE 3-way Rep.", and "DRC-AE EC (6D2P)") and CephFS with EC ("CephFS EC (6D2P)") against CephFS with three-way replication ("CephFS 3-way Rep.") under each workload. Note that the values above "CephFS 3-way Rep." for each workload give the actual performance of CephFS with three-way replication. To determine the effects of the different disk and network configurations, I conducted the same evaluations in the SSD and 10GbE NIC configuration (Fig. 4.9a), the NVMe SSD and 10GbE NIC configuration (Fig. 4.9b), and the NVMe SSD and 100GbE NIC configuration (Fig. 4.9c). In addition, Figure 4.10 shows the sum of CPU usage among eight servers during a test run under the OLTP workload in the NVMe SSD and 10GbE configuration. The CPU usage is shown as usage per thousand operations to normalize performance differences.

For the sequential read and random read workloads, we can see different tendencies depending on disk configurations. In the low-performance SSD configuration, DRC-AE shows 15% less sequential read performance and 39% less random read performance with three-way

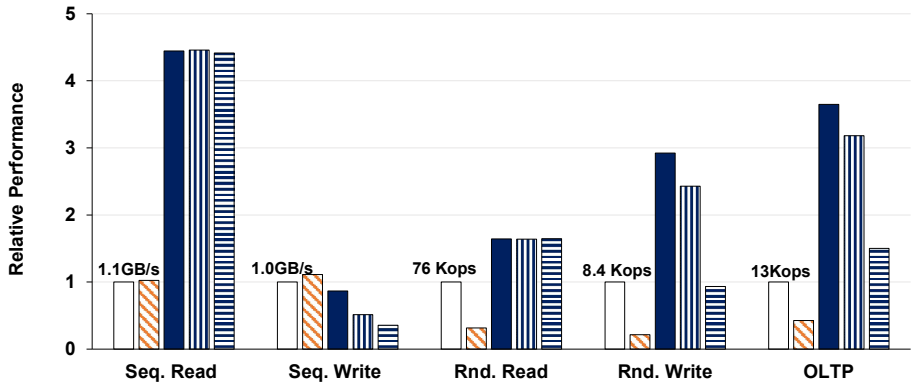
replication. In the NVMe SSD and 10GbE NIC configuration, DRC-AE shows 346% higher sequential read performance and 63% higher random read performance with the same three-way replication. In the NVMe SSD and 100GbE NIC configuration, the difference in the sequential read is reduced to 70% because the larger bandwidth of 100GbE NIC solved the network bandwidth bottleneck of CephFS. The important point here is that DRC-AE succeeded in fully utilizing the local disk performance under these workloads, whereas CephFS could not take full advantage of NVMe SSDs because of larger network processing and the network bandwidth limit. In addition, DRC-AE with EC achieves almost the same performance with replication, while CephFS with EC shows lower performance than replication. For these workloads, DRC-AE reads data from local disks regardless of whether the data are replicated or EC. By contrast, CephFS collects data from other servers when it reads EC data [KZK+17]. It is thought that this additional data transfer in CephFS decreases the EC read performance of CephFS in comparison with replication.

For the sequential write workload, CephFS shows better performance than DRC-AE with the same redundancy level in all hardware configurations. The largest difference can be seen in the SSD and 10GbE NIC configuration where DRC-AE with EC shows 90% less performance than CephFS with EC. Under the sequential write workload, the server where the benchmark tool runs became the performance bottleneck of DRC-AE. Disks were the performance bottleneck for the SSD configuration, while 10GbE NIC was the performance bottleneck in the NVMe SSD and 10GbE NIC configuration. In the NVMe SSD and 100GbE configuration, it is assumed that DRC-AE hit its performance limit of the current prototype implementation. Furthermore, DRC-AE shows lower performance with EC than replication, while CephFS shows higher performance with EC. Because DRC-AE encodes local data across multiple servers, sequential write to the local data causes partial overwrites to EC data while CephFS processes sequential write as a full-stripe-write over multiple servers. Partial overwrites to EC data entail larger network traffic and processing overhead than full-stripe-write [Pla13]. DRC-AE shows about half the sequential write performance of two-way replication with three-way replication in the NVMe SSDs and 10GbE NIC because of the limit of the 10GbE NIC bandwidth.

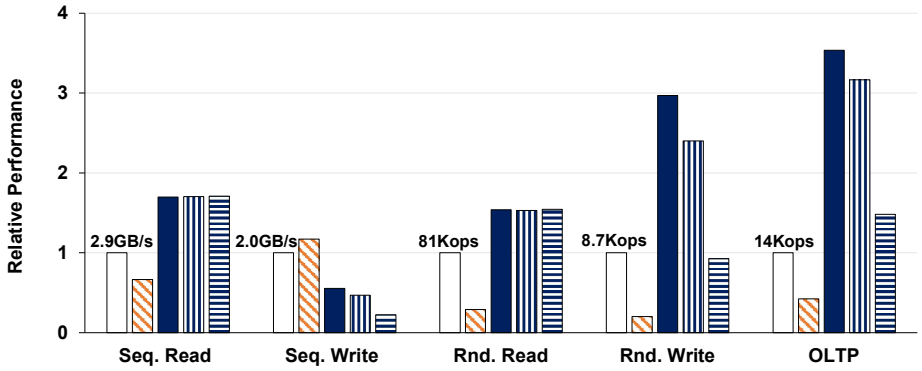
For the random write and OLTP workloads, DRC-AE shows superior performance than those of CephFS with the same redundancy level in all hardware configurations. DRC-AE shows up to 251% better random write performance and up to 230% better OLTP performance than CephFS with three-way replication. Because the random write performance did not reach the performance limit of SSD and NVMe SSD in these experiments, it is thought that less network access latency of DRC-AE is the reason for the higher random write performance of




(a) SSD and 10GbE NIC Configuration




(b) NVMe SSD and 10GbE NIC Configuration




(c) NVMe SSD and 100GbE NIC Configuration

Figure 4.9: Access Performance under Data-intensive Workload

DRC-AE. As for OLTP performance, Figure 4.10 shows a more substantial CPU requirement for CephFS than for DRC-AE. The less-onerous CPU requirement of DRC-AE contributed to its better OLTP performance while it also allows applications to use much more CPU resources

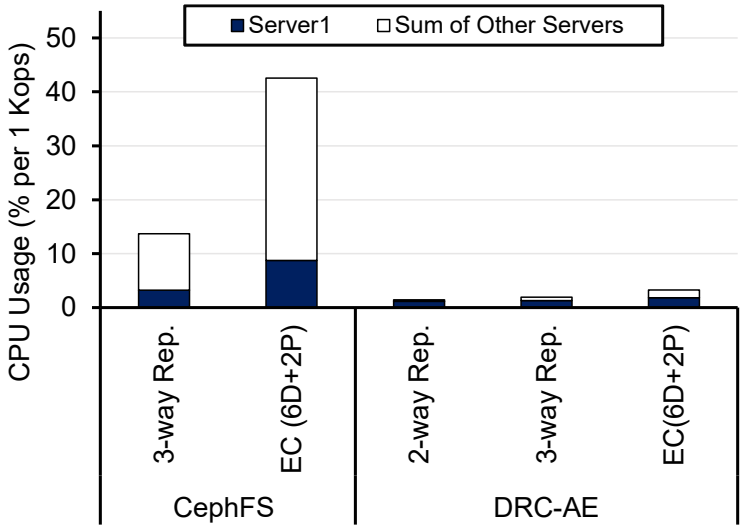



Figure 4.10: CPU Usage under OLTP Workload

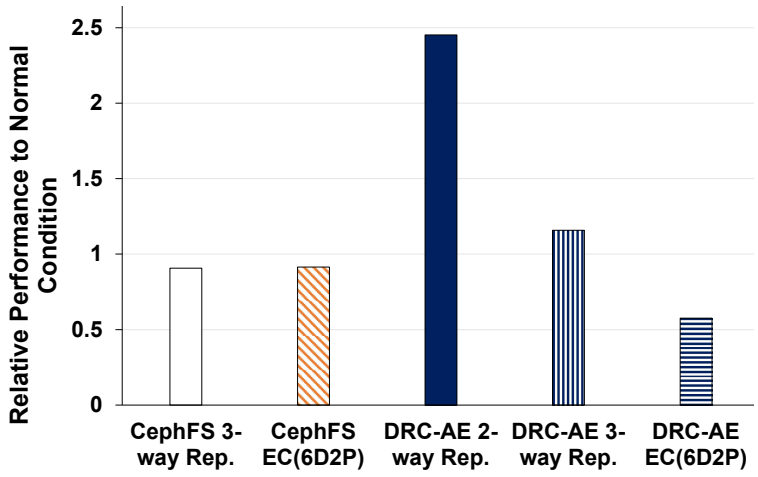



Figure 4.11: OLTP Performance under Server Failure

and achive higher application performance.

Figure 4.12 summarises the results of the performance comaprison between DRC-AE and CephFS. The quadrants in Figure 4.12 show whether DRC-AE or CephFS is advantageous for each combination of the network and disk configurations under each workload. These results show that DRC-AE has a performance advantage over CephFS except for the sequential write workload in a system using high-performance disks. Especially the much higher performance of DRC-AE for the OLTP workload indicates the obvious advantage of DRC-AE against CephFS under random-intensive workloads.

Figure 4.11 shows the OLTP performance of DRC-AE and CephFS under a server failure.

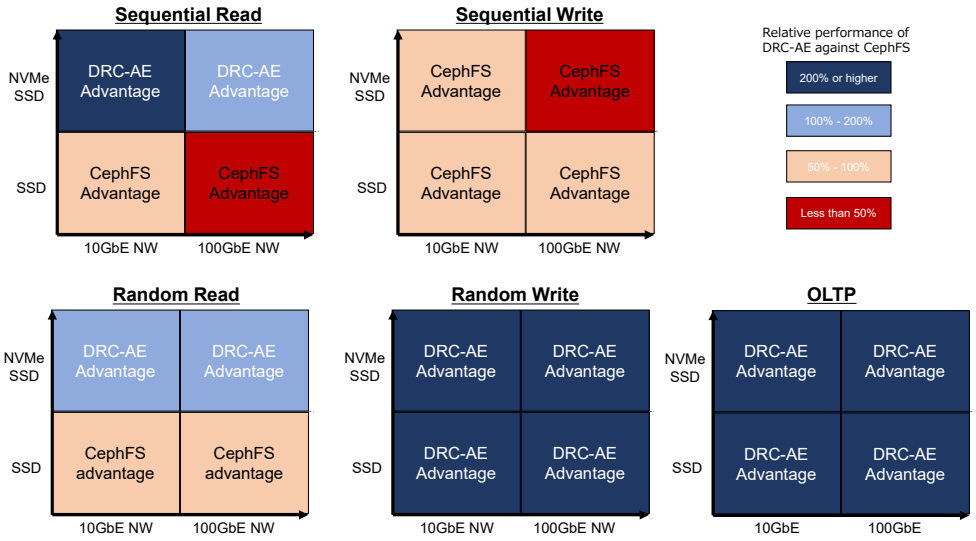



Figure 4.12: Performance Characteristics Comparison between DRC-AE and CephFS

The results show the relative performance of the failure condition to the normal condition for each configuration. In these evaluations, I ran the benchmark tool on the replication pair after I stopped the server where the benchmark tool worked. These results indicate that how much a failure affects application performance for each configuration.

There are different performance tendencies between DRC-AE and CephFS. While DRC-AE shows no performance degradation with replication and substantial performance degradation with EC, CephFS shows small performance degradations for both replication and EC.

DRC-AE with two-way replication and three-way replication shows higher performance under a server failure than the normal condition. Although this higher performance of DRC-AE with replication is simply because of reduced redundancy, the results indicate that a server failure does not decrease the access performance of replicated data. On the other hand, DRC-AE with EC shows 42% performance decrease because of EC data rebuild in comparison with its high performance in the normal condition. CephFS shows less than 10% performance decrease for both replication and EC. Because CephFS distributes data across all servers, the impact of a single server failure is limited. Also, it is assumed that the aforementioned low EC read performance of CephFS in the normal condition contributes to the less performance degradation of CephFS with EC under a server failure.

The substantial performance decrease of DRC-AE with EC indicates that excessive access to EC data under a server failure could degrade the access performance of DRC-AE to an unacceptable level. These results imply an essential role for the EC rate control to mitigate

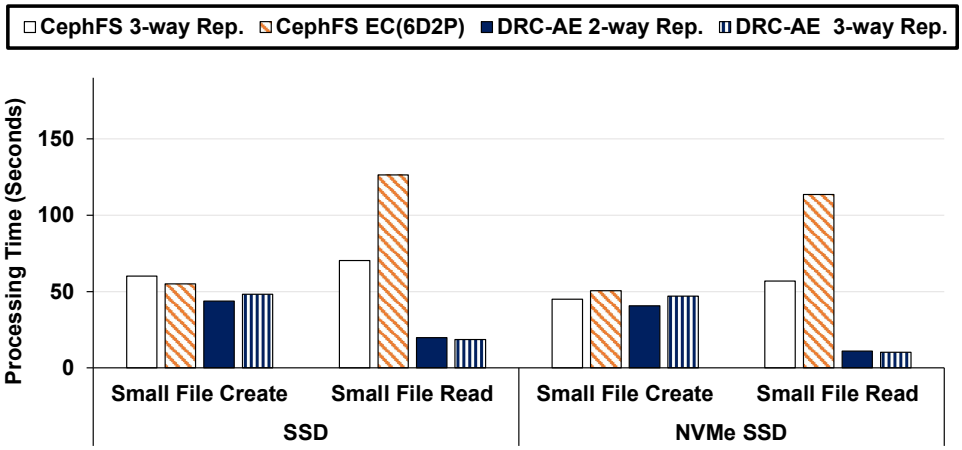



Figure 4.13: Access Performance under Metadata-intensive Workload

performance degradation when failure performance has a high priority.

### Metadata-intensive Workloads

I also investigated the access performance of DRC-AE for the metadata-intensive workloads.

For the metadata-intensive workload,I measured the small-file creation performance and the small-file read performance. I used the decompression time of compressed Linux[1] 5.7.9 source code [ker21] for the small-file creation. The Linux-kernel source code contains about 68,000 files with an average size of 16 KB, with decompression, representing the small-file creation workload. For the small-file read workload, I used the *grep* command to search for a word in all source files. Because DRC-AE does not encode small files, I excluded the performance of DRC-AE using EC from the experiment targets. I evaluated the performance in the SSD configuration and the NVMe configuration with 10GbE NIC.

Figure 4.13 shows the experimental results. Here, DRC-AE shows better performance than CephFS in both the SSD configuration and the NVMe SSD configuration. The largest superiority of DRC-AE is in the small-file read workload for the NVMe SSD configuration, where DRC-AE shows 82% less processing time than that of CephFS.

These results show that DRC-AE outperformed CephFS for the small-file read performance. Because DRC-AE stores file data and metadata in a local filesystem, its small-file read performance was better than that for CephFS.

[1]Linux is a registered trademark or trademark of Linus Torvalds in the United States and other countries. Other system names and product names used in this thesis are registered trademarks or trademarks of the associated company.

However, the small-file creation performance of DRC-AE is almost the same as CephFS. When DRC-AE creates a file, DRC-AE communicates with the other servers in the replication pair to create the replica files. Therefore, the performance difference between DRC-AE and CephFS is theoretically smaller than that for the small-file read. Moreover, the DRC-AE prototype uses FUSE, which is reported to perform worse for metadata-intensive workloads than data-intensive workloads [VTZ17]. I, therefore, think that a kernel-client-based approach similar to that of the Linux-kernel client of CephFS could improve the small-file creation performance of DRC-AE.

### 4.4.3 Performance against Capacity Efficiency

#### Synthetic Workload

To evaluate the effectiveness of the adaptive encoding,I measured the access performance and capacity efficiency of DRC-AE with various settings of the adaptive encoding. For these experiments, I changed the EC target rates, $R$, load limit, $L$, and read weight, $\alpha$, for each measurement. I used DRC-AE with 3-way replication and EC (6D2P) for these experiments.

I used the OLTP workload in the previous section with different sizes of working sets and operation sets. I used two sizes of working sets, the large working set and the small working set, to determine the effect of access locality of workloads. The ratio of accessed data to the entire dataset is one-third in the small working set, whereas the entire dataset is accessed in the large working set. In addition, to determine the effect of the operation mixes, I evaluated three different operation sets: the mixed operation set, which is the same operation set as the original OLTP workload, the read-only operation set, and the write-only operation set. I used the large working set with the mixed operation set (mixed large workload), small working set with the mixed operation set (mixed small workload), large working set with the read-only operation set (read-only large workload), and large working set with the write-only operation set (write-only large workload).

For parameters of the adaptive encoding, I used target EC rate, $R$, with 0, 0.33, 0.66, and 1; load limit, $L$, with 0 and 0.1; and $\alpha$ with 0 and 1. Here, $L = 1$ means DRC-AE does not conduct the EC rate control, $L = 0.1$ means the EC rate control limits the load of EC chunks to be less than 10%, $\alpha = 0$ means DRC-AE ignores the load of read operations, and $\alpha = 1$ means DRC-AE accounts for the load of read operations. In addition, I compared the results of the adaptive encoding with those of a naïve method that randomly selects the encoding target chunks, regardless of access frequency.

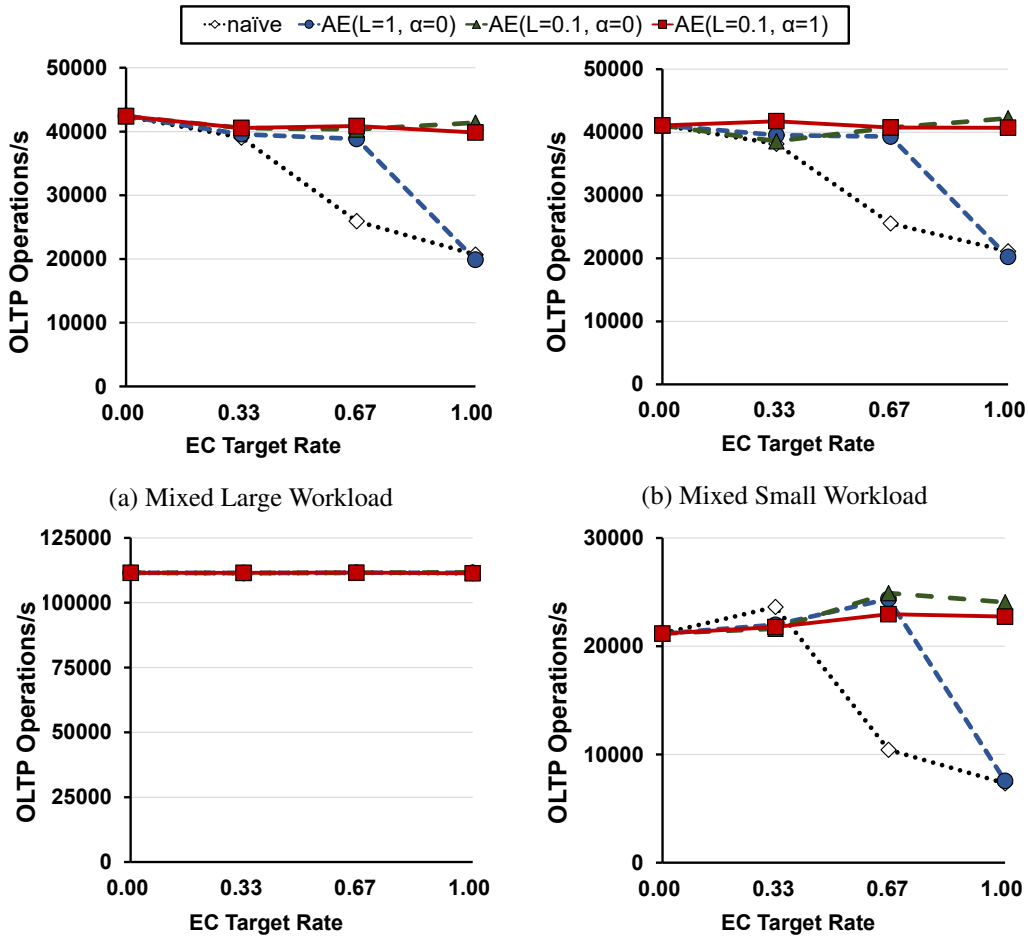
Table 4.2: Capacity Efficiency against Achieved EC Rate with Three-way Replication and EC (6D2P)

| Achieved EC Rate | Capacity Efficiency |
|---|---|
| 0 | 0.33 |
| 0.33 | 0.47 |
| 0.66 | 0.61 |
| 1.00 | 0.75 |

I measured performance in the normal condition, performance under a server failure, and via the achieved EC rate, which means the capacity ratio of encoded chunks against the total used capacity. Before measuring the performance in the normal condition, I ran the benchmark tool with the target workload and conducted the encoding process with the target parameters. After measuring performance in the normal condition, I stopped the server where the benchmark tools worked and then ran the benchmark tool on the replication pair to measure the performance under a server failure. I evaluated the achieved EC rate from the amount of encoded data in the encoding process. Table 4.2 shows the relationship between the achieved EC rate and capacity efficiency in the configuration of three-way replication and EC ($6D + 2P$), and a higher achieved EC rate means higher capacity efficiency. Here, I note that capacity efficiency improves further when the number of data symbols increases, as shown in Equation (4.4).

Figures 4.14, 4.15, and 4.16 show the results of these evaluations. The figures show the results of DRC-AE using the adaptive encoding with the different parameters ("AE (L = 1, $\alpha$ = 0)", "AE (L = 0.1, $\alpha$ = 0)", and "AE (L = 0.1, $\alpha$ = 1)") and DRC-AE using the naïve method ("naïve"). The x-axis shows the value of the target EC rate, $R$.

As for the mixed and write-only operation sets in the normal condition, the adaptive encoding without the EC rate control (i.e., $L = 0.1$) shows better performance than the naïve method when the target EC rate, $R$, was 0.66 and 1. This improved performance is because the EC priority calculation preferably selects the chunks with less performance impact to achieve higher performance for the same amount of the EC data. When the EC rate control was enabled (i.e., $L = 0.1$), the performance decrease was mitigated to less than 6.0%, whereas it was up to 65% without the EC rate control at the sake of the capacity efficiency, as shown in Figure 4.16. Setting $\alpha = 1$ (i.e., read operations are prioritized) does not show significant changes




(a) Mixed Large Workload

(b) Mixed Small Workload

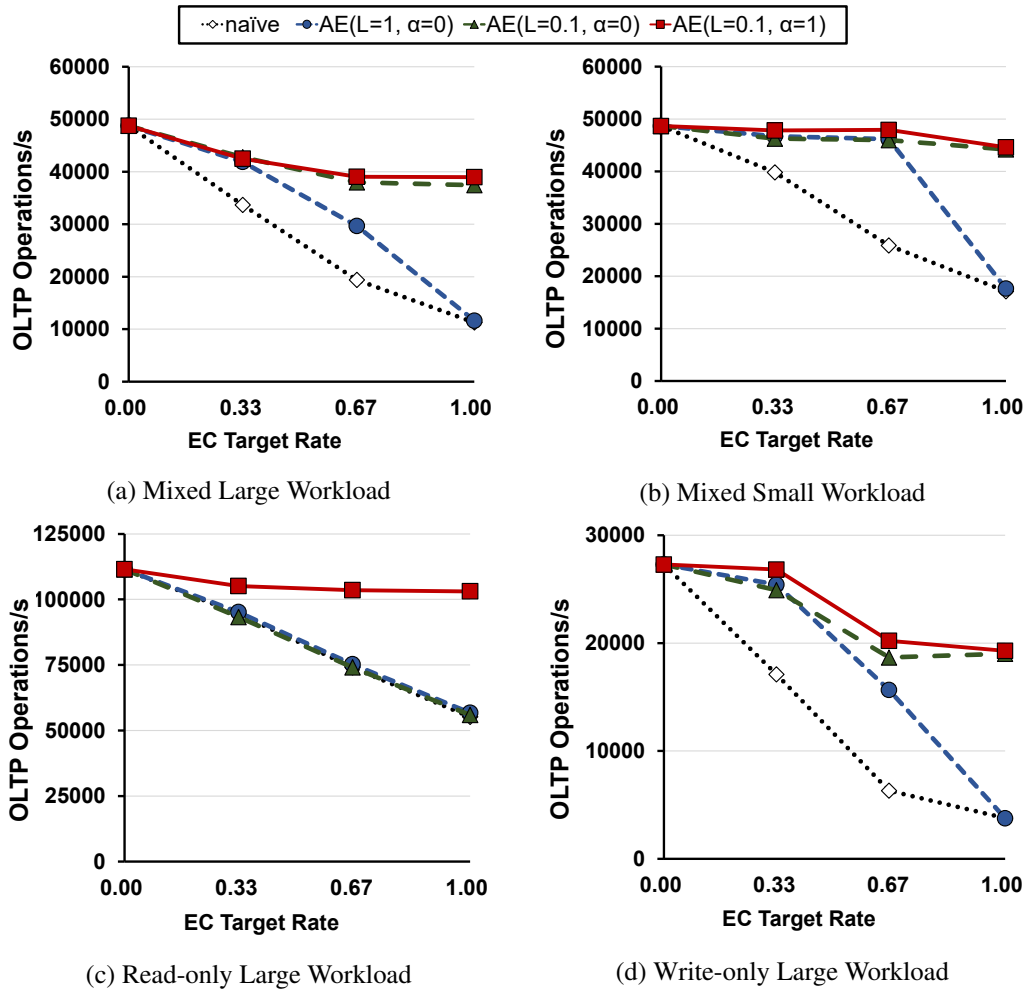
(c) Read-only Large Workload

(d) Write-only Large Workload

Figure 4.14: OLTP Performance against EC Target Rate in Normal Condition

in performance and capacity efficiency for these operation sets. This is because the working set size of read operations was the same as that of write operations in the mixed operation set, and there was no read operation issued in the write-only operation set. Therefore, prioritizing chunks with high read access frequency did not change the chunk selection of the adaptive encoding under these operation sets.

The important point here is that DRC-AE with the EC rate control (i.e., $L = 0.1$) shows a better achieved EC rate in the small working set than that in the large working set when the target EC rate, $R$, was 0.33 and 0.66. Figure 4.16 shows 32% higher achieved EC ratio of the mixed small workload than that of the mixed large workload when the EC target rate was 1. In the small working set, access was concentrated on the smaller amount of data than that of the large working set. Therefore, DRC-AE can encode the larger amount of cold data before the




(a) Mixed Large Workload

(b) Mixed Small Workload

(c) Read-only Large Workload

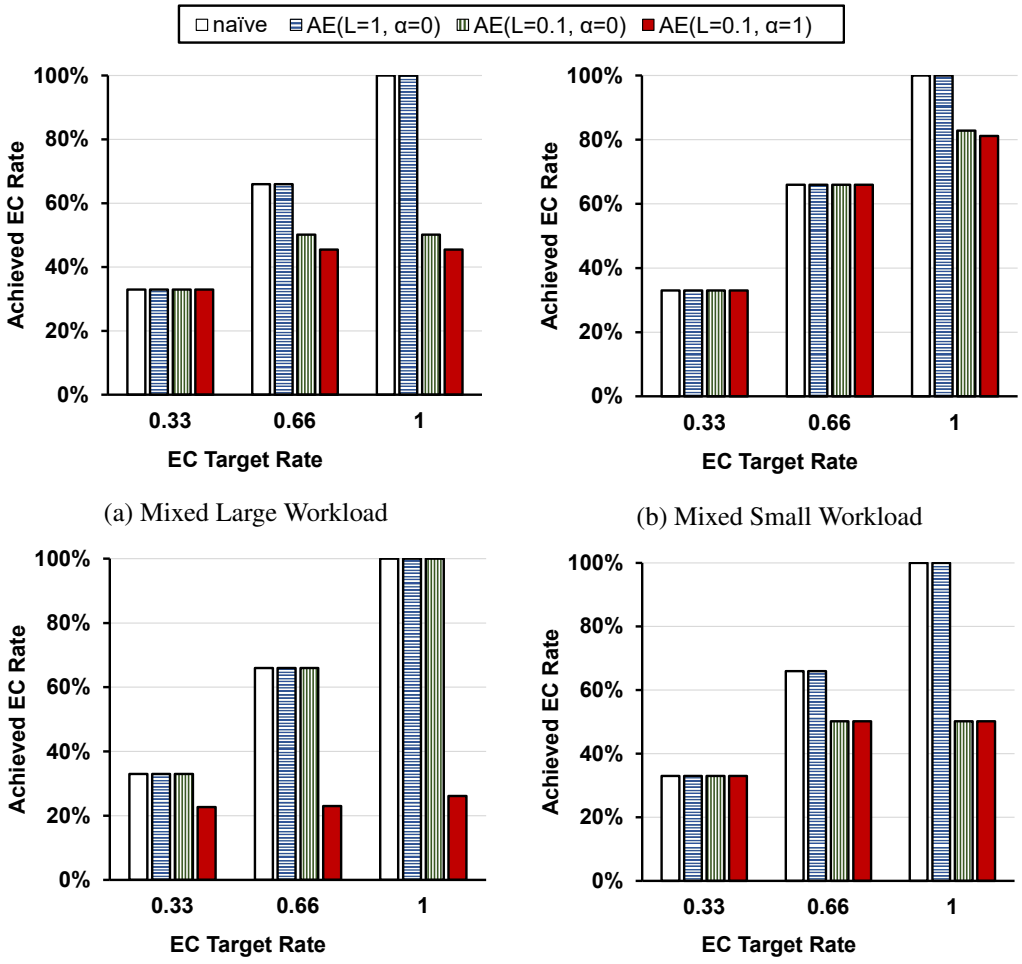
(d) Write-only Large Workload

Figure 4.15: OLTP Performance against EC Target Rate under Server Failure

load ratio of encoded chunks reaches the load limit, $L$. This means DRC-AE with the EC rate control achieves better capacity efficiency with the higher access locality.

For the read-only operation set in the normal condition, the adaptive encoding with all parameters shows almost equal performance and achieved EC rate with the naïve method when $\alpha$ was set as 0. This is because the adaptive encoding ignores read operations in the EC priority calculation if $\alpha$ is set as 0; therefore, it works in the same way with the naïve method under the read-only workload. By contrast, when $\alpha$ was set as 1, the achieved EC rate decreased by up to 74% without showing any performance increase. As read performance degradation because of EC does not occur in the normal condition, accounting read operations in the EC priority calculation only decreases capacity efficiency.

As for performance under a server failure, almost the same trends with the normal condition




(a) Mixed Large Workload

(b) Mixed Small Workload

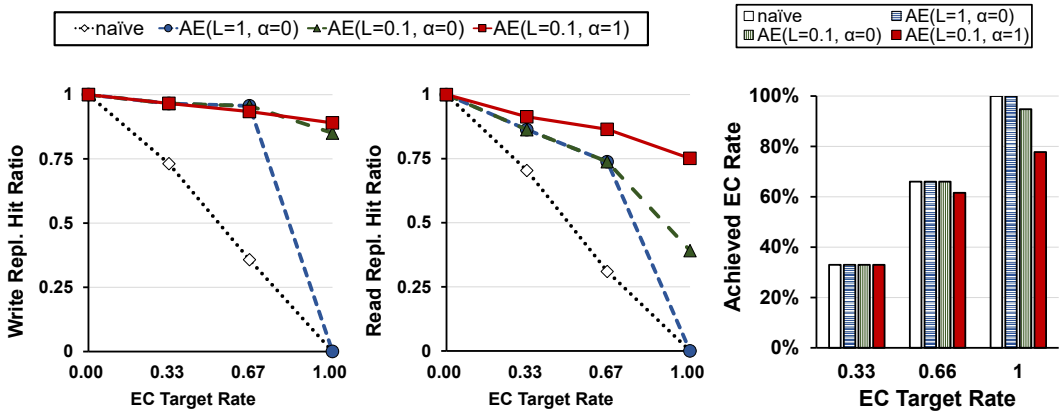
(c) Read-only Large Workload

(d) Write-only Large Workload

Figure 4.16: Achieved EC Rate against EC Target Rate

can be seen in the mixed working sets and the write working set. Although the configurations with the EC rate control show a larger performance decrease than those of the normal condition, it remained up to a 30% decrease, whereas it was up to 86% decrease without the EC rate control. By contrast, there was a large difference from the normal condition in the read-only large workload. Setting $\alpha$ to 1 mitigates the performance decrease from 50% to 7.6% when $L$ was set as 0.1 (i.e., the EC rate control was enabled). This is because the EC priority calculation with $\alpha = 1$ excludes chunks with high read access frequency from the encoding target. As a result, the decreased number of data rebuild during read operations reduces the performance decrease.

These results show that the EC priority calculation enables better capacity efficiency at the same level of performance, especially when a workload has a larger access locality. In addition,




(a) Write Replication Hit Ratio (b) Read Replication Hit Ratio (c) Achieved EC Rate

Figure 4.17: Replication Hit Ratio and Capacity Efficiency for MSR Traces

the EC rate control prevents performance degradation from exceeding an unacceptable level. Furthermore, a user can set a trade-off between these performance and capacity efficiency by setting the parameters of the adaptive encoding.

### Real-world Workload

To verify the effectiveness of DRC-AE in practice, I examined how DRC-AE provides the trade-off relationship between performance and capacity efficiency by using the MSR traces described in Section 4.1.3. I assumed that the same accesses with MSR traces were issued to virtual disk image files of VMs on a system using DRC-AE. I used six of the seven days of trace data for monitoring and the remaining one day for the evaluations. In these evaluations, I show the average results of all MSR traces.

First, I examined performance and capacity efficiency against the EC target rate for the same configurations with the synthetic workload evaluations in Section 4.4.3. Then, I examined the effectiveness of the proposed method against a method that emulates the HDFS-based conventional methods such as DiskReduce or HACFS.

For the performance evaluation, I used the write and read replication hit ratios, which show the write and read access ratio to replicated data. The higher write replication hit ratio means higher performance in the normal condition. The higher write and read replication hit ratios mean higher performance under a server failure. I also evaluated the capacity efficiency by using the achieved EC ratio. Because the actual used capacity is not available for MSR traces, I used the aggregated capacity of accessed 64 MB chunks as the total used capacity for the calculation of the capacity ratio.

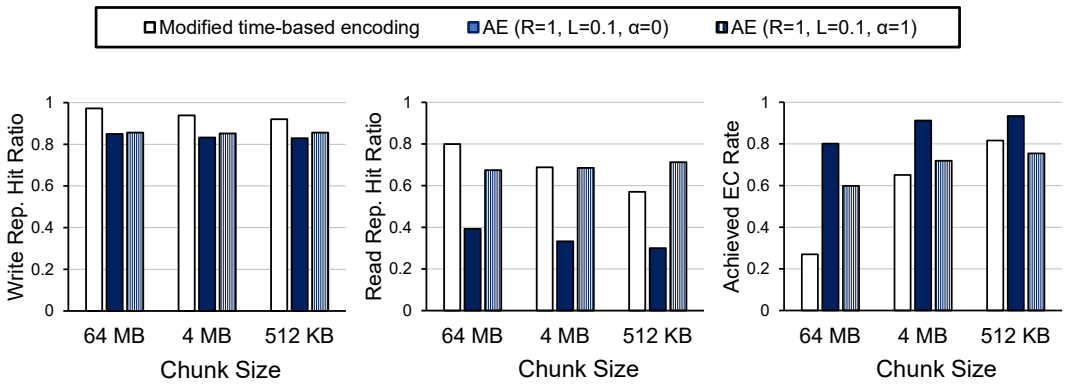
Figure 4.17 shows the results of the performance and capacity efficiency against the EC target rate for the same configurations with synthesized workload evaluations. The adaptive encoding shows higher write and read replication hit ratios in comparison with the naïve method when the target EC rate, $R$, was 0.33 and 0.66, as in the case of the synthesized workloads. With the EC rate control, the DRC-AE performance degradations were mitigated to 15% in write replication hit ratio and 61% in read replication hit ratio, while the achieved EC rate remains at 95% when read operations are not prioritized (i.e., $\alpha = 0$). When read operations are prioritized (i.e., $\alpha = 1$), the replication hit ratios were improved to 11% for write and 25% for read, and the achieved EC rate becomes 75%.

These results show that the use of adaptive encoding enables maximizing capacity efficiency while maintaining performance under real-world workloads.

Next, to compare the effectiveness of the proposed method with conventional methods, I evaluated the effects of different EC data selection algorithms and different chunk sizes. For the EC data selection algorithms, I evaluated the adaptive encoding with different read weights and the modified time-based EC data selection used in HACFS. In the evaluations, I set the EC target rate $R$ as 1 (i.e., all chunks could be encoded), the load limit $L$ as 0.1 (i.e., the EC rate control was enabled), and the read weight $\alpha$ as 0 (i.e., read operations are not prioritized) or 1 (i.e., read operations are prioritized) for the adaptive encoding. The modified time-based encoding switches replicated chunks that were not updated during the monitoring period into EC chunks. Unlike HACFS, I used the modified time of chunks instead of the modified time of files. Because I assumed huge virtual disk image files for the evaluations, I used chunk-based data selection as a practical variation of the file-based conventional method of HACFS. I also evaluated 64 MB chunks, which are the same as the HDFS-based conventional methods, 4 MB chunks, which are the same as CephFS, and 512 KB, which are the same as DRC-AE.

Figure 4.18 shows the comparison of the write replication hit ratio, the read replication hit ratio, and the achieved EC rate between the adaptive encoding ("AE (R = 1, L = 0.1, $\alpha$ = 0)" and "AE (R = 1, L = 0.1, $\alpha$ = 1)") and the modified time-based encoding with different chunk sizes. Note that the adaptive encoding with 512 KB corresponds to DRC-AE and the modified time-based encoding with 64 MB corresponds to the HDFS-based conventional methods.

For the write replication hit ratios, although the adaptive encoding with 512 KB chunks shows a 14% less hit ratio than the modified time-based encoding with 64 MB chunks, the performance difference can be seen as moderate. For the read replication hit ratios, the adaptive encoding with 512 KB chunks also shows a 50% less hit ratio when read operations are not prioritized (i.e., $\alpha = 0$) and a 9% less hit ratio when read operations are prioritized (i.e., $\alpha$ =




(a) Write Replication Hit Ratio (b) Read Replication Hit Ratio (c) Achieved EC Rate

Figure 4.18: Comparison of Performance and Capacity Efficiency of Adaptive Encoding and Conventional Methods for MSR traces

1) in comparison with the modified time-based encoding with 64 MB chunks. By contrast, for the achieved EC rate, the adaptive encoding with 512 KB chunks achieves 93% for $\alpha = 0$ and 75% for $\alpha = 1$, while the modified time-based encoding with 64MB chunks stays at 27%.

In addition, the results show that the finer-grained chunk size improves the capacity efficiency of the adaptive encoding considerably. When read operations are not prioritized (i.e., $\alpha = 0$), the adaptive encoding with 512 KB chunks achieves 93% of the achieved EC rate, whereas the adaptive encoding with 64 MB remains at 82%. No significant performance differences were confirmed among the different chunk sizes.

These results show that the adaptive encoding achieves much higher capacity efficiency than the HDFS-based conventional methods while mitigating performance degradations to a moderate level.

### 4.4.4 Overhead of Chunk and Parity Metadata Management

DRC-AE uses the chunk and parity metadata to manage chunks and parities over multiple servers. As the metadata requires additional disk space and metadata processing, capacity and performance overheads are unavoidable. In this section, I evaluated the capacity and performance overheads of the chunk and parity metadata management.

Table 4.3: Capacity Overhead of Chunk and Parity Metadata of DRC-AE

| | **User Data** | **Replica/ Parity Data** | **Chunk/Parity Metadata** | **Metadata Ratio** | **Capacity Efficiency** |
|---|---|---|---|---|---|
| 2-way Rep. | 0.5 MB | 0.5 MB | - | - | 0.50 |
| 3-way Rep. | 0.5 MB | 1.0 MB | - | - | 0.33 |
| EC (6D2P) | 3.0 MB | 1.0 MB | 1.2 KB | 0.030% | 0.75 |
| EC (6D3P) | 3.0 MB | 1.5 MB | 1.8 KB | 0.039% | 0.67 |
| EC (10D4P) | 5.0 MB | 2.0 MB | 3.5 KB | 0.048% | 0.71 |

**Capacity Overhead**

I investigated the increased capacity for the chunk and parity metadata among different redundancy methods. I calculated the capacity consumption of the metadata against the total capacity consumption. I assumed the chunk and parity metadata are stored with three-way replication. Table 4.3 shows the evaluation results.

The left three columns show the capacity consumption of the user data, the replica or parity data, and the sum of chunk and parity metadata for a replication unit or an EC unit of chunks. The metadata ratio is the percentage of the total capacity of the chunk and parity metadata against the total capacity consumption. Capacity efficiency is the ratio of user data against the total capacity consumption.

For two-way replication and three-way replication, DRC-AE stores the replica files in the same file path as their original user files. As these replication methods do not use the chunk and parity metadata, there is no capacity overhead of the metadata. By contrast, for EC, DRC-AE manages the fixed-length chunk metadata for each chunk and the parity metadata for the parity files. The chunk metadata contains 8-byte Parity IDs and an additional 5-byte header while the parity metadata contains 24-byte chunk IDs. In addition, both chunk and parity metadata are replicated to the replication pairs. With 6D2P configuration, the total capacity consumption of the chunk and parity metadata becomes about 1.2 KB, and the metadata ratio becomes 0.030%. The metadata ratio becomes 0.039% and 0.048% for 6D3P and 10D4P EC, respectively.

Considering that EC largely improves the capacity efficiency from replication, I think that the capacity overhead of the chunk and parity metadata management is sufficiently small.

**Performance Overhead**

For the performance overhead, I measured the response time of file operations. To evaluate the impact of the amount of the chunk metadata, I used files of different sizes: 512 MB, 1 GB, and 2 GB. The number of chunks in a 512 MB file is 1000 and increases linearly with the file size. I also evaluated the performance impact of the chunk access monitoring in DRC-AE. I evaluated the performance of DRC-AE using three-way replication with monitoring disabled (DRC-AE Rep.), DRC-AE using three-way replication with monitoring enabled (DRC-AE Rep. w/ monitoring), and DRC-AE using 6D2P EC (DRC-AE EC (6D2P)). I used the custom-made benchmark tool that measures the response time of file open, close, 4KB random read, and 4KB random write operations. To avoid the cache effect, the benchmark tool issues synchronous write operations and drops cache before read operations. I measured the average response time of 1000 operations for each configuration and operation type. Figure 4.19 shows the evaluation results.

The response time of open operations increases with the larger file size in the three-way replication with monitoring enabled and EC. The response time of open operations increases up to 1 ms for three-way replication with monitoring enabled and exceeds 3 ms for EC. The monitoring requires initialization of the on-memory monitoring counters for each chunk during file open, while EC requires reading chunk metadata from disks. The performance overhead of these initialization processes during file open increases linearly with the number of chunks.

For operations other than open, the evaluation results show ignorable performance overhead of the metadata management. The read and close operations show ignorable performance differences among the file sizes and redundancy methods. The write operation shows a longer response time in the EC configuration than the replication configurations. As the response time of the EC write operation is almost the same for the different file sizes, the longer response time can be seen as the overhead of the EC data update, rather than the metadata management. The chunk and parity metadata are updated when the chunk is encoded and read from disks when the file is opened. Because the read, write, and close operations are not accompanied by disk accesses for the metadata, the performance overhead of the metadata management is sufficiently small in these operations. The performance impact of read and write monitoring also seems ignorable.

These results show that open operations of DRC-AE are accompanied by the considerable performance overhead of the chunk and parity metadata management. The performance overhead could be an issue in a workload with frequent open operations on large-size files.

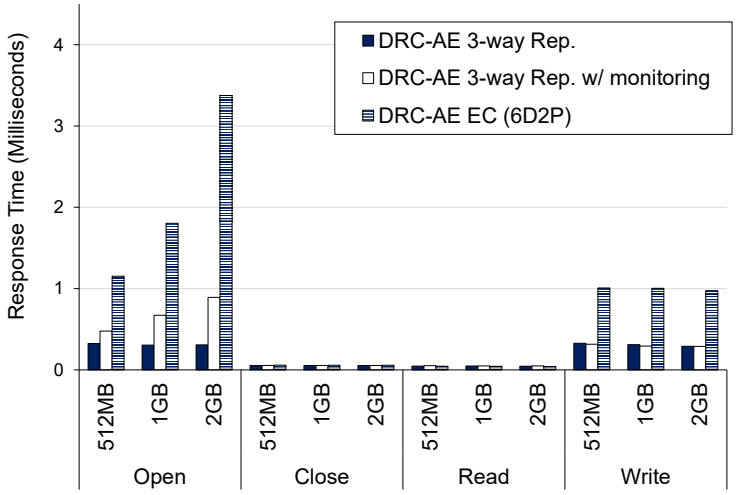



Figure 4.19: Performance Overhead of Chunk and Parity Metadata Management for Different File Sizes

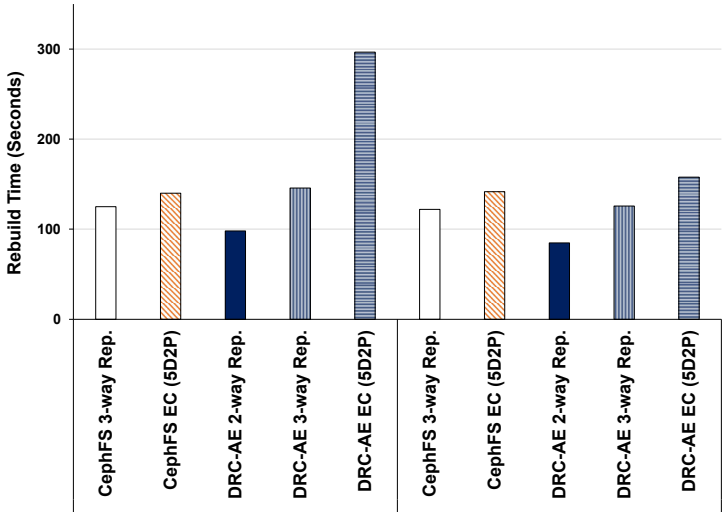
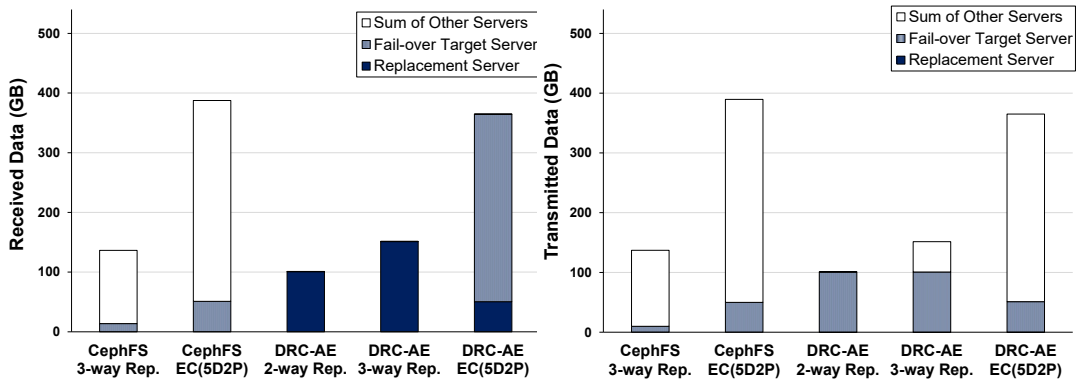



Figure 4.20: Rebuild Performance

However, the performance overhead should only be a small concern if an application keeps accessing a file without closing the file once it opens the file. For example, OLTP keeps table files open once it opens the files; therefore, the impact of the lower open performance should be small.




(a) Received Data (b) Transmitted Data

Figure 4.21: Amount of Network Traffic between Servers during Rebuild Processing

### 4.4.5 Rebuild Performance

In this section, I evaluated the rebuild performance of DRC-AE in comparison with CephFS. I measured the time and amount of network traffic to recover the data redundancy after a server failure.

I used standard rebuild methods for DRC-AE and CephFS, respectively. In DRC-AE, lost data in the failed server was recovered to a replacement server, while the lost data were recovered to the remaining servers in CephFS. For DRC-AE, I injected a failure into one of the eight servers and injected failed-over applications and their data to the replication pair (fail-over target server). After initializing the failed server, I reused the initialized failed server as a replacement server. I measured the time to recover user data, replica data, and parity data in the replace server. For CephFS, I evaluated the time to recover data redundancy in the remaining servers after injecting a failure to one of the eight servers. The applications on the failed server were failed over to another server in the cluster. I measured the time from the start of the rebuild process of CephFS until the server cluster gets back to the normal state.

Each server stored 48 1 GB user files; therefore, the total used capacity was 384 GB. I evaluated both 10GbE NIC and 100GbE NIC configurations with NVMe SSDs. For the configurations using EC, I used 5D2P EC so that CephFS could recover data redundancy on the remaining seven servers.

To maximize the rebuild performance of CephFS, I applied parameter tuning to CephFS [UHS17]. I set *osd_max_backfills* as 100 and *osd_recovery_max_active* as 100 to increase the priority of the rebuild process. Also, I turned off the placement group auto-scaling capability of

CephFS to avoid data rebalancing during the measurements. The number of placement groups was set to 256 for three-way replication and 128 for EC. I used the default values for other parameters.

Figure 4.20 shows the comparison of rebuild time between CephFS and DRC-AE. Figure 4.21 shows the amount of network traffic during the rebuild process. Figure 4.21a shows the amount of the received data for the replacement server, the fail-over target server, and the sum of the other servers, while Figure 4.21b shows the amount of the transmitted data.

These figures show that the rebuild time of DRC-AE is longer than CephFS with the same redundancy methods in the 10GbE NIC configuration and almost the same in the 100GbE NIC configuration. Although the total amount of data traffic is almost the same between CephFS and DRC-AE, the network traffic of DRC-AE heavily concentrated on the replacement server or the fail-over target server. In DRC-AE, all rebuilt data are transferred to the replica server from the fail-over target server. In addition, the fail-over target server collects the remaining chunk data and parity data to rebuild the lost EC data. As a result, the network bandwidth of the replacement server and the fail-over target server tends to be a performance bottleneck of DRC-AE in the 10GbE NIC configuration. The use of 100GbE NICs greatly improves the rebuild performance of DRC-AE because the larger bandwidth of 100GbE NICs alleviates the limit of network bandwidth. By contrast, because CephFS performs data rebuild in a distributed manner, the network bandwidth of each server did not become the performance bottleneck of CephFS, even in the 10GbE NIC configuration.

These results indicate that the rebuild performance of DRC-AE could be an issue in the configurations with low network bandwidth. However, the use of faster NICs enables DRC-AE to achieve competitive rebuild performance with CephFS.

## 4.5 Discussion

As shown in Sections 4.4.2 and 4.4.3, DRC-AE achieves better access performance compared with CephFS in a high disk performance environment except for sequential write while enabling higher capacity efficiency. These results indicate that DRC-AE succeeds in utilizing the advantages of both using the local filesystem and using EC. By contrast, DRC-AE requires an application to run on the same server with its data. This requirement could introduce unbalanced capacity and unbalanced load between servers in comparison with the scale-out distributed storage where capacity and load are balanced by equally distributing data among servers.

Because of these characteristics, the target applications of DRC-AE become applications that require fast response time, higher CPU power, or larger read throughput than network bandwidth. An OLTP system using NVMe SSD or SCM is an example of a DRC-AE's target application. I conclude that load balancing or intelligent resource allocation algorithms [WW17, MMP+12] could enable the balanced load and capacity between servers for DRC-AE. Investigating these solutions will be future work to expand the range of applications of DRC-AE.

Further scalability of encoding processing is another challenge of DRC-AE. DRC-AE has certain advantages in terms of the amount of network traffic among servers in data-access processing. The less network traffic mitigates the network bottleneck congestion and could enable higher data-access scalability once proper load balancing is enabled. On the other hand, in the encoding process, DRC-AE requires a large amount of data transfer between servers for the parity calculations. Further work remains as future work to reduce network traffic between servers or racks to achieve scalability of the encoding process. Some prior studies in the scale-out distributed storage [LHL17, WLXW17] could be applied to DRC-AE for this aim.

Improving the file open performance on large files is also another challenge of DRC-AE. Because of the initialization of the monitoring counters and the chunk metadata, the file open performance decreases with the number of chunks in the opened file, as described in Section 4.4.4. Although the file open performance is not very important in the target applications such as OLTP or VDI because of the low frequency of open operations on large files, it could be an issue for other applications that frequently open large files. In the current prototype, DRC-AE initializes the monitoring counters and the chunk metadata synchronously when it opens a file. Investigating the performance improvements such as asynchronous initialization in the file open operation will be future work of DRC-AE.

## 4.6 Related Work

Much research has been undertaken with respect to the dynamic control of data redundancy for scale-out distributed storage, as described in Section 4.1.2. Bin et al. [FTXG11] proposed DiskReduce, which improves capacity efficiency by changing the RAID level for user data in HDFS. Xia et al. [XSBP15] also proposed HACFS, which combines replication and two types of EC and dynamically adapts to workload changes in HDFS. Similar techniques were proposed in [CAA+17] for a propriety distributed storage system. In addition, some scale-out distributed storage implementations offer the tiering functions that can migrate data between

the replication pool and the EC pool [UHS17, Bor07, KC16]. The proposed method uses a local filesystem-based approach for redundancy control. The proposed method has advantages over other work with respect to access latency and resource consumptions on the network, CPU, and disk access for data-access processing. In addition, to the best of my knowledge, no preliminary work offers control of the amount of EC data based on workload, whereas adaptive encoding offers EC rate control. Therefore, I think the easier performance management is another advantage of the proposed method.

Much attention has been drawn to migrate legacy POSIX-based applications to cloud computing environments and on-premises infrastructures [ZLL+20, MNE+14]. CephFS is one of the most widely used types of POSIX-complaint distributed storage for these applications [WBM+06]. CephFS offers performance scalability by using the pseudo-random data placement algorithm that eliminates the need for a centralized data place management. However, because an application needs remote access to data and file metadata, the network access overhead remains a performance issue for these applications. The proposed method reduces the network access overhead for these applications by storing data and metadata in the local storage.

There have been many studies focused on improving the performance of EC data access [CDLC14, ZLL+20, ZT19] and rebuild performance in failure cases [VRP+18, RNW+15]. Those performance improvements realize higher performance in comparison with the traditional Reed–Solomon coding used in the proposed method. However, these methods assume that the distributing data among servers and the performance overhead discussed in Section 4.1.3 remain. The higher random and sequential read performance because of the proposed local filesystem-based approach is a major advantage of the proposed method in high-performance disk configurations. Furthermore, those performance improvements of EC are also applicable to the proposed method as well as other EC-based systems.

File metadata extension has been studied in the field of network-attached storage (NAS) for data backup and data migration [NSI17, MHS20]. These studies use additional file metadata to store reference pointers to data on other servers. They use the object ID of the object storage and the file path of the migration source file in the reference pointers. The proposed method uses global IDs to refer to parity files and chunks among the servers to realize EC, which differs from the approaches adopted in prior work. In addition, I previously proposed using custom file metadata to improve the network protocol coverage of an NAS product [FSN18]. This study can be seen as an application of the custom file metadata to the realization of DRC-AE.

This chapter developed adaptive encoding that improves the encoding process of the prior

work published in [FLY19]. Adaptive encoding introduces the EC rate control to the original encoding process of the prior work. EC rate control enables a user to avoid performance degradation by limiting a load of EC data based on access monitoring, whereas the prior work requires manual settings to maintain the required performance. In the field of hierarchical storage management, many studies use access monitoring for data migration control between storage tiers with different access characteristics [NXX18]. Most of these precedent works assume that each tier has fixed-size capacity and allocates data to higher tiers in accordance with access frequency. As far as I know, no precedent work adaptively changes the capacity of storage tiers based on the performance requirement and the access monitoring as in the adaptive encoding. In addition, I demonstrated that the parameters of the adaptive encoding enable a user to set the trade-off between performance and capacity efficiency. I believe the progress in this chapter dramatically improves the practicality of the prior work.

## 4.7 Chapter Summary

This chapter proposed a new lightweight redundancy control method, named DRC-AE, for a server-based storage system with random-intensive workloads. DRC-AE aims to achieve high performance and high capacity efficiency under random-intensive workloads.

DRC-AE adopts a local-filesystem-based redundancy control to enable an application to directly access its data in the local storage. This approach reduces the remote data and metadata access in data-access processing and enables fully utilizing the high performance of NMVe SSDs or SCM. I also propose using adaptive encoding to achieve higher capacity efficiency against the same level of performance while controlling the amount of EC data to avoid unacceptable performance degradation. Also, the adaptive encoding enables a user to set a trade-off between performance and capacity efficiency by setting parameters of the adaptive encoding.

The experiments showed that DRC-AE outperforms CephFS, one of the most frequently used types of POSIX-complaint distributed storage, in access performance except for sequential write performance in high-performance disk configurations. DRC-AE achieves up to 230% better OLTP performance than CephFS. In addition, I confirmed that DRC-AE achieves higher capacity efficiency than the HDFS-based conventional methods without deteriorating performance. DRC-AE increases the amount of EC data by up to 66% from the HDFS-based conventional methods under real-world workloads. I also confirmed that the presented parameters could be used to set a trade-off between performance and capacity efficiency. These results show that DRC-AE enables high performance and high capacity efficiency in a server-based

storage system with random-intensive workloads.

# Chapter 5

# Dynamic Redundancy Control with Delayed Parity Update

This chapter proposes another performance improvement method using the custom file metadata for server-based storage. This chapter uses the custom file metadata to solve the problem of the degraded write response time of EC data in DRC-AE proposed in Chapter 4. This chapter proposes DRC-DPU which is the third application of the custom file metadata.

With the progress of cloud computing environments, there is a trend of migrating legacy applications such as DB and VDI to the clouds [Kim09, MNE+14]. 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. Cloud computing environments enable cost reductions in equipment and operations compared to legacy systems that use dedicated servers.

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 [San03, WBM+06]. 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.

As described in Chapter 4, DRC-AE has been proposed as a method to improve the response time and the random access performance of server-based storage. DRC-AE 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-AE uses high-performance replication for frequently accessed data and capacity-efficient EC for infrequently accessed data as a redundancy method. By dynamically controlling redundancy methods based on data access characteristics, DRC-AE achieves high random performance while maintaining the capacity efficiency of conventional server-based storage.

However, DRC-AE still has a performance issue of poor write response time for EC data. DRC-AE 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 [DB13].

This chapter proposes DRC-DPU to improve the degraded write response time for EC data in DRC-AE. 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, I 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-AE and enables the stable write response time in server-based storage. Furthermore, I 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.

- this chapter proposes a method to improve the degraded write response time of EC data in DRC-AE by asynchronizing parity updates of EC data.

- this chapter proposes a method to reduce the EC overhead by consolidating parity updates for the same EC data.
- this chapter demonstrates that the proposed method achieves the better write response time than the conventional EC optimizations.
- this chapter demonstrates the effectiveness of our proposed method in parity update reduction and metadata capacity overhead.

This chapter is an extended version of our prior paper [FLY21a].

## 5.1 Conventional approaches

There is a trend of using server-based storage as back-end storage of legacy applications such as DB and VDI [Kim09, MNE+14, ZLL+20]. This section provides background knowledge of the use of server-based storage in legacy applications. Then, I discuss conventional approaches for improving the random performance of server-based storage.

### 5.1.1 Applying Server-based Storage to Legacy Applications

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. Even in on-premise environments, inexpensive server-based storage has been spreading as a practical alternative to the conventional storage appliances [vmw21, UHS17]. 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 [San03]. 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 [Fik10, HSX+12, KBP+12]. 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 [Pla13].

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.

### 5.1.2 Dynamic Redundancy Control with Adaptive Encoding

As described in Chapter 4, DRC-AE has been proposed as a method to improve the random performance of server-based storage. DRC-AE is a redundancy control module that makes local data in a server redundant between servers. DRC-AE stores an application and its data on the same server, and reduces the overhead of network communication. In addition, DRC-AE also dynamically switches the data redundancy method between replication and EC, depending on the performance requirements and workloads. DRC-AE achieves high capacity efficiency while maintaining high performance.

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

Furthermore, DRC-AE reduces network communication during metadata access by storing the redundancy metadata in the local file system. DRC-AE stores metadata of chunk and parity configuration in the custom file metadata, which is an extension of the local file system [FSN18]. By storing metadata in the local file system, DRC-AE avoids the metadata server becoming a performance bottleneck as in conventional methods [Gib10, XSBP15].

### 5.1.3 EC Write Optimizations

Several EC write optimizations have been proposed to improve the random write performance of EC. In the following, I describe the naïve method of EC write and explain two other prior methods: parity logging [CDLC14] and speculative partial write (SPW) [ZLL+20].

**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 [Pla13].

$$(p_1, \cdots, p_k)^T = A(d_1, \cdots, d_m)^T \tag{5.1}$$

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.

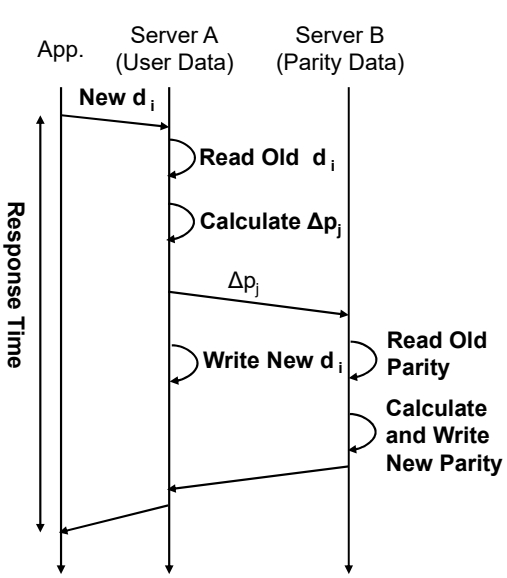
$$\Delta p_j = a_{ij} \times \Delta d_i \tag{5.2}$$

where $\Delta p_j$ denotes the difference between the old $p_j$ and the new $p_j$, $\Delta d_i$ 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 $\Delta p_j$ to the old $p_j$.

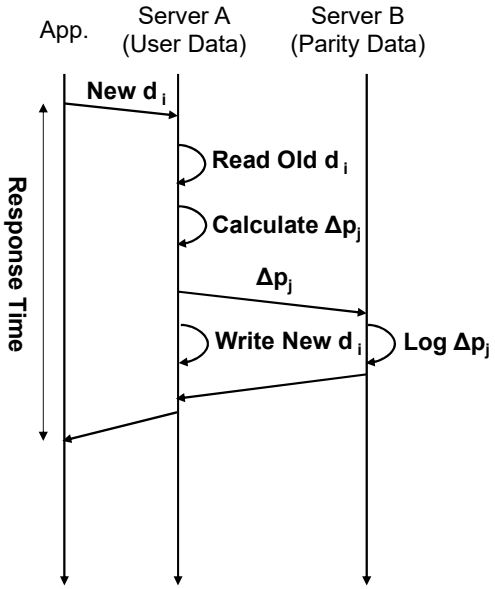
Figure 5.1a shows the process flow of the naïve method. The figure shows the communication between the servers when the application performs a partial write to EC data stored in Server A. Server A, which receives the write request from the application, reads the old data from the disks and calculates $\Delta p_j$. Then, Server A sends $\Delta p_j$ 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 $\Delta p_j$ 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 [CDLC14]. 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 5.1b shows the processing flow of parity logging.

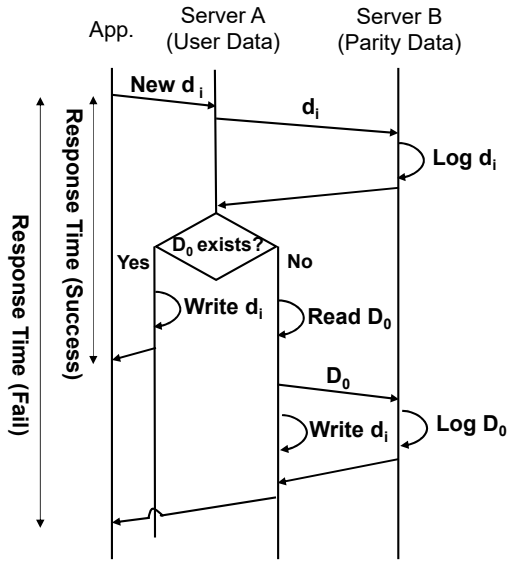
As the figure shows, $\Delta p_j$ is logged without reading the old parity on Server B. Parity logging avoids reading the old parity data from the disks and allows faster response time compared with the naïve method.




(a) Naive Method




(b) Parity Logging




(c) Speculative Partial Write

Figure 5.1: Conventional Parity Update Methods

**Speculative Partial Write (SPW)** 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 5.2 to calculate the sequence of $\Delta p_i$ from the logged new data in order. The failure recovery process calculates the latest parity data by adding all $\Delta p_i$ 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 5.1c 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.

### 5.1.4 Challenges

DRC-AE improves the random access performance of server-based storage. DRC-AE reduces the overhead of network communication by storing data in the local storage of servers. DRC-AE 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-AE is considerably worse than replicated data because of the overhead of parity updates. DRC-AE 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, I evaluated the write response time of the conventional methods. In the preliminary experiment, I evaluated the difference in write response times of the naïve method, parity logging, SPW, and DRC-AE compared with replication. For the experiment, I implemented parity logging and SPW to the DRC-AE prototype. I measured the cumulative distribution function of write response time in the same configuration as in Section 5.3.1. I used fio [fio21] for benchmarking. One thread issued 8 KB random writes to a 48 GB

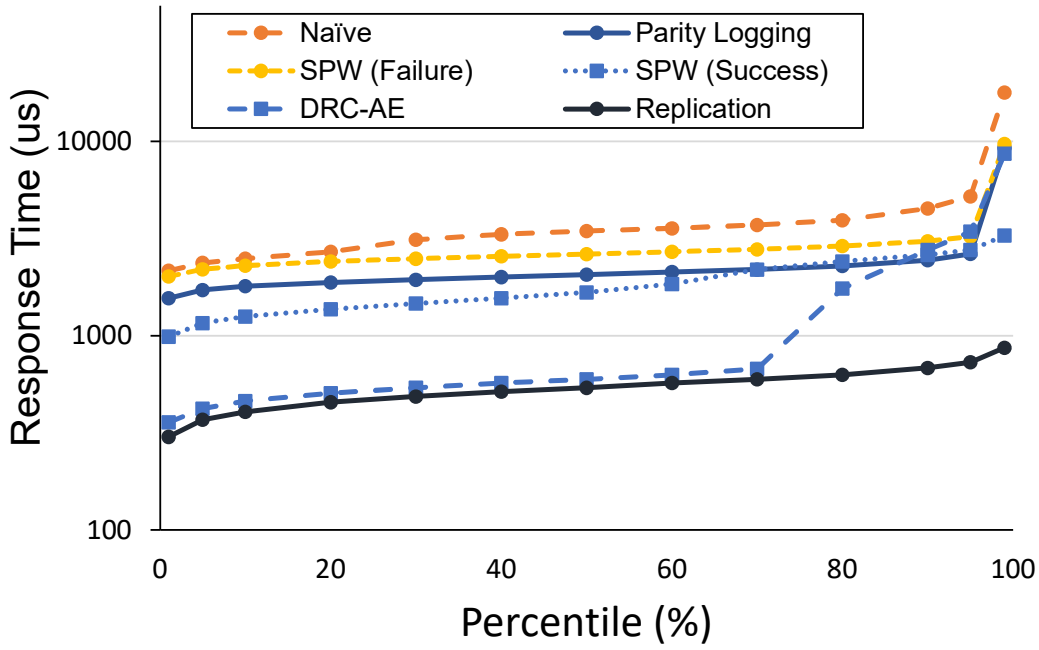



Figure 5.2: Write Response Time Comparison of Conventional Methods

file. In DRC-AE 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 5.2 shows the evaluation results.

The write response time of DRC-AE 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-AE 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.

This chapter aims to solve the problem of the degraded write response time of DRC-AE and enable stable storage access in server-based storage. I aim to make the write response time of EC data in DRC-AE equivalent to replication.

I 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.

## 5.2 Dynamic Redundancy Control with Delayed Parity Update

### 5.2.1 Overview

This section proposes DRC-DPU to improve the degraded write response time of EC data in DRC-AE. DRC-DPU improves the write response time by asynchronizing the parity update of EC data. Figure 5.3 shows an overview diagram of DRC-DPU.

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

In redirect on write, DRC-DPU redirects writes for 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 [GKDB09, Pet09]. The important point here is that DRC-DPU uses EC only for data that is updated infrequently. Therefore, it is unnecessary to take care of cases where the speed of merge processing is not fast enough for the incoming write requests.

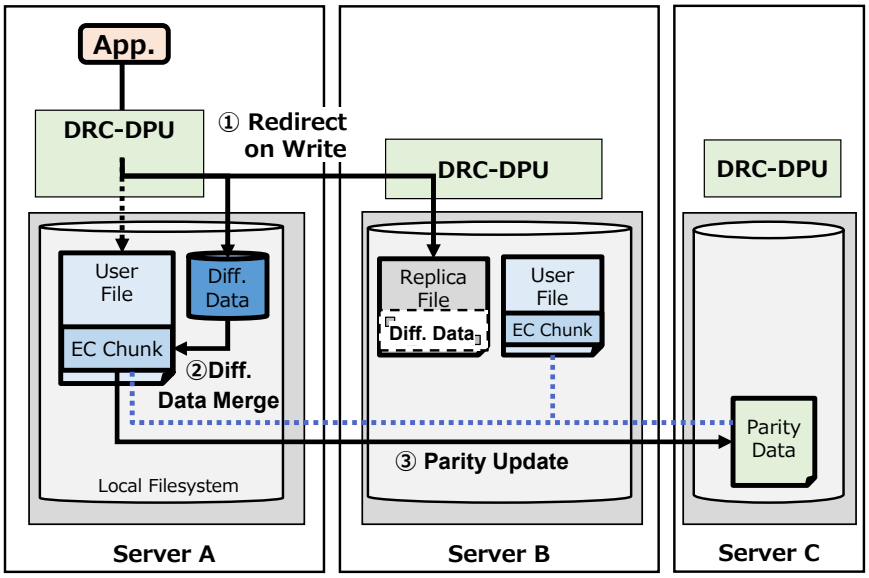



Figure 5.3: Dynamic Redunadncy Control with Delayed Parity Update

Figure 5.4 shows the entire processing flow of DRC-DPU.

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

### 5.2.2 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 5.5 shows the data layout of DRC-DPU.

DRC-DPU uses the custom file metadata [FSN18], 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.

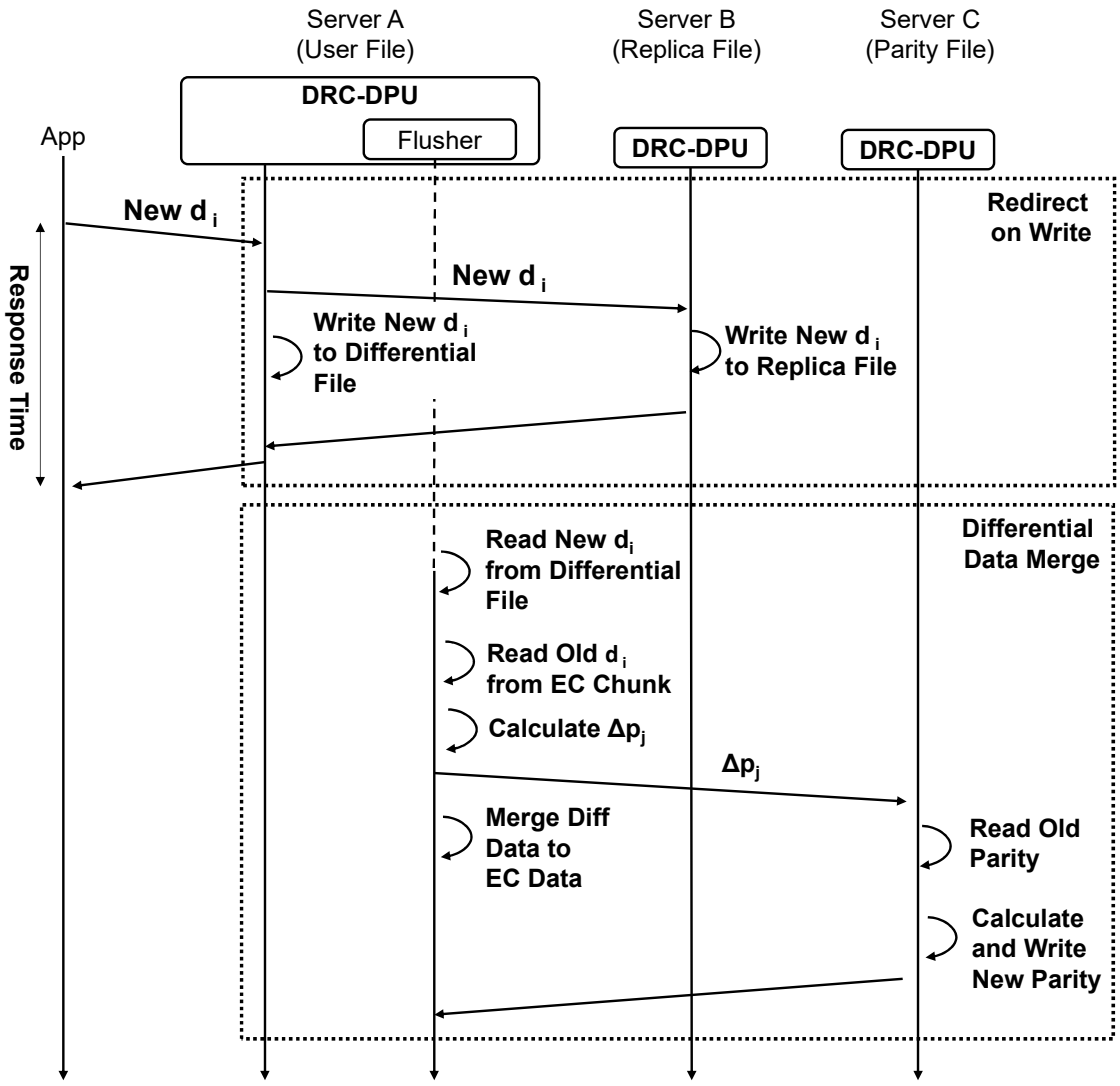



Figure 5.4: Flow of Delayed Parity Update

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

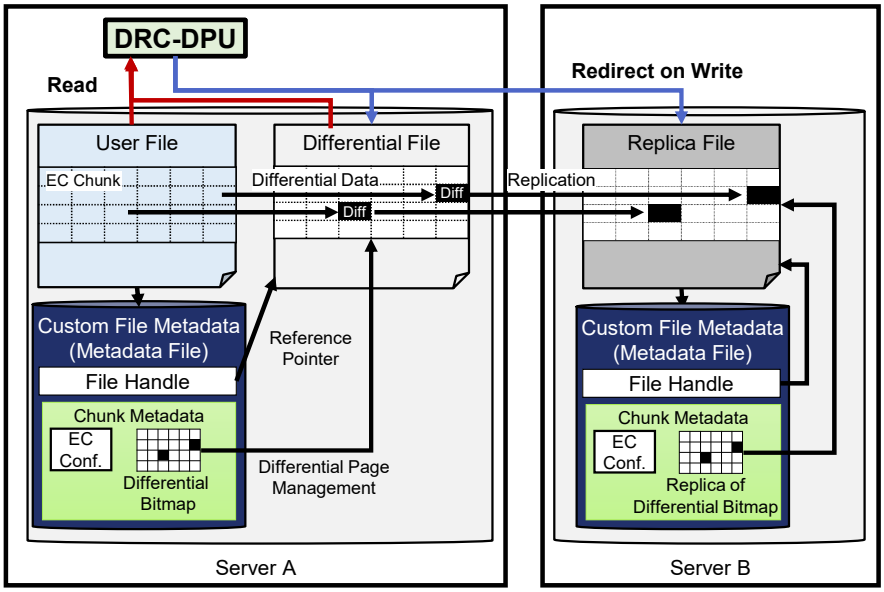



Figure 5.5: Data Layout in DRC-DPU

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. I 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 5.3.

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

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.

### 5.2.3 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.

$$ECRatio = R + B * r/(r - (m + k)/m) \tag{5.3}$$

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 5.3.2, 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.

Table 5.1: Experimental Environment

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

## 5.3 Evaluation

This section confirms the effectiveness of DRC-DPU. In Section 5.3.1, I evaluate the performance improvement of the proposed method using the synthesized workload. In Section 5.3.2, I evaluate the parity update reduction and metadata capacity overhead of the proposed method using the real-world workload.

### 5.3.1 Synthesized Workload Evaluation

#### Experimental Environment

Table 5.1 shows the experimental environment. I used three commodity servers and ran the prototype of proposed and conventional methods. I implemented parity logging, SPW, and DRC-DPU in the DRC-AE prototype, which I developed in the prior study [FLY19]. I ran the benchmarking tool *fio* on one of the servers [fio21]. As a redundancy method, I used two-way replication and 2D1P EC.

#### Experimental Results

I evaluated write response time and throughput under random write workload for the proposed method and the conventional methods. I evaluated the performance degradation from replication for the naïve method, parity logging, SPW, DRC-AE, and DRC-DPU. For SPW, I evaluated both the failure case and the success case. In DRC-AE and DRC-DPU measurements, 80% of the data were replicated, and 20% of the data were encoded into EC. Because *fio* has no access

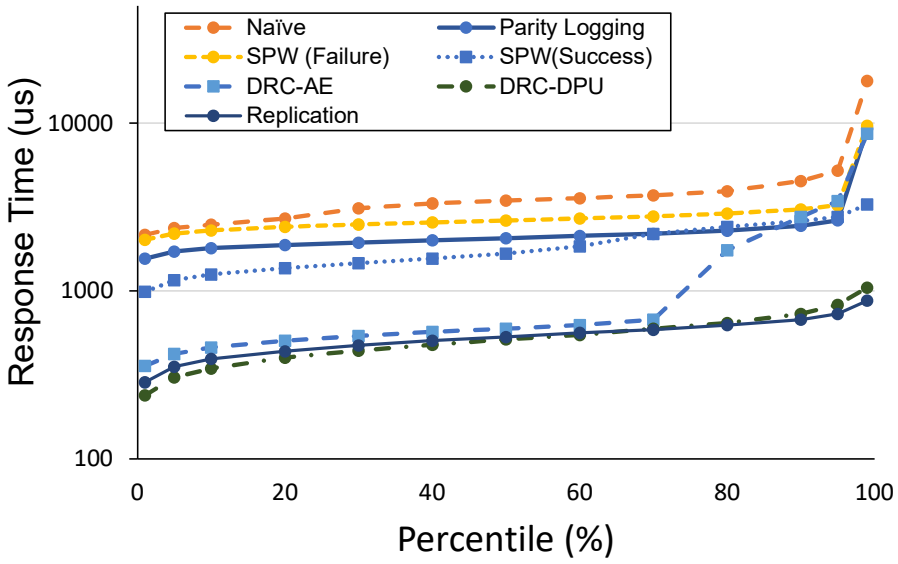



Figure 5.6: Comparison of Write Response Time (8 KB IO)

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 data can be encoded for the same EC data access ratio in actual use cases [YZ16]. To evaluate the performance impact of the read–modify–write processing, I 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 5.6–5.8 show the evaluation results.

Figure 5.6 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 5.7 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 5.8 shows that the throughput degradation of DRC-DPU against replication is only

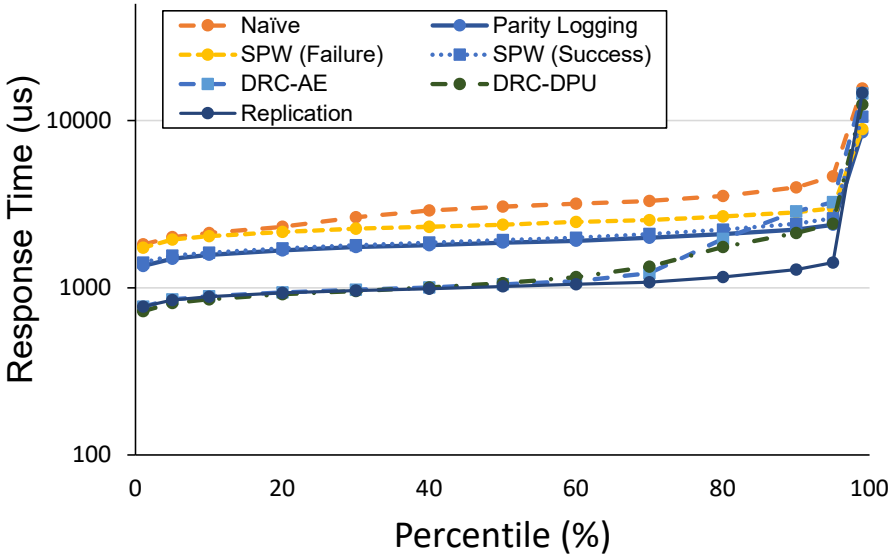



Figure 5.7: Comparison of Write Response Time (2 KB IO)

9% at 8 KB and 13% at 2 KB. In addition, DRC-DPU shows a higher throughput than other conventional methods.

### 5.3.2 Real-world Workload Evaluation

This section evaluates the efficiency of the proposed method using the real-world workload. I used Microsoft Research Cambridge (MSR) traces, which is 1-week I/O traces of enterprise servers, for the evaluations [NDR08]. I evaluated the reduction in the number of parity updates and the metadata capacity overhead of the proposed method.

#### Parity Update Reduction

To verify the effectiveness of differential data merge, I evaluated the reduction of the number of parity updates. I examined the change in the number of parity updates when the total capacity of the differential pages is changed.

I 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). I 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. I used the first six days in the MSR traces to encode low-access frequency chunks into EC chunks. I simulated the change in the number of parity updates for the remaining day in the MSR traces. Table 5.2 shows the write access characteristics of these

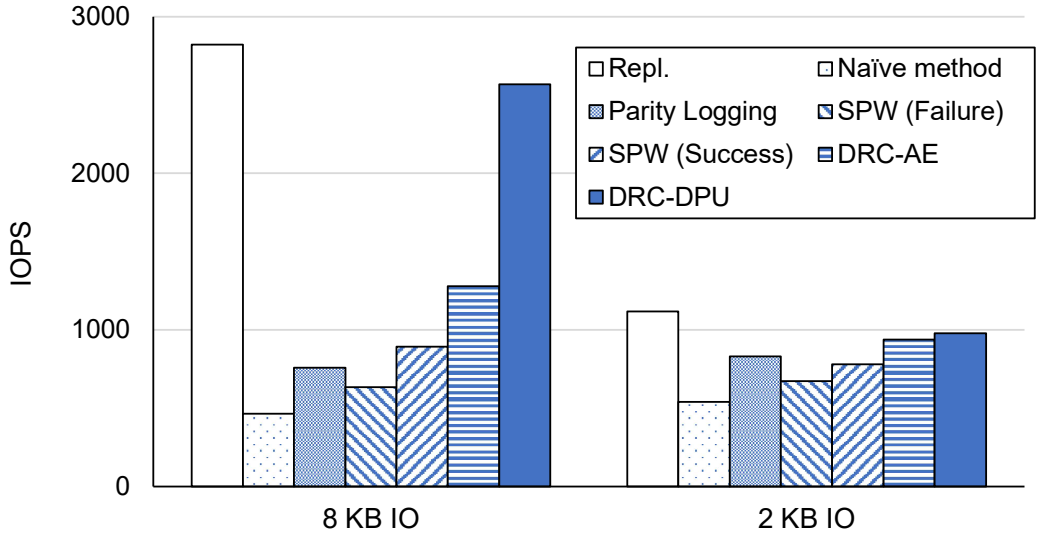



Figure 5.8: IOPS Comparison

Table 5.2: 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 |

traces for the evaluated trace periods.

I also set the EC target rate $R$ to 0.9 and used Equation 5.3 to determine the total amount of EC data. I examined the number of parity updates for the different capacity ratios $B$ of differential pages to the total amount of chunks accessed during the trace period. Figure 5.9 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 proxy_1 and proj_1. The reason for the poor results in proxy_1 and proj_1 is thought to be because of fewer writes to the same EC

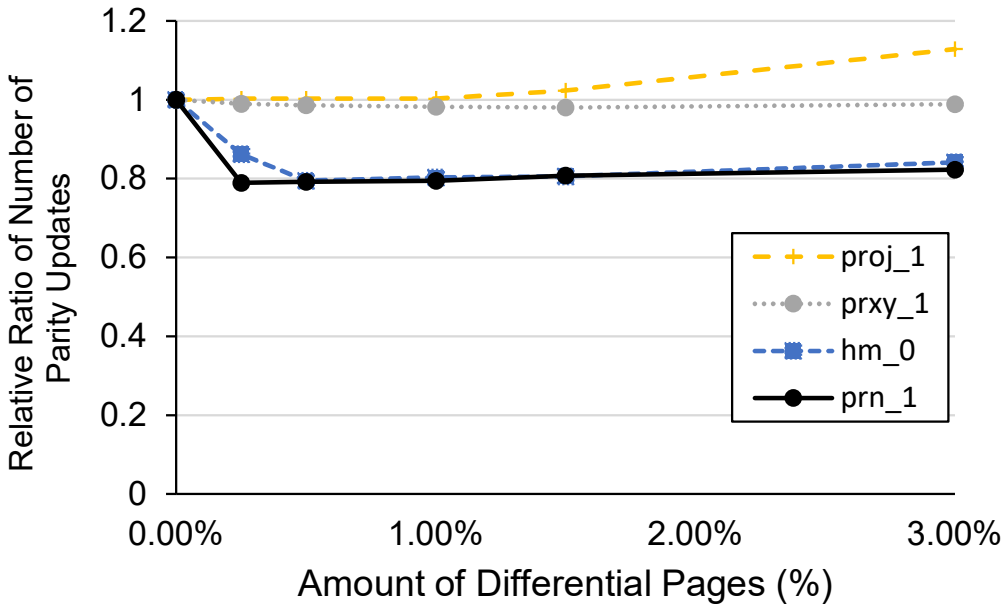



Figure 5.9: Parity Update Reduction in MSR Traces

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.

### Metadata Capacity Overhead

I evaluated the capacity overhead of the metadata used in DRC-DPU.

First, I evaluated the capacity consumption of the custom file metadata for DRC-AE and DRC-DPU with different page sizes of differential data. I calculated the capacity consumption of the custom file metadata for an EC stripe when using 2D1P and 6D2P EC. Then, I investigated the ratio of metadata to the total capacity consumption. Tables 5.3 and 5.4 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

Table 5.3: Capacity Overhead of Custom File Metadata of DRC-DPU (2D1P)

| | **User Data** | **Replica/ Parity Data** | **Custom File Metadata** | **Metadata Ratio** |
|---|---|---|---|---|
| DRC-AE | 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% |

small.

Next, I examined the impact of different page sizes on performance. In the evaluation, I examined the occurrence frequency of read–modify–write processing in the simulations using MSR trace as in the previous section. Figure 5.10 shows the evaluation result of the occurrence frequency of read–modify–write processing.

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.

## 5.4 Related Work

There has been extensive research on write buffers using volatile memory [GKDB09, Pet09]. 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

Table 5.4: Capacity Overhead of Custom File Metadata of DRC-DPU (6D2P)

| | User Data | Replica/ Parity Data | Custom File Metadata | Metadata Ratio |
|---|---|---|---|---|
| DRC-AE | 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% |

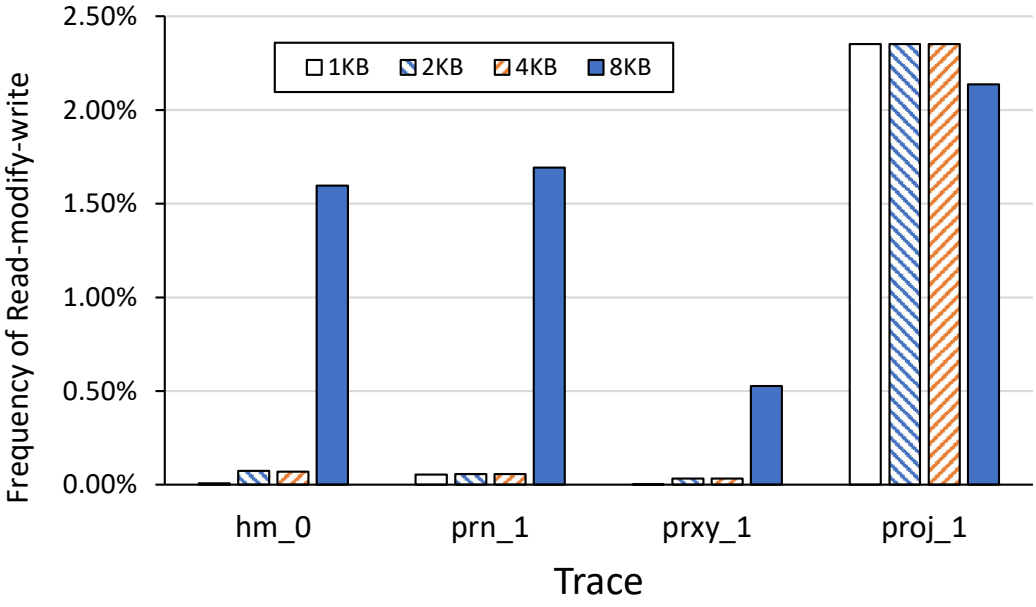



Figure 5.10: Evaluation of Number of Read-modify-write Processes

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 [UHS17]. 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 [NSI17, MHS20, FLY19]. Network-protocol-specific metadata is another application of file metadata extension to achieve higher protocol coverage of network protocol servers [FSN18]. 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.

In Chapter 4, I proposed the adaptive encoding, which dynamically determines the amount of EC data according to performance requirements, as a part of DRC-AE. DRC-DPU described in this chapter can be used in conjunction with the adaptive encoding. DRC-DPU and the adaptive encoding improve the write response time and operational management, respectively.

## 5.5 Chapter Summary

This chapter proposed DRC-DPU to improve the degraded write response time of DRC-AE 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, I 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.

# Chapter 6

# Discussions

This chapter discusses the overall effectiveness of combining all the proposed methods in this doctoral dissertation. I also identify the strengths and weaknesses of the server-based storage after applying the proposed methods against the storage appliances.

## 6.1 Overall Evaluations

This section evaluates the effectiveness of the proposed methods in this doctoral dissertation in terms of performance, functionality, and reliability.

### 6.1.1 Performance

I estimated the performance improvements of the proposed methods in six major network storage applications [spe17]. I estimated the performance improvements for each workload of the major applications using the results of performance experiments in Chapter 4. I used the operation ratio of each workload and the performance improvement rate for each operation to calculate the overall performance improvements of the workloads. Table 6.1 shows the evaluation results.

The evaluation results show that the proposed methods improve the performance in five out of six applications with a maximum of 4.4 times in the software build. Even with the video data acquisition, the only workload that showed a performance decrease, the performance decrease remains less than 5%.

These results indicate that the proposed methods have significantly improved the performance of server-based storage in the major applications of network storage.

| Workload | Operation | Operation Ratio | Per-operation Relative Performance | Relative Performance to Conventional Method |
|---|---|---|---|---|
| Database | Random Read | 66% | 1.64 | **2.38** |
| | Random Write | 20% | 2.92 | |
| | Sequential Read | 1% | 4.4 | |
| | Sequential Write | 0% | 0.86 | |
| | Metadata Read | 13% | 5.1 | |
| | Metadata Write | 0% | 1.1 | |
| Software Build | Random Read | 0% | 1.64 | **4.40** |
| | Random Write | 0% | 2.92 | |
| | Sequential Read | 6% | 4.4 | |
| | Sequential Write | 7% | 0.86 | |
| | Metadata Read | 78% | 5.1 | |
| | Metadata Write | 9% | 1.1 | |
| Electronic Design Automation | Random Read | 0% | 1.64 | **3.58** |
| | Random Write | 10% | 2.92 | |
| | Sequential Read | 21% | 4.4 | |
| | Sequential Write | 23% | 0.86 | |
| | Metadata Read | 41% | 5.1 | |
| | Metadata Write | 4% | 1.1 | |
| File Sharing | Random Read | 10% | 1.64 | **4.12** |
| | Random Write | 18% | 2.92 | |
| | Sequential Read | 0% | 4.4 | |
| | Sequential Write | 0% | 0.86 | |
| | Metadata Read | 66% | 5.1 | |
| | Metadata Write | 6% | 1.1 | |
| Virtual Desktop Infrastructure | Random Read | 20% | 1.64 | **2.59** |
| | Random Write | 64% | 2.92 | |
| | Sequential Read | 6% | 4.4 | |
| | Sequential Write | 9% | 0.86 | |
| | Metadata Read | 1% | 5.1 | |
| | Metadata Write | 0% | 1.1 | |
| Video Data Acquisition | Random Read | 9% | 1.64 | **0.97** |
| | Random Write | 0% | 2.92 | |
| | Sequential Read | 1% | 4.4 | |
| | Sequential Write | 90% | 0.86 | |
| | Metadata Read | 1% | 5.1 | |
| | Metadata Write | 0% | 1.1 | |

Table 6.1: Estimations on Performance Improvements of Proposed Methods

### 6.1.2 Functionality

I evaluated the effectiveness of the proposed methods in improving the functionality of server-based storage for six major applications of network storage. I compared the functional sufficiency of the network protocols required for each application between server-based storage with and without the proposed methods. Table 6.2 shows the results of the evaluation.

The evaluation results show that with the proposed methods, server-based storage meets the functional requirements of all six major applications. The proposed methods increase the functional coverage of the network protocols as described in Chapter 3. With this improved functional coverage, server-based storage meets the requirements for the file sharing workload.

These results show that server-based storage now meets the requirements of major applica-

tions by improving the functionality of the proposed method.

### 6.1.3 Reliablity

The proposed methods achieve high reliability by replication and EC as well as conventional server-based storage. In terms of reliability, the proposed method provides the same level of reliability as the conventional server-based storage. In addition, the capacity consumption associated with data redundancy for high reliability is also the same as before, as described in Chapter 4.

## 6.2 Comparison between Server-based Storage and Storage Appliance

I discuss the comparison between the server-based storage after applying the proposed methods in comparison with the storage appliances. I conducted a qualitative assessment of the strengths and weaknesses of server-based storage and the storage appliance in terms of performance, functionality, and reliability. Table 6.3 shows the evaluation results.

**Performance** The performance advantages of server-based storage are scalability and the use of the latest hardware such as NVMe SSDs, SCM, and 100GbE NICs. The proposed

| *Workloads* | *Requirements* | *Without Proposed Methods* | *With Proposed Methods* |
|---|---|---|---|
| Database | POSIX Support | OK | OK |
| Software Build | POSIX Support | OK | OK |
| Electronic Design Automation | POSIX Support | OK | OK |
| File Sharing | POSIX Support, High Protocol Functional Coverage | Not OK (Coverage: 78%) | OK (Coverage: 97%) |
| Virtual Desktop Infrastructure | POSIX Support | OK | OK |
| Video Data Acquisition | POSIX Support | OK | OK |

Table 6.2: Evaluation of Functional Sufficiency

methods enable local access to the internal disks and take advantage of the hardware performance as shown in Chapter 4. The combination of scalability and modern hardware provides server-based storage advantage over the storage appliance in read performance. On the other hand, one of the weaknesses of server-based storage is the overhead of network-based data redundancy among servers in random write performance. Furthermore, the storage appliance achieves high random write performance by using dedicated hardware such as NVRAM and high-speed interconnect. Because of these differences in architecture and hardware components, the random write performance of server-based storage becomes the weakness against the storage appliances even with the proposed methods.

**Function** The functional weakness of server-based storage is its less functionality in the file system layer. Server-based storage requires inter-server capabilities to provide file system layer functionality. Conventional distributed filesystems and proposed methods provide limited functionality in comparison with the storage appliance which provides functions such as cloning and deduplication. Therefore, server-based storage has a functional disadvantage over the storage appliances.

For server-based storage, lack of cloning, deduplication, and other features can lead to

| | *Strength* | *Weakness* |
|---|---|---|
| Performance | • Use of latest hardware (High-performance disks and broadband NICs)<br>• Scalability with the number of servers | • Write overhead in Inter-server Redundancy<br>• Lack of specialized hardware (High-bandwidth Interconnects, NVRAM) |
| Function | | •Less functions in file system layer (Clone, Deduplication etc.) |
| Reliability | • Large-stripe erasure coding | •Unstable performance under failure<br>•Unstable network-based architecture |

Table 6.3: Strength and Weakness of Server-based Storage Comapring to Storage Appliance

increased capacity and higher operation and management costs. However, since server-based storage has a much lower hardware cost than the storage appliances, the increase in cost due to lack of functionality should not be a major problem. I believe that the lack of functionality of server-based storage is a challenge for the future, but not a significant disadvantage to the storage appliances.

**Reliablity** The strength of server-based storage in terms of reliability is large-stripe EC. In server-based storage, data reliability can be arbitrarily increased by forming large-stripe EC among many servers such as the 100D4P EC configuration. The large-stripe EC enable high reliability with higher capacity efficiency. On the other hand, the weaknesses of server-based storage are the unstable performance in case of failures and the unstable network-based architecture. In server-based storage, when a failure occurs, a large amount of data transfer occurs between servers for data rebuild. Since server-based storage shares the network between the rebuild process and data access from applications, the rebuild process affects the data access performance. In addition, server-based storage consists of distributed storage over a network, which is more complex in architecture than the traditional standalone storage appliance. In contrast to the storage appliance, server-based storage, with its complex architecture and immaturity, is relatively unstable.

**Overall** Server-based storage has an advantage over the storage appliance in read performance, performance scalability, and reliablity in a system with many servers, while it has disadvantages in random write performance, functionality in the filesystem layer, and performance stability. As for random write performance, the proposed methods enable server-based storage to provide much improved performance as seen in Chapter 4. We believe that the improved performance enables the server-based storage to cover a wide range of applications except for the applications which require extreme random write performance. Regarding functionality in the filesystem layer, the lack of capacity reduction functions for server-based storage increases costs. However, since server-based storage itself is inexpensive, I believe the cost increase is acceptable. On the other hand, in terms of reliability, current server-based storage has the problem of unstable performance due to the rebuild processes in case of failure and unstable networks.

I believe that, server-based storage can replace the storage appliances for applications such as best-effort type online services which do not require strict service level guarantees, and large-scale systems with low-latency requirements. On the other hand, the storage appliances,

which provide stable high-performance services, are still superior in terms of stabilitity for infrastructure systems.

# Chapter 7

# Conclusion

This doctoral dissertation proposes methods that utilize the custom file metadata to achieve high-functionality and high-performance in server-based storage.

There is a trend of replacing expensive storage appliances with inexpensive server-based storage. However, the target applications of server-based storage are limited by the poor functionality and performance of server-based storage. Improvements in the functionality and performance of server-based storage have been required to expand its target applications.

The purpose of this doctoral dissertation is to improve the functionality and performance of server-based storage. I use the three approaches described below to achieve the purpose. These approaches use the custom file metadata to offer the extended file metadata capabilities on the commodity OS to achieve higher functionality and performance.

In Chapter 3, this doctoral dissertation proposes PMM that enables a portable protocol stack using the custom file metadata. I clarify the limitations of the commodity OS running on the commodity servers, and investigate the custom file metadata to extend the functionality of the commodity OS. Then, I develop PMM, which is a software module for using the custom file metadata on the commodity OS. PMM consists of user-space metadata management and journal management. PMM allows the protocol stack of the storage appliance to manage the protocol-specific metadata by using the custom file metadata on commodity servers. PMM improves the functional coverage of network protocols for server-based storage from 78% to 97%.

In Chapter 4, this doctoral dissertation proposes DRC-AE, which is a redundancy control module using the custom file metadata. DRC-AE improves random access performance by placing an application and its data on the same server by making local data redundant between servers. DRC-AE uses the custom file metadata to manage the metadata of the redundancy

configuration between servers. DRC-AE stores the custom file metadata locally and avoids the bottleneck of the metadata server in conventional methods. In addition, DRC-AE dynamically switches the redundancy method of user data according to performance requirements and workloads. DRC-AE changes the amount of EC data to achieve high capacity efficiency while maintaining high performance. DRC-AE improves the random performance up to 2.92 times over the conventional method.

In Chapter 5, this doctoral dissertation proposes DRC-DPU that uses the custom file metadata to enable delayed update of parity data. DRC-DPU enables asynchronous parity updates by redirecting write requests to EC data to replicated differential data. DRC-DPU uses the custom file metadata to manage the metadata of the differential data. DRC-DPU makes the response time of writes to EC data at the same level as replication. DRC-DPU avoids the degradation of the write response time of EC data in DRC-AE. DRC-DPU improved the 99% percentile response time to one-eighth of the conventional method.

This doctoral dissertation confirms that the proposed methods enable server-based storage to significantly improve performance while meeting the functional and reliablity requirements in the major applications of network storage.

The works in the dissertation contribute directly for designing server-based storage systems that support a wider range of target applications for large-scale IT systems.

## 7.1 Open Issues

There are several directions for extending this work.

**Application-level Load-balancing between Servers** DRC-AE improves the random performance of server-based storage by placing an application and its data on the same server. This approach assigns data access and capacity of an application to a specific server. To achieve scalability with respect to the number of servers, it is necessary to balance the load and capacity usage of applications among servers. Establishing the optimal placement control of applications to balance the load and capacity among servers will be a future challenge.

**Stabilize Performance under Server Failures** The proposed methods require the rebuilding process of EC data when a server failure occurs. The impact of the rebuild process on request processing destabilizes data access performance. DRC-AE and DRC-DPU, which achieve high-performance with local access under normal conditions, experience large performance

degradation when a failure occurs. Establishing performance control methods to avoid performance degradation when failures occur will be a future challenge.

**Maturity of distributed systems** To further expand the target applications of server-based storage, it is essential to mature distributed system control. Network-based distributed systems are more complex than the storage appliance, and the software needs to be mature to provide stable services. In addition, the server-based storage has a short history, whereas the storage appliance has been in use for decades. The future challenge is to operate server-based storage and accumulate know-how in solving problems and troubleshooting.

# Acknowledgments

This dissertation would never have been possible without help and support of many people.

First and foremost, I would like deeply thank my advisor Prof. Haruo Yokota for his guidance, understanding, patience, and continuing support and encouragement during my doctoral studies at Tokyo Institute of Technology.

I also would like thank Asst. Prof. Hieu Hanh LE of Tokyo Institute of Technology for his invaluable advices throughout my doctoral studies.

I would like to express my appreciation to the other thesis committee members for inspecting and valuable comments to improve the quality of this thesis, Prof. Jun Miyazaki, Prof. Takuo Watanabe, Prof. Kenji Kise, Assoc. Prof. Haruhiko Kaneko.

I would also like to thank all my colleagues (past and present) of Yokota Lab.: Dr. Satoshi Hikida, Dr. Takamitsu Shioi, Dr. Tian Hao, Dr. Nesrine Berjab and other young members. I enjoyed discussing with them at weekly seminars, summer workshops, and conferences. It was a good opportunity for me to keep a distance from my full-time job and to touch new research ideas. I am also grateful to Secretary Harumi Miura for her support and kindness during my time at Yokota Lab.

I would like to express my appreciation to my superiors and colleagues of Hitachi. Special thanks go to, Mr. Kenta Shiga, Dr. Norio Shimozono, and Dr. Mitsuo Hayasaka, my past and present superiors, for permitting me to work toward a doctorate; Mr. Atsushi Sutoh and Mr. Takahiro Nakano, my past superiors and the best research partners in Hitachi, for their well-grounded discussions and their considerable support for improbing my study and presentations.

I dedicate this dissertation to my better half, Ayako; my precious boys Rikuto and Kota for their endless support and encouragement. Last but not least, I could not express my deep appreciation for Ayako. Without her love and support, I could not complete this works.

# Bibliography


[AAG+16] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *11th European Conference on Computer Systems (Eurosys 2016)*, pages 1–17, 2016.

[AHB+99] Bridget Allison, Robert Hawley, Andrea Borr, Mark Muhlestein, and David Hitz. File system security: Secure network data sharing for nt and unix. *Network Appliance, Inc. Tech Library*, page 16, 1999.

[Apa21] Apache.org. Welcome to apache zookeeper, 2021. visited on 2021-05-29.

[AWK+19] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*, pages 353–369. ACM, 2019.

[BBP12] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Lab.(LANL), 2012.

[BC05] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc., 2005.

[BCX+12] Patricia V Beserra, Alessandro Camara, Rafael Ximenes, Adriano B Albuquerque, and Nabor C Mendonça. Cloudstep: A step-by-step decision process to support legacy application migration to the cloud. In *2012 IEEE 6th international workshop on the maintenance and evolution of service-oriented and cloud-based systems (MESOCA)*, pages 7–16. IEEE, 2012.

[BHH+04] Mark Bakke, J Hafner, J Hufferd, K Voruganti, and M Krueger. Internet small computer systems interface (iscsi) naming and discovery. *The Internet Society*, 2004.

[BHRM18] Luiz A Barroso, Urs Holzle, Parthasarathy Ranganathan, and Margaret Martonosi. *The datacenter as a computer: designing warehouse-scale machines*. Morgan & Claypool Publishers, 3rd edition, 2018.

[BKL+16] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*, pages 83–98, 2016.

[Bor98] Andrea J Borr. Secureshare: safe unix/windows file sharing through multiprotocol locking. In *2 nd USENIX Windows NT Symposium*, pages 3–4, 1998.

[Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[BSR+09] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Kaladhar Voruganti, and Garth R Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX Annual technical conference*. San Diego, CA, 2009.

[BWB+04] Geoffrey S Barrall, Trevor Willis, Simon Benham, Michael Cooper, Jonathan Meyer, Christopher J Aston, and John Winfield. Apparatus and method for hardware implementation or acceleration of operating system functions, November 30 2004. US Patent 6,826,615.

[CAA+17] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-managing storage for enterprise clusters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 51–66, 2017.

[CDLC14] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. 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)*, pages 163–176, 2014.

[Cor10a] Johnathan Corbet. Open by handle, 2010. visited on 2021-05-29.

[Cor10b] Johnathan Corbet. Punching holes in files, 2010. visited on 2021-05-29.

[cor21a] Intel corporation. intel.com, 2021. visited on 2021-05-29.

[Cor21b] Microsoft Corporation. [ms-smb]: Server message block (smb) protocol, rev 48.0., 2021. visited on 2021-05-29.

[Cor21c] Microsoft Corporation. [ms-smb2]: Server message block (smb) protocol versions 2 and 3, rev 62.0., 2021. visited on 2021-05-29.

[Cor21d] Microsoft Corporation. Windows 10 os, 2021. visited on 2021-05-29.

[CSGK11] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2015)*, pages 43–56, 2011.

[CYS+14] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. Software defined storage. *Storage Networking Industry Assoc. working draft*, pages 20–24, 2014.

[DB13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[DLL07] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. Ganesha, a multi-usage with large cache nfsv4 server. In *Linux Symposium*, volume 113, 2007.

[DVS12] Scott Dawkins, Kaladhar Voruganti, and John D Strunk. Systems research and innovation in data ontap. *ACM SIGOPS Operating Systems Review*, 46(3):1–3, 2012.

[Fik10] Andrew Fikes. Storage architecture and challenges. *Talk at the Google Faculty Summit*, 2010.

[fio21] Flexible io tester, 2021.

[FLY19] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. Lightweight dynamic redundancy control for server-based storage. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS 2019)*, pages 353–369. IEEE, 2019.

[FLY21a] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. Delayed parity update for bridging the gap between replication and erasure coding in server-based storage. In *the twelfth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS2021) in conjunction with the 47th International Conference on Very Large Data Bases (VLDB 2021)(to be appeared)*, 2021.

[FLY21b] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. Lightweight dynamic redundancy control with adaptive encoding for server-based storage. *ACM Transactions on Storage (TOS)(to be appeared)*, 2021.

[For95] Internet Engineering Task Force. Nfs version 3 protocol specification, 1995. visited on 2021-05-29.

[FSN18] Takayuki Fukatani, Atsushi Sutoh, and Takahiro Nakano. 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):23–42, 2018.

[FTXG11] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *Technical Report Technical Report CMU-PDL-11-112, Carnegie Mellon Univsersity, Parallel Data Laboratory*, 2011.

[Gib10] Garth Gibson. Diskreduce v2.0 for hdfs, 2010. visited on 2021-05-29.

[GKDB09] Binny S Gill, Michael Ko, Biplob Debnath, and Wendy Belluomini. Stow: A spatially and temporally optimized write caching algorithm. In *2009 USENIX Annual Technical Conference (USENIX ATC '09)*, pages 29–42, 2009.

[HLM94] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[HN15] Tom Haynes and David Noveck. Network file system (nfs) version 4 protocol. 2015.

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

[IG18] IEEE and The Open Group. The open group base specifications issue 7, 2018 edition (ieee std 1003.1™-2017), 2018.

[Inc21] Dell Inc. Dell technologies - enterprise data storage: Cloud, nas & flash, 2021. visited on 2021-05-29.

[ISI21] DELL EMC ISILON. Dell emc isilon onefs: A technical overview, 2021.

[IYZ+19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[KBP+12] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 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)*, page 20, 2012.

[KC16] Mohammed Rafi KC. Efficient data tiering in glusterfs. In *Storage Developer Conference (SDC) 2016, SNIA*, 2016.

[ker21] kernel.org. The linux kernel archives, 2021. visited on 2021-05-29.

[KGB10] Aneesh Kumar KV, Andreas Grünbacher, and Greg Banks. Implementing an advanced access control model on linux. In *Linux Symposium*, page 19. Citeseer, 2010.

[Kim09] Won Kim. Cloud computing: Today and tomorrow. *J. Object Technol.*, 8(1):65–72, 2009.

[KZK+17] Sungjoon Koh, Jie Zhang, Miryeong Kwon, Jungyeon Yoon, David Donofrio, Nam Sung Kim, and Myoungsoo Jung. Understanding system characteristics of

online erasure coding on scalable, distributed and large-scale ssd array systems. In *2017 IEEE International Symposium on Workload Characterization (IISWC 2017)*, pages 76–86. IEEE, 2017.

[LHL17] Runhui Li, Yuchong Hu, and Patrick PC Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Transactions on parallel and distributed systems*, 28(9):2500–2513, 2017.

[lib21] libfuse. libfuse, 2021. visited on 2021-05-29.

[LJH+17] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. Understanding write behaviors of storage backends in ceph object store. In *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology (MSST 2017)*, volume 10, 2017.

[LKM+17] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR 2017)*, pages 1–11. ACM, 2017.

[LLS+15] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. {WALDIO}: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 235–247, 2015.

[LPGM08] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *2008 USENIX Annual Technical Conference (USENIX ATC '08)*, volume 1, pages 5–2, 2008.

[Mat12] Steve Mattos. Server-based storage. In *Flash Memory Summit 2012*, 2012.

[MB12] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4), 2012.

[MCB+07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[MHS20] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. Practical quick file server migration. *ACM Transactions on Storage (TOS)*, 16(2):1–30, 2020.

[Mic20] Microsoft. Microsoft azure lsv2-series, 2020. visited on 2021-05-29.

[MMP+12] Harsha V Madhyastha, John McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C Snoeren, and Amin Vahdat. scc: cluster storage provisioning informed by application characteristics and slas. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*, 2012.

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

[Nar16] Srinivasan Narayanamurthy. Modern erasure codes for distributed storage systems. In *Storage Developer Conference (SDC) 2016, SNIA*, 2016.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[Net21] NetApp. Netapp - data management solutions for the cloud, 2021. visited on 2021-05-29.

[NSI17] Jun Nemoto, Atsushi Sutoh, and Masaaki Iwasaki. Directory-aware file system backup to object storage for fast on-demand restore. *International Journal of Smart Computing and Artificial Intelligence (IJSCAI)*, 1(1):1–19, 2017.

[NXX18] Junpeng Niu, Jun Xu, and Lihua Xie. Hybrid storage systems: A survey of architectures and algorithms. *IEEE Access*, 6:13385–13406, 2018.

[Pas16] Tudor Pascu. Ontap select product architecture and best practices. Technical report, NetApp Technical Report 4517, NetApp, 2016.

[PBM+17] Rogério Pontes, Dorian Burihabwa, Francisco Maia, Joao Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. Safefs: a modular architecture for secure user-space file systems: one fuse to rule them all. In *Proceed-*

*ings of the 10th ACM International Systems and Storage Conference (SYSTOR 2017)*, page 9. ACM, 2017.

[PCA+14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 433–448, 2014.

[Pet09] Scott Peterson. Using persistent memory and rdma for ceph client write-back caching. In *USENIX Annual Technical Conference*, pages 29–42, 2009.

[Pla13] James S Plank. Erasure codes for storage systems: A brief primer. *Login: The USENIX Magzine*, 38(6):44–50, 2013.

[PSS07] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications. *Technical Report CS-07–603, University of Tennessee*, 2007.

[RNW+15] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 81–94, 2015.

[RSI12] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows® Internals (6th Edition)*. Microsoft Press, 2012.

[sam21] samba.org. Samba - opening windows to a wider world, 2021. visited on 2021-05-29.

[San03] Ghemawat Sanjay. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP 2003)*, pages 29–43. ACM, 2003.

[Ser20] Amazon Web Services. Amazon ec2 i3 instances, 2020. visited on 2021-05-29.

[Sha20] Shachar Sharon. Pmem file-system in user-space. In *Storage Developer Conference (SDC) EMEA 2020, SNIA*, 2020.

[spe17] spec.org. Spec sfs® 2014, 2017. visited on 2021-05-29.

[SS12] G Somasundaram and Alok Shrivastava. *Information storage and management, 2nd Edition*. Wiley Publishing, Inc., 2012.

[SWZ05] Gopalan Sivathanu, Charles P Wright, and Erez Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36, 2005.

[TCW16] Dam Quang Tuan, Seungyong Cheon, and Youjip Won. On the io characteristics of the sqlite transactions. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2016)*, pages 214–224, 2016.

[TJAP16] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *11th European Conference on Computer Systems (Eurosys 2016)*, pages 1–16, 2016.

[TZS16] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login: The USENIX Magazine*, 41(1):6–12, 2016.

[UHS17] Vikhyat Umrao, Michael Hackett, and Karan Singh. *Ceph cookbook: practical recipes to design, implement, operate, and manage Ceph storage systems*. Packt Publishing Ltd, 2017.

[Van21] Hitachi Vantara. Hitachi vantara - data storage and analytics, dataops, iot, cloud, consulting, 2021. visited on 2021-05-29.

[VMP+15] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P Kelly, and Charles B Morrey III. Failure-atomic updates of application data in a linux file system. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 203–211, 2015.

[vmw21] vmware. Vmware vsan 7 technical overview, 2021. visited on 2020-05-29.

[VRP+18] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, pages 139–154, 2018.

[VTZ17] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 59–72, 2017.

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

[WBYS99] Andy Watson, Paul Benn, Alan G Yoder, and HT Sun. Multiprotocol data access: Nfs, cifs, and http. *Technical Report TR3014, Network Appliance*, 1999.

[WDQ+12] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*, pages 1–16, 2012.

[WLXW17] Shuzhan Wei, Yongkun Li, Yinlong Xu, and Si Wu. Dsc: Dynamic stripe construction for asynchronous encoding in clustered file system. In *Proceedings of IEEE INFOCOM 2017 International Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[WW17] Jake Wires and Andrew Warfield. Mirador: an active control plane for datacenter storage. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 213–228, 2017.

[www21] www.sqlite.org. Atomic commit in sqlite, 2021.

[XFS13] XFS.org. Xfs.org, 2013. visited on 2021-05-29.

[XSBP15] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A Pease. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 213–226, 2015.

[XSG+15] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on realworld databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR 2015)*. ACM, 2015.

[XZC+20] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. Spool: Reliable virtualized nvme storage pool in public cloud infrastructure. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pages 97–110, 2020.

[YZ16] Yue Yang and Jianwen Zhu. Write skew and zipf distribution: evidence and implications. *ACM transactions on Storage (TOS)*, 12(4):21, 2016.

[ZLL+20] Yiming Zhang, Huiba Li, Shengyun Liu, Jiawei Xu, and Guangtao Xue. Pbs: An efficient erasure-coded block storage system based on speculative partial writes. *ACM Transactions on Storage (TOS)*, 16(1):1–25, 2020.

[ZT19] Tianli Zhou and Chao Tian. Fast erasure coding for data storage: a comprehensive study of the acceleration techniques. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 317–329, 2019.

[ZTH+14] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 449–464, 2014.

# Publications and Awards

## Journals

- Takayuki Fukatani, Atsushi Sutoh, Takahiro Nakano, ”A Method to Adapt Storage Protocol Stack Using Custom File Metadata to Commodity Linux Servers”, International Journal of Smart Computing and Artificial Intelligence, Vol.2, No.1, P.23 - 42, 2018

- Takayuki Fukatani, Le Hieu Hanh, Haruo Yokota, ”Proposal and Evaluation of a Redundancy Control Method Considering Update Frequency for Server-based Storage”, DBSJ Japanese Journal, Vol. 18-J, No. 15, 2020

- Takayuki Fukatani, Hieu Hanh Le, Haruo Yokota, ”Lightweight Dynamic Redundancy Control with Adaptive Encoding for Server-based Storage”, ACM Transactions on Storage (accepted, 40 pages), 2021

## Proceedings on international conference (refereed)

- Takayuki Fukatani, Atsushi Sutoh, Takayhiro Nakano, ”Adapting Storage Protocol Stack Using Custom File Metadata to Commodity Linux Servers”,in Proc. of the 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), pp. 249-254, 2017

- Takayuki Fukatani, Hieu Hanh Le, Haruo Yokota, ”Lightweight Dynamic Redundancy Control for Server-based Storage”,in Proc. of the 38th International Symposium on Reliable Distributed Systems (SRDS 2019), pp. 295-304, 2019

- Takayuki Fukatani, Hieu Hanh Le, Haruo Yokota, ”Delayed Parity Update for Bridging the Gap between Replication and Erasure Coding in Server-based Storage”, in Proc. of the twelfth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS2021) in conjunc-

tion with the 47th International Conference on Very Large Data Bases (VLDB 2021) (accepted, 9 pages), 2021

## Proceedings on domestic conference


- Takayuki Fukatani, Hieu Hanh Le, Haruo Yokota, ”Dynamic Redundancy Control Using Update Frequency for Server-based Storage”, in Proc. of the 11th Forum on Data Engineering and Information Management(DEIM), J4-2, 2019
- Takayuki Fukatani, Hieu Hanh Le, Haruo Yokota, ”Utilizing Workload Characteristics for Improving Capacity Efficiency of Server-based Storage”, in Proc. of the 12th Forum on Data Engineering and Information Management(DEIM), H1-1, 2020


## Honors and Awards

- Won the Outstanding Paper Award in 6th International Congress on Advanced Applied Informatics (AAI), 2017.