IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 20, 2022, accepted March 22, 2022, date of publication March 31, 2022, date of current version April 8, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3163729

Concurrent and Robust End-to-End Data Integrity Verification Scheme for Flash-Based Storage Devices

HWAJUNG KIM[1], INHWI HWANG[1], JEONGEUN LEE[1], HEON Y. YEOM[1], (Member, IEEE), AND HANUL SUNG[2]

[1]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[2]Department of Game Design and Development, Sangmyung University, Seoul 03016, South Korea

Corresponding author: Hanul Sung (hanul.sung@smu.ac.kr)

This work was supported in part by the Ministry of Science, ICT (MIST), South Korea, under the National Program for Excellence in Software (SW), supervised by the Institute of Information and Communications Technology Planning and Evaluation (IITP), in 2022, under Grant 2019-0-01880; and in part by the National Research Foundation of Korea (NRF) Grant through the Korea Government (MSIT) under Grant NRF-2021R1A2C2003618 and NRF-2022R1G1A1011433.

ABSTRACT The amount of data generated by scientific applications on high-performance computing systems is growing at an ever-increasing pace. Most of the generated data are transferred to storage in remote systems for various purposes such as backup, replication, or analysis. To detect data corruption caused by network or storage failures during data transfer, the receiver system verifies data integrity by comparing the checksum of the data. However, the internal operation of the storage device is not sufficiently investigated in the existing end-to-end integrity verification techniques. In this paper, we propose a concurrent and reliable end-to-end data integrity verification scheme considering the internal operation of the storage devices for data transfer between high-performance computing systems with flash-based storage devices. To perform data integrity verification including data corruptions that occurred inside the storage devices, we control the order of I/O operations considering the internal operations of the storage devices. Also, to prove the effectiveness of the proposed scheme, we devise a prototype that injects faults on the specific layer of the storage stack and examines detection of faults. We parallelize checksum computation and overlap it with I/O operations to mitigate the overhead caused by I/O reordering. The experimental results show that the proposed scheme reduces the entire data transfer time by up to 62% compared with the existing schemes while ensuring robust data integrity. With the prototype implementation, our scheme detects failures on NAND flash memory inside storage devices that cannot be detected with the existing schemes.

INDEX TERMS Data transfer, I/O scheduling, parallel processing, data integrity.

I. INTRODUCTION

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 [6]. 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 [3]. 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 [2]–[5]. 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 [1], [12], [13]. Such flash-based storage devices have a cache layer, called a buffer cache, that buffers data [10], [11], [14], [15], [28]. When writing data to SSD, the data are cached in the buffer cache of SSD for a

The associate editor coordinating the review of this manuscript and approving it for publication was Ligang He.

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 [31]–[33]. 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.* [3] and Globus [5] overlapped a data transfer and checksum computation with file and block granularity, respectively. Arslan *et al.* [2] introduced FIVER, which cooperates during data transfer and integrity verification processes to reduce the entire data transfer time. Different from these approaches [2], [3], [5] that focus on reducing data transfer time, RIVA [4] 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 paper, we propose 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 [6], [9], [18]. 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.

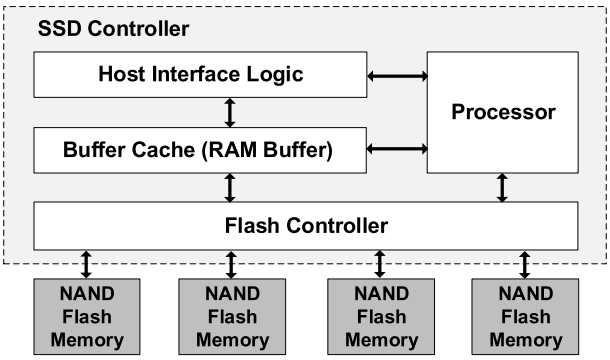


FIGURE 1. Internal structure of a flash-based SSD.

- 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.

The rest of this paper is organized as follows: Section II describes the background and motivation. Section III discusses the related works. Section IV presents the design and implementation of the proposed scheme. Section V presents the implementation details of the prototype that injects and detects faults on different layers of the storage stack. Section VI shows the experimental results. Finally, Section VII concludes this paper.

II. BACKGROUND AND MOTIVATION

In this section, we describe the internal structure and I/O operations of flash-based storage devices. Then, we provide background on data integrity verification procedures and present our preliminary experimental results that motivated our study.

A. THE INTERNAL STRUCTURE AND OPERATIONS OF FLASH-BASED SSD

The internal structure of flash-based SSD is illustrated in Fig. 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 [10], [11], [28].

When the host system requests writing data to the storage device, the SSD controller first caches the data in the

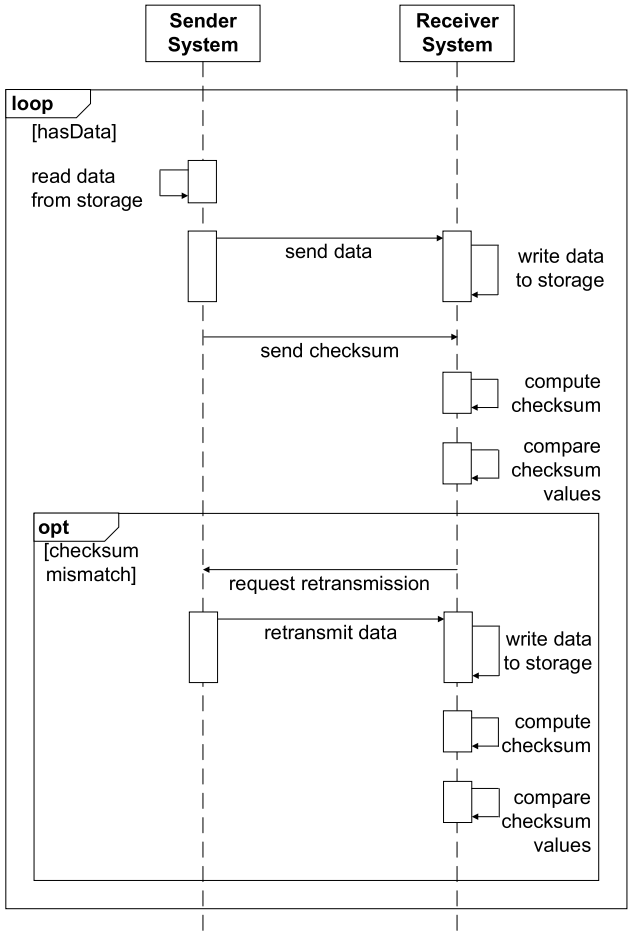


FIGURE 2. Data transfer procedure including verification.

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 [19]–[22], [24], [25]. In addition, repetitive data writes shorten the lifetime of SSD because the endurance of flash memory is limited [23]–[27]. Therefore, the data are 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 [31]–[33]. The internal failures of SSDs are relatively common, and the failure rates of various SSDs range from 4.2% to 34.1% [31]. 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 [32], [33].

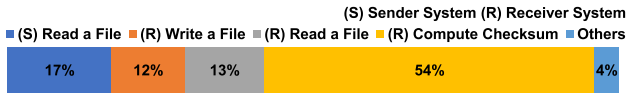


FIGURE 3. Major functions of the entire data transfer procedure including verification.

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.

B. DATA INTEGRITY VERIFICATION PROCEDURE

The existing implementation of data transfer including verification is shown in Fig. 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.

C. BOTTLENECK ANALYSIS IN DATA TRANSFER INCLUDING INTEGRITY VERIFICATION

To analyze the main bottlenecks in data transfer including integrity verification between remote systems, we performed a simple preliminary experiment. 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 [17] as the storage device. The detailed specifications of the sender and receiver system are described in Section VI. The analysis result is illustrated in Fig. 3. 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 account for 12% to 17% 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.

III. RELATED WORK

A. DATA INTEGRITY VERIFICATION

Several studies have analyzed how to provide data integrity verification efficiently. Globus [5] introduced file-level overlapping data transfer and checksum computation to optimize the entire data transfer time. Liu *et al.* [3] 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.* [2] 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.

Our scheme is similar to these approaches [2], [3], [5] in that it overlaps the data transfer and verification operations to optimize the large-scale data transfer time. However, our scheme uses multiple threads to optimize the verification operation itself. In addition, we focused on scheduling data I/O operations while considering the internal structure of SSD to perform an integrity verification of the data written to NAND flash memory.

To provide the integrity verification with the data stored in the storage device, Charyyev *et al.* [4] 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.

Our scheme is similar to that for RIVA in that it provides the integrity verification with the data stored on the storage device. However, our scheme schedules I/O operations and checksum computations by considering the internal structure of SSD so that we perform the integrity verification of the data written to NAND flash memory, which is the actual final destination.

B. DATA CORRUPTION INSIDE FLASH-BASED SSD

Several studies have investigated possible data corruption using the characteristics of flash-based SSDs. Ahmadian *et al.* [30] analyzed and investigated the various failures of SSD. They implemented a platform that detects physical failures inside SSD. Cai *et al.* [29] explored the fundamentals and recent research on flash-based SSD reliability.

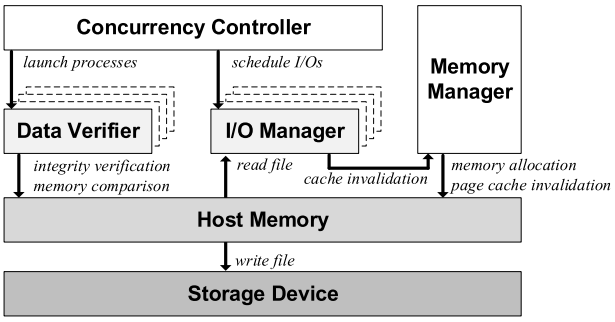


FIGURE 4. Overall architecture.

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.* [33] investigated the characteristics of several commodity SSDs in terms of performance, power, and reliability. Based on the characteristics, they performed application case studies, improving the performance while extending the lifetime of the storage devices.

Jaffer *et al.* [12] 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.* [31] 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.* [32] 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.

Our scheme is in line with these approaches [12], [29]–[33] in terms of investigating data corruption inside flash-based SSDs. However, these approaches concentrate on data corruption in a single system, whereas the proposed scheme focuses on detecting data corruption while transferring data over the network between remote systems.

IV. DESIGN AND IMPLEMENTATION

A. 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 [4] 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.

Fig. 4 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 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.

B. 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.

C. CONCURRENCY CONTROLLER

The *concurrency controller* is responsible for scheduling I/O operations and integrity verification procedures so that the

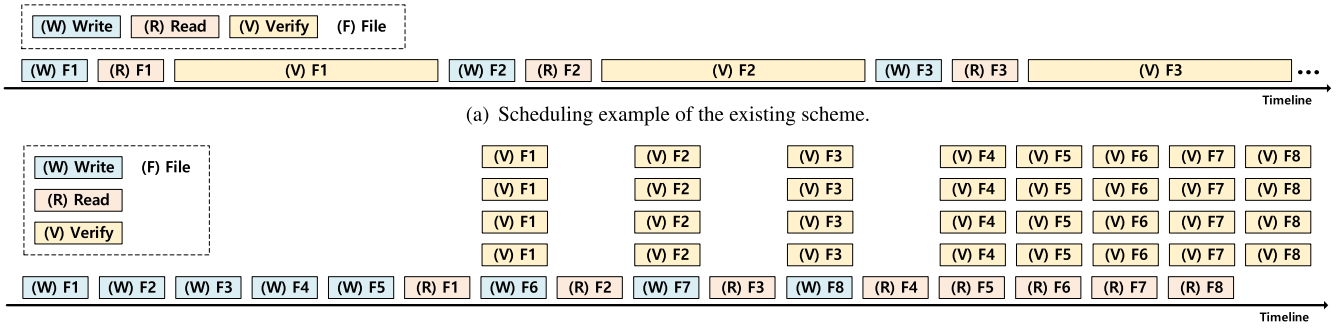


FIGURE 5. **Scheduling example of the existing and proposed schemes. Each box represents the execution of corresponding process, such as write a file to the storage device, read a file from the storage device, and verify integrity of the file. Each number indicates a different file.**

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*.

1) 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 [15]. 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.

Fig. 5 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 (Fig. 5(a)), 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 Fig. 5(b), 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.

2) 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 II-C, 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 Fig. 5(b), 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.

3) 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 II, 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 Fig. 5, 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 VI.

D. 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.

E. 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.

V. 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 Fig. 6. 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.

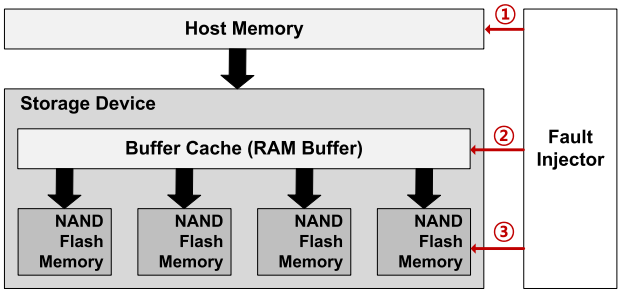


FIGURE 6. Prototype implementation to inject faults.

TABLE 1. Specification of the machines.

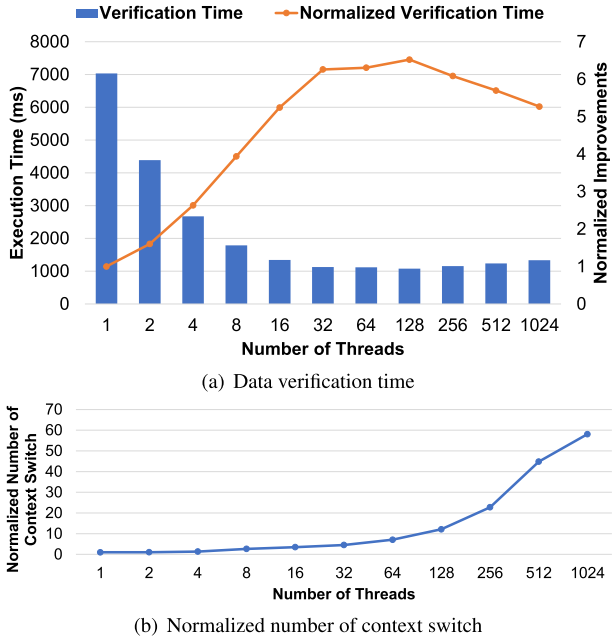
Processor	4-way Intel E7-8870@2.1GHz
Memory	256 GiB
Storage	Samsung PM983 NVMe SSD, 3.84TB
OS	Ubuntu 16.04
Kernel	kernel v4.4.0

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 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.

VI. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETUP

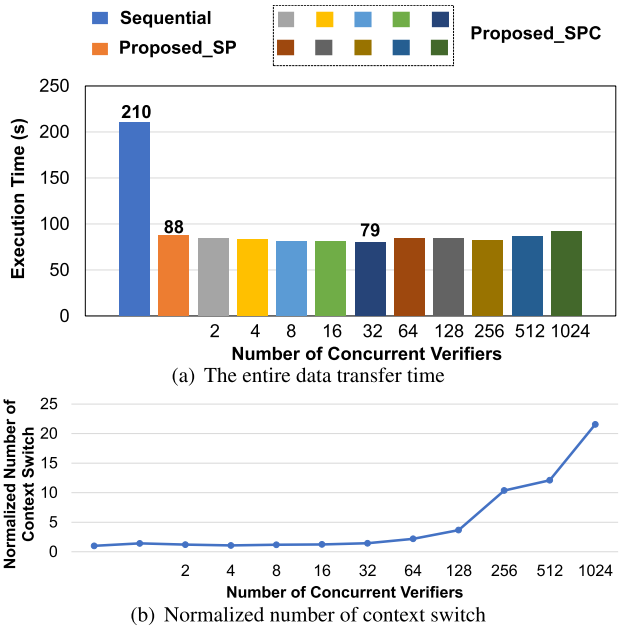
We evaluated the performance of our integrity verification scheme using two machines connected by a 10 Gbps network. Each machine has 72 physical cores that support



(a) Data verification time

(b) Normalized number of context switch

FIGURE 7. Data verification time.



(a) The entire data transfer time

(b) Normalized number of context switch

FIGURE 8. The entire data transfer time.

hyperthreading and is equipped with a 3.84 TB Samsung PM983 NVMe SSD. The detailed specifications of the machine are described in Table 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 20% 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 GiB 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 [3], and RIVA [4] which 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.

B. DATA VERIFICATION TIME

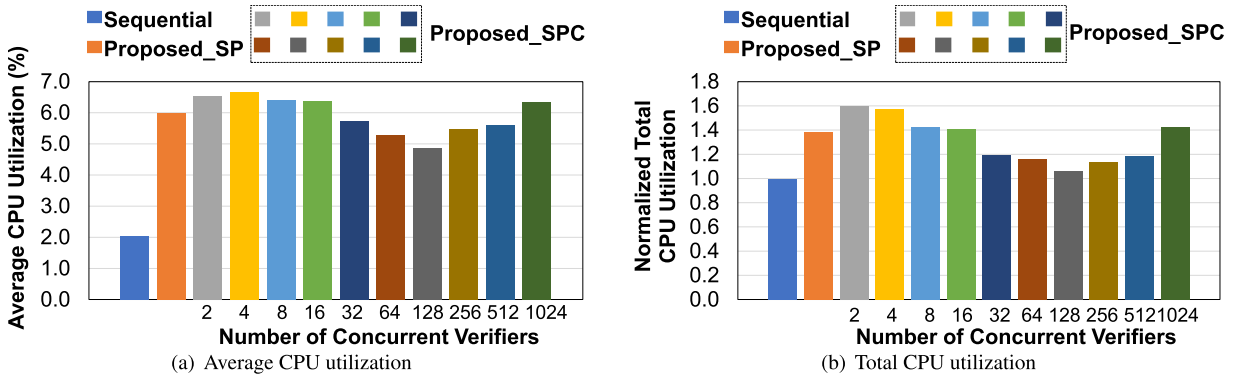
We used a single file of size 1 GiB to demonstrate the effectiveness of concurrent computation. In this experiment, we focused on the effectiveness of concurrent verification rather than other techniques, so we used the file already stored in the storage device. Fig. 7(a) presents the data verification 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 32. 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 32, 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. Fig. 7(b) shows the normalized number of context switches as the number of threads increases. As can be seen, until the number of threads reaches 32, the number of context switch increase is insignificant. However, the number of context switches increases exponentially when more than 128 threads are used to perform the verification. As a result, the verification time increases despite doubling the number of threads.

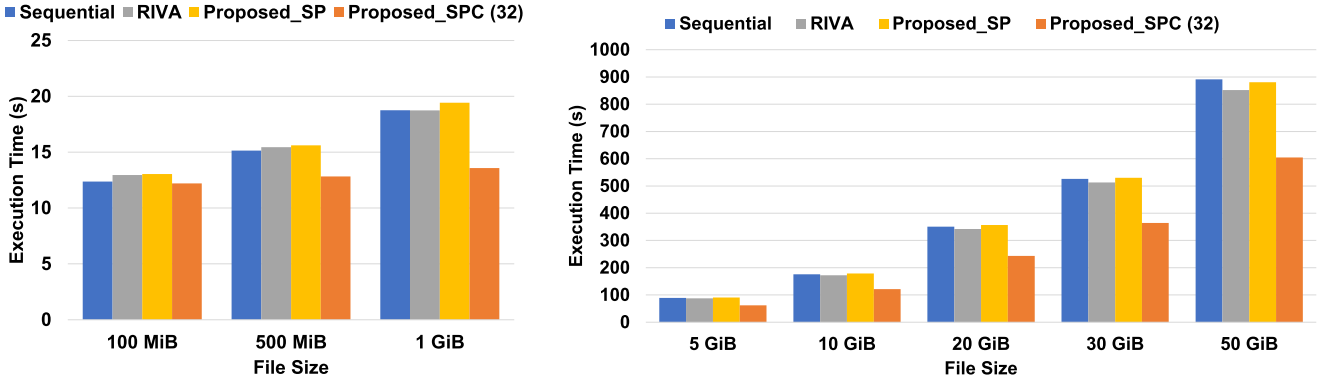
C. ENTIRE DATA TRANSFER TIME

Fig. 8(a) presents the entire data transfer time of the existing sequential scheme and the proposed scheme. The entire



(a) Average CPU utilization (b) Total CPU utilization

FIGURE 9. CPU utilization comparison.



(a) The entire data transfer time with file size under 5 GiB (b) The entire data transfer time with file size over 10 GiB

FIGURE 10. Data transfer time comparison changing file size.

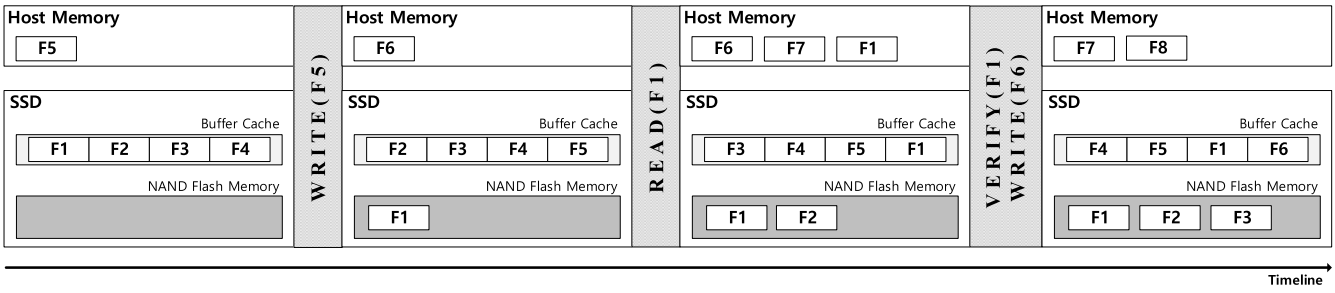


FIGURE 11. Host memory and SSD status changes over time.

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 58% with the Proposed_SP scheme compared with the Sequential scheme. Similar to the file-level pipelining introduced in the previous study [3], the overlapping computation and I/O operation reduced the entire transfer time by 58%. 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 62% and 10% compared with the existing sequential scheme (Sequential) and proposed scheme without concurrent verification (Proposed_SP), respectively.

The entire data transfer time gradually decreased until the number of concurrent verification threads reaches 32. As depicted in the figure, when we performed the integrity verification in parallel with 32 threads, the entire data transfer time was the most shortened, which is a 62% reduction compared with the existing sequential scheme.

However, using more than 32 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 Section VI-B. Fig. 8(b) reveals that the number of context switches increased rapidly when we used more than 64 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 increased.

D. 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 Fig. 9. Figs. 9(a) and 9(b) list the average and total CPU utilization for each scheme, respectively. The average CPU utilization of the proposed scheme is 6%, which is 3 times higher than that of the existing sequential scheme (2%). However, Fig. 8(a) indicates that the entire data transfer time of the existing sequential scheme is up to 2.7 times longer than that of the proposed scheme. As a result, the total CPU utilization of the proposed scheme is from 10% to 60% higher than that of the existing sequential scheme (Fig. 9(b)).

The difference in the gap depends on the number of concurrent verifiers. For example, using 32 concurrent verifiers, the proposed scheme completes the data transfer and verification 2.7 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 transfer and verification time by up to 62% by intensively investing computing resources in a short time.

E. ENTIRE DATA TRANSFER TIME WITH DIFFERENT FILE SIZES

Fig. 10 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) [4]. We used a single file of different sizes from 100 MiB to 50 GiB to demonstrate the effectiveness of concurrent verification with various file sizes. As shown in the aforementioned experiments, using 32 concurrent verifiers is the most efficient in terms of performance and resource utilization, we configured the number of data verifiers to 32 in subsequent experiments.

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 MiB. Different from the Proposed_SP scheme, the Proposed_SPC scheme is effective for data 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 GiB or more. The reason is that the Proposed_SPC scheme reduces the time of data verification by involving multiple threads to perform concurrent verification.

F. 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. Fig. 11 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-GiB 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.

G. 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

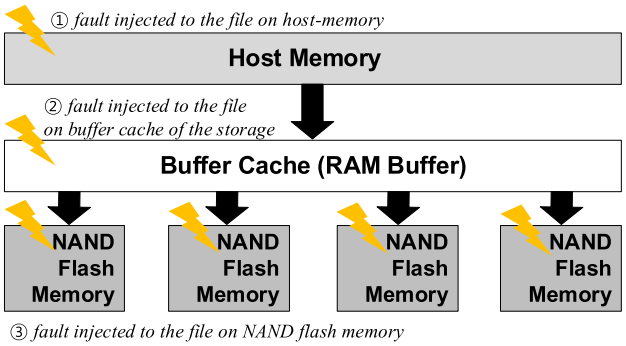


FIGURE 12. Robustness test by injecting faults in different data storage layers on the receiver system.

TABLE 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	✓	✓	✓

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 Fig. 12.

Table 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 in some layers. For example, RIVA [4] could not detect errors that occurred in NAND flash memory of SSD. On the other hand, file-level pipelining (File-LevelPpl) [3] 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.

VII. CONCLUSION

In this paper, we propose a concurrent and reliable 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.

REFERENCES

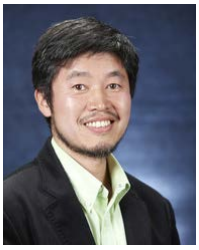
[1] Y. Hu, S. Song, S. Xiao, Q. Xu, N. Xiao, and Z. Qin, "A dominating error region strategy for improving the bit-flipping LDPC decoder of SSDs," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 6, pp. 578–582, Jun. 2015.
[2] E. Arslan and A. Alhussen, "A low-overhead integrity verification for big data transfers," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 4227–4236.
[3] 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 *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 3002–3007.
[4] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyoul, M. H. Gunes, and E. Arslan, "Towards securing data transfers against silent data corruption," in *Proc. 19th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2019, pp. 262–271.
[5] (2022). *Globus*. Accessed: Jan. 20, 2022. [Online]. Available: https://www.globus.org/
[6] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–32, 2013.
[7] J. E. Boritz, "IS practitioners' views on core concepts of information integrity," *Int. J. Accounting Inf. Syst.*, vol. 6, no. 4, pp. 260–279, Dec. 2005.
[8] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, vol. 2, 2007, pp. 21–33.
[9] 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 *Proc. FAST*, 2010, pp. 29–42.
[10] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choi, S. Yoon, and J. Cha, "VSSIM: Virtual machine based SSD simulator," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol. (MSST)*, May 2013, pp. 1–14.
[11] 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 Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 1–19, Jan. 2015.
[12] S. Jaffer, S. Maneas, A. Hwang, and B. Schroeder, "Evaluating file system reliability on solid state drives," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 783–798.
[13] Y. Lee, L. Barolli, and S.-H. Lim, "Mapping granularity and performance tradeoffs for solid state drive," *J. Supercomput.*, vol. 65, no. 2, pp. 507–523, Aug. 2013.
[14] 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 *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 266–277.
[15] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2014, pp. 529–540.
[16] (2022). *NVMe-Flush—Flush Command*. Accessed: Jan. 20, 2022. [Online]. Available: https://www.mankier.com/1/nvme-flush
[17] (2022). *Samsung PM983 Product Brief*. Accessed: Jan. 20, 2022. [Online]. Available: https://samsungsemiconductor-us.com/labs/pdfs/Samsung_PM983_Product_Brief.pdf
[18] (2022). *XFS*. Accessed: Jan. 20, 2022. [Online]. Available: https://wiki.archlinux.org/index.php/XFS
[19] W. Xie and Y. Chen, "A cache management scheme for hiding garbage collection latency in flash-based solid state drives," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2015, pp. 486–487.

[20] M. Wajahat, A. Yele, T. Estro, A. Gandhi, and E. Zadok, "Distribution fitting and performance modeling for storage traces," in *Proc. IEEE 27th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Oct. 2019, pp. 138–151.
[21] M.-K. Seo and S.-H. Lim, "Deduplication flash file system with PRAM for non-linear editing," *IEEE Trans. Consum. Electron.*, vol. 56, no. 3, pp. 1502–1510, Aug. 2010.
[22] 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 *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 957–968.
[23] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design," in *Proc. 44th ACM/IEEE Design Autom. Conf.*, Jun. 2007, pp. 212–217.
[24] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537–1550, Jan. 2016.
[25] D. Park and D. H. C. Du, "Hot data identification for flash-based storage systems using multiple Bloom filters," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–11.
[26] M. Murugan and D. H. C. Du, "Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–12.
[27] G. Wu and X. He, "Delta-FTL: Improving SSD lifetime via exploiting content locality," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 253–266.
[28] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery," 2017, *arXiv:1711.11427*.
[29] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proc. IEEE*, vol. 105, no. 9, pp. 1666–1704, Sep. 2017.
[30] S. Ahmadian, F. Taheri, and H. Asadi, "Evaluating reliability of SSD-based I/O caches in enterprise storage systems," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 4, pp. 1914–1929, Oct. 2021.
[31] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 1, pp. 177–190, Jun. 2015.
[32] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. M. Khessib, and K. Vaid, "SSD failures in datacenters: What? When? and Why?" in *Proc. 9th ACM Int. Syst. Storage Conf.*, 2016, pp. 1–11.
[33] J. K. Wolf, P. H. Siegel, E. Yaakobi, S. Swanson, J. D. Coburn, A. M. Caulfield, and L. M. Grupp, "Characterizing flash memory: Anomalies, observations, and applications," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2009, pp. 24–33.
[34] H. Kim, I. Hwang, and H. Y. Yeom, "Efficient and robust data integrity verification scheme for high-performance storage devices," in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, Mar. 2021, pp. 1199–1202.

HWAJUNG KIM received the B.S. degree in computer science and engineering from the Pohang University of Science and Technology, South Korea, in 2006, and the M.S. degree from the Department of Computer Science and Engineering, Seoul National University, South Korea, in 2018, where she is currently pursuing the Ph.D. degree in computer science and engineering. From 2006 to 2018, she was a Research Engineer at Samsung Electronics. Her research interests include operating systems, distributed systems, and database systems.

INHWI HWANG received the B.B.A. and B.S. degrees in computer science and engineering from Seoul National University, in 2020, where he is currently pursuing the M.S. degree in computer science and engineering. His research interests include operating systems and distributed systems.

JEONGEUN LEE received the B.S. degree in software convergence technology from Ajou University, in 2020. He is currently pursuing the M.S. degree in computer engineering with Seoul National University. His research interests include distributed computing and storage systems.



HEON Y. YEOM (Member, IEEE) received the B.S. degree in computer science from Seoul National University, in 1984, and the M.S. and Ph.D. degrees in computer science from Texas A&M University, in 1986 and 1992, respectively. From 1986 to 1990, he worked with the Texas Transportation Institute as a Systems Analyst, and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University, in 1993, where he currently is a Professor with the School of Computer Science and Engineering. He teaches and researches on distributed systems and transaction processing.

HANUL SUNG received the B.S. degree in computer science from Sangmyung University (SMU), Seoul, South Korea, in 2012, and the M.S. and Ph.D. degrees in computer science and engineering from Seoul National University (SNU), Seoul, in 2020. She is currently an Assistant Professor with the Department of Game Design and Development. Her main research interests include distributed systems, operating systems, high performance storage systems, and cloud computing.