Ph.D. DISSERTATION

Optimizing Components for Data Durability on Different Software Layers

다양한 소프트웨어 계층에서 데이터 내구성을 위한 구성 요소 최적화

February 2023

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hwajung Kim

Ph.D. DISSERTATION

Optimizing Components for Data Durability on Different Software Layers

다양한 소프트웨어 계층에서 데이터 내구성을 위한
구성 요소 최적화

February 2023

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hwajung Kim

Optimizing Components for Data Durability on Different Software Layers

다양한 소프트웨어 계층에서 데이터 내구성을 위한 구성 요소 최적화

지도교수 염헌영

이 논문을 공학박사 학위논문으로 제출함

2022 년 11 월

서울대학교 대학원

컴퓨터 공학부

김화정

김화정의 공학박사 학위논문을 인준함

2022 년 11 월

Abstract

Data movement between remote systems frequently occurs for various purposes, such as backup, replication, or analysis. Data can be corrupted or lost during transmission due to hardware failures or system crashes. Many systems provide procedures for data durability, such as integrity verification or periodic backups to detect data corruption in a timely manner. These procedures either increase data processing time or require additional I/O operations.

In this dissertation, we focus on eliminating redundant operations that degrade system performance in the process of ensuring data durability; we present three optimization schemes on the storage, application, and network layers. For the storage layer, we focus on providing robust data durability on flash-based storage systems without adding delays to the entire data processing time. By involving idle cores, we parallelize the checksum computation and overlap it with I/O operations to mitigate the overhead caused by I/O reordering for robust data durability. On the application layer, we present an efficient backup and recovery scheme by exploiting write-ahead logging (WAL) to provide data durability for database systems. We substantially eliminate additional I/O operations by keeping log data for backup and recovery. Archiving log data, on the other hand, degrades the system performance because the amount of log data to manage grows, and logging operations on the critical path slow down data processing times. To address these limitations, we present an in-transit logging scheme that logs important data by inspecting packets on the network layer at the destination system. The proposed scheme guarantees fault-tolerance by delivering the original requests through local clients on the target system. With the proposed schemes, it is possible to provide data durability on different software layers without adding delays to data processing times or redundant I/O operations.

We have implemented and evaluated our schemes on real multi-core systems to show the effectiveness of our approach. Using machines equipped with high-performance storage devices connected via 10 Gbps, we evaluated and compared our schemes to existing schemes. Experimental results demonstrate that the proposed schemes efficiently provide data durability by leveraging idle resources and ready-to-use data on the storage, application, and network layers while providing better system performance.

Keywords: Storage Systems, Database Systems, Data Durability, Integrity Verification, Write-ahead-Logging, Packet Inspection

Student Number: 2018-36618

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The amount of data transferred to remote systems grows exponentially as the amount of data grows. Previous studies predict that global data center traffic will reach around 250 ZB by 2030 [1, 2]. Data is transferred to remote storage systems for various purposes, such as backup, replication, or analysis. During this process, data can be corrupted or lost due to various reasons. According to [3], the reasons for data corruption or loss are as follows: hardware failures occurred inside the storage device, software failures including program bugs, system crashes, or human errors.

Storage and database systems provide procedures for ensuring data durability in order to prevent data corruption or loss as a result of such malfunctions. When detecting data corruption caused by network failures during data transfer, the receiver node verifies data integrity by comparing the checksum of data. Storage engines in database systems usually perform periodic backups or logging operations to protect the data from failures. However, these procedures increase the entire data transfer time between remote systems, require additional I/O operations, and introduce delays on the critical path; as a result, these procedures slow down the whole system.

1.2 Overview

In this dissertation, we present three optimization schemes to provide data durability on the storage, application, and network layers, without adding delays to data processing times or additional I/O operations. For the storage layer, we first analyze the bottlenecks in the entire transfer procedure including integrity verification between remote systems. We also investigate the internal operations of flash-based storage devices in order to provide robust data durability. We present optimization scheme to provide robust and efficient data durability for the storage layer by reordering and parallelizing each step without increasing the entire data transfer time. On the application layer, we investigate existing database backup and recovery techniques and present an efficient backup and recovery scheme to provide data durability for database systems by leveraging ready-to-use data, write-ahead log (WAL). Although archiving log data eliminates redundant I/O operations for backup and recovery, it incurs management overhead, and logging operations on the critical path degrade system performance. In order to overcome the overhead of preserving log data, we present an in-transit logging scheme that provides data durability on the network layer by inspecting original requests from remote clients. We allocate separate cores to inspect packets while avoiding interference with normal operations and successfully moving logging operations out of the critical path.

We have implemented and evaluated our schemes on real multi-core systems using realistic scenarios on different software layers. Experimental results demonstrate that the proposed scheme provides robust data durability by overlapping and parallelizing steps for the storage layer by employing idle computing resources. The proposed schemes also efficiently provide data durability on the application and network layers without compromising system performance by utilizing ready-to-use data, such as write-ahead log (WAL) and incoming packets.

1.3 Contributions

The contributions of this dissertation can be summarized as follows:

- For optimizing procedures of providing data durability for the storage layer, we introduce a concurrent and robust end-to-end integrity verification scheme considering the internal operation of the storage devices for data transfer between remote systems with flash-based storage devices. To perform data integrity verification including possible data corruptions that occurred inside the storage devices, we control the order of I/O operations considering the internal operations of the storage devices. We also parallelize checksum computation and overlap it with I/O operations to mitigate the overhead caused by I/O reordering.

- For optimizing procedures of providing data durability on the application layer, we introduce an efficient backup and recovery scheme by exploiting write-ahead logging (WAL) in database systems. For backup, we devise a backup system that uses log data generated by the existing storage engine to eliminate the additional I/O operations. Our scheme restores a backup by leveraging and optimizing the existing crash recovery procedure of storage engine to reduce recovery time. For example, we divide the recovery range and apply backup data for each range independently via multiple threads.

- For optimizing procedures of providing data durability on the network layer, we introduce an in-transit logging scheme that logs the important data in the network at the destination system as soon as the data arrives before being processed. We filter packets whose destination is the target system, such as database servers, and store the data that need to be logged by extracting data from the packet's payload. For recovery, we use recorded original request packets to mimic clients and restore the database by making the server believe that clients are sending normal requests.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 first presents background information on the existing approaches to guarantee data durability in storage and database systems. We discuss related work in Chapter 3. Chapter 4 presents a concurrent and robust end-to-end data integrity verification scheme for flash-based storage devices to efficiently provide data durability for the storage layer. Chapter 5 presents an efficient database backup and recovery scheme using write-ahead logging (WAL) to efficiently provide data durability on the application layer. Chapter 6 presents an in-transit logging scheme that performs logging operations by extracting data from original requests to efficiently provide data durability on the network layer. Chapter 7 discusses how to provide data durability for storage and database systems, and some limitations of the proposed schemes. Finally, we conclude and discuss potential future research directions in Chapter 8.

The technical material is adapted from existing published or under review.

Chapters 2 and 3 have some material adapted from [4–6]. Chapter 4 is adapted from [4], chapter 5 is adapted from [5], and chapter 6 is adapted from [6].

Chapter 2

Background

2.1 The Internal Structure and Operations of Flash-based SSD

The internal structure of flash-based SSD is illustrated in Figure 2.1. As illustrated in the figure, flash-based SSD has a certain amount of DRAM buffer, called buffer cache, to cache data. Flash-based SSDs cache data in the buffer cache to increase throughput and improve NAND flash endurance [7–9].

When the host system requests writing data to the storage device, the SSD controller first caches the data in the buffer cache inside SSD. The primary role of the buffer cache is caching data to be written to or read from NAND flash memory. The reason for caching data before writing to NAND flash memory is the out-of-place update nature of flash memory. When the data stored in NAND flash memory are modified, the new data are written to another location. The old data are invalidated, and the block containing the data is a candidate for garbage collection that causes performance degradation of SSD [10–15].

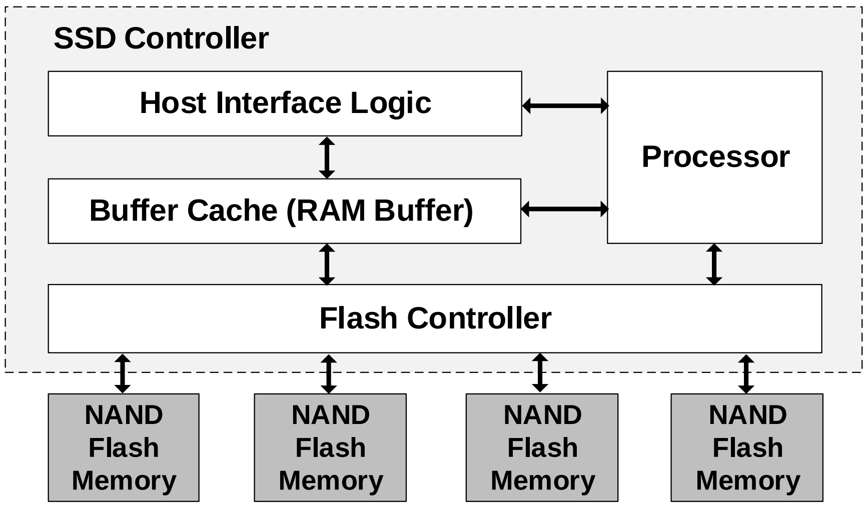


Figure 2.1: Internal structure of a flash-based SSD

In addition, repetitive data writes shorten the lifetime of SSD because the endurance of flash memory is limited [14–18]. Therefore, the data is cached in the buffer cache to prevent the modified data from being repeatedly written to NAND flash memory in a short time.

Data can become corrupted while moving from the buffer cache to NAND flash memory [19–21]. The internal failures of SSDs are relatively common, and the failure rates of various SSDs range from 4.2% to 34.1% [19]. The causes of data corruption inside SSD are metadata corruption (i.e., FTL metadata mapping disruption), shorn writes (i.e., incomplete writes), dropped writes (i.e., data cached in the buffer cache are not written to NAND flash memory), misdirected writes (i.e., writes in the wrong location), and so on [20,21]. Therefore, data transfer including verification should consider the internal structure and operations of SSD.

In SSDs, there exist error correction code (ECC) which can detect and correct errors in SSDs. However, it is not sufficient for end-to-end integrity verification. Because ECC can check only internal errors in SSD, it is not able to detect errors of a transmitted file on the path to buffer cache in SSD. Therefore, a new end-to-end integrity verification scheme other than ECC is necessary.

2.2 Data Integrity Verification Procedure

The existing implementation of data transfer including verification is shown in Figure 2.2. A sender system first reads a file from the storage device and sends it over the network to a receiver system. After the file is transferred, the sender computes the checksum of the file and transfers it to the receiver. The receiver system writes the file to the storage device and computes the checksum of the file. Then, the receiver compares the checksum value with the value of the sender to detect data corruption in the entire procedure. If the checksum values match, the data transfer proceeds to the next files. If the checksum values are different, the receiver requests retransmission of the corrupted file to the sender.

To perform integrity verification on the receiver system, the receiver reads a file from the storage device to compute the checksum of the file. However, without explicit eviction of the page cache of host memory, the verification is performed with the cached data, not the data written to the storage device. In other words, it is impossible to detect data corruption caused by errors occurring while writing files from host memory to the storage device or the aforementioned internal failures of the storage device.

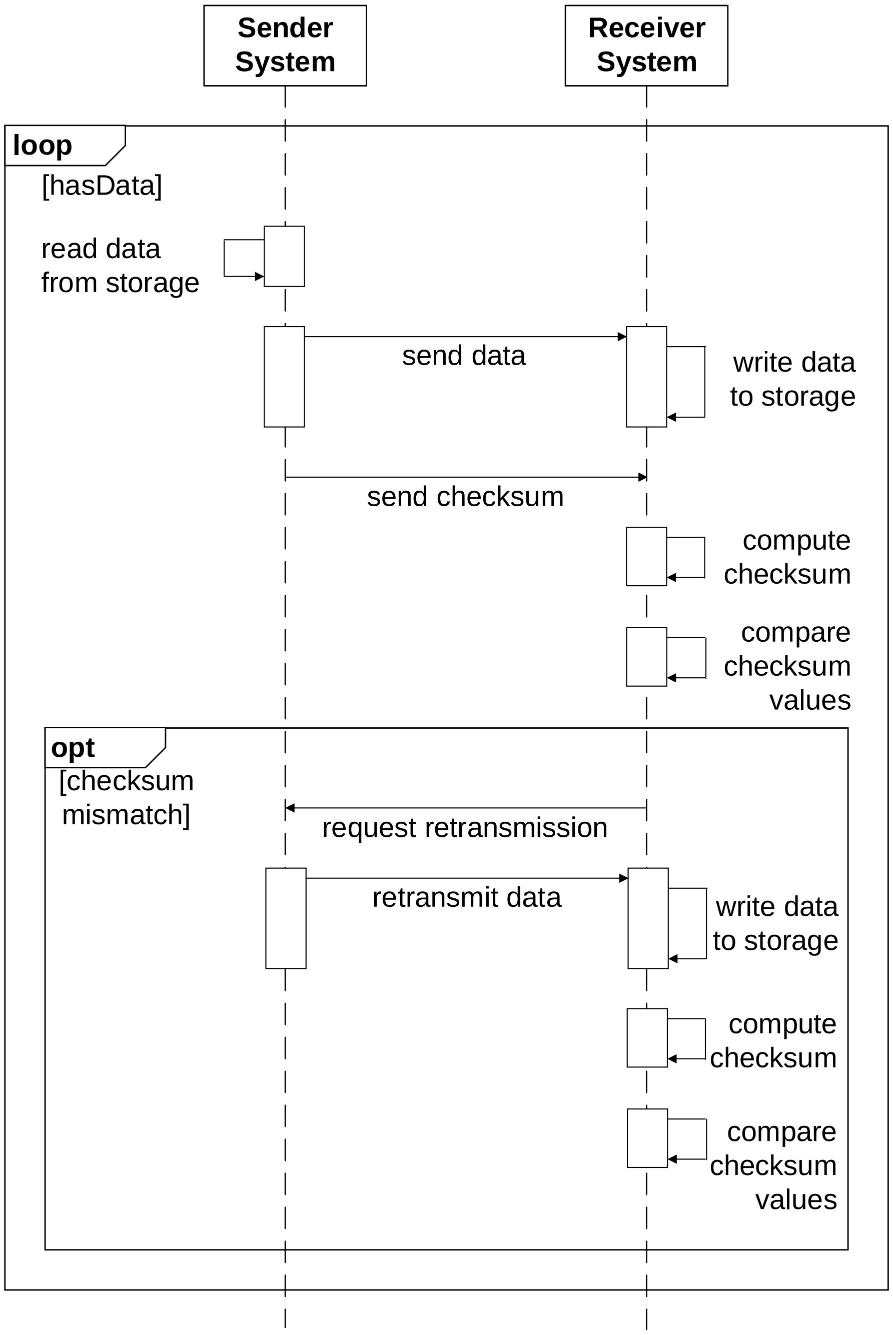


Figure 2.2: Data transfer procedure including verification

2.3 Existing Database Backup and Recovery Schemes

There are two major mechanisms to provide database backup and recovery, physical backup and logical backup. Physical backups use raw data for backup and recovery whereas logical backups use the data in a form of query statements. Both mechanisms are provided at the application layer as in the proposed scheme. In this section, we will explain how these mechanisms perform backup and recovery.

2.3.1 Physical Backup and Recovery

Xtrabackup of Percona [22] is one of the most popular applications that provide physical backup for MySQL database. To provide full backup, Xtrabackup copies all the files of the data directory of the database to the designated backup directory. Similarly, to restore a full backup, Xtrabackup copies back files in the backup directory to the data directory of the database. For the incremental backup, Xtrabackup compares the log sequence number (LSN) of each data page between the current database and the previous backup. If the page LSN of the current database differs from the previous backup, Xtrabackup writes the page to the delta file.

2.3.2 Logical Backup and Recovery

On the other hand, mysqldump [23], one of the tools provided as a utility for MySQL, provides the logical backup. To perform the full backup with mysqldump, it first scans all the tables in the database, and generate query statements that can create tables and insert data. All the query statements are stored in a single SQL file. Mysqldump interprets and executes all the query statements in the backup file to restore the full backup. Meanwhile, mysqldump uses binary log (binlog) [24] to perform the incremental backup. The binary log contains a collection of the events that change the database such as table creation and data insertion. Therefore, using binlog, mysqldump does not require database scan or data extraction to generate query statements for incremental backup. Instead, mysqldump extracts query sequences from binlog between the previous and current backup time to generate query statements for the incremental backup. However, every execution of a query statement requires the extra write of query history to a separate binlog file.

2.4 Packet Inspection

The extended Berkeley Packet Filter (eBPF) [25] originated from BPF [26] that provides functionalities of capturing and filtering network packets. The BPF is evolved to eBPF to provide more extensive functionalities by introducing BPF map data structure and supporting 64-bit registers. The eBPF detects specific kernel events (e.g., packet arrival at the network driver) and triggers a user-defined function to be executed. With eBPF, we can easily inspect and filter network packets of specific target systems.

When an application sends data through the network, data is encapsulated as it goes through the network stack of the kernel. Figure 2.3 illustrates the brief packet format of the TCP/IP protocol stack. The packet arriving at the network driver on the host system is encapsulated by TCP, IP, and Ethernet layers as shown in the figure. That is, in order to extract the application data, it is necessary to go through the process of parsing the header information of each layer.

Let's suppose that we want to extract application data from the request packets at the server system, using TCP/IP protocol stack. This requires us to

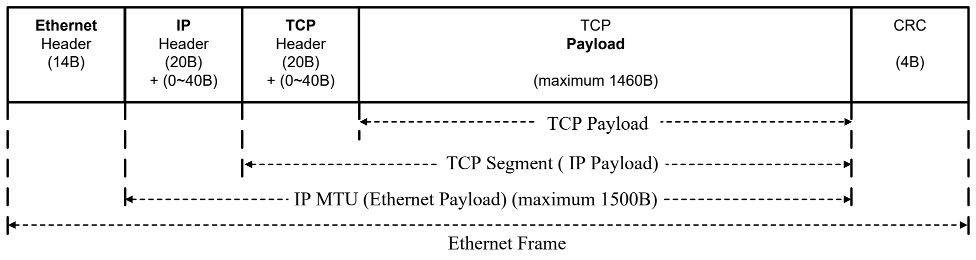

Ethernet Header (14B)
IP Header (20B) + (0~40B)
TCP Header (20B) + (0~40B)
TCP Payload (maximum 1460B)
CRC (4B)
TCP Payload
TCP Segment (IP Payload)
IP MTU (Ethernet Payload) (maximum 1500B)
Ethernet Frame

Figure 2.3: Ethernet Packet Format

inspect the header fields of each layer in order. The first layer is ethernet frame, which contains the internet layer protocol information. For packets of TCP/IP protocol stack, the internet layer protocol should be IPv4. After it is confirmed that the internet layer protocol is IPv4, then we check the IP layer header. In inspecting IP protocol header, we first check the destination IP address matches the server system's IP address. If the IP address matches, we further check if the transport layer protocol is TCP and the length of the IP header to determine where the IP layer payload begins. The IP and TCP layer headers include an optional field of variable-length, it is necessary to check the header length in both layers. If the IP address does not match, we stop inspecting the packet. Inspecting the TCP header is the last step to check whether the target application data is transmitted in the payload. Different from inspecting the other layers, inspecting the TCP layer requires the extraction of information as well as checking and filtering of specific packet fields. In the checking and filtering step, we first check whether the destination port matches the target application's service port. Then we get the length of the TCP header to locate where the TCP layer payload begins. In the information extracting step, we check and record the following fields: sequence number, source port, and payload length. We record the sequence number for preserving the request orders of the extracted data from the packets. The source port information is required to isolate requests from distinct clients. We check the length of the TCP payload and proceed to inspect the packet with a length greater than zero, indicating that the packet contains the application data that we need. Also, we extract the application data from the TCP payload.

Chapter 3

Related Work

3.1 Data Corruption inside Flash-based SSD

Several studies have investigated possible data corruption using the characteristics of flash-based SSDs. Ahmadian et al. [27] analyzed and investigated the various failures of SSD. They implemented a platform that detects physical failures inside SSD. Cai et al. [28] explored the fundamentals and recent research on flash-based SSD reliability. They investigated several studies on error mitigation and data recovery and suggested a system-memory codesign to enhance the reliability of flash-based SSD. Grupp et al. [21] investigated the characteristics of several commodity SSDs in terms of performance, power, and reliability. Based on the characteristics, they focused on performing application case studies, improving the performance while extending the lifetime of the storage devices.

Jaffer et al. [29] investigated the resilience of popular file systems to various errors in flash-based SSDs. They also introduced a fault injection framework and performed an extensive study over a thousand error cases. Meza et al. [19] performed an extensive analysis of flash-based SSD reliability trends in the field. They analyzed various internal and external characteristics of SSDs and investigated how these characteristics affect failures. Narayanan et al. [20] presented an extensive SSD failure characterization in production data centers using field data. They also investigated several factors that affect SSD failures and used machine learning approaches to evaluate the influence of relevant factors on failures.

3.2 Data Integrity Verification

Several studies have analyzed how to provide data integrity verification efficiently. Globus [30] introduced file-level overlapping data transfer and checksum computation to optimize the entire data transfer time. Liu et al. [31] also overlapped data transfer and checksum computation but performed an integrity verification at the block level. They divided a single file into different sized blocks and performed experiments to determine the optimal block size for different datasets. Arslan et al. [32] introduced FIVER, which cooperates with the data transfer and integrity verification process. They proposed overlapping during the data transfer and verification using cooperating processes to perform the verification process that takes a longer time. However, these approaches [30–32] perform integrity verification with data resides on host memory.

To provide the integrity verification with the data stored in the storage device, Charyyev et al. [33] proposed a robust integrity verification algorithm (RIVA). RIVA focuses on detecting possible data corruption in the process of moving the data from host memory to the storage device. To detect such silent data corruption, RIVA deletes the file content in the page cache of host memory by invalidating the page cache and reading the content directly from the disk to perform the integrity verification. Previous studies, including RIVA [33], still overlooked the possibility of data corruption inside the storage device.

3.3 Backup and Recovery

There have been several studies to provide backup and recovery functionalities at different layers. Ext3cow [34] and BTRFS [35] support backup and recovery via snapshot by exploiting a copy-on-write (CoW) strategy at the file system layer. Ext3cow [34] introduced file versioning and file system snapshot using the CoW by extending the ext3 file system. BTRFS [35] manages files and directories in the subvolumes that are the unit of creating snapshots or clones. However, the CoW-based backup and recovery mechanism involves extra write operations caused by data copy or garbage collection which damages the performance of normal operations on flash-based SSDs [36, 37]. In addition, both techniques have dependencies on specific file systems.

On the other hand, many researchers have investigated providing snapshots or versioning to support backup and recovery functionality at the block layer. Peabody [38] introduced a network block storage device that supports roll the state back to an arbitrary point in time by sector versioning and log keeping. TRAP [39] proposed a new disk array architecture that provides point-in-time recovery by leveraging exclusive-OR operations on top of RAID4/5 controllers. These approaches [38, 39] are file system independent but require modification to operating systems.

A number of studies have taken advantage of the characteristics of flash-based SSDs to support backup and recovery functionalities at the storage device [36, 37, 40, 41]. BVSSD [36] introduced a block-level versioning system by exploiting the property of flash. A flash-optimized snapshot system, ioSnap [40], adds the remap-on-Write capability to the FTL to provide snapshot functionality. Also, ioSnap keeps track of blocks for each snapshot through a snapshot tree that traces the relationships between the snapshots. LTFTL [41] introduced lightweight time-shift FTL that maintains multiple versions of storage states to provide rollback capability. BR-SSD [37] proposed a backup and recovery scheme by extending the out-of-place update nature of flash-based SSD. These approaches [36, 37, 40, 41] take advantage of the nature of flash-based SSDs so that they require modifications of the firmware of the SSDs.

To support instantaneous recovery from system failures, several studies introduced a recovery protocol that consists of on-demand single-tuple redo and single-transaction undo processes [42, 43]. However, these approaches only provide on-demand single page reconstruction.

3.4 Logging Optimization

Several prior works have been proposed to optimize the logging performance by adopting Non-Volatile memory (NVM) including Intel Optane Persistent Memory Module [44–49].

LeanStore [45] introduced logging and commit protocol by adding a few steps to Wang and Johnson's scheme [44] to exploit the low write latency characteristics of NVM. LeanStore's protocol enables low-latency transactions on persistent memory by avoiding remote partition flushes. NVWAL [48] and NV-Logging [49] reduced the logging overhead of transaction systems by exploiting byte-addressable NVM. These approaches [44, 45, 48, 49] can be employed on machines with NVM and logging operation is still in the critical path of execution.

ATOM [46] introduced a hardware log manager that removes delays in the critical path caused by logging operations. Inspired by data movement task offloading to a DMA engine, ATOM offloads logging functions (e.g., allocating log space, writing log entries, and truncating logs) by introducing primitives for exposing atomic durable regions to hardware. ATOM relies on hardware primitives to persist the log at the destination.

PASV [50] optimized the logging performance by removing redundant log operations in LSM-tree based relational database systems (RDBs). For RDBs employing the LSM-tree based storage engine, there is a performance overhead due to logging performed independently at each of the RDB and storage engine layers. PASV addresses double-logging problem in LSM-tree based RDBs but logging operation is still in the critical path of execution.

3.5 Packet Inspection

Deep packet inspection (DPI) is widely used for network security, bandwidth management, user profiling for monetization, and other applications by inspection the packet contents including headers [51–54]. Several studies have taken advantage of the DPI technique, but mainly focused on network security [55–57] or traffic management [55, 56, 58–62].

Cilium [55, 56] provides eBPF-based networking, security, and observability solution for cloud native environments. Linux Security Module (LSM) introduced a patchset that facilitates a unified and dynamic MAC and audit policy by allowing eBPF programs to be attached to LSM hooks [57]. Cloudflare [58] and Katran [59, 60] provide layer 4 load balancing functionality by adopting eBPF. They leverage XDP and eBPF to enable fast packet processing by providing an in-kernel capability. Syrup [61] introduced a framework that enables application developers can specify their preferred scheduling policies. Syrup allows developers to define application-specific scheduling policies and executes them in the kernel with the help of eBPF. CRAB [62] proposed an alternative load balancing scheme that removes the load balancing from the critical path by redirecting TCP connections. CRAB inspects packets using an in-kernel middlebox implementation based on eBPF that allows connection redirection, thus removing the load balancer from the data path. These approaches [55–62] utilize deep packet inspection for network security or traffic management.

3.6 eBPF

Recently, using eBPF for I/O acceleration has been investigated to improve application performance [63–66].

BMC [63] introduced an in-kernel cache for accelerating Memcached [67]. BMC intercepts requests from the network driver before being processed by the host's network stack and handles them within the in-kernel cache. XRP [64] accelerates I/O operations by resubmitting I/O with the assistance of eBPF. XRP resubmits I/Os based on user-defined storage functions by using eBPF hook in the NVMe driver's interrupt handler to bypass the kernel's storage stack. ExtFUSE [65] introduced extensible user file system framework by leveraging eBPF. ExtFUSE allows applications to load in-kernel request handlers for their specialized functionality and to serve low-level file system requests. Zhong et al. [66] explored the potential of using BPF to accelerate storage by injecting user-defined functions into the kernel's storage stack. However, these approaches [63–66] require the developer's effort in providing their own handlers for application-specific optimization.

Chapter 4

Providing Data Durability for the Storage Layer

4.1 Overview

Data integrity verification is one of the most important features of storage systems. For example, file systems (e.g., BtrFS) detect data corruption by performing a cyclic redundancy check (CRC) in the units of pages of 4 KiB [68]. As the volume of data grows rapidly, it has become common to move data to remote storage systems. Transferring data to remote storage systems increases the possibility of data corruption caused by network failure, packet loss, or storage corruption of the receiver system [31]. Therefore, many systems have adopted end-to-end integrity verification by comparing the checksum of each file using secure hash functions such as MD5 and SHA1 [30–33]. When the receiver system detects data corruption, it reconstructs the corrupted data by requesting a data retransmission from the sender system. It is important to detect data corruption at the right time because detecting and reconstructing data corruption after a certain period affects the results of the data processing already done.

In recent years, flash-based storage devices, solid-state drives (SSDs), are increasingly replacing hard-disk drives in modern storage systems [29, 69, 70]. Such flash-based storage devices have a cache layer, called a buffer cache, that buffers data [7–9, 71, 72]. When writing data to SSD, the data are cached in the buffer cache of SSD for a while. Data cached in the buffer cache are flushed to NAND flash memory when the buffer cache is full, or according to SSD controller's buffer management policy. As the capacity of SSD increases, the size of the buffer cache also increases proportionally. In other words, the data written to the storage device stay in the buffer cache for a longer time. As a result, reading data immediately after writing to flash-based SSD returns data from the buffer cache, which has not yet been flushed to NAND flash memory. During the process of flushing data from the buffer cache to NAND flash memory, data can become corrupted due to the internal failures of SSD [19–21]. These data corruptions that occurred during the data processing inside the SSD are difficult to detect.

The general implementation of data transfer, including verification, is as follows. As a first step, the sender system reads a file from its storage device and sends it over the network to the receiver system. The sender system also computes and sends the checksum value of the file to the receiver system. When the file is transferred to and stored in the storage device of the receiver, the receiver performs an integrity verification by computing the checksum of the file to detect errors during transmission. Finally, the receiver compares the checksum value with the value sent from the sender. If the two values are different, the receiver considers that the file has been corrupted during the transmission and requests the sender to retransmit the file. In the existing implementation, data corruption that may occur in the procedure of flushing the data to the storage device is not sufficiently considered because the integrity verification is performed with data buffered in host memory.

To catch undetectable errors that occur while flushing data from the buffer cache to NAND flash memory, the integrity verification must be performed after a sufficient number of new data have been written to ensure that the data are flushed to NAND flash memory. If the receiver system clears the cached content of the file in page cache of host memory to provide robust data integrity verification, the entire data transfer time is significantly increased due to additional I/O operations and cache eviction. Therefore, the receiver system should schedule I/O operations considering the internal structure of SSD without extra overhead to perform a full coverage integrity verification in the data transfer process.

To address these issues, many researchers have investigated to optimize the entire time of data transfer, including integrity verification. For example, Liu et al. [31] and Globus [30] overlapped a data transfer and checksum computation with file and block granularity, respectively. Arslan et al. [32] introduced FIVER, which cooperates during data transfer and integrity verification processes to reduce the entire data transfer time. Different from these approaches [30–32] that focus on reducing data transfer time, RIVA [33] introduces a robust integrity verification algorithm that considers undetected write errors that occur on disk drives. However, RIVA focuses on detecting corruption that occurs during data is stored on the storage device from the host memory, and does not consider corruption that may occur inside the storage device. Considering the internal structure of flash-based storage devices, room still exists for reconsideration of robust and reliable integrity verification.

In this chapter, we introduce a reliable data transfer through robust data integrity verification scheme considering the internal operations of flash-based storage devices. We still provide efficient data transfer through concurrent verification procedure. To support robust and reliable data integrity verification, we schedule procedures for data verification in consideration of the internal structure of flash-based storage devices. After receiving a file and writing it to the storage device, we delete the memory mapping information of the file to invalidate the data stored in host memory of the receiver system. By doing this, we perform integrity verification with the data written on NAND flash memory which is the actual final destination. We delay the integrity verification process until the file to be verified is flushed to NAND flash memory. Integrity verification is performed by reading the file from the storage device directly after sufficient new data are stored on the storage device. To hide the overhead due to cache invalidation, we parallelize the checksum computation procedure. We investigate the end-to-end data transfer procedure including integrity verification and identify the main bottleneck is checksum computation. Based on the results of bottleneck analysis, we adopt page-level integrity verification using the CRC32C algorithm, a variant of CRC, that enables parallel execution of integrity verification for a single file. We reduce the entire time for data transfer including verification by efficiently parallelizing the verification procedure. We use the CRC32C algorithm because the algorithm is used in many file systems (e.g., BtrFS, ZFS, and XFS) to ensure the integrity of the data and metadata of a file in units of pages of 4 KiB [68,73,74]. With page-level integrity verification, we involve multiple threads to perform concurrent checksum computation for a single file. To examine that our proposed scheme detects errors in each layer of the storage stack, we implement a prototype that intentionally injects faults on the specific layer of the storage stack. After transferred file is stored, we manipulate the data of each storage layer to inject faults. Then, data integrity verification is performed to ensure that the proposed scheme can detect faults correctly.

For evaluations, we measured the entire data transfer time with realistic data transfer scenarios. With the prototype implementation, the proposed scheme detects data corruptions on each layer of the storage stack. Specifically, using fault injection, we demonstrate that the proposed scheme can detect errors occurring in NAND flash memory inside storage device that cannot be detected by the exiting scheme. Moreover, the experimental results demonstrate that the verification time for a single file and the entire data transfer time is reduced by up to 84% and 62% compared with the existing scheme, respectively.

In summary, our main contributions are as follows:

- We study and analyze the main bottleneck of the end-to-end data transfer procedure including integrity verification.
- We schedule I/O operations considering the internal structure of flash-based storage device to ensure integrity verification including data corruption that occurs inside the storage device.
- We parallelize checksum computation procedure and overlap it with I/O operations to provide efficient data transfer while ensuring the robustness of data integrity verification.
- We implement a prototype that intentionally injects faults on the specific layer of the storage stack and examines detection of data corruptions.
- The experimental results show that our scheme provides robust and reliable data transfer while efficiently performing computations and I/O operations.

4.2 Motivation

In this section, we demonstrate and analyze a simple preliminary experiment conducted to investigate the main bottlenecks in data transfer including integrity verification between remote systems. We transferred a single 1 GiB file for the analysis and stored it to the storage device of the receiver system. Then, we read the file directly from the storage device and performed the integrity verification. We used a Samsung PM983 3.84 TB NVMe SSD [75] as the storage device. The detailed specifications of the sender and receiver system are described in Section 4.4. The analysis result is illustrated in Figure 4.1. As shown in the figure, the checksum computation time in the receiver system occupies 54% of the entire data transfer time. Moreover, the write and read operations on the receiver system account for 12% and 13% of the entire data transfer time. If we consider the internal structure of SSD, the entire data transfer time with verification becomes longer, because we should perform data verification after it is guaranteed that the data has been written to NAND flash memory of SSD. Therefore, to perform integrity verification efficiently, we adopted page-level checksum computation, which verifies a single file concurrently using multiple threads.

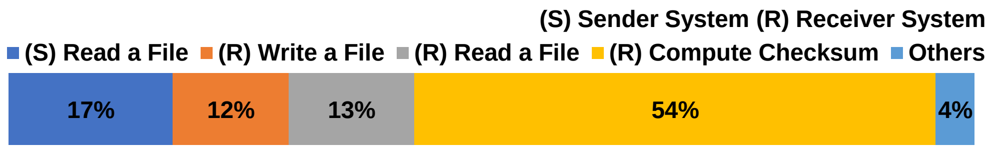


Figure 4.1: Major functions of the entire data transfer procedure including verification

4.3 Design and Implementation

4.3.1 Overall Architecture

Previous studies have mainly focused on reducing the data transfer time, and robustly providing integrity verification of transferred data has been overlooked. To address this problem, research such as RIVA [33] that invalidates data cached during the transmission process and performs integrity verification with data stored in the storage device has been proposed. Considering the internal structure of SSD, there is still a possibility that corruption may occur in the process of data is stored on the final destination, which is hard to be detected using existing schemes.

Performing integrity verification by reading the data after stored on NAND flash memory increases the entire data transfer time and degrades the performance. To efficiently perform the entire data transfer while ensuring robust integrity verification, we parallelize checksum computation procedure by involving multiple threads and overlap it with I/O operations. We control the order of I/O operations to detect data corruption that occurred inside the storage device; thus retransmission can be performed immediately to prevent corrupted data from being used for subsequent processing.

Figure 4.2 illustrates the overall architecture of the proposed system. As illustrated in the figure, our system consists of four components, including a *memory manager*, *I/O manager*, *data verifier*, and *concurrency controller*. When a file is transferred over the network, the *memory manager* first allocates memory to store the file content. After the transfer is complete, the *I/O manager* writes the file to the storage device. When the file is completely written to the storage device, the *memory manager* deletes the file content in memory by invalidating the page caches to prevent performing integrity verification with cached

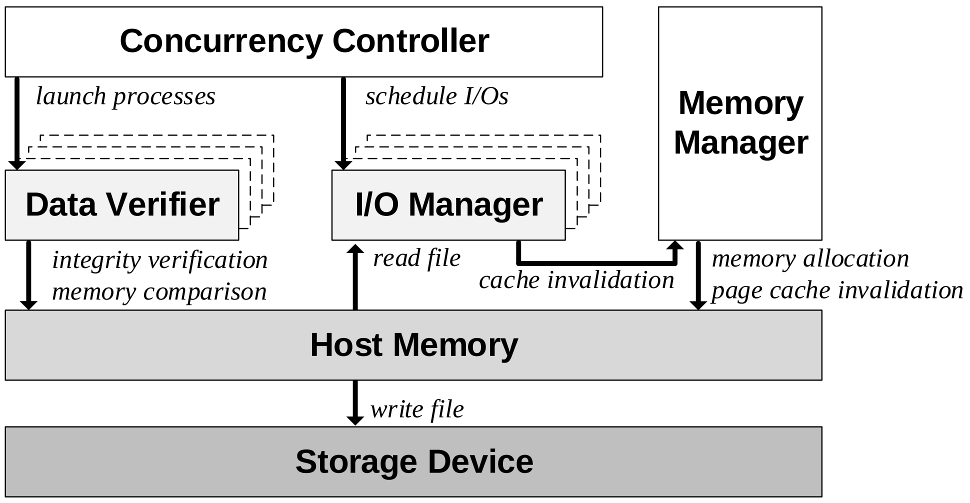


Figure 4.2: Overall architecture

data. The *concurrency controller* schedules I/O operations so that the file to be verified is written to NAND flash memory. Then, the *concurrency controller* requests the *I/O manager* to read the file to be verified. The *concurrency controller* launches multiple *data verifier* threads to perform integrity verification for a single file. The *data verifier* computes the checksum of a partial range of the file and compares it with the values from the sender system. The *concurrency controller* schedules multiple *data verifier* threads and I/O operations in parallel to minimize the entire data transfer time.

4.3.2 Design and Implementation

Memory Manager

The *memory manager* is responsible for memory allocation, deallocation, and cache invalidation after the transferred data are completely written to the storage device. The *memory manager* allocates and deallocates memory using the `mmap`, `munmap`, and `mincore` system calls to efficiently manage page caches of the receiver system. During file transfer, the data are buffered in the memory region created by the `mmap` system call. When receiving and writing the file to the storage device are finished, the memory region used to buffer the file should be cleared to ensure the data verification is performed with the file stored in the storage medium, which is the actual final destination. To remove data of the file buffered in memory, we use the `munmap` and `mincore` system calls. Because the `munmap` system call deletes the mappings for the specified address range, it causes a page fault on further references to the unmapped memory region. Although the `munmap` system call deletes the mappings at once, the *memory manager* uses the `mincore` system call to ensure that all pages in the specified range have been deleted. By invalidating cached data, our scheme ensures robust data integrity for transferred data so that prevents corrupted data is used in any subsequent processing.

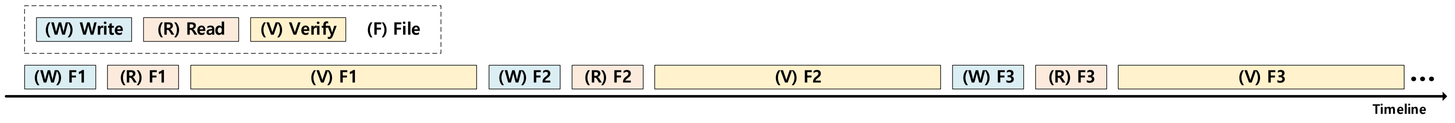
Concurrency Controller

The *concurrency controller* is responsible for scheduling I/O operations and integrity verification procedures so that the entire data transfer is performed efficiently. The integrity verification procedure performs two types of I/O operations: writing the transferred file to the storage device and reading the file for integrity verification. The *concurrency controller* reduces the data verification time by overlapping the I/O operations and the verification process performed by the *data verifier*.

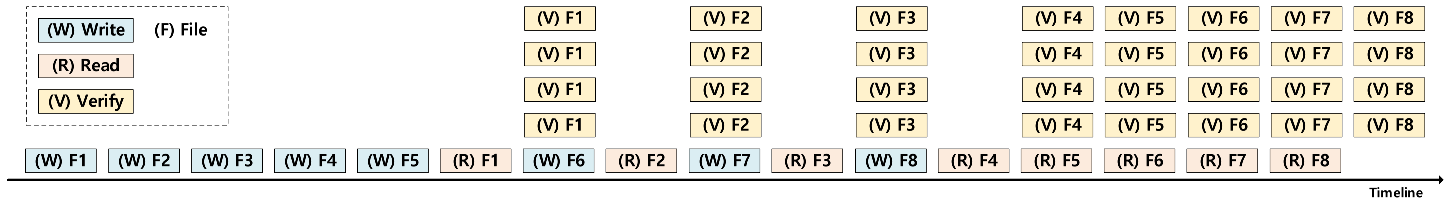
Supporting Data Verification Scheduling The important role of the *concurrency controller* is to schedule I/O operations regarding the size of the buffer cache of SSD, which in general is 0.1% of the storage capacity [72]. We ensure the robustness of data integrity by verifying the file written in NAND flash memory. To do this, we read the file to be verified from the storage device after sufficient data are written. To write sufficient data to the storage device, the *concurrency controller* maintains the accumulated size of files that stay in the buffer cache before being flushed to NAND flash memory of SSD. In addition, using the verification waiting list, the *concurrency controller* memorizes the files for which the integrity verification should be performed next.

For example, suppose that the buffer cache size of SSD is 4 GiB and that the dataset to be transferred consists of eight 1-GiB files. When a single file of 1 GiB is transferred, the *concurrency controller* schedules the *I/O manager* to write the file and adds the file information to the waiting list. After the written data exceed the buffer cache size, the *concurrency controller* schedules the *I/O manager* to read the file to be verified directly from the storage device. In this example, after writing five files, the *concurrency controller* reads the file and performs the verification. Then, the *concurrency controller* launches the *I/O manager* and *data verifier* simultaneously to write the transferred file to the storage device and perform the checksum computation for the file in the waiting list in parallel.

Figure 4.3 illustrates the scheduling example of the *concurrency controller* in the case of transferring the dataset consisting of eight files of 1 GiB. In the existing scheme (Figure 4.3a), each file is read from the storage device immediately after write performed, then perform integrity verification using a single thread. As can be seen, the procedures for write, read, and verify data are performed sequentially for each file in the existing scheme. In contrast, as shown in Figure 4.3b, we control the order of each procedure corresponding to individual files to guarantee that the integrity verification is performed with data written on NAND flash memory.



(a) Scheduling example of the existing scheme



(b) Scheduling example of the proposed scheme

Figure 4.3: Scheduling example of the existing and proposed schemes. Each box represents the execution of corresponding process, such as write a file to the storage device, read a file from the storage device, and verify integrity of the file. Each number indicates a different file

Supporting Pipelined Data Verification Another key role of the *concurrency controller* is overlapping I/O operations with checksum computation procedures. Based on the bottleneck analysis described in Section 4.2, we identify that checksum computation occupies 54% of the entire procedure. Thus, we overlap the procedure of integrity verification for a given file with I/O operations, such as write the file to the storage device or read the file from the storage device.

In the illustrated example of Figure 4.3b, integrity verification for file 1 ((V) F1) is executed concurrently with the write operation for file 6 ((W) F6). Read operations can also be executed concurrently with integrity verification. In the example, read operations for files 5 – 8 ((R) F5 – F8) are overlapped with the integrity verification for files 4 – 7 ((V) F4 – F7), respectively.

Through such overlapping, our scheme enables efficient data transfer including verification despite the overhead of read files after write them to NAND flash memory.

Supporting Concurrent Data Verification The last key role of the *concurrency controller* is executing multiple verification procedures for a single file in parallel by dividing the range. As described in Section 4.2, the checksum computation time is a major bottleneck in the integrity verification procedure. Therefore, we perform verification for a single file using multiple threads by dividing the file depending on the number of threads, in addition to pipelined data verification. To simplify the task of dividing the file by a varied number of threads, we adopt a page-level checksum. The *concurrency controller* divides the memory area that contains file content according to the number of threads executing the verification and launches multiple *data verifier* threads. When launching a thread, the *concurrency controller* passes the memory range information (e.g., start address, length) of the data to be verified to each thread and checksum value of the corresponding range transferred from the sender.

In the example of Figure 4.3, the number of threads that are concurrently executed for the integrity verification is four. If we increase the number of threads, the verification time and entire data transfer time decreases. However, if we decrease the number of threads, it takes a longer time to verify and transfer the data.

The number of threads to perform verification is configurable depending on the computation resource (e.g., CPU) and the size of the file to be verified. We analyze the correlation between the number of threads and the computation resources through evaluation results in Section 4.4.

I/O Manager

The *I/O manager* is responsible for I/O operations scheduled by the *concurrency controller*. The *I/O manager* writes file contents from host memory to the storage device when the file transfer is completed. Before closing the file, the *I/O manager* synchronizes the file state with the storage device using the `fsync` system call. After writing the file to the storage device, the *I/O manager* notifies the *memory manager*, so that the *memory manager* deletes the content of the file from host memory. In addition, the *I/O manager* reads the file to perform the integrity verification from the storage device and transfers the file location in memory to the *data verifier*. The *I/O manager* uses synchronous calls to ensure that all I/O operations are performed immediately.

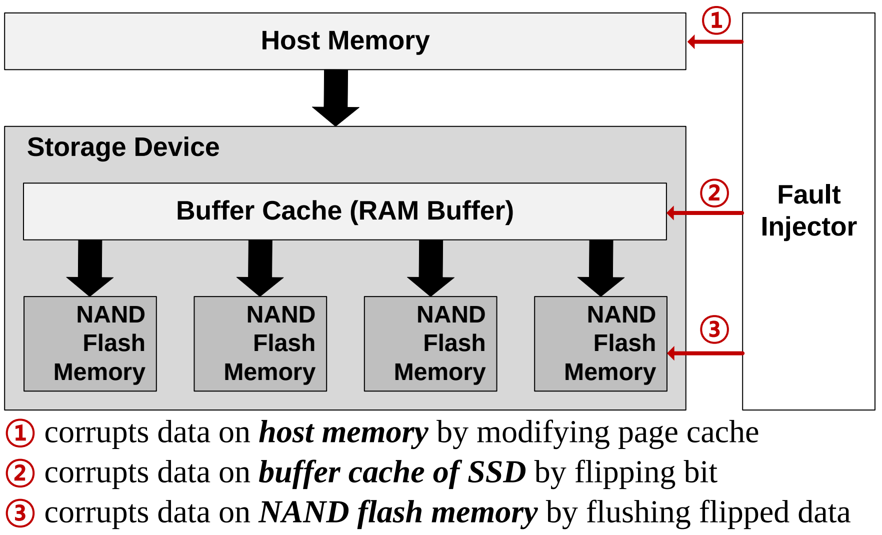
Data Verifier

The *data verifier* performs the checksum computation for a single file scheduled by the *concurrency controller*. When performing the verification, multiple verification threads can be executed in parallel by the *concurrency controller*. The *concurrency controller* divides and assigns the range of the file to be verified in each thread. Then, the *concurrency controller* passes the range to each thread with a list of checksum values transferred from the sender. Each verification thread computes the checksum values for pages that correspond to the specific range of the file. When the computation is finished, the *data verifier* validates the results by comparing the memory region of the corresponding checksum values using the `memcmp` system call. The reason for validating results by comparing memory is to provide a constant validation time. Instead of comparing memory regions, if we validate results by comparing values, the verification time increases in proportion to the number of pages verified in each thread, which can lead to a new bottleneck.

Error Detection by Injecting Faults

To investigate our scheme detects errors on each layer of the storage stack, we implement a prototype that intentionally corrupts data written on the specific storage layer, as shown in Figure 4.4. Basically, we inject faults by modifying the stored file in each layer as follows. Transferred file is first written to host memory, then stored on the storage device. In the storage device, a file is first stored on the buffer cache and then stored on NAND flash memory, the actual final destination. Thus, the prototype fault injector corrupts the data in the order in which the file is written to the storage stack.

In the case of injecting faults into files on host memory, we change the file contents by manipulating cached page data(①). However, in the case of injecting faults into files on the buffer cache or NAND flash memory, the process of injecting faults is more complicated. Because observing the internal failure of SSD in the real system is costly, we injected faults into the files in NAND



① corrupts data on ***host memory*** by modifying page cache
② corrupts data on ***buffer cache of SSD*** by flipping bit
③ corrupts data on ***NAND flash memory*** by flushing flipped data

Figure 4.4: Prototype implementation to inject faults

flash memory in SSD, the final destination of the file transmission. We directly injected faults into a file in the buffer cache of SSD(②) and wrote dummy data sequentially to flush the corrupted data from the buffer cache to NAND flash memory(③). We write dummy data that are larger than the capacity of SSD buffer cache, so that guarantee corrupted data are written to NAND flash memory. More specifically, at first, we copied the data from a transferred 1-GiB file and modified it by flipping the least significant bit in every 4-KiB page. After the original file is flushed to NAND flash memory, the fault injector overwrites it with the corrupted data and writes enough dummy data to flush the corrupted data to NAND flash memory.

4.4 Evaluation

4.4.1 Experimental Setup

We evaluated the performance of our integrity verification scheme using two machines connected by a 10 Gbps network. The machine has 72 physical cores but we only use 18 physical cores on a single socket. We use a 3.84 TB Samsung PM9A3 NVMe SSD for storage device. The detailed specifications of the machine are described in Table 4.1.

Through the experiments, we focused on demonstrating that our scheme provides robust and reliable data transfer without sacrificing performance with the help of additional computing resources. To demonstrate the efficiency of the proposed scheme, we first conduct the experiment to investigate the effect of the concurrent integrity verification using multiple threads. We evaluated the proposed scheme with a realistic workload that transfers 20 files of 1 GB over the network. We evaluated the entire data transfer time of the proposed scheme by changing the number of *data verifiers* and compare it with the sequential approach. In order to investigate how much more computing resources required for the proposed scheme, we analyzed the average and total CPU utilization during data transfer including integrity verification. Then, we evaluated the entire data transfer time by changing file sizes and compare it with the existing schemes, including the sequential, file-level pipelining [31], and RIVA [33] which

Table 4.1: Specification of the machines.

Processor	4-way Intel E7-8870@2.1GHz
Memory	256 GiB
Storage	Samsung PM9A3 NVMe SSD, 3.84TB
OS	Ubuntu 16.04
Kernel	kernel v4.4.0

is the most recent state-of-the-art scheme. Finally, to validate the proposed scheme guarantees robust and reliable data integrity verification, we analyzed the changes of each layer of the storage stack as data move from host memory to NAND flash memory. With the prototype implementation, we investigated fault detection by the existing and proposed schemes by injecting faults into different storage layers of the receiver system.

In the following experiments, each evaluation point is obtained by averaging the results of 5 independent executions.

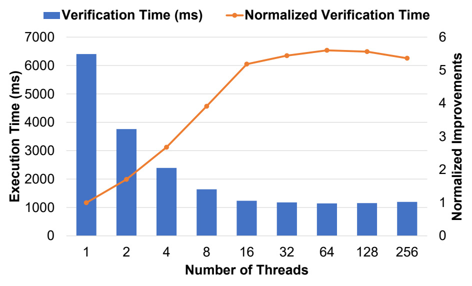
4.4.2 Performance Results and Analysis

Data Verification Time

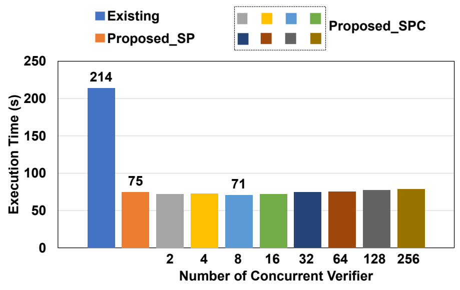
We used a single file of size 1 GiB to demonstrate the effectiveness of concurrent computation. In this experiment, we focused on the effectiveness of concurrent verification rather than other techniques, so we used the file already stored in the storage device. Figure 4.5a presents the data verification time when performing multiple integrity verification procedures concurrently using multiple threads.

When we measured the verification time by doubling the number of threads, the verification time reduced until the number of threads reached 16. The reason is that the data size assigned to each thread is halved when we double the number of threads performing the computation. However, when the number of threads is more than 16, the verification time is almost the same, but when the number of threads is more than 256, the verification time increases. This is because, when numerous threads are executed concurrently, the management overhead, such as thread creation and context switching, greatly increases.

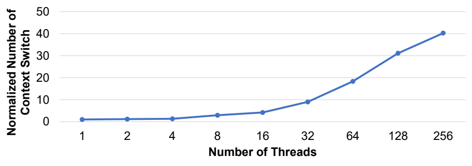
To identify the management overhead, we measure the number of context switches during data verification. Figure 4.5b shows the normalized number



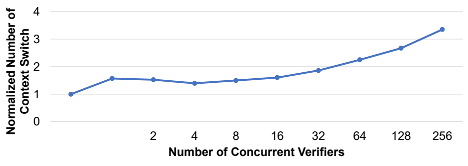
(a) Data verification time



(a) The entire data transfer time



(b) Normalized number of context switch



(b) Normalized number of context switch

Figure 4.5: Data verification time

Figure 4.6: The entire data transfer time

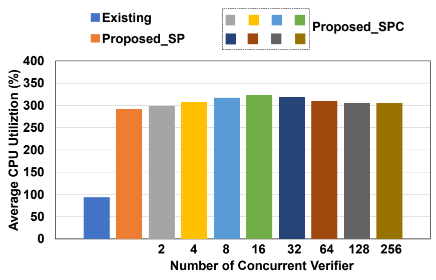
of context switches as the number of threads increases. As can be seen, until the number of threads reaches 16, the number of context switch increase is insignificant. However, the number of context switches increases exponentially when more than 32 threads are used to perform the verification. As a result, the verification time increases despite doubling the number of threads.

Entire Data Transfer Time

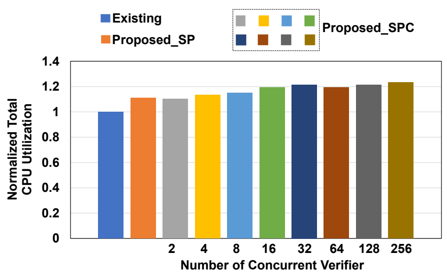
Figure 4.6a presents the entire data transfer time of the existing sequential scheme and the proposed scheme. The entire data transfer time includes the data transfer time between the sender and receiver systems, the data writing time to the storage device of the receiver system, and the data integrity verification time. In the figure, we denote the existing sequential scheme as Sequential and the proposed scheme as Proposed_SP and Proposed_SPC. In the proposed scheme notations, S, P, and C stand for scheduling, pipelining, and concurrent verification, respectively. The evaluation results indicate that the entire transfer time is reduced by 65% with the Proposed_SP scheme compared with the Sequential scheme. Similar to the file-level pipelining introduced in the previous study [31], the overlapping computation and I/O operation reduced the entire transfer time by 65%. The difference between file-level pipelining and the proposed scheme is that the proposed scheme delays performing the integrity verification until the file is written to NAND flash memory. As a result, the proposed scheme provides robust integrity verification compared with the file-level pipelining scheme while efficiently transferring files.

When we applied the concurrent verification using multiple threads (Proposed_SPC), the entire data transfer time was reduced by 67% and 5% compared with the existing sequential scheme (Sequential) and proposed scheme without concurrent verification (Proposed_SP), respectively. The entire data transfer time decreased until the number of concurrent verification threads reaches 8. As depicted in the figure, when we performed the integrity verification in parallel with 8 threads, the entire data transfer time was the most shortened, which is a 67% reduction compared with the existing sequential scheme.

However, using more than 8 threads for integrity verification increased the entire data transfer time. When too many threads were executed in parallel, thread management tasks (e.g., thread creation, clean up, context switch, etc.) took a longer time, as mentioned in the previous section. Figure 4.6b reveals that the number of context switches increased rapidly when we used more than 32 concurrent verifiers. Moreover, the *concurrency controller* maintained the total size of the verified data as a global variable. Because each verification thread accesses the global variable, performing verification with many threads causes memory access contention. As a result, the entire data transfer time



(a) Average CPU utilization



(b) Total CPU utilization

Figure 4.7: CPU utilization comparison

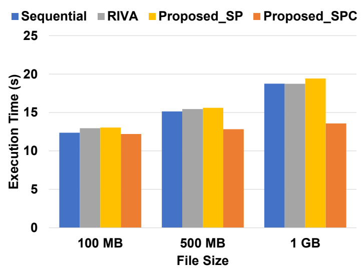
increased.

Resource Utilization

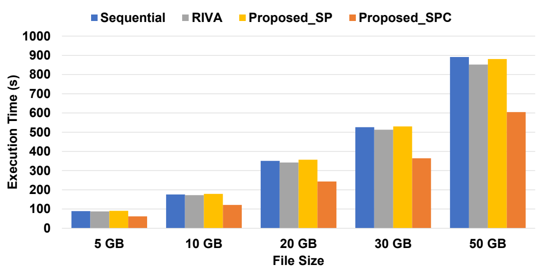
As we created multiple threads to concurrently verify the integrity of the file, we measured the CPU utilization of each scheme over time, as presented in Figure 4.7. Figures 4.7a and 4.7b list the average and total CPU utilization for each scheme, respectively. The average CPU utilization of the proposed scheme with 8 concurrent verifiers is 316%, which is 3.4 times higher than that of the existing sequential scheme (92%). However, Figure 4.6a indicates that the entire data transfer time of the existing sequential scheme is up to 3.01 times longer than that of the proposed scheme. As a result, the total CPU utilization of the proposed scheme is from 10% to 20% higher than that of the existing sequential scheme (Figure 4.7b).

The difference in the gap depends on the number of concurrent verifiers. For example, using 8 concurrent verifiers, the proposed scheme completes the data transfer and verification 3.01 times faster using 20% more of the total resources, compared with the existing sequential scheme.

In summary, with the proposed scheme, we can reduce the entire data trans-



(a) The entire data transfer time with file size under 1 GB



(b) The entire data transfer time with file size over 5 GB

Figure 4.8: Data transfer time comparison changing file size

fer and verification time by up to 67% by intensively investing computing resources in a short time.

Entire Data Transfer Time with Different File Sizes

Figure 4.8 shows the entire data transfer time of each scheme with various file size. In the experiments, we compare the entire data transfer time including verification with the existing schemes that performs data verification in the order in which files were transferred (Sequential) and the state-of-the-art robust data transfer scheme (RIVA) [33]. We used a single file of different sizes from 100 MB to 50 GB to demonstrate the effectiveness of concurrent verification with various file sizes.

With the Proposed_SP scheme, there is an overhead due to scheduling and time to write and read data on the final destination, NAND flash memory, but it is negligible as shown in the experimental results. The entire data transfer time of the Proposed_SP scheme takes 1% to 5% longer than those of Sequential and RIVA, regardless of the file size. With the Proposed_SPC scheme, the entire data transfer time difference is insignificant for the file size of 100 MB. Different from the Proposed_SP scheme, the Proposed_SPC scheme is effective for data

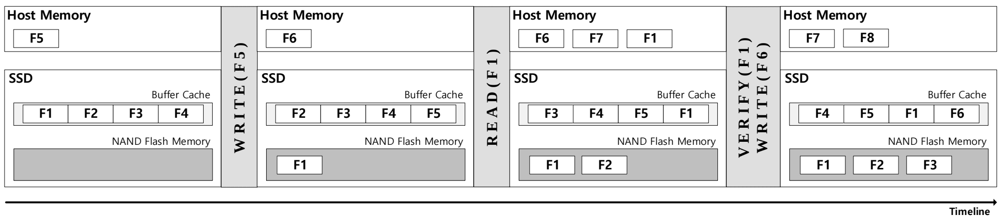


Figure 4.9: Host memory and SSD status changes over time

transfer including integrity verification as the file size increases. As the file size increases, the entire transfer time reduction reaches about 30% when the file size is 1 GB or more. The reason is that the Proposed_SPC scheme reduces the time of data verification by involving multiple threads to perform concurrent verification.

Host Memory and SSD Status Analysis

To validate that the proposed scheme reads a file from NAND flash memory of SSD, we analyzed the changes in the data residing on each storage medium (e.g., host memory, buffer cache of SSD, and NAND flash memory) over time. Figure 4.9 illustrates changes of data residing on each storage medium during data transfers. The figure presents the data change of each storage medium from the time when the data written to the buffer cache starts flushing to NAND flash memory.

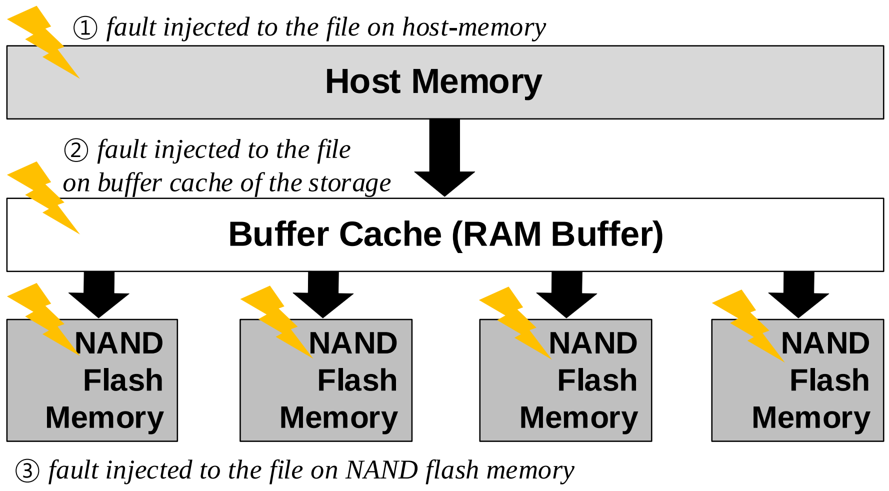
When a newly transferred file (e.g., F5 in the figure) starts to write to SSD, the SSD controller starts to flush the first 1-GB file to NAND flash memory (e.g., F1 in the figure). Simultaneously, a new file, F6, is transferred over the network and written to host memory. The second state of the figure displays the state of each storage medium after flushing file F1 and writing file F5. In the second state, file F1 exists only in NAND flash memory and is removed from the buffer cache of SSD. Therefore, the *concurrency controller* schedules the *I/O manager* to read file F1 to start the verification. After reading file F1 from SSD, the data on each storage medium are shown in the third state of the figure. The SSD controller flushes file F2 to make space in the buffer cache so that file F1 can be cached. In the third state, the *concurrency controller* performs the integrity verification of file F1 in parallel using multiple threads. The state after the verification completes is presented in the last state of the figure. The *concurrency controller* schedules the *I/O manager* so that file F6 is written to the storage device while the verification is performed. The file transfer continues, and the new file F8 is transferred to host memory.

The proposed scheme provides the improved robustness of the integrity verification by scheduling the read and write operations alternately. Moreover, the proposed scheme performs the integrity verification after the file is written to NAND flash memory.

Detecting Data Corruption

To evaluate the robustness of the proposed scheme, we conducted experiments to measure how many errors can be detected in each scheme. In the experiments, we focused on whether each scheme can detect errors occurring in each layer of the storage stack where data resides until the data transmitted to the receiver system is stored in NAND flash memory in SSD, the actual final destination of the transferred data. With the prototype implementation, we adopted a mechanism of injecting faults into files in different storage layers, host memory, the buffer cache of SSD, and NAND flash memory, as shown in Figure 4.10.

Table 4.2 presents the result of detecting errors of each scheme when injecting faults into different storage layers. In the experiments, the proposed scheme (Proposed_SP, Proposed_SPC) detects all errors in the transferred file in NAND flash memory, while the other schemes do not detect the errors that occurred



① fault injected to the file on host-memory
Host Memory
② fault injected to the file on buffer cache of the storage
Buffer Cache (RAM Buffer)
NAND Flash Memory
NAND Flash Memory
NAND Flash Memory
NAND Flash Memory
③ fault injected to the file on NAND flash memory

Figure 4.10: Robustness test by injecting faults in different data storage layers on the receiver system

Table 4.2: The results of fault injection experiments in different storage stack of the receiver system

	Host memory	Buffer cache	Flash memory
File-LevelPpl	✓	-	-
RIVA	✓	✓	-
Proposed_SP	✓	✓	✓
Proposed_SPC	✓	✓	✓

in some layers. For example, RIVA [33] could not detect errors that occurred in NAND flash memory of SSD. On the other hand, file-level pipelining (File-LevelPpl) [31] only detects errors that occurred in host memory. This is because data verification is performed with the data stored in the buffer cache of SSD or host memory with each scheme, not the data stored on NAND flash memory in SSD. In contrast, the proposed scheme detects the internal failures of the storage device because the *data verifier* reads files to be verified directly from NAND flash memory after guaranteeing the data are written to NAND flash memory.

4.5 Summary

In this chapter, we propose a concurrent and robust end-to-end data integrity verification scheme for flash-based storage devices. We schedule I/O operations considering the internal structure and operation of the storage device to perform an integrity verification with the data written on the final storage media. By doing this, the proposed scheme detects data corruption that occurred across storage layers including inside the storage device. To provide data transfer without sacrificing performance, we concurrently perform I/O operations and integrity verification with the fine-grained data unit. With the prototype implementation and storage medium state analysis, we guarantee that the proposed scheme performs robust integrity verification with the data written on the actual final destination. The experimental results with realistic scenarios demonstrate that the proposed scheme provides robust and reliable data transfer while reducing the entire data transfer time by up to 62% compared with the existing scheme.

Chapter 5

Providing Data Durability in the Application Layer

5.1 Overview

Database systems are widely deployed in cloud services and applications to store their data. One important feature of the database system is data backup and recovery. Database administrators prevent loss and corruption of data by periodic backup including structure information (e.g., schema, indexes, and so on) and stored data [76–79]. When a disaster happens to the database system, such as hardware failure, data corruption due to bugs, system crashes, and other reasons, the administrators restore the database from the backup. Especially, services and applications of cloud computing require a mechanism to efficiently backup continuously increasing data.

For database backup and recovery, two techniques are widely used [80]. One of the techniques is a physical backup [22] that copies files containing the architecture of the database and the table files. Physical backup is intuitive since backup and recovery can be done by copying the files of the database. The other technique is a logical backup [80] that scans the entire database to generate sequences of query statements that can reproduce the database itself. Since logical backup is based on query statements, so that backup data is portable across the underlying file systems, OS, and MySQL versions.

Both techniques support two backup and recovery strategies: a full backup and an incremental backup. The full backup is performed by storing the data of the entire database, whereas the incremental backup is performed by copying or scanning data that changed since the latest backup [77, 81–84]. For example, Xtrabackup of Percona [22] provides a backup and recovery solution using the physical backup. Xtrabackup performs the full backup and recovery by copying the entire files of the database. Meanwhile, the incremental backup is performed by comparing each data page between the current database and the latest backup. Xtrabackup writes the pages that have changed since the last backup to the delta files.

On the other hand, MySQL provides a tool for logical backup, mysqldump [23]. Mysqldump scans and extracts the contents of the database to generate a sequence of query statements for rebuilding the database. When restoring a backup, mysqldump interprets the query statements and executes them. For the incremental backup, mysqldump uses a binary log (binlog) that contains a collection of the events that modify the database. The binlog contains all the query statements in the same order in which they were executed. Therefore, with binlog, mysqldump performs an incremental backup without scanning or extracting the data from the database. However, both techniques require additional I/O operations such as extracting data from the database or copying files when performing a backup. In addition, both techniques take a long time to restore the database from backup.

In addition to the aforementioned utilities (e.g., Xtrabackup and mysqldump), many researchers have investigated to provide backup and recovery functionality across different layers of the storage stack. For example, ext3cow [34] and BTRFS [35] support snapshot-based backup and recovery functionality depending on copy-on-write (CoW) strategies. In addition, many researchers have been investigated providing backup and recovery functionalities on the storage device itself by modifying the flash translation layer (FTL) of the flash-based solid-state devices (SSDs) [36, 37, 40, 41]. However, snapshot-based backup provided by the file systems disrupts normal operations and SSD-supported backup and recovery functionalities are not available on commodity SSDs.

In this chapter, we introduce an efficient backup and recovery scheme for database systems by exploiting write-ahead logging (WAL). Our key idea is to exploit log data created by the existing WAL for database backup without additional query or I/O operations. Furthermore, we reduce recovery time by modifying the existing recovery procedure of the database system. To do this, for backup, we devise a backup system based on the existing log system. In the backup system, we maintain the backup state of the system by managing a backup information list. Each node in the list contains backup information regarding WAL. The backup system stores the system backup state to the separate file if the state changes, such as creating a new backup or deleting old backups. Also, we modify the existing procedure of WAL management of the database system to preserve the log data of WAL. To reduce the recovery time, we divide the entire recovery range based on the buffer pool size. We reduce the time to apply log data to the data files by involving multiple I/O threads of the database system. Besides, we maintain the latest recovery information to provide data consistency in the case of a system crash during a recovery operation.

We implement the proposed scheme on MySQL 8.0.15 and evaluate the performance of backup and recovery under synthetic backup and recovery scenarios with `sysbench` benchmark. To evaluate the impact of the storage device performance on the database backup and recovery, we run the scenario with different storage device configurations (e.g., the number of devices). The experimental results show that the proposed scheme provides instant backup and fast recovery. In addition, regarding the backup, the proposed scheme does not require additional I/O operations for storing backup data.

In summary, our main contributions are as follows:

- We analyze the existing backup and recovery schemes for the database system.
- We introduce an efficient backup and recovery scheme using write-ahead logging to provide instant backup and fast recovery.
- We implement the proposed scheme on MySQL and evaluate its performance with a synthetic scenario.
- The experimental results show that the proposed scheme provides fast backup and recovery speeds compared with the existing backup/recovery techniques.

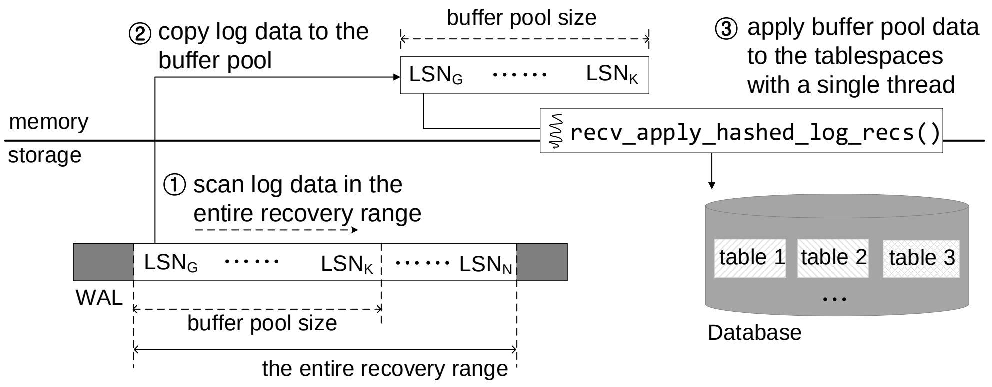
5.2 Motivation

The database systems use write-ahead logging (WAL) to provide data consistency and durability. Since WAL contains all the changes of data in the database, it is one of the most essential components of the database system. When the data are changed, it must be recorded in WAL first. Before changes are applied to the data files, the log must be written to the persistent storage.

Thus, WAL has a record of all data changes in the database. However, the data recorded in WAL is temporary. The data of WAL is deleted after they are applied to their data files. In the proposed scheme, we keep the log data even they are applied, so that using them as backup data. By using log data of WAL as backup data, the proposed scheme does not perform additional data manipulation such as copying when performing a backup.

The recovery procedure of MySQL is executed at MySQL startup and does not run again until it terminates. The procedure reads and applies log data in WAL that are not written to their appropriate tablespaces due to unexpected system terminations. Starting from log data with a checkpoint label, MySQL scans forward and copies the logged changes to the buffer pool first. If the buffer pool becomes full while loading log data into the buffer pool, the recovery procedure pauses loading and applies log records in the buffer pool to the persistent storage device. However, it is done by a single thread responsible for recovery because it is performed in the middle of the recovery procedure. Since flushing data in the buffer pool with a single thread is inefficient, the proposed scheme modifies the existing recovery procedure so that multiple threads can be involved for the flush operation.

Figure 5.1 presents the existing crash recovery procedure of WAL in MySQL. Each log data has a unique log sequence number (LSN) that represents the ordering of the changes. For example, assume that we want to recover pages between LSN_G to LSN_N of WAL. The recovery procedure scans log data in WAL sequentially from the first page of the entire recovery range (①). While scanning log data, the procedure also copies log data into the buffer pool if the LSN value of the corresponding data page is smaller than that of the log data (②). When the buffer pool is full of log data that needs to be recovered, the procedure pauses scanning and copying log data and begins applying data



② copy log data to the buffer pool
buffer pool size
③ apply buffer pool data to the tablespaces with a single thread
LSN_G …… LSN_K
memory
storage
recv_apply_hashed_log_recs()
① scan log data in the entire recovery range
LSN_G …… LSN_K …… LSN_N
WAL
buffer pool size
the entire recovery range
table 1
table 2
table 3
…
Database

Figure 5.1: Existing Crash Recovery Procedure of Write-Ahead Logging in MySQL

from the buffer pool to the corresponding data files (③). The existing recovery procedure performs I/O operations with a single thread, since I/O threads are sleeping during recovery.

5.3 Design and Implementation

5.3.1 Overall Procedure

Figure 5.2 presents the overall procedure of the proposed scheme. We perform a full backup by copying the files of the entire database. In the case of performing a full backup and restore, we copy the files of the entire database. Meanwhile, regarding incremental backup and restore, we use log data of WAL to provide backup without additional I/O operations and provide restore by modifying the existing crash recovery procedure. In addition, we maintain a separate backup/recovery information file that contains information about all backups and the last recovery.

First, we create the file in the backup directory to store backup information persistently and to ensure data consistency in the event of a crash during a

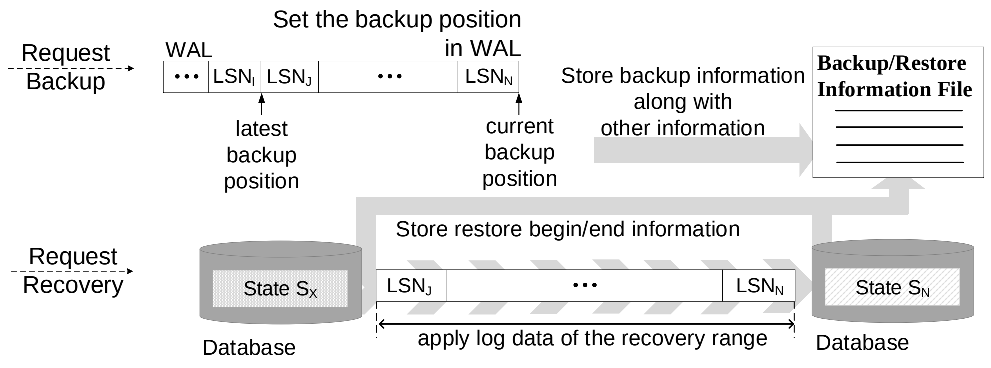

Set the backup position in WAL
Request Backup
WAL
LSN_I
LSN_J
LSN_N
latest backup position
current backup position
Store backup information along with other information
Backup/Restore Information File
Store restore begin/end information
Request Recovery
State S_X
Database
LSN_J
LSN_N
apply log data of the recovery range
State S_N
Database

Figure 5.2: Overall Procedure

recovery operation. The file contains summarized backup information, backup information list, and last recovery information. All information included in the file is generated and stored during backup and recovery operations.

When the database administrator requests a backup, our system first sets the current backup position in WAL. For example, as can be seen at the top of the figure, LSN_I is the backup position of the previous backup and LSN_N is the current backup position. After setting the backup position in WAL, we generate the corresponding backup information including the start position, end position, timestamp, and so on. Then we update backup information in the backup/recovery information file along with previous backup information.

When the administrator wants to change the state of the database to a certain point in the past, we restore the database by applying log data of WAL according to the backup information we stored in a file. Suppose that the current state of the database is S_X, and the administrator wants to change the database state to the past, S_N. Our system gets the corresponding backup information that the administrator wants to restore. Then the system applies log data from WAL to the database between the recovery range of that backup. In this example, the log data between LSN_J to LSN_N is in the recovery range,

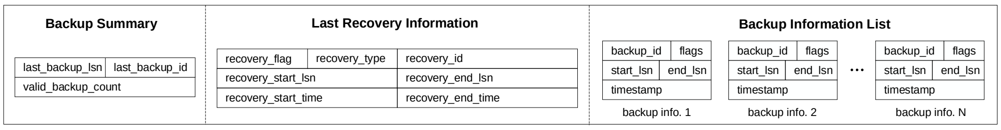


Figure 5.3: Backup/Recovery Information File Format

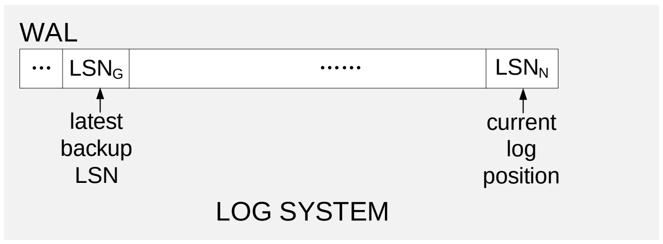
and we apply that data to the database to change the database state from S_X to S_N. To ensure data consistency when a system crash occurs during the recovery operation, we record the recovery status with corresponding recovery information to the backup/recovery information file at the beginning and the end of each operation.

We will explain each procedure in more detail in the following sections.

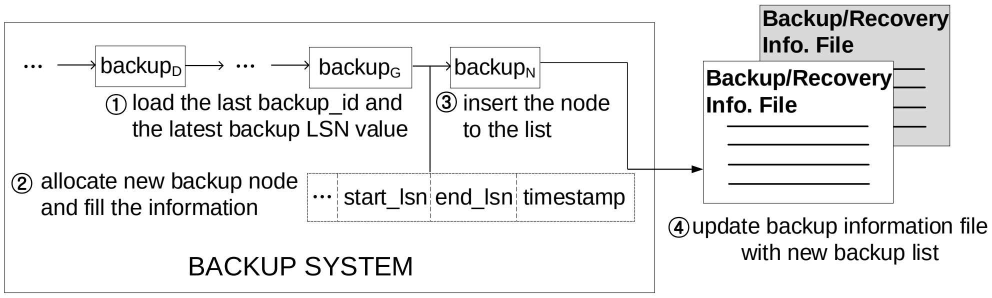
5.3.2 Design

Backup/Recovery Information Management

We maintain backup/recovery related information in a separate backup/recovery information file as shown in Figure 5.3. The figure presents the file format to store backup/recovery information. We create the backup/recovery information file in the backup directory to store information persistently. The file is composed of three parts, the backup summary, the last recovery information, and the backup information list. The backup summary and the backup information list contain information about all backups of the current database. We manage summarized backup information such as the number of valid backups, and backup information list that maintains data related to each backup. We also record information about the last recovery in the backup/recovery information file, such as the status (e.g., PROCESSING or FINISHED), the type (e.g., full or incremental), and so on. If a crash occurs during a recovery operation, the `recovery_state` of the file will remain recorded as PROCESSING, so at the



(a) Log System


···
$backup_D$
···
$backup_G$
$backup_N$
① load the last backup_id and the latest backup LSN value
③ insert the node to the list
② allocate new backup node and fill the information
··· start_lsn end_lsn timestamp
BACKUP SYSTEM
Backup/Recovery Info. File
Backup/Recovery Info. File
④update backup information file with new backup list

(b) Backup System

Figure 5.4: Backup Operations in Proposed Scheme

time restart the database system, we restore the database according to the last recovery information.

Backup Operations

The proposed scheme performs backup based on the physical backup strategy. Especially, we provide an efficient backup by exploiting log data of WAL. We first copy the entire files of the database to the designated backup directory to perform a full backup. To use log data as backup data, we design a structure to manage backup information as shown in Figure 5.3. We maintain summarized backup information such as the last LSN value of the latest backup, the latest ID assigned to the backup, and the number of valid backups. Meanwhile, data related to each backup are contained in the backup information list.

Figure 5.4 depicts the procedure of backup operations in the proposed scheme. When performing a backup, our backup system first sets the current log position in WAL and get the current LSN value of the log from the log system. Then, the backup system loads the last backup_id and the latest backup LSN value from the node at the end of the list (①). For each backup, the backup system assigns a unique ID and store the timestamp together to distinguish between different backups. Then, the system allocates a new backup node and fill the backup information (②).

To use log data of WAL as backup data, the backup system records the position of WAL at the time of each backup. To indicate the start and end of each backup, two LSN values are used. To identify the end of the backup, the backup system reads the LSN value of the last page flushed to WAL from the log system at the beginning of the backup operation. The LSN value guarantees that all previous pages have been flushed to WAL. The system stores the LSN value and uses the value as the end position of the current backup and start position of the next backup. As a start point for a backup, we use the LSN value from the previous backup, except for the full backup. Since the full backup has no preceding backups, the value zero (0) is used for the start point of the full backup. For example, in the case shown in the figure, LSN_G and LSN_N are the `start_lsn` and `end_lsn` of the backup, respectively.

We also define some flag values that represent information of the corresponding backup such as the validity, type (e.g., full or incremental), and so on. After all information of the corresponding backup is filled in the backup node, the backup system inserts the node to the end of the backup list (③). Then, the system updates the backup/recovery information file with the newly updated list along with the backup summary at the end of each backup operation (④).

To support backup using log data of WAL, we modify the existing WAL maintenance structure to keep the log data. Instead of overwriting log data

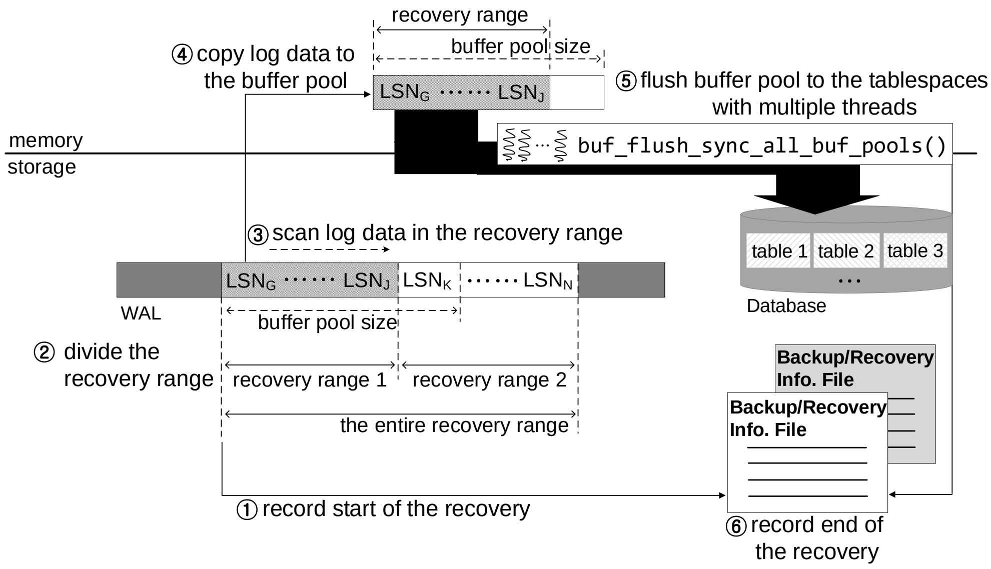

recovery range
buffer pool size
④ copy log data to the buffer pool
LSN$_G$ ······ LSN$_J$
⑤ flush buffer pool to the tablespaces with multiple threads
memory
storage
buf_flush_sync_all_buf_pools()
③ scan log data in the recovery range
table 1
table 2
table 3
Database
LSN$_G$ ······ LSN$_J$
LSN$_K$
······ LSN$_N$
WAL
buffer pool size
② divide the recovery range
recovery range 1
recovery range 2
the entire recovery range
Backup/Recovery Info. File
Backup/Recovery Info. File
① record start of the recovery
⑥ record end of the recovery

Figure 5.5: Recovery Operations in Proposed Scheme

to the existing WAL files, we create a new WAL file to store new log data. However, since we copy all files of the database as full backup, so we do not need to preserve log data of WAL in the full backup range. So, after a full backup is completed, WAL files that contain log data in the full backup range can be removed.

Recovery Operations

In the proposed scheme, restoring a full backup is performed by copying the entire files from the backup directory to the data directory. Figure 5.5 describes the proposed scheme for restoring an incremental backup using log data of WAL. We propose a recovery scheme by modifying the existing crash recovery procedure using WAL in MySQL.

Before starting the recovery operation, we record the corresponding recovery information to the backup/recovery information file to indicate that the recovery is started and is in progress (①). By recording the status of the latest recovery to the file, we provide data consistency by redoing recovery when the recovery procedure is not completed normally. Unlike the existing procedure, we first estimate the size of the entire recovery to prevent applying log data in the buffer pool to the data files by a single thread. If the estimated size exceeds the buffer pool size, we divide the entire recovery range so that a single recovery range does not exceed the buffer pool size (②). Then we start the recovery procedure of each recovery range. For example, in the case shown in the figure, we divide the entire recovery range from LSN_G to LSN_N in two. One range is from LSN_G to LSN_J and the other range is from LSN_K to LSN_N. Therefore, we start recovery from LSN_G to LSN_J corresponding to recovery range 1. We scan and read log data in WAL, as the existing procedure (③). While scanning log data between the recovery range in WAL, we copy log data to the buffer pool (④). However, we skip log data that are belonging to the system tablespaces such as MySQL user because we copy files containing system information from the backup directory. When all log data in the recovery range are loaded in the buffer pool, we stop the recovery procedure and flush the buffer pool by waking up the I/O threads those are sleeping (⑤). Since MySQL 8.0 supports scalable WAL [85], we leverage this feature to flush log data with multiple threads. After the data in the buffer pool are flushed to the tablespaces, we restart the recovery procedure of the next recovery range. In other words, we divide the whole recovery range into several independent recoveries and handle them sequentially. If the recovery procedure completes normally, we update the backup/recovery information file to indicate that the last recovery completes successfully (⑥).

5.3.3 Implementation

We implement our scheme in MySQL 8.0.15. We define a set of SQL queries to use the proposed scheme for backup and recovery of the database such as `CREATE_[FULL|INC]_BACKUP`, `RESTORE_[FULL|INC]_BACKUP`, `SHOW_BACKUPS`, and `DELETE_BACKUP`. Among these SQL queries, `RESTORE_[FULL|INC]_BACKUP` takes a `backup_id` as an argument from the user. The users acquire the `backup_id` as the return value of the backup operations or find it in the list of valid backups returned by the `SHOW_BACKUPS` query.

Backup Operations

We define several data structures to manage backup information and add them to the `log_t` data structure that manages the state of the redo log system of MySQL. We add atomic variables that represent the state of our backup system such as `backup_id`, `last_backup_LSN`, `backup_count` and so on. We guarantee the atomicity of such state variables by using the atomic template class provided by C++11. We manage each backup information with `backup_node_t` data structure that contains `backup_id`, `flags`, `start_lsn`, `end_ lsn`, `timestamp`, and so on.

When performing a backup, our backup system assigns a new ID to the corresponding backup. Then, the system allocates a new node to store corresponding backup information. At the beginning of each backup, we load the current value of `flushed_to_disk_lsn` from the `log_t` structure. This value is used as `end_lsn` of the corresponding backup and `start_lsn` of the next backup. The backup system sets the flags of the node and changes state to the valid, then inserts a backup node to the global `backup_info_list` that manages information of the entire backups. In the case of performing a full backup, we copy the entire files of the database to the designated backup directory whereas incremental backup copies the files containing system information. Before terminating the backup operation, we update the state of the backup system and update the backup/recovery information file to maintain the latest state persistently.

Recovery Operations

The recovery procedure is performed by passing the `backup_id` that wants to restore to the `RESTORE_[FULL|INC]_BACKUP` query. Before we perform a recovery procedure, we first pause MySQL service to not accept any transactions from the users by calling `log_stop_background_threads()`. Then we set the `restore_on` flag to inform that our recovery procedure is running. Since the contents of the backup/recovery information file are loaded at the time of MySQL startup, the recovery procedure reads the recovery range of the corresponding `backup_id`.

We first record the corresponding recovery information to the backup/recovery information file, such as type (e.g., full or incremental), ID, the `start_lsn` and `end_lsn`, and so on. At this time, the `recovery_flag` value is set to PROCESSING to indicate that the recovery operation has started and is in progress. The `recovery_flag` does not change until the recovery operation completes normally. Then we estimate the size of pages between `start_lsn` to `end_lsn`, which is the entire recovery range. If the estimated size is larger than the buffer pool size, we divide the entire recovery range so that data in each recovery range are smaller than the buffer pool size. For example, pages between LSN_G to LSN_N are the entire recovery range in Figure 5.5. In this case, we divide the recovery range 1 by LSN_G to LSN_J and recovery range 2 by LSN_K to LSN_N because the entire recovery range is larger than the buffer pool size. Recovery is performed sequentially for each range to preserve the data order.

For each recovery range, we scan and read log data from WAL. In contrast to the existing recovery procedure, we stop the recovery procedure and wake up the I/O threads to flush the buffer pool at the end of each recovery range. We repeat the recovery procedure until the last range is recovered. After restoring data in all recovery range is done, we resume MySQL service by calling `log_start_background_threads()`. When the recovery procedure for all recovery ranges completes normally, we set the `recovery_flag` value to FINISHED and update the backup/recovery information file.

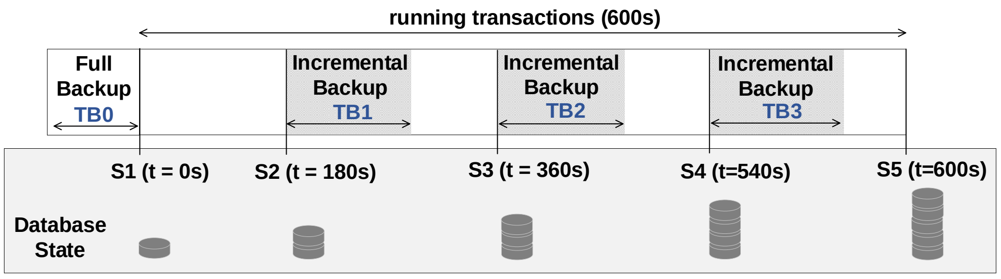
5.4 Evaluation

5.4.1 Experimental Setup

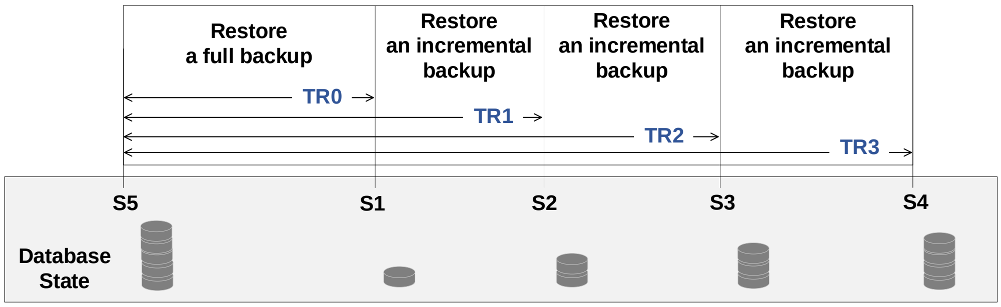
The machine we used in the following experiments is equipped with an Intel Xeon 2-way E5-2620 which has 16 physical cores with 16GiB memory. We use Samsung 860 PRO 256GB [86] as storage devices.

We use one SSD to evaluate the performance of each scheme when the data and backup directories are on the same storage device. On the other hand, we separate the data and backup directories physically by using two SSDs and evaluate backup and recovery performance.

We implement our scheme on MySQL 8.0.15 and evaluate its performance with `sysbench` OLTP_READ_WRITE workload [87]. We set the number of transactions to 35 million and run the workload for 600 seconds resulting in the entire data size 9GB approximately. We also run the same backup and recovery scenario with mysqldump [23] and Xtrabackup [22] and compare the performance.



(a) Backup Scenario



Restore a full backup
Restore an incremental backup
Restore an incremental backup
Restore an incremental backup
TR0
TR1
TR2
TR3
S5
S1
S2
S3
S4
Database State

(b) Recovery Scenario

Figure 5.6: Backup and Recovery Scenario

5.4.2 Performance Results and Analysis

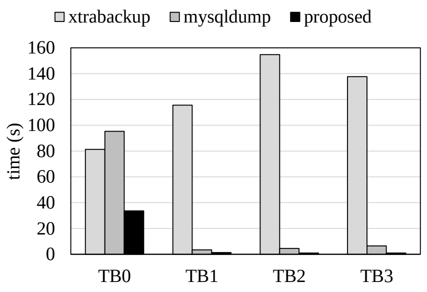
Backup and Recovery Scenario

In this section, we describe a synthetic scenario that we designed to evaluate the backup and recovery performance of the proposed scheme compared with the existing schemes. Figure 5.6 depicts a backup and recovery scenario used in the following experiments. In the scenario, we perform a full backup before running transactions. The transactions are executed for 600 seconds while executing an incremental backup every 180 seconds. As shown in Figure 5.6a, the database is growing from state $S1$ to $S5$ over time. The time for a full backup is denoted by $TB0$ and incremental backups are denoted by $TB1$ to $TB3$.

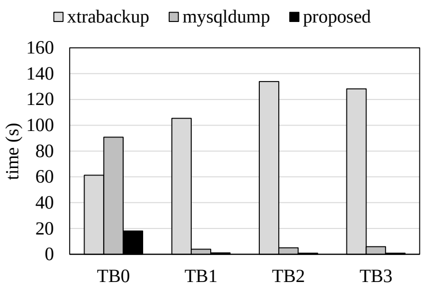
We present the entire recovery scenario in Figure 5.6b. To restore an incremental backup, all the previous backups must be restored sequentially. Therefore, we evaluate recovery time as the accumulated time for restoring the current backup. Starting at the database state $S5$, we restore a full backup and change the database state to $S1$. The time for restoring a full backup is denoted by $TR0$. The following incremental backups are restored sequentially and change the database state from $S1$ to $S4$. The entire time for restoring each incremental backup is denoted by $TR1$ to $TR3$. For example, in the case of restoring an incremental backup and changing the database state to $S2$, we first restore a full backup and then the first incremental backup in order. The recovery time is denoted by $TR1$, which is the total time for changing the database state from $S5$ to $S2$.

Backup and Recovery Performance

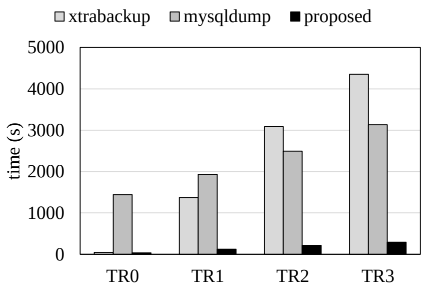
Figure 5.7 demonstrates the database backup and recovery time between different schemes. Figures 5.7a and 5.7b show the time for each backup with Xtrabackup, mysqldump and proposed scheme according to the backup scenario described in Section 5.4.2. As can be seen, the time for a full backup of existing schemes takes longer than that of the proposed scheme. In the case of the full backup, using two SSDs to physically separate the backup data from the data is 26% and 43% faster than using one SSD in Xtrabackup and the proposed schemes, respectively. Since Xtrabackup and the proposed scheme copy the files of the entire database to the designated backup directory for the full backup, so increasing the total bandwidth of the underlying storage device reduces backup time. Different from Xtrabackup, the proposed scheme spends less time for full backup because the proposed scheme performs no additional operations other than copying files. In contrast, mysqldump generates SQL query sequences by extracting table structures and data from the database using SELECT queries,



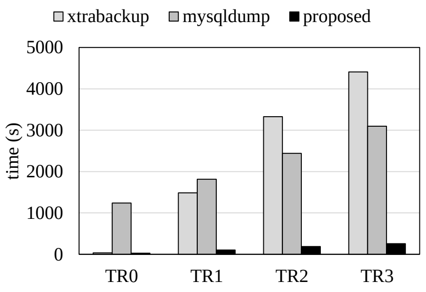
(a) Backup Time with One SSD



(b) Backup Time with Two SSDs



(c) Recovery Time with One SSD



(d) Recovery Time with Two SSDs

Figure 5.7: Backup and Recovery Time with SSDs

the total bandwidth of the underlying storage device does not affect the backup time.

However, in the case of the incremental backup, mysqldump and the proposed scheme show the instant backup speed. Xtrabackup takes the longest incremental backup time since it requires scanning the entire tables in the database and previous backup data to generate delta files. For mysqldump, the time for each incremental backup is 4/4, 6/5, and 8/6 seconds using one/two SSDs, respectively. Mysqldump uses binlog for incremental backups, which records that a backup event has occurred in the binlog file as described in Section 2.3. In the proposed scheme, each incremental backup takes only one second regardless of the number of the underlying storage device. The reason for performing the incremental backup instantly is that the backup operation is performed by utilizing log data of WAL without copying or comparing data.

Meanwhile, unlike backup time, mysqldump takes the longest time to restore a full backup. Figures 5.7c and 5.7d present the time for each recovery with Xtrabackup, mysqldump and proposed scheme according to the recovery scenario described in Section 5.4.2. Mysqldump interprets and executes SQL queries in the backup SQL sequences, whereas Xtrabackup and the proposed scheme copy back the files from the backup directory to the data directory. However, Xtrabackup takes more time as we restore incremental backups because it restores the database from the delta file page by page by a single thread. Different from Xtrabackup, the proposed scheme copies files containing system information and restores data that does not belong to it. In addition, the proposed scheme flushes the buffer pool with multiple threads by modifying the existing recovery procedure of MySQL. As a result, after restoring all the incremental backups, the recovery time of the proposed scheme is much faster than that of other schemes. Regarding recovery, using two SSDs to separate backup data reduces time to restore the full backup compared with using one SSD by 32% and 27% with Xtrabackup and the proposed scheme, respectively. Using two SSDs increases the total bandwidth of the underlying storage device, resulting in reduced data copy time.

In summary, the proposed scheme provides instant backup and fast recovery compared with the existing schemes. Using log data of WAL eliminates additional I/O operations such as data scanning or copying, so the proposed scheme provides instant backup time.

5.5 Summary

In this chapter, we propose an efficient backup and recovery scheme for database systems. Our key idea is using log data of write-ahead logging (WAL) of the database for backup and recovery. We develop a backup system that manages backup information by using log data of the existing system without additional I/O operations. Furthermore, we optimize the existing crash recovery procedure of WAL by utilizing multiple threads. We implemented and evaluated our scheme in a popular database system, MySQL. The experimental results demonstrate that the proposed scheme provides instant backup by eliminating additional I/O operations. Moreover, the proposed scheme provides fast recovery performance compared with the existing schemes.

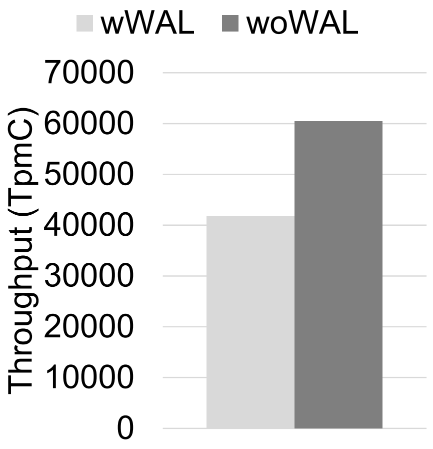
Chapter 6

Providing Data Durability in the Network Layer

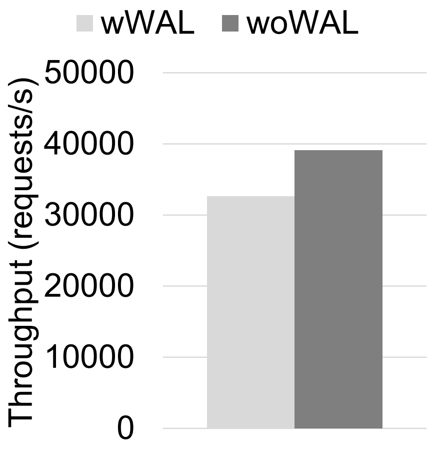
6.1 Overview

Write-ahead logging is a common technique in mission critical systems, such as database systems, file systems, and transaction processing systems, for ensuring data integrity and reliability [88–90]. However, to provide short-term data durability, most systems bear a high cost in terms of resource utilization and performance. For example, in MySQL [91], write-ahead logging file size is 48MB[1] to store 4KB page entries, and in RocksDB [92], write-ahead logging file size is 64MB[1] to store key-value pairs. Previous studies [45, 46, 93] claim that write-ahead logging in the critical path slows down system performance by up to 70%. Our experiments shown in Figure 6.1 also indicate that write-ahead logging in the critical path degrades throughput by 45% and 20% on average for transaction processing workload and social graph workload, respectively. In

[1]default configuration, configurable value



(a) Transaction processing



(b) Social graph

Figure 6.1: Throughput comparison with and without WAL

other words, we should pay indispensable costs (e.g., performance degradation, and additional resources) to provide short-term data durability that is rarely used.

Many researchers have investigated optimization techniques that persist log data on non-volatile memory to reduce logging overhead [44–49]. For example, NVWAL [48] introduced byte-granularity differential logging and user-level heap management to take advantage of the byte-addressability of NVM. NV-Logging [49] proposed NVRAM-aware per-transaction logging to efficiently exploit the byte-addressability of NVM. Different from these software logging approaches [48,49], Joshi et al. [46] and Shin et al. [47] suggested hardware log manager that provides logging operations out of the critical path. Despite the fact that several research efforts have been devoted to reduce logging overhead, logging operations still remain on the critical path (software logging) [48,49] or require special hardware instructions (hardware logging) [46,47].

In this chapter, we propose an in-transit logging (*ITLogging*) scheme to provide fault-tolerance by logging the important data in the network layer using deep packet inspection technique. Deep packet inspection is commonly used for network security or traffic management [51–62]. Unlike the common approaches, we employ deep packet inspection to identify the important data to be logged before being processed, allowing us to move logging operations out of the critical path without the help of special hardware or instructions. For recovery, we mimic clients using recorded packets of *ITLogging* and restore the database by making the server believe that clients are sending normal requests. *ITLogging* preserves the global ordering of data by replaying packets maintaining the order of arrival and distinguish packets based on their original source port number.

To demonstrate the effectiveness of the proposed scheme, we evaluate logging and recovery performance on the MySQL database system using two types of workload, transaction processing and social graph. We evaluate the performance of each workload while performing logging operations in the network layer through packet inspection using a dedicated core. Also, we measure the recovery time according to the logging duration. The experimental results show that *ITLogging* enables to log the important data without sacrificing workload performance by using separate cores.

In summary, our main contributions are as follows:

- We propose an in-transit logging scheme, *ITLogging*, that records the important data in the network layer using deep packet inspection technique.
- We provide fault-tolerance with almost no cost by mimicking a client on the target host system and replaying original requests.
- We demonstrate that *ITLogging* provides fault-tolerance in the database system using small amount of computing resources without adding any delays in the critical path.

6.2 Motivation

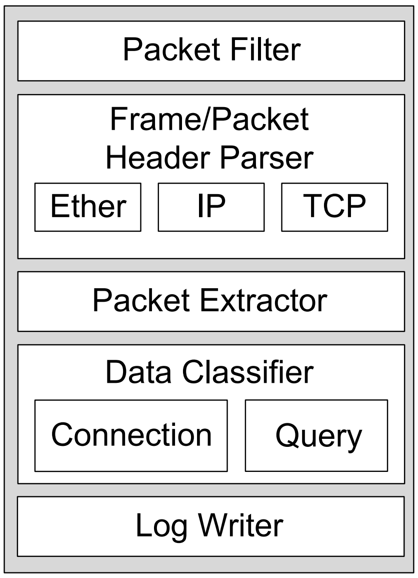
Logging is commonly used for providing fault-tolerance when unexpected system malfunctions occur. In most database systems, logging occurs during the processing of clients' write requests. However, performing logging operations during write request handling incurs delays which lead to performance degradation. According to several previous studies, in the case of transaction processing, logging operations degrade workload throughput by 40% on average and up to 70% even exploiting the characteristics of emerging non-volatile memory because logging operations occur in the critical path [46, 47]. On the other hand, RocksDB spends 68.1% and 81.0% of its overall write processing time on logging on a SATA and Optane SSD, respectively, according to SpanDB's analysis of write request processing stages [94].

We conduct preliminary experiments to check the performance degradation in MySQL caused by logging operations under different workloads. Figure 6.1 presents the difference in request processing throughput with and without logging operations under transaction processing and social graph workloads. As can be seen, logging operations during request processing decrease MySQL performance by up to 45%. That is, moving logging operations out of the critical path may result in performance improvements.

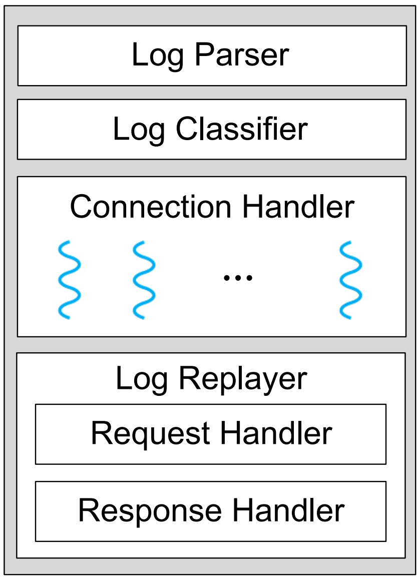
6.3 Design and Implementation

6.3.1 Design

In this section we introduce the design details for *ITLogging*, in-transit logging scheme for providing fault-tolerance by preserving the important data out of the critical path. We first introduce the logging procedure using the deep packet inspection technique and then explain how to perform recovery with preserved



(a) Logging Component



(b) Recovery Component

Figure 6.2: Component diagram for logging and recovery

data in the network layer. We then describe how to guarantee the correctness and global ordering in the recovery procedure.

Logging Procedure

The component diagram for preserving the important data using deep packet inspection is presented in Figure 6.2a.

In order to inspect whether the packet contains the data of the target application, we first analyze the header information by traversing each layer of the network stack, as explained in Section 2.4. All packets whose destination does not match the IP address and port of the target server system are filtered out by the *Packet Filter*. The *Frame/Packet Header Parser* inspects headers of ethernet, IP, and TCP by sequentially, leaving only the packets containing the data of the target application. In the ethernet and IP layers, we check the protocol of the encapsulated packet is IPv4 and TCP, respectively. We then check the payload length at the TCP layer, remaining only packets with payload lengths greater than zero and filtering out the rest; the packets whose payload length is greater than zero only contain the application data that we want.

The packets that have survived so far contain data to be used for recovery in the case of system malfunctions, thereby before further proceeding to inspect the payload, the *Packet Extractor* extracts some important values of the fields from the TCP header. The *Packet Extractor* extracts and records the following field values from the TCP packet: source port, sequence number, and payload length. The source port is used for isolating transactions from different clients during the recovery. If we can not distinguish which transaction each request belongs to, the consistency of the database can not be guaranteed. Assume there are two independent transactions executed by two different clients. If two transactions are guaranteed to execute independently, even if one is aborted by rollback, the other should not be affected. In such cases, if transaction isolation is not guaranteed, the database consistency will be broken; a transaction that should be aborted will be committed or a transaction that should be committed will be aborted. We record the source port information along with the payload data so that requests from different transactions can be distinguished during recovery, thereby preventing database consistency from being broken. The sequence number is used to preserve request orders between packets from the same client. The payload length is used for identifying the total length of the data that will be logged because payload data is variable length.

After completing the extraction of header fields required for recovery, the *Data Classifier* then inspects the payload of the packet and classifies packets into two types; those containing data for establishing the initial connection

Source Port (2B)	**Sequence Number** (4B)	**Payload Length** (4B)	**Payload Data** (*length* B)

Figure 6.3: Log Format

(*Connection*) and those containing the queries modifying the database state (*Query*). When the data of type *Connection* arrives, the *Data Classifier* first stores the data in a separate data structure, a hash table. In *ITLogging*, the log file is periodically truncated because we only keep log data for a short period of time. Thus, the data of type *Connection*, which is required for connection establishment, should be maintained separately and always be included whenever a log file is truncated and newly generated. In order to distinguish the connection data of different clients, the *Connection* type data is stored separately based on its source port; and then *Data Classifier* passes the data together with extracted information such as source port, sequence number, and payload length to the *Log Writer*. Unlike the *Connection* type data, *Query* type data is not reused and only needs to be logged, so when the *Query* type data arrives, the *Data Classifier* passes the data together with information extracted by the *Packet Extractor* to the *Log Writer*.

The final step of logging is persisting log data in the non-volatile storage medium. The *Log Writer* is responsible for storing the log data safely in the storage medium. A single log entry consists of source port, sequence number, payload length, and variable-length payload data, as shown in Figure 6.3. When all information composing a single log entry is transferred from the *Data Classifier*, the *Log Writer* first concatenates it into a single data stream. Then the *Log Writer* atomically writes a single log entry to the log file.

Checkpointing

Checkpointing is accomplished by periodically recording the processed packets. According to MySQL policy, we set the recording period to one second. Every second, the completed packet is recorded in a separate file for storing checkpointing information including packet number, time, and packet payload. We maintain two files for storing packets, each of which contains packets for ten seconds. The file size is configurable depending on the type of workload. For example, we configure the file size for storing packets for transaction processing, TPC-C, to 40 MB, whereas we configure it to 120 MB for social graph, LinkBench. Every file contains packets for initial connections through individual source port. We support partial recovery with this information, which avoids re-executing already completed packets and only processes packets that have been requested but not yet been processed.

Recovery Procedure

According to the logging procedure, the payload data of the original request packet is stored in the log file without any modification. That is, we can easily restore database status by sending the original requests. Therefore, we mimic the original clients on the host system, making the illusion of multiple clients sending requests to the server. The component diagram for mimicking the client that performs recovery is presented in Figure 6.2b.

The first step for recovery is read log entries from the log file. The *Log Parser* first reads the log file that stores log entries that need to be restored. The *Log Parser* loads log entries into the memory based on the log format depicted in Figure 6.3.

The *Log Classifier* identifies the type of the log entry, *Connection* or *Query*, which was classified by the *Data Classifier* during logging procedure. If the log entry is *Connection* type, it is collected separately based on the source port and used for connection establishment. If the log entry is *Query* type, the *Log Classifier* assigns a unique and global recovery sequence number. The *Query* type log entries are maintained independently for each source port.

The *Connection Handler* establishes distinct connections equal to the number of original connections, N_C. We launch N_C threads with *Connection* data of each source port and establish independent N_C connections within each thread. Individual connections are mapped to each source port and are used to replay requests sent from each source port to ensure transaction isolation between clients. After establishing a connection that corresponds to each source port within each thread, the connection information, such as socket descriptor, is mapped and maintained one by one to each source port.

Because multi-threaded log replay introduces a high overhead for preserving global order between original requests, the *Log Replayer*, a key component of recovery, operates within a single thread. The *Log Replayer* is composed of two network packet handlers; the *Request Handler* and *Response Handler.* The *Request Handler* sends requests as the original client did by sending log data through the connection mapped to its source port according to the global recovery sequence number assigned by the *Log Classifier.* In order to guarantee the global order, the *Request Handler* does not send the next request until the response to the previous request in each transaction (i.e., each connection) arrives. In other words, within a single transaction (single connection), the request will be send after the preceding request processing is completed. When the server has completed processing the request, it sends a response in the same manner as it did in response to the original client request. The response is managed by the *Response Handler.* Unlike the original clients, recovery does not need

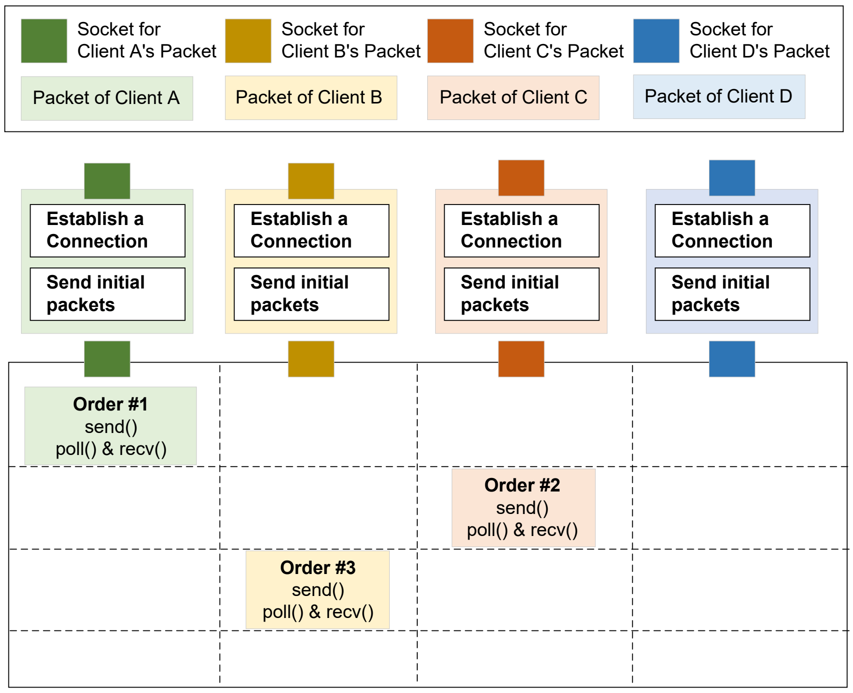


Figure 6.4: Recovery procedure by replaying original requests

to process response data since recovery aims to restore the database server's state. However, in order to confirm that the request has been processed by the server and to proceed with subsequent requests, the response data should be read from the TCP buffer.

The *Log Replayer* performs recovery by replaying log entries one by one, just like the original request execution. The *Log Replayer* consists of the *Request Handler* and *Response Handler*. The *Request Handler* replays log entries one by one. The *Request Handler* sends only one request pertaining to a specific connection across all connections. That means log entries are restored one by one sequentially. Assume that original requests have performed with four distinct connections as shown in Figure 6.4. First, the request originating from client

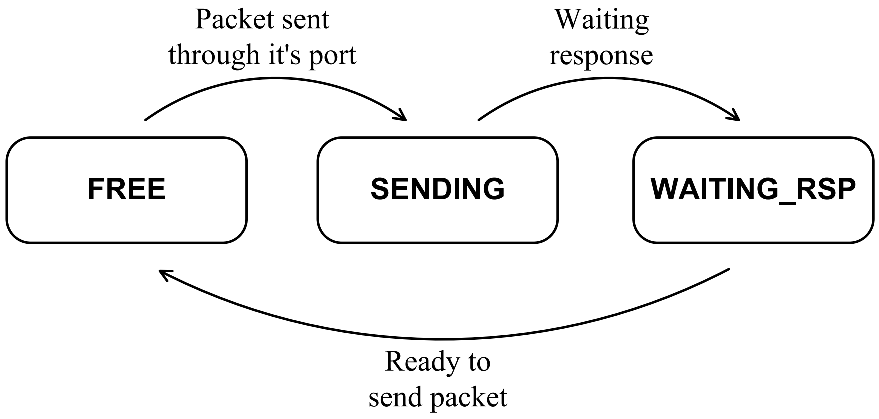


Figure 6.5: Finite state machine of connection

A is sent through the corresponding connection to the server and waits until the response arrives. Other connections are idle while the first log entry is being processed. After the first log entry has been processed, the subsequent log entry, whose global recovery sequence number is two, is sent through the corresponding connection to the server. Until the last log entry, all log entries are processed in the same way, one by one.

Figure 6.5 describes the finite state machine for each connection. There are three states a single connection can have: FREE, SENDING, and WAITING_RSP. All connections are set to a FREE state initially. During recovery, a log entry belongs to a connection can be processed through the connection with the FREE state; when the request is submitted, the connection's state changes to SENDING. Then the *Response Handler* waits and processes corresponding response. When response processing is completed, the state of the connections is changed to FREE and the *Request Handler* continues restoring the database.

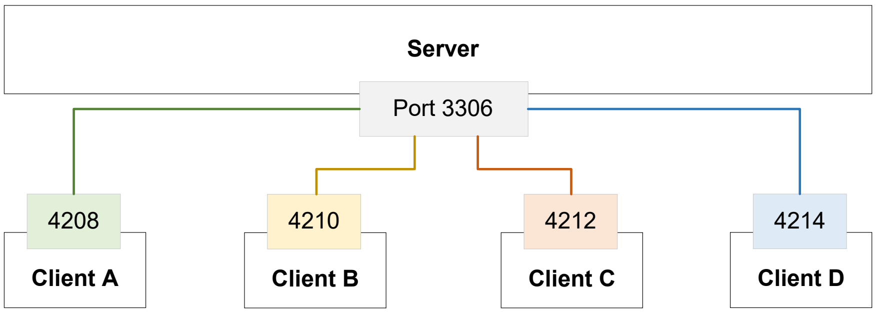


Figure 6.6: Example of packet serialization on the server system

Global Ordering

Usually, service server provides a single port to receive requests from multiple clients. For example, MySQL receives requests through port 3306 (default configuration). That is, the request packets from the number of clients are serialized in the server system's service port. All the requests sent from multiple clients are serialized in the order they arrived at the server port as in the example shown in Figure 6.6. Therefore, our scheme guarantees global ordering of the original requests by sending packets in the order they captured on the server side during recovery. At the time of recovery, the *Log Classifier* assigns a unique number, the global recovery sequence number, to each *Query* type log entry based on the entry's sequence number, source port, and the written order in the log file. The *Log Replayer* sends requests to the server according to its global recovery sequence number.

However, ensuring global order between requests from multiple clients introduces overhead of restoring the database during recovery procedure. We will explore the overhead caused by ensuring global ordering in Section 6.4.2

6.4 Evaluation

6.4.1 Experimental Setup

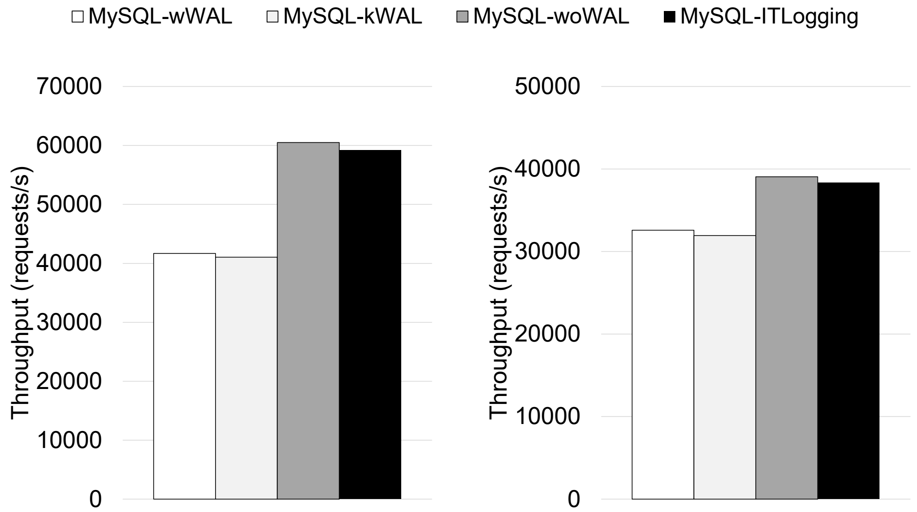
We evaluated the proposed in-transit logging scheme using two machines directly connected by a 10Gbps network. Each machine is equipped with an Intel Xeon W-2245 3.9GHz processor (16 cores with hyper threading), 32GB memory, and Intel P4510 4TB NVMe SSD. We used Ubuntu 18.04 with Linux Kernel 5.13 and ext4 file system, with MySQL 8.0. We evaluated our scheme using two workloads, transaction processing and social graph workload. For transaction processing workload, we used TPC-C [95] implementation, tpcc-mysql, developed and provided by Percona [96]. For social graph workload, we used LinkBench [97] on MySQL with MyRocks [98] storage engine.

6.4.2 Performance Results and Analysis

Logging Performance

We measure and compare the performance of transaction processing and social graph workload on the following MySQL configurations.

- **MySQL-wWAL** represents unmodified MySQL server that providing fault-tolerance using write-ahead logging.
- **MySQL-kWAL** represents MySQL server that keeps log data for backup and recovery (proposed scheme in Chapter 5).
- **MySQL-woWAL** represents unmodified MySQL server but does not provide fault-tolerance by disabling write-ahead logging. Transaction throughput under **MySQL-woWAL** gives us an ideal performance baseline.



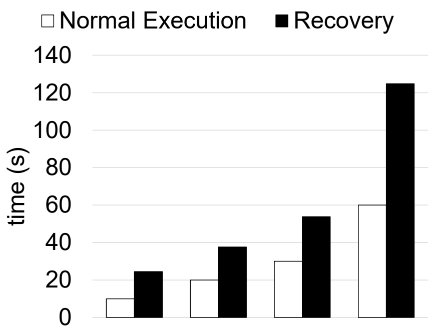
(a) Transaction processing (b) Social graph

Figure 6.7: In-transit logging throughput comparison

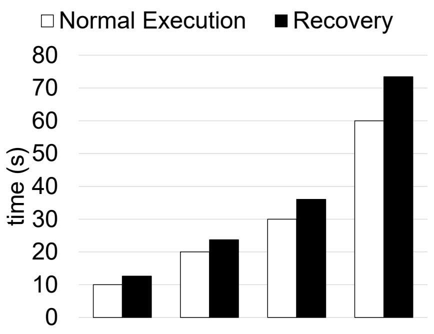
- **MySQL-ITLogging** represents that the proposed logging scheme is used for providing fault-tolerance.

For transaction processing workload, we run TPC-C transaction for 600 seconds using 16 clients and measure transaction throughput of different MySQL configurations. Figure 6.7a shows transaction throughput of TPC-C on different MySQL configurations. The throughput improves by 1.45x when MySQL does not perform write-ahead logging (**MySQL-woWAL**). With *ITLogging*, MySQL server does not need to perform write-ahead logging since we guarantee fault-tolerance by logging the important data in the network layer using deep packet inspection. Moving procedures for providing fault-tolerance out of the critical path results in throughput improvement, and the proposed scheme improves throughput by 1.42x compared to the baseline (**MySQL-wWAL**), as shown in the figure.

For social graph workload, we run 500,000 requests using 16 clients and mea-



(a) Transaction processing



(b) Social graph

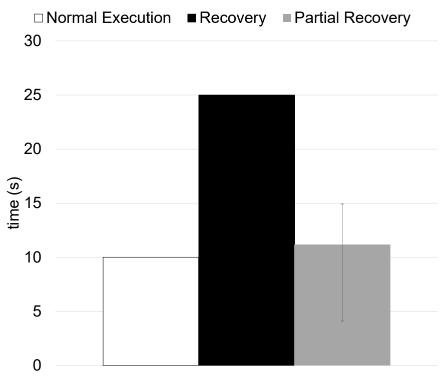
Figure 6.8: Recovery time comparison

sure the number of requests processed per second. Figure 6.7b shows LinkBench throughput on different MySQL configurations. The throughput improves by 1.20x when MySQL does not perform write-ahead logging (**MySQL-woWAL**). The proposed scheme improves social graph processing throughput by 1.18x compared to the baseline (**MySQL-wWAL**), as shown in the figure.

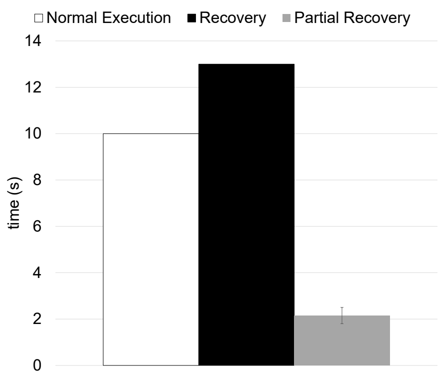
The overhead incurred by packet inspection is under 2%, which is negligible, in both transaction processing and social graph workloads. We also measure the throughput of two workloads while keeping log data for backup and recovery as proposed in Chapter 5. As shown in the figure, keeping write-ahead log data (**MySQL-kWAL**) incurs 2% throughput degradation on average compared with the baseline (**MySQL-wWAL**) on both workloads.

Recovery Time

We measure the recovery time of the proposed scheme under various sizes of log files. We measure the recovery time with log files of requests that are normally executed during 10, 20, 30, and 60 seconds. The results are shown in Figure 6.8. Recovery takes longer than normal execution because the proposed scheme



(a) Transaction processing



(b) Social graph

Figure 6.9: Partial recovery time comparison

guarantees global ordering and isolation between requests from different clients. However, recovery occurs rarely and only needs to be performed at system reboot after the system malfunctions.

Partial Recovery Time

Our scheme support partial recovery by leveraging checkpointing. With log files of requests that are normally executed during 10 seconds, we measure the partial recovery time. The results are shown in Figure 6.9. The error bars indicate minimum and maximum partial recovery time depending on trials. We measure 10 times of partial recovery for each workload. Partial recovery substantially reduces the recovery time by processing only packets that have not been processed after checkpointing.

Resource Utilization

Table 6.1 presents a single log file size according to logging duration. The size of the log file grows in proportion to the logging duration, as shown in the table.

	Transaction processing	Social graph
10s	40MB	114MB
20s	85MB	227MB
30s	128MB	362MB
60s	260MB	724MB

Table 6.1: Log file size

	Transaction processing	Social graph
MySQL-wWAL	12.24	11.66
MySQL-woWAL	11.16	11.77
MySQL-ITLogging-Server	10.84	11.38
MySQL-ITLogging-Logging	0.22	0.12

Table 6.2: CPU core utilization

Depending on the workload, the size of the log file may vary even for the same duration. For example, for requests of 10 seconds, the log file size of the transaction processing is 40MB whereas that of the social graph is 114MB, which is 2.85 times bigger. This is because the encoding format of each workload's request is different. For social graph workloads, clients send query statements, but for transaction processing workloads, clients send encoded data using MySQL command.

To minimize the performance interference (e.g., context switch) due to logging operations, we allocate a separate core to inspect packets and preserve data in the network layer. Table 6.2 shows the required number of cores on average under different MySQL configurations. As can be seen, the proposed scheme requires less than 25% of a single core to provide fault-tolerance for database systems; the average number of cores required for transaction processing and social graph workload is 0.22 and 0.12, respectively. In other words, the proposed scheme guarantees fault-tolerance for database systems while using less than 2% of the total CPU usage.

Correctness

To verify the correctness of recovery by the proposed scheme, we performed the entire database comparison by performing select queries to each table. We compared all the data stored in each table of the database as a result of normal execution and all the data stored in each table of the database restored by the log file generated by the proposed scheme, which is generated by inspecting packets in the network layer. We performed row-by-row comparisons of all data using the `diff` utility. The results are completely matched, which means the proposed scheme ensures database correctness.

6.5 Summary

In this chapter, we propose an in-transit logging (*ITLogging*) scheme using deep packet inspection technique. Our key idea is providing fault-tolerance without any delay in the critical path by inspecting packets in the network at the destination. *ITLogging* enables to log the important data with negligible performance overhead by inspecting incoming packets in the network layer before being processed. Our proposed scheme guarantees fault-tolerance by delivering original requests mimicking clients on the target host system. The experimental results demonstrate that *ITLogging* can improve workload performance while providing fault-tolerance only with small amounts of computing resources.

Chapter 7

Discussion

7.1 Providing Data Durability for Storage and Database Systems

We proposed three optimization schemes to provide data durability efficiently on different software layers. First, for the storage layer, we presented a concurrent and robust end-to-end data integrity verification scheme by reordering I/O operations and overlapping them with checksum computations. Second, on the application layer, we presented an efficient backup and recovery scheme by exploiting write-ahead logging. Finally, we presented an in-transit logging scheme that performs logging operations on the network layer by inspecting packets to overcome the limitations of archiving log data. In this section, we will discuss about how the proposed schemes can be integrated into a single system.

Suppose a scenario that transfers a large amount of data and uses a database to organize the information for archiving them. First, we can consider scheduling I/O operations with database system query processing operations. To provide robust and efficient data durability for storage and database systems, we overlap and reorder I/O operations for storing data in storage systems while processing queries for indexing data on the database system. Second, we may consider delaying checksum computation within the database system itself after data is stored in the storage device's final destination. Database systems, such as MySQL, have their own checksum computation to ensure data durability within the system. We could delegate checksum computation to the database system, allowing storage systems to focus on I/O operations in the correct order. Putting them all together, we are able to provide data durability for storage and database systems by delegating checksum computation to the database system, scheduling I/O operations to perform integrity verification after data is stored in the final destination, and overlapping them. Furthermore, by supporting logging operations through packet inspection on the network layer, we are able to support robust and efficient data durability on different software layers at the same time.

7.2 Limitations

In previous chapters, we demonstrated how our schemes work to efficiently provide data durability on different software layers. Although we address the problems raised in previous chapters, such as long data processing times and additional I/O operations, some limitations remain.

In the case of providing data durability for the storage layer, we utilize idle CPU resources to reduce the entire data transfer time including integrity verification. Our scheme requires at least three physical CPU cores to handle each operation without interfering with each other in order to overlap I/O operations with checksum computations. Our scheme involves more than three dedicated physical CPU cores to handle operations efficiently, depending on the number of concurrent verifiers and data sizes. For example, in the scenario described in section 4.4, the maximum CPU utilization is 800% when eight concurrent verifiers are launched, implying that we use eight physical CPU cores at the same time. This is due to the fact that we divide and assign specific data ranges to dedicated CPU cores for concurrent verification. In the future, we expect that computational storage devices with built-in processors and memory will allow us to offload checksum computations to storage devices, reducing the number of CPU cores required and allowing them to be used for other purposes.

For the database systems, the proposed scheme for providing data durability on the application layer successfully eliminates additional I/O operations for backup and recovery. However, as previously stated, archiving log data degrades system performance, and logging operations are still taking place on the critical path. To address the problem, we introduced an in-transit logging scheme in Chapter 6. Although successfully demonstrating that our scheme allows us to remove logging operations from the critical path, we do have some limitations. Our scheme dedicates CPU cores to inspect incoming packets without affecting system performance. Despite the fact that the number of CPU cores required for inspection is small, core separation efforts are necessary. In the future, we may consider employing programmable network interface cards to delegate inspection of incoming network packets to overcome such limitations.

Chapter 8

Conclusion

In this dissertation, we presented three optimization schemes to efficiently provide data durability on different software layers: the storage, application, and network. By leveraging idle resources and exploiting ready-to-use data including write-ahead log and incoming packets, we efficiently provide data durability for storage and database systems.

For the storage layer, we first explored the internal structure and operations of flash-based storage devices and analyze the bottleneck of the data transfer procedure including verification. We identify two problems: the internal operations of the storage device are not sufficiently considered in existing data integrity verification schemes and the checksum computation takes more than 50% of the time in the entire procedure. We presented a concurrent and robust end-to-end data integrity verification scheme that employs idle CPU resources to reorder I/O operations and parallelize checksum computations to provide robust data durability without increasing data processing time. On the application layer, we investigated existing database backup and recovery techniques and identify that existing schemes require additional I/O operations for extracting data from the database or copying the entire database files. We presented an efficient database backup and recovery scheme by exploiting write-ahead logging (WAL) to eliminate additional I/O operations. Archiving log data for backup, on the other hand, still requires logging operations on the critical path and introduces additional management overheads. To mitigate the overhead by archiving log data, we introduced an in-transit logging scheme, inspired by the fact that important data usually comes from external sources through the network. We preserve important data in the network layer by inspecting incoming packets and restore them by sending original requests through emulated clients. Experimental results under realistic scenarios using several benchmarks show that the proposed schemes efficiently provide data durability for storage and database systems on different software layers by leveraging idle resources and ready-to-used data.

Although we successfully provide data durability without increasing data processing times or additional I/O operations, there are some limitations. As aforementioned in section 7.2, the proposed schemes require additional computing resources (i.e., higher average and total CPU utilization) and resource management (i.e., core separation). One of the possible future research directions is offloading procedures for data durability to programmable hardware, such as computational storage devices or programmable network interface cards (a.k.a. SmartNIC), to reduce CPU overhead. Computational storage devices, such as SmartSSD [99], enable storage functions to be offloaded to storage devices without transferring data between storage devices and host systems. Relying on built-in processors and memory, we expect that storage system to be able to provide a higher level of robustness in procedures of integrity verification without extra computing resource consumption. This allows the storage system to concentrate on its main I/O operations rather than I/O and computation for integrity verification, which is expected to improve system performance. Like storage systems, database systems can also employ programmable hardware, such as SmartNIC, to offload functions to ensure data durability without additional CPU consumption or management. For example, the proposed scheme that provides data durability on the network layer, in-transit logging, enables filtering and storing incoming packets in the network device without separating cores to inspect packets. We believe that adopting programmable hardware will provide more strong data durability without requiring additional resources.

Bibliography

[1] A. S. Andrae, "Prediction studies of electricity use of global computing in 2030," *International Journal of Science and Engineering Investigations*, vol. 8, no. 86, pp. 27–33, 2019.

[2] A. S. Andrae and T. Edler, "On global electricity usage of communication technology: trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.

[3] "What Does 11 Nines of Durability Really Mean? - Wasabi." https://wasabi.com/blog/11-nines-durability/, 2022 (accessed 10 October 2022).

[4] H. Kim, I. Hwang, J. Lee, H. Y. Yeom, and H. Sung, "Concurrent and Robust End-to-End Data Integrity Verification Scheme for Flash-Based Storage Devices," *IEEE Access*, vol. 10, pp. 36350–36361, 2022.

[5] H. Kim, H. Y. Yeom, and Y. Son, "An efficient database backup and recovery scheme using write-ahead logging," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pp. 405–413, IEEE, 2020.

[6] H. Kim and H. Y. Yeom, "In-Transit Logging: Fault-Tolerance with Almost No Cost," *under review*.

[7] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha, "Vssim: Virtual machine based ssd simulator," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, 2013.

[8] D. He, F. Wang, H. Jiang, D. Feng, J. N. Liu, W. Tong, and Z. Zhang, "Improving hybrid FTL by fully exploiting internal SSD parallelism with virtual blocks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–19, 2014.

[9] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery," *arXiv preprint arXiv:1711.11427*, 2017.

[10] W. Xie and Y. Chen, "A Cache Management Scheme for Hiding Garbage Collection Latency in Flash-Based Solid State Drives," in *2015 IEEE International Conference on Cluster Computing*, pp. 486–487, 2015.

[11] M. Wajahat, A. Yele, T. Estro, A. Gandhi, and E. Zadok, "Distribution Fitting and Performance Modeling for Storage Traces," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 138–151, 2019.

[12] M.-K. Seo and S.-H. Lim, "Deduplication flash file system with PRAM for non-linear editing," *IEEE transactions on consumer electronics*, vol. 56, no. 3, pp. 1502–1510, 2010.

[13] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 957–968, 2012.

[14] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2015.

[15] D. Park and D. H. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, 2011.

[16] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design," in *2007 44th ACM/IEEE Design Automation Conference*, pp. 212–217.

[17] M. Murugan and D. H. Du, "Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, 2011.

[18] G. Wu and X. He, "Delta-FTL: improving SSD lifetime via exploiting content locality," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 253–266, 2012.

[19] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 177–190, 2015.

[20] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid, "SSD failures in datacenters: What? when? and why?," in *Proceedings of the 9th ACM International on Systems and Storage Conference*, pp. 1–11, 2016.

[21] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: anomalies, observations, and applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 24–33, 2009.

[22] "Percona XtraBackup - MySQL Database Backup Software." https://www.percona.com/software/mysql-database/percona-xtrabackup, 2022 (accessed 10 October 2022).

[23] "mysqldump." https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html, 2022 (accessed 10 October 2022).

[24] "The Binary Log." https://dev.mysql.com/doc/refman/8.0/en/binary-log.html, 2022 (accessed 10 October 2022).

[25] "A thorough introduction to eBPF." https://lwn.net/Articles/740157/, 2022 (accessed 10 October 2022).

[26] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture.," in *USENIX winter*, vol. 46, 1993.

[27] S. Ahmadian, F. Taheri, and H. Asadi, "Evaluating Reliability of SSD-Based I/O Caches in Enterprise Storage Systems," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[28] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[29] S. Jaffer, S. Maneas, A. Hwang, and B. Schroeder, "Evaluating file system reliability on solid state drives," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 783–798, 2019.

[30] “Globus.” https://www.globus.org/, 2022 (accessed 10 October 2022).

[31] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, “Towards optimizing large-scale data transfers with end-to-end integrity verification,” in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 3002–3007, 2016.

[32] E. Arslan and A. Alhussen, “A low-overhead integrity verification for big data transfers,” in *2018 IEEE International Conference on Big Data (Big Data).*

[33] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyoul, M. H. Gunes, and E. Arslan, “Towards securing data transfers against silent data corruption,” in *IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing, IEEE/ACM*, 2019.

[34] Z. Peterson and R. Burns, “Ext3cow: A time-shifting file system for regulatory compliance,” *ACM Transactions on Storage (TOS)*, vol. 1, no. 2, pp. 190–212, 2005.

[35] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-tree filesystem,” *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[36] P. Huang, K. Zhou, H. Wang, and C. H. Li, “BVSSD: Build built-in versioning flash-based solid state drives,” in *Proceedings of the 5th Annual International Systems and Storage Conference*, pp. 1–12, 2012.

[37] Y. Son, J. Choi, J. Jeon, C. Min, S. Kim, H. Y. Yeom, and H. Han, “SSD-assisted backup and recovery for database systems,” in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 285–296, IEEE, 2017.

[38] C. Morrey and D. Grunwald, “Peabody: The time travelling disk,” in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pp. 241–253, IEEE, 2003.

[39] Q. Yang, W. Xiao, and J. Ren, “Trap-array: A disk array architecture providing timely recovery to any point-in-time,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 289–301, 2006.

[40] S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Snapshots in a Flash with ioSnap,” in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[41] K. Sun, S. Baek, J. Choi, D. Lee, S. H. Noh, and S. L. Min, “LTFTL: Lightweight time-shift flash translation layer for flash memory based embedded storage,” in *Proceedings of the 8th ACM international conference on Embedded software*, pp. 51–58, 2008.

[42] G. Graefe, W. Guy, and C. Sauer, “Instant recovery with write-ahead logging: Page repair, system restart, and media restore,” *Synthesis Lectures on Data Management*, vol. 6, no. 5, pp. 1–85, 2014.

[43] T. Härder, C. Sauer, G. Graefe, and W. Guy, “Instant recovery with write-ahead logging,” *Datenbank-Spektrum*, vol. 15, no. 3, pp. 235–239, 2015.

[44] T. Wang and R. Johnson, “Scalable logging through emerging non-volatile memory,” *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[45] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis, “Rethinking logging, checkpoints, and recovery for high-performance storage engines,” in

Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 877–892, 2020.

[46] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 361–372, IEEE, 2017.

[47] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 178–190, 2017.

[48] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 385–398, 2016.

[49] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[50] K. Huang, Z. Shen, Z. Jia, Z. Shao, and F. Chen, "Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pp. 101–116, 2022.

[51] C. Xu, S. Chen, J. Su, S.-M. Yiu, and L. C. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2991–3029, 2016.

[52] M. Finsterbusch, C. Richter, E. Rocha, J.-A. Muller, and K. Hanssgen, "A survey of payload-based traffic classification approaches," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 1135–1156, 2013.

[53] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE communications surveys & tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[54] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *IEEE communications surveys & tutorials*, vol. 11, no. 3, pp. 37–52, 2009.

[55] "Cilium - Linux Native, API-Aware Networking and Security for Containers." https://cilium.io/, 2022 (accessed 10 October 2022).

[56] "Cilium." https://github.com/cilium/cilium, 2022 (accessed 10 October 2022).

[57] "MAC and Audit policy using eBPF." https://lwn.net/Articles/813057/, 2022 (accessed 10 October 2022).

[58] "Cloudflare architecture and how BPF eats the world.." https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/, 2022 (accessed 10 October 2022).

[59] "Open-sourcing Katran, a scalable network load balancer." https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, 2022 (accessed 10 October 2022).

[60] "Katran." https://github.com/facebookincubator/katran, 2022 (accessed 10 October 2022).

[61] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-defined scheduling across the stack," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 605–620, 2021.

[62] M. Kogias, R. Iyer, and E. Bugnion, "Bypassing the load balancer without regrets," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 193–207, 2020.

[63] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 487–501, 2021.

[64] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, *et al.*, "XRP:In-Kernel Storage Functions with eBPF," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 375–393, 2022.

[65] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 121–134, 2019.

[66] Y. Zhong, H. Wang, Y. J. Wu, A. Cidon, R. Stutsman, A. Tai, and J. Yang, "BPF for storage: an exokernel-inspired approach," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 128–135, 2021.

[67] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[68] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[69] Y. Hu, S. Song, S. Xiao, Q. Xu, N. Xiao, and Z. Qin, "A dominating error region strategy for improving the bit-flipping LDPC decoder of SSDs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 6, 2015.

[70] Y. Lee, L. Barolli, and S.-H. Lim, "Mapping granularity and performance tradeoffs for solid state drive," *The journal of supercomputing*, vol. 65, no. 2, pp. 507–523, 2013.

[71] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 266–277.

[72] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, "Durable write cache in flash memory SSD for relational and NoSQL databases," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 529–540, 2014.

[73] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end Data Integrity for File Systems: A ZFS Case Study.," in *FAST*, pp. 29–42, 2010.

[74] "XFS." https://wiki.archlinux.org/index.php/XFS, 2022 (accessed 10 October 2022).

[75] "Samsung PM983 Product Brief." https://samsungsemiconductor-us.com/labs/pdfs/Samsung_PM983_Product_Brief.pdf, 2022 (accessed 10 October 2022).

[76] S. Bhattacharya, C. Mohan, K. W. Brannon, I. Narang, H.-I. Hsiao, and M. Subramanian, "Coordinating backup/recovery and data consistency between database and file systems," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 500–511, 2002.

[77] G. Amvrosiadis and M. Bhadkamkar, "Identifying trends in enterprise data protection systems," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 151–164, 2015.

[78] G. Amvrosiadis and M. Bhadkamkar, "Getting back up: Understanding how enterprise data backups fail," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pp. 479–492, 2016.

[79] J. Kaiser, T. Süß, L. Nagel, and A. Brinkmann, "Sorted deduplication: How to process thousands of backup streams," in *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, IEEE, 2016.

[80] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High performance MySQL: optimization, backups, and replication.* " O'Reilly Media, Inc.", 2012.

[81] Y. Allu, F. Douglis, M. Kamat, R. Prabhakar, P. Shilane, and R. Ugale, "Can't We All Get Along? Redesigning Protection Storage for Modern Workloads," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 705–718, 2018.

[82] A. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting file systems: A survey of backup techniques," in *Joint NASA and IEEE Mass Storage Conference*, vol. 99, Citeseer, 1998.

[83] Y. Qin, B. Hoffmann, and D. J. Lilja, "Hyperprotect: Enhancing the performance of a dynamic backup system using intelligent scheduling," in

2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC), pp. 1–8, IEEE, 2018.

[84] Y. Allu, F. Douglis, M. Kamat, P. Shilane, H. Patterson, and B. Zhu, "Backup to the future: How workload and hardware changes continually redefine data domain file systems," *Computer*, vol. 50, no. 7, pp. 64–72, 2017.

[85] "MySQL 8.0: New Lock free, scalable WAL design." https://mysqlserverteam.com/mysql-8-0-new-lock-free-scalable-wal-design/, 2022 (accessed 10 October 2022).

[86] "Samsung SSD 860 PRO." https://semiconductor.samsung.com/consumer-storage/internal-ssd/860pro/, 2022 (accessed 10 October 2022).

[87] A. Kopytov, "Sysbench: a system performance benchmark," *http://sysbench.sourceforge.net/*, 2004.

[88] J. M. Hellerstein, M. Stonebraker, J. Hamilton, *et al.*, "Architecture of a database system," *Foundations and Trends® in Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[89] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: A Scalable Log-structured Database System in the Cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, 2012.

[90] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," *ACM SIGOPS operating systems review*, vol. 47, no. 1, pp. 9–15, 2013.

[91] "MySQL." https://www.mysql.com/, 2022 (accessed 10 October 2022).

[92] "RocksDB — A Persistent key-value store." http://rocksdb.org/, 2022 (accessed 10 October 2022).

[93] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 981–992, 2008.

[94] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A Fast,Cost-Effective LSM-tree Based KV Store on Hybrid Storage," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 17–32, 2021.

[95] "TPC-C Benchmark." http://www.tpc.org/tpcc/, 2022 (accessed 10 October 2022).

[96] "tpcc-mysql." https://github.com/Percona-Lab/tpcc-mysql, 2022 (accessed 10 October 2022).

[97] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: a database benchmark based on the facebook social graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1185–1196, 2013.

[98] Y. Matsunobu, S. Dong, and H. Lee, "MyRocks: LSM-tree database storage engine serving Facebook's Social Graph," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[99] "SmartSSD — SSD Card — Samsung Semiconductor Global." https://semiconductor.samsung.com/ssd/smart-ssd/, 2022 (accessed 10 October 2022).

요약

원격 시스템 간의 데이터 이동은 백업, 복제, 혹은 분석과 같은 다양한 목적으로 빈번하게 발생한다. 데이터는 하드웨어 오류 또는 시스템 충돌로 인해 전송 과정에서 손상되거나 손실될 수 있다. 많은 시스템들은 데이터 손상을 적시에 감지하기 위해 무결성 검증 또는 주기적인 백업과 같은 데이터 내구성을 보장하기 위한 절차를 제공한다. 이러한 절차들은 데이터 처리 시간을 지연시키거나 추가적인 I/O 작업을 필요로 한다.

본 논문에서는 데이터 내구성을 확보하는 과정에서 시스템 성능을 저하시키는 중복 작업을 제거하는 데 중점을 두어, 다양한 소프트웨어 계층 (저장장치, 응용, 그리고 네트워크 계층) 에서 데이터 내구성을 효율적으로 제공하기 위한 세 가지 최적화 기법들을 제안한다. 첫째, 저장장치 계층에서 데이터 내구성을 효율적으로 제공하기 위해서, 저장장치 내부 동작을 고려한 동시적이고 안정적인 종단 간 데이터 무결성 검증 기법을 제시한다. 유휴 CPU 리소스를 활용하여 무결성 검증 계산 과정을 병렬화 하고 I/O 작업과 중첩시킴으로써 안정적인 무결성 검증을 위한 I/O 순서 조정으로 야기되는 오버헤드를 완화한다. 둘째, 응용 계층에서 데이터 내구성을 효율적으로 제공하기 위해서, 데이터베이스 시스템의 미리 쓰기 로깅 (Write-ahead logging, WAL) 을 활용한 효율적 백업 및 복구 기법을 제시한다. 백업 및 복구를 위해 로그 데이터를 보관함으로써 추가적인 I/O 작업을 대부분 제거할 수 있다. 반면, 로그 데이터를 유지하는 것은 로그 데이터 양이 증가함에 따라 관리 부담이 증가하게 되고, 또한 여전히 critical path 상에서 로깅 연산을 수행함으로써 시스템 성능을 저하시키게 된다. 이러한 한계점을 극복하기 위해서, 목적지 시스템의 네트워크 계층에서 패킷을 검사하여 중요한 데이터 로깅을 수행하는 전송 중 로깅 (in-transit logging) 기법을 제시한다. 이 기법은 목적지의 로컬 클라이언트를 통해 원래의 요청들을 전달함으로써 내결함성 (fault-tolerance) 을

보장한다. 제안하는 기법들은 데이터 처리 시간 지연이나 추가적인 I/O 작업 없이도 다양한 소프트웨어 계층에서 데이터 내구성을 제공할 수 있다.

제안하는 기법들의 효과를 증명하기 위해서 실제 멀티 코어 시스템에 구현하여 성능을 평가하였다. 우리는 고성능 저장 장치가 장착된 시스템이 10 Gbps 로 연결된 환경에서 제안하는 기법들의 성능을 평가하고 기존의 기법들과 성능 비교를 수행하였다. 실험 결과는 제안하는 기법들이 유휴 리소스와 즉시 사용할 수 있는 데이터들을 활용하여 스토리지, 응용 및 네트워크 계층에서 효율적으로 데이터 내구성을 제공하면서 더 나은 시스템 성능을 제공할 수 있음을 보여준다.

주요어: 스토리지 시스템, 데이터베이스 시스템, 데이터 내구성, 무결성 검증, 미리 쓰기 로깅, 패킷 검사

학번: 2018-36618