IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 4, 2022, accepted January 22, 2022, date of publication January 27, 2022, date of current version February 8, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3146852

An Efficient Block Address Transformation Scheme in Block Layer for Flash-Based SSDs

JAEHYUN HAN AND YONGSEOK SON
School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Yongseok Son (sysganda@cau.ac.kr)

This work was supported in part by the Chung-Ang University Research Grant, in 2021; and in part by the National Research Foundation of Korea under Grant 2021R1C1C1010861.

ABSTRACT Flash-based solid state drives (SSDs) are widely adopted in both industry and the academia since SSDs offer higher performance, lower latency, and lower power consumption compared with traditional hard disk drives (HDDs). Unfortunately, the performance of SSDs can be affected by I/O access patterns. For example, random I/O operation degrades the SSD performance compared with sequential I/O operation since the random I/O operation reduces the spatial locality and increases garbage collection (GC) overhead. To handle this issue, in this article, we propose an efficient block address transformation scheme in the block layer to improve both performance and portability. To do this, we first transform random access patterns to sequential access patterns by sequentializing the block addresses in the block layer. Second, we devise a mapping table for managing transformed block addresses. Third, we support correct read operations and transaction processing for updating the mapping table to avoid sacrificing the consistency. Finally, we provide a cleaning scheme to reclaim invalid data generated by the transformed sequential access. This scheme increases spatial locality, reduces GC overhead, and operates well on any file systems or devices. Experimental results show the proposed scheme improves performance by up to 2x, 1.86x, 2.15x, and 42.8% compared with existing scheme in the case of diverse file systems such as EXT4, XFS, BTRFS, and F2FS, respectively.

INDEX TERMS Flash-based SSDs, block layer, I/O performance, garbage collection.

I. INTRODUCTION

Flash-based solid state drives (SSDs) offer several advantages over traditional hard disk drives (HDDs) such as higher throughput, lower access latency, lower power consumption, etc [9]. Thus, SSDs are becoming mainstream high performance storage devices in both industry and academia to improve I/O performance. Even though SSDs provide much higher I/O performance compared with HDDs, the performance of SSDs can be affected by access patterns and garbage collection (GC) from applications due to their specific features [18].

For example, sequential access is reading or writing of data in sequential order. In this case, there is high probability that sequential write requests have a similar lifetime inside the SSD [21], [28]. If a block is filled with pages with similar lifespan, it is also likely that all pages inside the block will be invalidated at a relatively similar time (i.e., high spatial locality [9]). Thus, the number of migrations for valid data during GC can be reduced, hence reducing write amplification within SSDs. On the contrary, in the case of random access, spatial locality is reduced due to writing data in random order. Thus, random access increases the number of migrations and hence write amplifications during GC.

Previous studies have investigated SSD performance by considering access patterns. SHRD [9] proposed an address reshaping technique called sequentializing for the host and randomizing for the device, which transforms random write requests into sequential write requests in the host block device driver by assigning address space of a reserved log area in the SSD. F2FS [13] is a flash-friendly file system that supports append-only logging for underlying flash devices. Our work is in line with these previous studies, in terms of improving the random write performance of SSDs by remmaping random writes into sequential writes.

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

In contrast, we focus on transforming random writes into sequential writes at the block layer. Our proposed scheme can be easily applied to the existing file system without any modification.

In this article, we propose a block address transforming technique in the block layer to improve the I/O performance of SSDs. We aim to improve the I/O performance by exploiting the feature of block layer without the modification of file systems and devices. To do this, we first transform the block address in the block layer by changing random access patterns to sequential access patterns. Second, we devise a mapping table that manages original and transformed block addresses. Third, we support correct read operations using the mapping table and perform transaction processing to protect the mapping table when an unexpected failure occurs. Finally, we provide a cleaning scheme to reclaim the invalid data generated by the transformed sequential access.

We evaluate our proposed scheme on a multi-core machine with SATA and NVMe SSDs. Experimental results show the our proposed scheme improves the performance by up to 2x, 1.86x, 2.15x, and 42.8% compared with the existing scheme in the case of existing file systems EXT4 [17], XFS [29], BTRFS [22], and F2FS [13], respectively.

Our contribution are as follows:

- We analyze how I/O performance is affected by block layer and access pattern.
- We design and implement our transforming technique for improving I/O performance in the block layer without any modification to file systems and devices.
- We demonstrate that file systems with our scheme can improve I/O performance compared with those with the existing scheme.

The rest of the article is as follows: Section II describes the background and motivation for study. Section III presents the design and implementation of the proposed scheme. Section IV presents the experimental results. Section V discusses the related work. Section VI summarizes and concludes the article.

II. BACKGROUND AND MOTIVATION

A. MOTIVATIONAL EVALUATION

Figure 1 shows the performance of random and sequential writes in the different file systems such as EXT4 [17], XFS [29], BTRFS [22], and F2FS [13] by using FIO benchmark with same configuration described in Section IV-B1. The performance of sequential writes is better than random writes by up to 2x, 1.86x, 2.6x, and 1.32x in EXT4, XFS, BTRFS, and F2FS, respectively. In the case of EXT4 and XFS, the performance gap is relatively similar since these are journaling file systems and they perform in-place updates. In the case of BTRFS, it performs copy-on-write (COW) operations based on B-tree. In particular, the random write pattern generates randomly tree updates which increases the garbage collection overheads. Thus, it results in a write amplification which decreases the I/O performance. F2FS is a flash memory-friendly file system, and there is a logic that converts data access patterns within the file system. F2FS builds on append-only logging to turn random writes into sequential ones except for metadata writes. Therefore, F2FS has much better random write performance than that of other file systems.

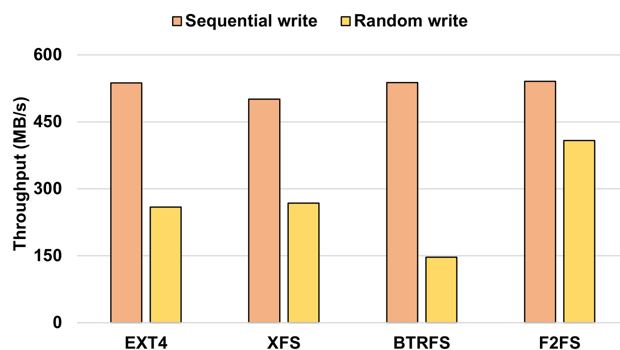


FIGURE 1. Write performance for various file systems.

Flash-based SSDs have many different characteristics compared with existing HDDs. In particular, Flash-based SSDs do not allow in-place update [16]. Thus, when a page in a block needs to be updated, the existing page gets invalidated and the updated data is written into a clean page, producing in an out-of-place update. Garbage collection (GC) is subsequently performed to reclaim this invalidated page. GC selects a victim block for cleaning based on given policy. Then, valid data pages in the block are migrated to a free block and the victim block is cleaned by an erase operation. This write amplification negatively affects the lifetime and endurance of SSDs [5].

Especially random writes can negatively affect the GC process by causing internal fragmentation of SSDs and decreasing spatial locality [9]. Also, random write requests cannot be merged in block layer, which leads to increase the number of data transmission between host and SSD. Since randomized pages makes uneven lifetime, there is much lower probability of similar lifetime inside the SSDs when random writes are requested compared with sequential write [21], [28]. Thus, there is a lower chance that all pages inside a block will be invalidated together. Thus, random writes show much lower performance than sequential writes in a NAND flash based SSDs.

B. PROCESSING I/O IN THE BLOCK LAYER

In general case of file I/O, there is a storage stack. For example, the virtual file system (VFS) first processes the request, manages page cache, and transfers the request to a file system when an application performs a read/write operation, and the file system layer provides system-specific implementation on top of the block storage [14]. The request is transferred to the block layer via `submit_bio()`. The block I/O layer serves as a bridge between the file system layer and device driver and provides OS-level block request/response management and block I/O scheduling [6]. The device driver transfers a

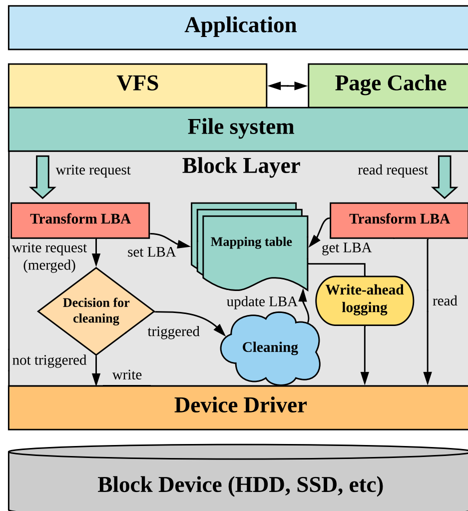


FIGURE 2. Overall architecture of proposed scheme.

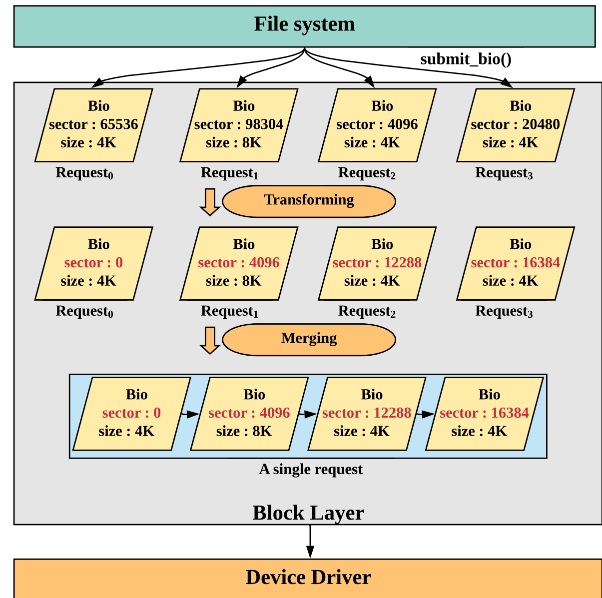


FIGURE 3. Transforming access patterns in the block layer.

read/write command according to the requested storage (e.g., HDD, SSD).

The block layer is an abstract layer that performs I/O operations regardless of device types or file systems [3]. When a file system passes an I/O request to the block layer, the file system makes a structure (i.e., `bio`). The `bio` structure is a basic unit to perform a block I/O operation that contains a variety of information (read/write, LBA, size, actual data, etc.). The block layer receives `bio` structure and manages the I/O request. The `bio` structure includes requested page(s) by maintaining a starting sector (i.e., `bi_sector`) and whole size of I/O request (i.e., `bi_size`).

More specifically, the block layer merges different requests and makes them into a single request if the requests are contiguous. For example, the block layer verifies whether the newly requested `bio` structure can be merged into a waiting request queue using the merge function (`blk_attempt_plug_merge()`). This merge mechanism in the block layer can increase storage bandwidth by reducing I/O between host and device. Thus, the proposed scheme accelerates and fully utilizes this merge mechanism by sequentializing requests at the block layer.

III. DESIGN AND IMPLEMENTATION

This section discusses design and implementation of the proposed scheme to improve SSD I/O performance.

A. OVERALL ARCHITECTURE

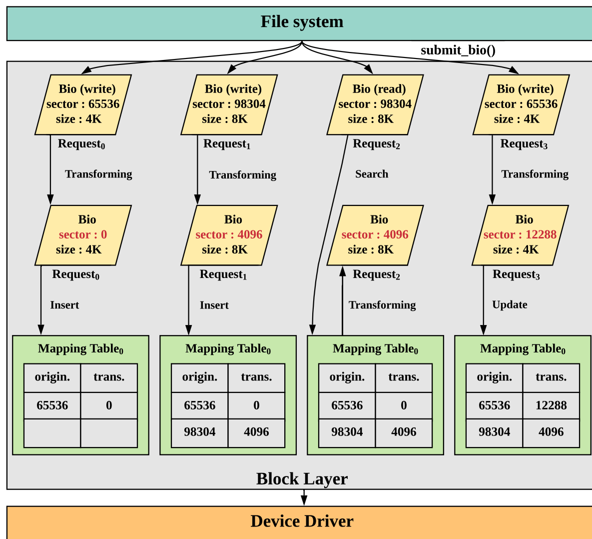
We present an efficient address transformation scheme in the block layer to improve the performance of file systems on SSDs. In our scheme, we transform random access patterns to sequential access patterns by sequentializing block addresses in the block layer.

Figure 2 shows our overall architecture of proposed scheme. As shown in the figure, when a request comes to the block layer, our proposed scheme identifies whether the request is write or read. If a write request is issued, we transform LBA of the request to sequential one which can be merged in the block layer. When the transform operation is executed, our proposed scheme stores information of original and transformed LBAs to the mapping table. To support transaction processing for the mapping table against a system or hardware failure, we adopt write-ahead logging scheme. On the other hand, if a read request is issued, we transform the original LBA to mapped LBA that was set in the write operation. After completing the processing for each request, our proposed scheme transfers the request to the device driver. If the number of invalid data is larger than a threshold value, our proposed scheme performs a cleaning operation to reclaim contiguous space and updates the mapping table.

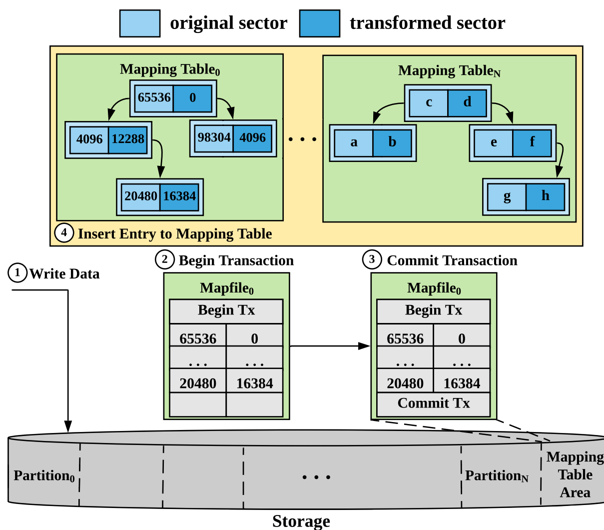
B. TRANSFORMING BLOCK ADDRESS IN THE BLOCK LAYER

To transform block addresses in the block layer sequentially, we intercept the write request and transform the original LBA (i.e., the sector in a `bio` structure) in a sequential order. To do this, we first check whether the request is read or write via the `bio` structure. If the request is write, we transform the original LBA from the `bio` structure to a sequential address. Figure 3 shows an example of transforming access patterns in the proposed scheme. As shown in the figure, there are four new write requests (request$_0$, request$_1$, request$_2$, and request$_3$) which are issued from the file system to the block layer. Each request has random sectors (i.e., 65536, 98304, 4096, and 20480) and different sizes (i.e., 4K, 8K, 4K, and 4K), respectively.

To transform the random sectors, we first obtain available sequential sectors (i.e., 0, 4096, 12288, and 16384) from



(a) Mapping table operations



(b) Mapping table layout and transaction processing

FIGURE 4. Managing mapping table.

a contiguous space. Then, we transform the original sectors (i.e., 65536, 98304, 4096, and 20480) to the sequential sectors (i.e., 0, 4096, 12288, and 16384) by considering request sizes, respectively. After these requests are made in a sequential manner, the existing block layer can merge contiguous requests into a single large request using its own merge algorithm [27]. Thus, our scheme accelerates this merging operations at the block layer by increasing the probability of fragmented requests to be merged into one request. After transforming sectors, we insert the pair of original and transformed sectors to a mapping table. Also, we manage the sector information for correct read operations, support transaction processing against failures, and perform a cleaning operation. We discuss them in following Sections III-C and III-D.

C. MANAGING THE MAPPING TABLE

Our scheme can achieve higher performance by transforming block addresses at the block layer. However, there can be challenges to be handled. For example, due to transformed the block addresses, we should transform the transformed block address to the original block address again to support correct read operations. Therefore, we need to manage the information of original and transformed block addresses carefully. To do this, we manage the block addresses in a mapping table. First, we insert pairs of original and transformed sectors to the mapping table in the case of a write operation. Second, we find the transformed sectors according to the original sector in the case of a read operation. Third, we find the original sector and update the transformed sectors in the case of a overwrite operations. Finally, we support transaction processing for the mapping table against a sudden system failure.

Figure 4 shows how to manage the mapping table in our proposed scheme. Figure 4(a) shows mapping table operations. As shown in the figure, four requests (i.e., $request_0$, $request_1$, $request_2$, and $request_3$) arrive at the block layer from the file system with different I/O types (i.e., write, write, read, and write). They have their own sectors (i.e., 65536, 98304, 98304, and 65536) and sizes (i.e., 4K, 8K, 8K, and 4K). In the case of write $request_0$, since this request is a write request, we transform the original sector (i.e., 65536) to the sequential sectors (i.e., 0) (transforming) if 0 is a starting block address. Then, we insert the pair of original and transformed sectors (i.e., 65536, 0) into mapping table (insert). In the case of write $request_1$, since this request is also a write request, we transform the original sector (i.e., 98304) to the next sequential sectors (i.e., 4096) which is calculated based on the size (4K) of the previous write request ($request_0$). Likewise to $request_0$, we insert the pair of original and transformed sectors (i.e., 98304, 4096) into mapping table (insert). In case of read $request_2$, we first search original sector (i.e., 98304) in mapping table (search). Then, we transform the original sector (i.e., 98304) to the corresponding transformed one (i.e., 4096) (transforming). Finally, in the case of write $request_4$, its origin sector (i.e., 65536) is already exist in the mapping table which is an overwrite operation. We update transformed sector (i.e., 0) in mapping table to new one (i.e., 12288) (update) which is calculated by the size (8K) of the previous write request ($request_2$).

Figure 4(b) shows the mapping table layout and transaction processing. For better efficient management of mapping table, we separate storage into several partitions logically and associate each partition (i.e., $partition_{0-N}$) with each mapping table (i.e., mapping $table_{0-N}$). Thus, each partition is managed by each associated mapping table which consists of a red-black tree [31]. We use four fixed-size partitions in the experiment and each partition is a basic unit for a cleaning operation. This multiple mapping table provides two main benefits. First, we can reduce the cost of finding the desired sectors. Second, we can minimize the overheads of

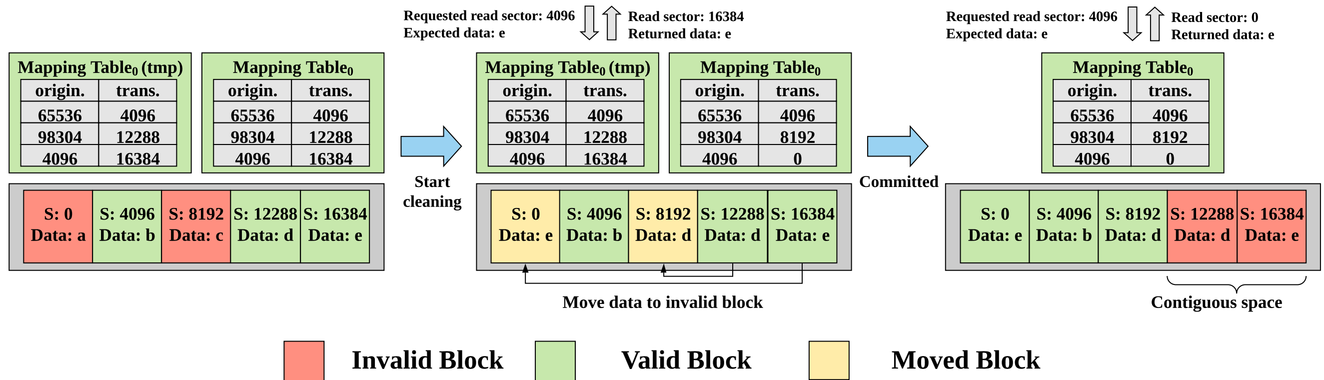


FIGURE 5. Cleaning operation (S: Sector, tmp: temporary mapping table).

cleaning operation. We will describe the cleaning operation in Section III-D.

As shown in Figure 4(b), to provide transaction processing for the mapping tables, we use a write-ahead logging scheme to store the mapping tables to the storage device. When write request arrives, we write data to the transformed sector (①). After the data is written to the storage device, we begin a transaction by recording the original and transformed sectors to the mapping table file (i.e., mapfile$_0$) (②). Then, we commit the transaction by writing the mapping table file to the mapping table area in the storage device (③). Finally, we update the mapping table by inserting sectors into the mapping table in memory (④). When a system failure occurs, our proposed scheme reads the mapping table in the mapping table area and reorganizes the mapping table based on the transaction commit. By doing so, we ensure the consistency of the mapping information and data integrity of the system.

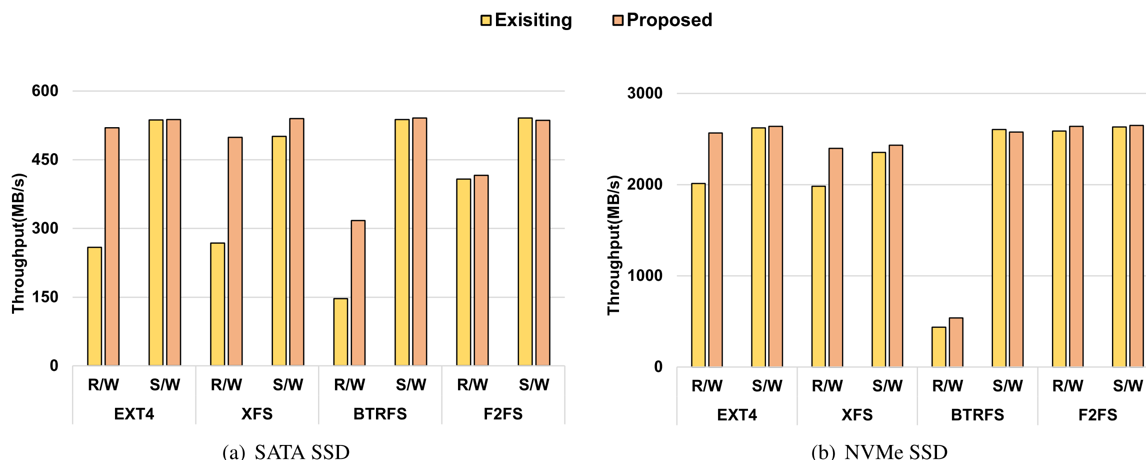
Meanwhile, managing the mapping tables can have issues in terms of scalability and performance. For example, when the size of the storage increases, the size of the mapping table will increase linearly. Thus, there can be scalability issue due to the limit of memory space. Also, as the speed of SSDs is increasingly faster, the performance overhead of the mapping table can be a bottleneck. To handle this limitation, we can adopt an on-demand map loading scheme for managing the mapping table. In this scheme, we can load only a subset of an entire mapping table. Thus, we load only the subset consisting of the sectors to be transformed and unload other subsets. This scheme can reduce the memory space overhead so that they can increase the scalability and performance. We leave this memory space overhead reduction and optimization as a future work.

D. CLEANING OPERATION

In our scheme, overwrite operations generate invalid data since we perform out-of-place updates. In order to collect this invalid data and create a continuous and large space, our proposed scheme perform cleaning operation [7], [13]. Cleaning operation is triggered for a partition when the next write request is placed on the next partition. Since we cannot process user requests during cleaning operation, with multiple mapping tables and partitions, we can perform cleaning operation for a partition and processing upcoming user requests in other partitions simultaneously. To allow correct read operations during a cleaning operation, before starting the cleaning operation, we copies the corresponding targeted mapping table to a temporary mapping table. Thus, when a read request arrives for the targeted mapping table, we perform the read operation on the temporary mapping table.

During the cleaning operation, we use a bitmap to check the data is valid or not. Thus, we move the valid data to the targeted invalid data location and update the corresponding entries of the mapping table. We find valid data from tail to head and invalid data from head to tail in the targeted partition. Note that if a write request arrives in the middle of cleaning operation, the request is processed on another partition in which a cleaning operation is not performed. Thus, even write requests can be processed without blocking due to multiple mapping tables and partitions. As a result, this cleaning operation fills the holes generated by invalid data and is repeated until there are no holes, producing continuous free space for new write requests. After completing the whole process, we delete the temporary mapping table.

Figure 5 shows an example of our cleaning operation. As shown in the figure, there are five contiguous sectors (i.e., 0, 4096, 8192, 12288, and 16384). In this case, sectors 0 and 8192 are invalid, creating non-contiguous space. Before starting a cleaning operation, we create a temporary mapping table (mapping table$_0$ (tmp)) by copying the targeted mapping table (mapping table$_0$). During the cleaning operation, we move the last valid data (i.e., sector 16384) to the hole (i.e., sector 0) and move the next valid data (i.e., sector 12288) to the next hole (i.e., sector 8192). After then, we update entries (<98304, 8192> and <4096, 0>) in the targeted mapping table. If a read request (i.e., original sector: 4096) arrives during the cleaning operation, we process the read



(a) SATA SSD (b) NVMe SSD

FIGURE 6. FIO benchmark.

request on the temporary mapping table, read the data in the temporary sector (i.e., 16384), and return the data (i.e., e). Thus, the read request is processed safely without sacrificing read consistency even though the cleaning operation is running.

After cleaning all invalid data, we commit the cleaning operation with the updated mapping table and remove the temporary mapping table. If a read request (i.e., 4096) arrives again, we read data (i.e., e) from the updated transformed sector (i.e., 0) in the committed mapping table. Thus, we obtain contiguous space (i.e., sectors: 12288-16384) for future write requests.

IV. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETUP

For the experiment setup, we used Intel i9-9900K (5.0 GHz) environment with eight physical cores, eight hyper-threading, and 16 GiB of memory. For the storage devices, we used a SATA SSD (Micron CT250MX500SSD1)(250GB) and NVMe SSD (Samsung 970 EVO M.2)(1TB). We used Linux Kernel 4.9.1 and Ubuntu 16.04.6 LTS. We used FIO [1] as a micro benchmark, and used Flexible Filesystem Benchmark (FFSB) [2] and Postmark [8] as macro benchmarks. All experimental results are average of five runs.

B. EXPERIMENT RESULTS

1) MICRO BENCHMARK

a: FIO BENCHMARK

For FIO configuration, we used 8 threads writing 3GB of files each and 4KB as a block size. We performed sequential and random writes on SATA and NVMe SSDs. Figure 6(a) shows the result of the FIO benchmark with existing and our proposed schemes in SATA SSD environment. As shown in the figure, except for F2FS [13] as a log-structured file system, other local file systems show a large performance difference between sequential and random writes in the existing scheme. The EXT4, XFS, BTRFS, and F2FS file systems with our proposed scheme improves the random write performance by up to 2x, 1.86x, 2.15x, and 1.01x compared with EXT4, XFS, BTRFS, and F2FS file systems with existing scheme, respectively. The performance improvement is achieved by transforming random LBAs to sequential LBAs at the block layer, which results in making a large requests. In the case of F2FS, there is almost no performance improvement since the nature of F2FS already sequentializes LBAs at the file system level.

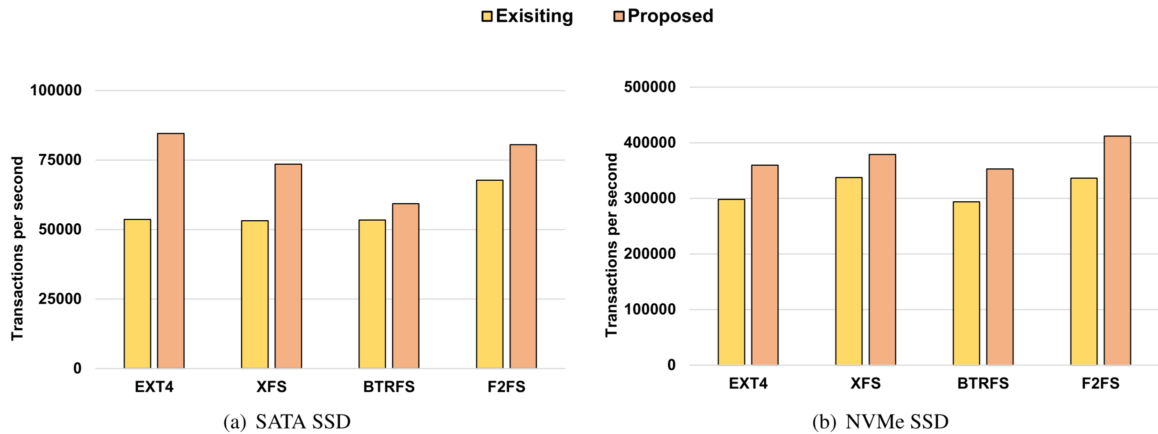
Figure 6(b) shows the result of the FIO benchmark in the case of existing and our proposed schemes in NVMe SSD. NVMe SSD does not shows significant performance difference between random and sequential writes. However, the performance of random write is still lower than sequential write. As shown in the figure, in existing file systems, the random write performance is lower than the sequential writes performance EXT4, XFS, BTRFS, and F2FS, by up to 23.3%, 15.7%, 82.2%, and 1.74%, respectively. Our proposed scheme with EXT4, XFS, BTRFS, and F2FS improves the random write performance by up to 21.6%, 17.3%, 18.9%, and 1.9% compared with an existing scheme with EXT4, XFS, BTRFS, and F2FS, respectively. This result demonstrates our scheme has still effectiveness even if NVMe SSD is used.

2) MACRO BENCHMARK

According to the performance metrics provided by each benchmark, we use transaction per second in FFSB and total execution time in PostMark to measure the performance.

a: FLEXIBLE FILE SYSTEM BENCHMARK

The Flexible Filesystem Benchmark (FFSB) is a file system performance measurement tool [2]. For FFSB configuration, we use 100 directories, 4GB file size, 16GB insert size, 8 threads, 4KB write size, 4KB write block size, 4KB read size, and 4KB read block size.



(a) SATA SSD
(b) NVMe SSD

FIGURE 7. **Flexible file system benchmark (FFSB).**

We evaluate the performance of existing and proposed schemes by using FFSB on SATA SSD. Figure 7(a) shows the FFSB's result when SATA SSD is used. As shown in the figure, the proposed scheme improves the transactions per second by up to 57.7%, 38.3%, 11.0%, and 18.8% compared with an existing scheme in the case of EXT4, XFS, BTRFS, F2FS file systems, respectively. We achieve high performance by up to 57.7% transactions per second in the case of EXT4 file system. The experimental results show that our proposed scheme is also effective in a macro benchmark.

Figure 7(b) shows the FFSB's result when NVMe SSD is used. As shown in the figure, our proposed scheme improves the transactions per second by up to 17.1%, 10.9%, 16.8%, and 18.3% compared with an existing scheme in the case of EXT4, XFS, BTRFS, and F2FS file systems, respectively. We achieve high performance by up to 18.3% transactions per second in the case of F2FS file system. This result shows that our scheme can improve the performance of F2FS file system even though the file system already provides a logging scheme in the file system-level. The experimental results show that our proposed scheme is also effective even in NVMe SSD.

b: PostMark

PostMark is a benchmark that demonstrates system performance for short-lived small files in internet applications (e.g., electronic mail or web commerce) [8]. Parameter settings for Postmark are as follows: 900MB file size (low and high bounds of files), 20 files (number of simultaneous files), read/write size 4KB(read/write block size), transaction 20 (number of transactions), bias create 10 (the chance of choosing read over append), bias read 10 (the chance of choosing create over delete).

We evaluate the performance of existing and proposed schemes by using PostMark on SATA SSD. Figure 8 shows the PostMark benchmark results in SATA SSD. As shown in the figure, our proposed scheme improves total transaction time up to 33.9%, 18.6%, 21.5%, and 16.4% compared with existing scheme in the case of EXT4, XFS, BTRFS, and F2FS file systems, respectively. We achieve high performance by up to 33.9% total time in the case of EXT4 file system. The experimental results demonstrate that our proposed scheme is effective for small file workloads.

Figure 8(b) shows the result of PostMark when NVMe SSD is used. As shown in the figure, our proposed scheme improves the transaction time up to 33.4%, 40%, 29.7%, and 42.8% compared with an existing scheme in the case of EXT4, XFS, BTRFS, and F2FS file systems, respectively. We achieve high performance by up to 42.8% total time in the case of F2FS file system. The experimental result show that our proposed scheme is effective for NVMe SSD.

C. CLEANING OVERHEADS

Figure 9 shows transient write throughput from FIO benchmarks when cleaning operation in our proposed scheme occurs in the case of SATA SSD. To measure the cleaning overhead, we first write data approximately 50GB and perform the overwrite operations from the FIO benchmark with time-based configuration. We force this cleaning operation to be performed during the experimental time.

Figures 9(b), 9(c), 9(d), and 9(e) shows overheads of cleaning operations in the case of EXT4, XFS, BTRFS, and F2FS, respectively. The cleaning operation decreases the performance by up to 62.4%, 55.3%, 89%, and -7.5% in the case of EXT4, XFS, BTRFS, and F2FS, respectively, compared with that of the proposed scheme without cleaning operation. This result demonstrates the cleaning operation has overhead. In the case of F2FS, even though the cleaning operation occurs, the performance degradation is not large. It is because F2FS already transforms the random patterns to the sequential patterns at the file system level. Thus, at the block layer, the number of cleaning operations decreases. Although the throughput decreases during the cleaning operation, the

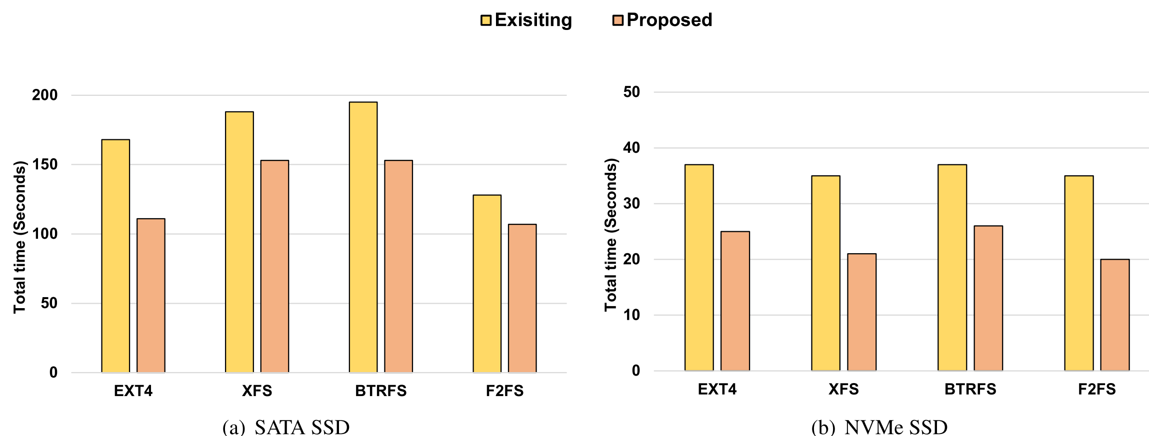


FIGURE 8. PostMark.

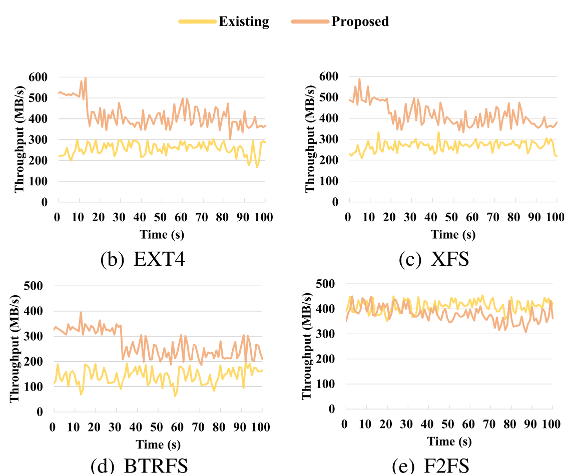


FIGURE 9. Cleaning overheads in different file systems on SATA SSD.

throughput of our proposed scheme still is higher than that of existing scheme. It is because a cleaning operation is performed on a partition and other partitions still perform I/O services without blocking I/O operations. In addition, we note that a cleaning operation is not performed frequently in normal situations. To measure and show the cleaning overhead in this evaluation, we force a cleaning operation continuously.

V. RELATED WORK

A. OPTIMIZING BLOCK LAYER

Bjørling *et al.* [4] proposes a new design for the block layer. They design two levels of queues in order to reduce contention and promote thread locality. Lee *et al.* [14] provides an asynchronous I/O stack which leverages a lightweight block layer specialized for NVMe SSDs. The I/O stack eliminates unnecessary components and overlaps I/O-related CPU operations with device I/O operations.

Falcon [12] propose a new design of per-drive I/O processing that separates two key functionalities of I/O batching and I/O serving in the I/O stack. Specifically, Falcon presents Falcon I/O Management Layer that batches the incoming I/Os at the volume level, and Falcon Block Layer that parallelizes I/O serving on the SSD level in a new block layer.

Isotope [26] presents a new block store that supports ACID transactions over block reads and writes. It uses a new multi-version concurrency control protocol that exploits fine-grained, sub-block parallelism in workloads and offers both strict serializability and snapshot isolation guarantees. Our study is in line with these studies [4], [12], [14], [26] in terms of improving the performance of I/O in block layer. In contrast, our study focuses on improving the performance by transforming block addresses in the block layer.

B. LOG-STRUCTURED SCHEME IN FILE SYSTEMS, DATABASE SYSTEMS, AND STORAGE DEVICES

Log-structured file system (LFS) [23] writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. Wang *et al.* [30] provide an evaluation of flash SSDs in transaction processing systems by using TPC-C benchmark with various configurations. Ho *et al.* [7] separates random write requests from sequential write requests and transforms the address of random write requests for eMMC storage devices. F2FS [13] presents a new file system optimized for flash storage devices. The file system basically builds on append-only logging to turn random writes into sequential ones. SHRD [9] supports log-structured data write operation by transforming random write requests into sequential write requests in the device level. In case of power failure, it restores the remapping information using the data stored in the out-of-band area of SSD to ensure correct recovery. Kim *et al.* [11] target distributed file systems and provide an address reshaping technique which reshapes random write requests into sequential ones in the distributed file system level.

Ryu *et al.* [24] address the problems of the existing log based flash memory file systems analytically and propose an efficient log-based file system, which produces higher

performance, less memory usage and mount time than the existing log-based file systems. SFS [19] is similar to the traditional log-structured file system (LFS) but SFS takes a new on writing data grouping strategy. IPL [15] has demonstrated its potential for considerable improvement of write performance for OLTP-type applications by exploiting the advantages of flash memory such as no mechanical latency and high read bandwidth. Oh *et al.* [20] proposes a cost-effective and reliable SSD host cache solution. It provides cost-effectiveness by using multiple low-cost SSDs and reliability by retaining data redundancy through RAID.

Our study is in line with these studies in terms of improving the performance of storage systems and devices using log-structured schemes. Meanwhile, these studies have a limitation that has a dependency with the file system or hardware. In contrast, our proposed scheme is based on the block layer and so that our scheme can be more easily applied to a system without the dependency with any file system or hardware.

C. LOG-STRUCTURED SCHEME IN THE BLOCK LAYER

Gecko [25] is a log-structured design that eliminates read-write contention by chaining together a small number of drives into a single log. Thus, writes proceed to the tail drive without contention from either GC reads or first-class reads. SWAN, a novel All Flash Array (AFA) management scheme [10] aims to alleviate the performance interference caused by GC at both levels. Unlike the commonly-used temporal separation approach that performs GC at idle time, SWAN take a spatial separation approach that partitions SSDs into the front-end SSDs dedicated to serve write requests and the back-end SSDs where GC is performed.

Our study is in line with these studies [10], [25] in terms of improving the performance in block layer. In contrast, our study focus on improving the performance by transforming block addresses instead of improving the layout or management of multiple storage devices.

VI. CONCLUSION

In this article, we design and implement an efficient log-structured block I/O Scheme for flash-based SSDs to improve I/O performance. To this end, we transform block addresses to turn random write requests into sequential write requests at the block layer. Also, we provide efficient management of mapping tables and cleaning operation without sacrificing consistency. We evaluate various file systems on our scheme with several micro and macro-benchmarks. Our experimental results show that the file systems (i.e., EXT4, XFS, BTRFS, and F2FS) with our proposed scheme can improve the I/O performance compared with those with the existing scheme.

REFERENCES

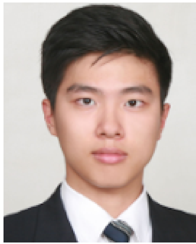
[1] *Fio Benchmark*. [Online]. Available: https://github.com/axboe/fio
[2] *The Flexible Filesystem Benchmark*. [Online]. Available: https://github.com/FFSB-Prime/ffsb
[3] J. Axboe, "Linux block IO-present and future," in *Proc. Ottawa Linux Symp.*, 2004, pp. 51–61.
[4] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: Introducing multi-queue SSD access on multi-core systems," in *Proc. 6th Int. Syst. Storage Conf. (SYSTOR)*, 2013, pp. 1–10.
[5] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Perform. Eval.*, vol. 67, no. 11, pp. 1172–1186, Nov. 2010.
[6] J. Guerra, L. Useche, M. Bhadkamkar, R. Koller, and R. Rangaswami, "The case for active block layer extensions," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 6, pp. 3–9, Oct. 2008.
[7] C.-C. Ho, Y.-H. Chang, and T.-W. Kuo, "Access pattern reshaping for eMMC-enabled SSDs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 30–37.
[8] J. Katcher, "Postmark: A new file system benchmark," Netw. Appliance, Sunnyvale, CA, USA, Tech. Rep. TR3022, 1997.
[9] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 271–284.
[10] J. Kim, K. Lim, Y. Jung, S. Lee, C. Min, and S. H. Noh, "Alleviating garbage collection interference through spatial separation in all flash arrays," in *Proc. USENIX Annu. Tech. Conf. (USENIX)*, 2019, pp. 799–812.
[11] S. Kim, J. Han, H. Eom, and Y. Son, "Improving I/O performance in distributed file systems for flash-based SSDs by access pattern reshaping," *Future Gener. Comput. Syst.*, vol. 115, pp. 365–373, Feb. 2021.
[12] P. Kumar and H. H. Huang, "Falcon: Scaling fIOg performance in multi-SSD volumes," in *Proc. USENIX Annu. Tech. Conf. (USENIX)*, 2017, pp. 41–53.
[13] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 273–286.
[14] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2019, pp. 603–616.
[15] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2007, pp. 55–66.
[16] D. Ma, J. Feng, and G. Li, "LazyFTL: A page-level flash translation layer optimized for NAND flash memory," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2011, pp. 1–12.
[17] 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. Princeton, NJ, USA: Citeseer, 2007, pp. 21–33.
[18] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. FAST*, vol. 12, 2012, pp. 1–16.
[19] C. Min, S.-W. Lee, and Y. I. Eom, "Design and implementation of a log-structured file system for flash-based solid state drives," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2215–2227, Sep. 2014.
[20] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Improving performance and lifetime of the SSD RAID-based host cache through a log-structured approach," *ACM SIGOPS Operating Syst. Rev.*, vol. 48, no. 1, pp. 90–97, May 2014.
[21] L. D. Phan and D. S. Suryabudi, "Non-volatile semiconductor memory segregating sequential data during garbage collection to reduce write amplification," U.S. Patent 8 316 176, Nov. 20, 2012.
[22] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–32, Aug. 2013.
[23] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *Proc. 13th ACM Symp. Operating Syst. Princ. (SOSP)*, 1991, pp. 1–15.
[24] J. Ryu and C. Park, "A technique to enhance performance of log-based file systems for flash memory in embedded systems," in *Proc. 2nd Int. Conf. Digit. Inf. Manage.*, Oct. 2007, pp. 580–582.
[25] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Gecko: Contention-oblivious disk arrays for cloud storage," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 285–297.
[26] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Isotope: Transactional isolation for block storage," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 23–37.
[27] Y. Son, H. Y. Yeom, and H. Han, "Optimizing I/O operations in file systems for fast storage devices," *IEEE Trans. Comput.*, vol. 66, no. 6, pp. 1071–1084, Jun. 2017.

[28] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *FAST*, vol. 10, 2010, pp. 101–114.

[29] R. Y. Wang and T. E. Anderson, "XFS: A wide area mass storage file system," in *Proc. IEEE 4th Workshop Workstation Operating Syst. (WWOS-III)*, Oct. 1993, pp. 71–78.

[30] Y. Wang, K. Goda, M. Nakano, and M. Kitsuregawa, "Performance evaluation of flash SSDs in a transaction processing system," *IEICE Trans. Inf. Syst.*, vol. E94-D, no. 3, pp. 602–611, 2011.

[31] C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam, and W. Fun, "Fairness and interactive performance of O(1) and CFS Linux kernel schedulers," in *Proc. Int. Symp. Inf. Technol.*, Aug. 2008, pp. 1–8.



JAEHYUN HAN received the B.S. degree from the School of Computer Science and Engineering, Chung-Ang University, where he is currently pursuing the M.S. degree with the School of Computer Science and Engineering. He was an Intern at bosornd, in 2018 and 2019. His research interests include distributed systems and operating systems.

YONGSEOK SON received the B.S. degree in information and computer engineering from Ajou University, in 2010, and the M.S. and Ph.D. degrees from the Department of Intelligent Convergence Systems and Electronic Engineering & Computer Science, Seoul National University, in 2012 and 2018, respectively. He was a Postdoctoral Research Associate in electrical and computer engineering with the University of Illinois at Urbana–Champaign. He is currently an Assistant Professor with the School of Computer Science and Engineering, Chung-Ang University. His research interests include operating, distributed and database systems.