IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 27 September 2022, accepted 4 November 2022, date of publication 14 November 2022,
date of current version 17 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3221733

RESEARCH ARTICLE

Improving I/O Performance via Address Remapping in NVMe Interface

DONG KYU SUNG[1], YONGSEOK SON[2], HYEONSANG EOM[1], AND SUNGGON KIM[3]
[1]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[2]Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea
[3]Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul 01811, South Korea

Corresponding author: Sunggon Kim (sunggonkim@seoultech.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) through the Korean Government under Grant RS-2022-00166541, Grant NRF-2021R1C1C1010861, Grant NRF-2022R1A4A5034130, and Grant 2021R1F1A1106343812; and in part by the BK21 FOUR Intelligence Computing (Department of Computer Science and Engineering, SNU) funded by NRF under Grant 4199990214639.

ABSTRACT Recently, flash-based solid-state drives (SSDs) are widely used in industry and academia due to their higher bandwidth and lower latency compared with traditional hard disk drives (HDDs). Furthermore, SSDs with the Non-Volatile Memory Express (NVMe) interface can provide higher performance and ultra-low latency compared with the Serial AT Attachment (SATA) SSDs. Due to their high performance, NVMe SSDs are adopted in many systems as fast storage devices. However, the performance of NVMe SSDs can be negatively affected by I/O access patterns. For example, random write access patterns can have negative impacts on performance due to the unique characteristics of SSDs such as out-of-place update and garbage collection. In this paper, we propose an address remapping scheme to improve the I/O performance of NVMe SSDs. Our proposed scheme transforms random access patterns into sequential access patterns in the NVMe device driver. This allows our scheme to improve the I/O performance of NVMe SSDs while supporting widely used file systems such as EXT4, XFS, BTRFS, and F2FS without any modification to the device. Experimental results show that our proposed scheme can improve the performance of NVMe SSD by up to 64.1% compared with the existing scheme.

INDEX TERMS Flash-based SSDs, NVMe interface, device driver, I/O performance, garbage collection.

I. INTRODUCTION

As emerging big data and machine learning applications produce and process a large amount of data, the storage performance is becoming more and more important [1], [2], [3]. To improve storage performance, flash-based solid-state drives (SSDs) have been widely adopted in both industry and academia as they provide higher bandwidth and lower latency compared with the existing hard disk drives (HDDs) [4]. Especially, SSDs with the Non-Volatile Memory Express (NVMe) interface can provide higher performance than SSDs with the Serial AT Attachment (SATA) [5].

While NVMe SSDs provide high I/O performance, they have a higher cost per GB compared with other storage devices. As a result, many systems utilize tiered storage architecture with heterogeneous devices to improve the cost efficiency of the systems [6], [7], [8]. In these systems, low-performance storage devices such as hard disk drive (HDD) and tape storage are used to store a large amount of less frequently accessed data such as backup data and archives. In contrast, high-performance storage devices such as NVMe SSD are used to handle frequent random writes and a large amount of data from I/O-intensive applications. For example, database systems store frequently updated logs and data entries using NVMe SSDs [9], [10]. In addition, NVMe SSDs are employed to support I/O-intensive workloads and checkpointing operations in HPC systems [11], [12]. Thus, NVMe SSDs are widely used to handle I/O requests from many applications which have various access patterns and request sizes.

To efficiently store data with various patterns and sizes, it is important to understand the unique characteristics of

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

SSDs and their implication for performance. For example, the performance of random writes can be lower than that of sequential writes in SSDs [13]. Flash memory which is used in SSDs has erase-before-write constraints that require memory cells to be free before writing data for the first time. Thus, in case of update, SSDs perform out-of-place update which writes updated data to a free page in the same or different block. This creates blocks with a lot of invalidated pages with old data which need to be cleaned through garbage collection operation. When garbage collection is performed, it reads the entire victim block, copies data to the empty block, and erases the victim block. This incurs larger page copy overheads when invalid pages are more dispersed throughout the blocks. As sequential write workloads store data with a similar lifetime in a single block, more data in a single block can be cleared with a single operation. This leads to higher garbage collection efficiency and performance compared with random write workloads [14], [15], [16]. Thus, the access patterns of applications need to be carefully considered to fully exploit the performance of SSDs.

Previous studies have attempted to improve the performance of SSDs by addressing I/O access patterns. F2FS [17] is a file system for flash-based storage devices that improves the performance using append-only logging. SHRD [18] transforms random writes into sequential writes and stores the data in reserved log space. ReSSD [19] identifies small random writes and changes them to sequential writes at the virtual block device level. Our study is in line with these studies in terms of improving I/O performance by transforming random writes into sequential writes. In contrast, we focus on remapping addresses of random writes into those of sequential writes in the NVMe device driver layer. This allows our proposed scheme to be applied in the various file systems and storage devices without any modification in other layers.

In this paper, we propose an address remapping scheme in the NVMe device driver to improve the I/O performance. The proposed scheme transforms I/O requests from the applications into sequential requests. For example, our scheme checks if the request is a write or read request in the device driver. If the request is a write request, we transform the block address of the request into a sequential address, record the original and transformed address in a remapping table, and perform the I/O operation using the transformed address. If the request is a read request, we search the transformed block address from the remapping table using the original block address and perform a read operation using the transformed address. This allows our scheme to improve the performance by creating sequential write patterns and perform correct read operation using the transformed address.

Especially, by transforming requests in the device driver layer, our scheme has two main advantages: (1) the proposed scheme can be easily adapted to many storage systems since the modifications are limited to the device driver. (2) the proposed scheme based on the device driver can be applied to the individual device in heterogeneous tiered storage systems that have various devices with different characteristics.

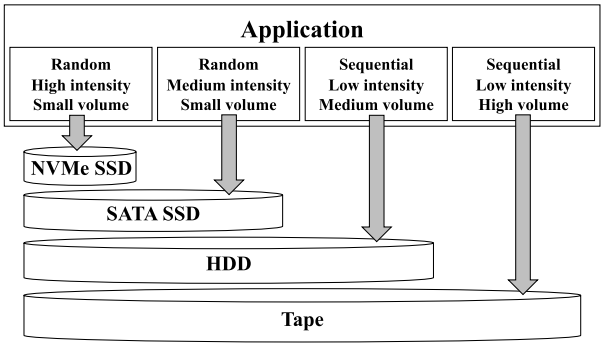


FIGURE 1. Overall architecture of a tiered storage system. Random and high I/O intensive workloads are handled by NVMe SSD tier.

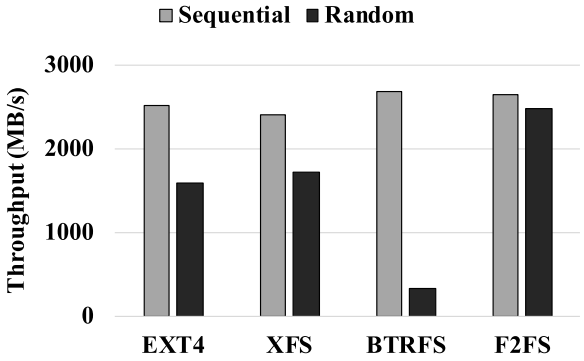


FIGURE 2. Sequential and random write performance of NVMe SSD on various file systems.

For evaluation, we used an enterprise-grade NVMe SSD (Samsung NVMe SSD PM1725b) with widely used file systems such as EXT4 [20], XFS [21], BTRFS [22], and F2FS [17]. The experimental results show that our proposed scheme can improve the I/O performance up to 64.1% compared with the existing scheme.

Our contributions are as follows:

- We analyze the I/O operation in NVMe SSDs.
- We design and implement an address remapping scheme for NVMe SSDs in the device driver layer.
- We demonstrate that the proposed scheme can improve the performance of random writes compared with the existing scheme on various file systems.

The rest of the paper is organized as follows: Section II presents the background and motivation of the paper. Section III presents the overall design of the proposed scheme. Section IV shows the experimental results. Section V discusses the related work. Section VI concludes the paper.

II. BACKGROUND AND MOTIVATION

A. NVMe SSD

NVMe SSDs provide higher bandwidth and lower latency compared with HDDs and SATA SSDs. However, they also have a high cost per GB and low capacity. Thus, to fully exploit the performance of NVMe SSDs, they are used as fast storage layer in multi-tiered storage systems. Figure 1 shows an overall architecture of a tiered storage system. As shown in the figure, each of the storage tiers is constituted by a

different type of storage device and applications are sending I/O requests with various characteristics to each storage tier. For example, HDD and tape storage tiers that have high capacity but low performance are used to store sequential access and low I/O intensity requests such as backup and archive data. On the other hand, SSD tiers are used to store data that have random access patterns and high I/O intensity.

Especially, the NVMe SSD tier is adopted to handle random and highly intensive I/O requests from applications with different characteristics. For example, SpanDB [23] adopts NVMe SSDs as a storage tier for processing parallel write-ahead log and flush/compaction of top levels of LSM-tree, both of which incur bottlenecks in the critical path of I/O operations in KV stores. In addition, many burst buffer systems utilize NVMe SSDs to process bursty I/O traffics [11], [24]. Thus, NVMe SSDs can be utilized as a storage tier that helps mitigate I/O bottlenecks from many applications and systems. However, as applications have different I/O characteristics, it is important to understand the effect of access patterns on NVMe SSDs. Moreover, it is necessary to support the patterns efficiently to improve the overall performance of the storage system.

To show how the I/O access pattern impacts the performance of NVMe SSDs, we performed a motivational evaluation. We ran FIO benchmark [25] with 8 threads, 8 GiB file per thread, 4 KiB block size, I/O queue depth of 16, and buffered I/O mode. Figure 2 shows the performance of sequential and random writes in NVMe SSD with various file systems such as EXT4 [20], XFS [21], BTRFS [22], and F2FS [17]. As shown in the figure, sequential write outperforms random write in all file systems and the performance of random write can be degraded by up to 87.4% compared with that of sequential write. BTRFS is a copy-on-write (CoW) file system that does not directly overwrite data. Thus, random writes in SSDs with BTRFS cause huge garbage collection overhead compared with sequential writes. On the other hand, F2FS is a log-structured file system that writes data in a sequential manner by append-only logging. Thus, random writes do not introduce significant overheads in SSDs with F2FS. These results show that the performance of NVMe SSDs can be affected by I/O access patterns and this can be observed in many traditional file systems. Since NVMe SSDs are adopted in many systems to handle workloads with various access patterns including random writes, it is important to address I/O access patterns to minimize the performance degradation of NVMe SSDs with many file systems.

B. CHARACTERISTICS OF SSDs

Flash-based SSDs have unique characteristics that need to be considered for fully exploiting the performance. SSDs are composed of flash memory cells. A page is the smallest unit in SSDs and a group of pages is referred to as a block [26]. When write operations are executed, SSDs can store data in the unit of page. However, SSDs can only erase data in the unit of block. Therefore, in the case of overwriting data, SSDs need to perform out-of-place update [27]. When overwrite operations are executed, a flash translation layer (FTL) of SSD writes new data in free pages in other blocks and invalidates the original data. If the number of free pages is below the threshold, a garbage collection operation is triggered by FTL to reclaim the invalidated pages in SSDs. During the garbage collection, all valid pages in the victim block are copied to another block. Then, the entire victim block is erased for future use. This internal mechanism of SSDs causes write amplification which decreases the performance and endurance of SSDs.

There have been many studies to investigate the overheads of garbage collection in SSDs and improve the performance of SSDs by reducing the overheads. Previous studies [28], [29] evaluated the relationship between garbage collection and write amplification when uniformly-distributed small random writes are performed. Since random writes cause more write amplification from garbage collection, many works have tried to resolve issues of random I/O access patterns to reduce garbage collection overheads in various approaches. SWAN [30] is an All Flash Array (AFA) management scheme in the block I/O layer designed to alleviate performance degradation from garbage collection in flash memory. It partitions SSDs in the array into two groups and handles bulk write and garbage collection separately. SFS [13] is a file system optimized for SSDs that transforms random writes into sequential writes by a log-structure scheme. Also, by dividing data blocks based on update likelihood, SFS reduces segment cleaning overheads. LBA scrambler [31] is designed to fill up unused space in fragmented pages with newly written data to reduce the number of valid pages in the next erase block. With this approach, the scrambler can reduce the number of page copy operations and the latency of garbage collection can be minimized.

As explained above, many studies improved the performance of SSD by investigating the effects of garbage collection operations and optimizing the I/O layers such as the file system and block I/O layer. However, optimizations in the file system and block layer can affect all storage devices in the system. In a tiered storage system where multiple storage devices are used in a single system, devices have different characteristics. Thus, it is important to propose an optimization for each of the storage tiers with unique characteristics. In order to improve the performance of NVMe SSDs in the tiered storage systems, the optimization needs to be based on the system layer that is specific to the NVMe SSD storage tier.

III. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of our proposed scheme that improves the I/O performance of NVMe SSDs.

A. OVERALL ARCHITECTURE

To improve the I/O performance of NVMe SSDs, we propose an address remapping scheme. The proposed scheme transforms random access patterns into sequential access patterns

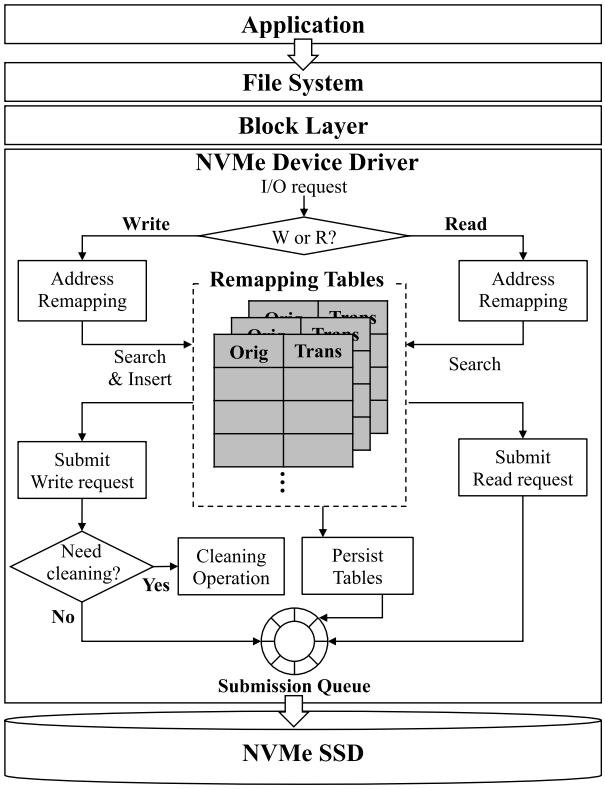


FIGURE 3. Overall architecture of the proposed scheme.

by modifying block addresses of I/O requests in the NVMe device driver. This is because modifications on the file system and block layer can affect the entire devices in the system as all I/O operations are handled by these layers. However, by applying our scheme in the NVMe device driver layer, the proposed scheme can improve the performance of the NVMe device without affecting other types of devices in the multi-tiered storage system. In the existing scheme, the I/O request issued from an application goes through the file system and block layer and arrives at the NVMe device driver. The request is then submitted to a submission queue of the NVMe device driver with information such as logical block address (LBA) and request size. After the request is submitted, the NVMe controller in the device processes the requests and notifies the host after the request is processed through a completion queue. An entry of the completion queue contains the submission queue and command identifiers of the completed request to identify the request. Therefore, the host can acknowledge that the request is successfully handled and continue processing the data.

On the other hand, the proposed scheme performs the address remapping operation of the I/O request before submitting the request to the submission queue of the NVMe SSD. Figure 3 shows the overall architecture of the proposed scheme. As shown in the figure, the proposed scheme first identifies if the request is a write or read request. If the request is a write request, the LBA of the request is transformed into a sequential LBA by the address remapping operation. To do this, we use the transformed LBA and the size of the previous request to generate a new sequential LBA for the request. Then, we search by the original LBA to check if the data page already exists. If it exists, we invalidate old data and insert a pair of original and transformed LBAs into a remapping table. Otherwise, we continue to the next step by simply inserting the new pair of LBAs. After the pair is inserted into the remapping table, we submit the write request with the sequential LBA to the submission queue. If the request is a read request, the transformed LBA must be used to access the data since our scheme stores the original data in a different address by transforming the original LBA. To obtain the transformed LBA, we search the remapping table and find the transformed LBA which is paired with the original LBA of the request. With the transformed LBA found in the table, we submit the read request to the submission queue. To submit the write or read request with transformed LBA, we intercept struct `nvme_rw_command` which is used for carrying information such as address and size of request in the NVMe device driver and modify field `slba` to change the original LBA to transformed LBA.

In addition to processing the I/O requests, the proposed scheme performs the cleaning operation. The cleaning operation is necessary as the data are invalidated when overwrite and delete operations are performed. Through the cleaning operation, the proposed scheme cleans the invalidated data and creates a large contiguous address space that can be used for transformed write requests. Finally, the proposed scheme persists the remapping tables in the memory to the persistent device by using logs and transactions to ensure atomicity and persistence of address remapping operations. This is to protect the remapping information and support crash consistency and recovery in the case of unexpected power failure.

B. REMAPPING OPERATIONS

1) WRITE OPERATION

In the case of a write request, the proposed scheme performs sequential access by transforming the LBA of the request into a sequential LBA. This allows our scheme to improve the write performance even if the applications have diverse access patterns. To transform the request, we first search remapping tables to check if the original data page is in the storage. If there is no data page, we generate a new sequential LBA that is consecutive to the sequential LBA generated by the last remapping operation. Then, we assign a new transformed LBA to `slba` in `nvme_rw_command` struct of the request. To manage the original and transformed LBAs of the request, we create an entry using `remap_node` struct which consists of variables such as `orig_lba`, `trans_lba`, and `invalid_flag`. We store the original LBA and transformed LBA in `orig_lba` and `trans_lba` of the entry and insert the entry into the remapping table for other I/O operations such as read and overwrite. This is because we need to redirect read and overwrite requests to refer to the transformed LBA associated with the original LBA in the remapping table. After inserting the entry of the LBA pair,

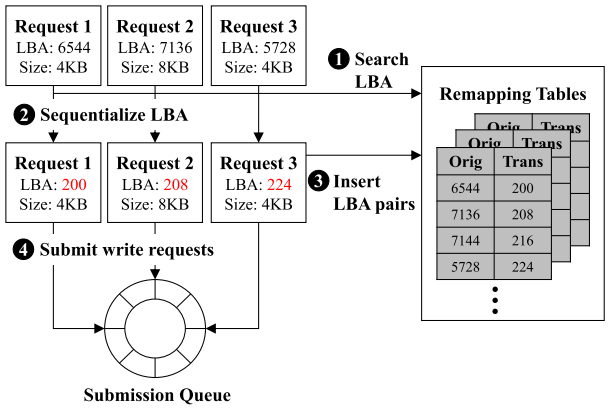


FIGURE 4. Remapping operations for write requests in the proposed scheme (The unit of LBA is 512 bytes. Thus, 4KB request increases the LBA by 8).

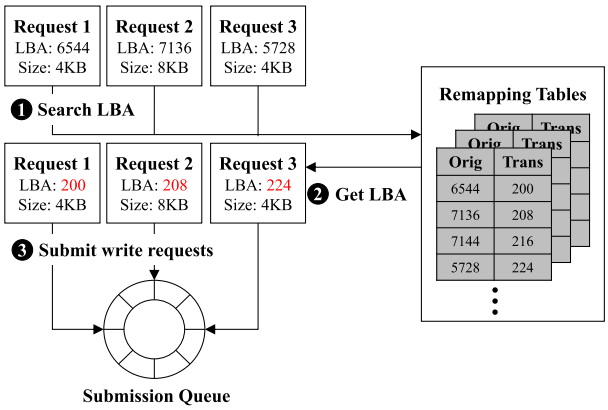


FIGURE 5. Remapping operations for read requests in the proposed scheme.

we submit the write request with the transformed LBA to the submission queue. Thus, the proposed scheme can improve the random write performance of NVMe SSD by issuing the write request in a sequential pattern.

Figure 4 shows an example of performing write operations in the proposed scheme. As shown in the figure, there are three write requests (request 1, 2, and 3) from the application. Since the original LBAs of the requests (i.e., 6544, 7136, and 5728) are not in sequential order, the requests are random write requests. Before sequentializing LBA to improve performance, we check if the data already exist by searching the entry in the remapping table (❶). If not found, the proposed scheme sequentializes LBAs of the requests by transforming them into sequential addresses (❷). To generate a new sequential address, we use the transformed LBA and size of the previous request. In this example, we transform the LBA of request 1 into 200 which is consecutive to the previous write request. We transform the LBA for the next requests using the request size and the transformed LBA of the previous request. Then, we assign a transformed LBA (i.e., 208 and 216) to request 2 that is consecutive to the transformed LBA of request 1. In the same way, request 3 obtains a transformed LBA (i.e., 224). After transforming LBAs for the requests, the original and transformed LBAs are inserted into the remapping table (❸). Note that the granularity of table entry is 4KB data; therefore, request 2 requires two entries with split original LBAs (i.e., 7136 and 7144) and transformed LBAs (i.e., 208 and 216) to be inserted in the table. Finally, the write requests with transformed LBAs will be submitted to the submission queue in the NVMe device driver (❹). Thus, by generating sequential access patterns for write requests, our proposed scheme can improve the performance of NVMe SSD.

2) READ OPERATION

In the case of a read request, the proposed scheme performs the address remapping operation to remap the original LBA of the read request to the transformed LBA. This is because the data page to be read is written in a different address which is transformed by the proposed scheme when a write request for the data was performed. Thus, it is necessary to find the transformed LBA from the remapping table to read the correct data. When the read request arrives at the NVMe device driver, we first search for the entry of `remap_node` struct in the remapping table. When the entry of `remap_node` containing the original LBA (`orig_lba`) is found, we change `slba` of `nvme_rw_command` struct for the request to the value of transformed LBA (`trans_lba`) in the entry. Finally, the read request is submitted to the submission queue. Thus, the proposed scheme can perform correct read operation by managing and searching transformed LBA in the remapping table.

Figure 5 shows an example of performing read operations in the proposed scheme. As shown in the figure, there are three read requests (request 1, 2, and 3) and they have original LBAs of 6544, 7136, and 5728. However, the data for the requests are not stored in the original LBAs. Therefore, we need to find the correct locations for the data. To do this, we use the original LBAs of the requests to search for the corresponding transformed LBAs in the remapping table (❶). After searching and finding each LBA pair in the remapping table, the read requests obtain the transformed LBAs (i.e., 200, 208, and 224) from the remapping table (❷). Note that these transformed LBAs were generated when previous write requests were performed. Then, with found transformed LBAs, the read requests are submitted to the submission queue (❸). Thus, this read procedure of the proposed scheme ensures correct read operation by forwarding the request to the transformed LBA where the page of the original data is located.

3) OVERWRITE OPERATION

In the case of an overwrite request, the proposed scheme performs the address remapping operation to generate a new sequential LBA for the request. This is because the type of the overwrite request is identified as a write request. Therefore, it is important to check if the entry for the request already exists in the remapping table to distinguish the overwrite request from normal write requests. When the overwrite

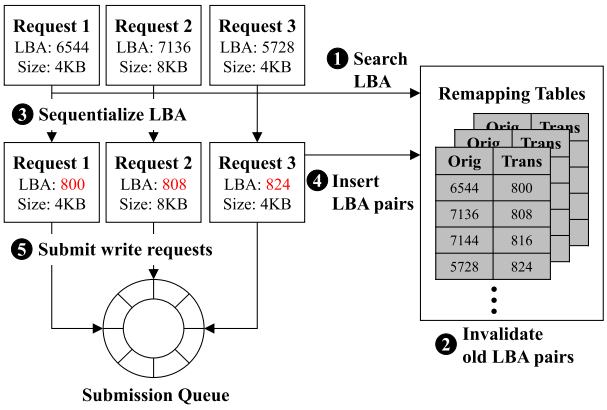


FIGURE 6. Remapping operations for overwrite requests in the proposed scheme (The unit of LBA is 512 bytes. Thus, 4KB request increases the LBA by 8).

request arrives, we first search table entries using the original LBA to find whether the data exist. If an entry with the same original LBA of the request is found in the remapping table, our scheme can notice that the request is not trying to write new data but overwriting previously written data. Since we overwrite previous data, we need to invalidate the previous data and write new data in the new transformed LBA. Therefore, the old `remap_node` struct containing outdated transformed LBA will be marked as invalid since it will be no longer accessed. To indicate invalidation of data, we set `invalid_flag` of the struct to true. When the invalidation of the old data is finished, we can proceed with generating a new table entry. To do this, we generate a new transformed LBA and store the original and transformed LBAs of the overwrite request in `orig_lba` and `trans_lba` of newly created `remap_node` struct. Then, we insert the entry into the remapping table. After successfully inserting the new entry, the overwrite request can be submitted with the new transformed LBA. Thus, the proposed scheme can accurately process the overwrite request by managing entries in the remapping table and using the flag to indicate the validity of data.

Figure 6 shows an example of performing overwrite operations in the proposed scheme. As shown in the figure, there are three overwrite requests (request 1, 2, and 3). Before generating a new LBA for the request, we check if the table entries with the same original LBAs (i.e., 6544, 7136, and 5728) already exist (❶). If the old entries (with transformed LBAs 200, 208, and 224) are found, we invalidate the entries by modifying `invalid_flag` fields (❷). Then, we give new sequential LBAs (i.e., 800, 808, and 824) to the requests (❸). Note that, as explained in III-B1, the granularity of the data unit is 4KB. Therefore, entries with transformed LBAs 208 and 216 are invalidated and new entries with transformed LBAs 808 and 816 for request 2 are generated. Before submitting the requests, new entries with transformed LBAs are inserted into the table (❹). Finally, the overwrite requests are submitted to the submission queue (❺). Thus, by checking the remapping tables and invalidating old data, our proposed scheme can perform the correct overwrite operation.

C. REMAPPING TABLES

Since the proposed scheme transforms LBAs of write requests, a list of transformed LBA needs to be maintained for correct I/O operations. To do this, we create a remapping table which is a table used to manage remapping information such as original and transformed LBAs. When processing I/O requests, we search the remapping table and insert new entries in the table. To enable searching the entries, we sort entries of the remapping table by the original LBA.

A single table entry is generated for 4KB of data and it requires approximately 3MB of memory for storing remapping information of 1GB of data in the storage. If the write operations are continuously executed, the number of entries in the table increases gradually. This can cause large memory space of the host occupied by remapping tables and the bottleneck in table operations. If the size of remapping entries in memory gets larger and takes up a large portion of memory capacity, we flush tables that are least recently updated to reserved space in the storage device for memory efficiency. Also, to reduce the remapping table management overhead, we divide the block address space into multiple partitions and create a remapping table for each partition (default size is 4GB). This allows our scheme to reduce the number of entries per remapping table. We note that partitions are filled in order of transformed LBA; therefore, consecutive partitions are filled up in order as write requests are sent. For implementation, we used red-black tree [32] which has the search time of O(logN). By partitioning a large address space into multiple partitions and using a red-black tree that supports fast tree operations, our scheme can manage the remapping information efficiently.

D. CLEANING OPERATIONS

In the proposed scheme, the data are written in a contiguous and sequential address space as it transforms the random write requests into sequential requests. However, when overwrite and delete operations are performed, the data and their LBAs are invalidated and no longer accessed. This creates holes of invalidated data in a contiguous address space. To remove these holes and reclaim the invalid LBAs, we perform the cleaning operation of the remapping table. Through cleaning operations, the proposed scheme creates a contiguous free space that can be used to store data from transformed random writes.

When the cleaning operation is performed, our proposed scheme first sorts table entries of consecutive partitions by transformed LBA. To find an invalid LBA to be reclaimed in sorted space, we traverse the `remap_node` entries from the beginning of the remapping table and find an entry with `invalid_flag` set to true. If the entry is found, we find another entry with valid data of which `invalid_flag` is set to false from the end of the remapping table. Then, we move the valid data to the LBA of the invalid entry. After

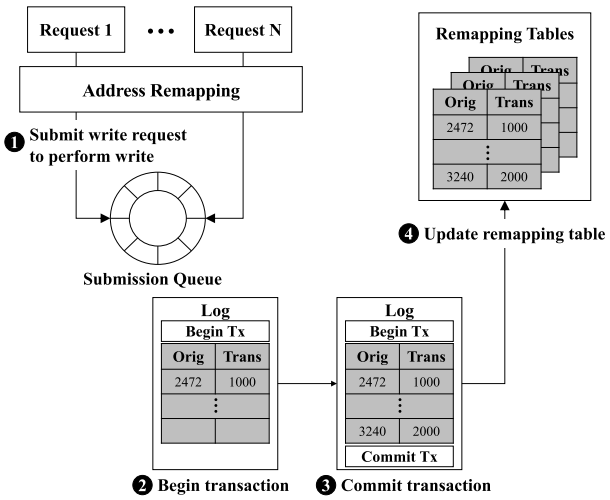


FIGURE 7. Procedure of transaction processing for crash consistency.

moving the data, we set `invalid_flag` to false, indicating that the valid data page has been copied into the LBA. Finally, the transformed LBA of the previously valid data is cleaned by removing the entry from the remapping table and deleting the data. By repeating the process, the proposed scheme can fill holes of invalid LBA with valid data. We note that we directly move the valid data to the invalid address, we do not need additional space for cleaning operations. In addition, the valid data page is moved to the holes starting from the end of the address space, creating a new contiguous address space at the end of the remapping table. The reclaimed space can be used for transforming future write requests into sequential patterns.

E. CRASH CONSISTENCY AND RECOVERY

The remapping table which stores the original and transformed LBAs resides in the memory to support fast table operations. However, as they are stored in volatile memory, the information can be lost in the case of a system shutdown. In addition, the remapping tables can be in an inconsistent state if the system crashes while remapping operations are being performed. Therefore, the proposed scheme uses the write-ahead logging (WAL) technique and transaction processing to provide atomicity and durability of remapping information in the case of system failure and restart.

To support crash consistency and recovery, we perform address remapping operations on write requests in the unit of transaction and record the operations on the remapping table in a log before modifying the remapping table. If a write request is transformed and successfully submitted to the submission queue, we start a new transaction and record the event by marking it at the beginning of the log. As more write requests are processed, we record pairs of original and transformed LBAs in the log. When a set of operations is executed successfully, we commit the transaction and the commit mark is written at the end of the log. After the commit operation, the remapping information of the transaction is reflected on the remapping tables in the memory. The log is saved in reserved log space inside persistent storage for later use such as the reconstruction of remapping tables.

Figure 7 shows an example of transaction processing for supporting crash consistency in the proposed scheme. As shown in the figure, incoming write requests are transformed by remapping operations, and the request is submitted to the submission queue (❶). As the write is performed which modifies the remapping table, a transaction starts and we record the original and transformed LBAs of the request in the log (i.e., 2472 and 1000) (❷). Note that the mark (Begin Tx) at the beginning of the log indicates the start of the transaction. After a set of remapping information is recorded in the log, we commit the transaction by writing the commit mark at the end of the log (❸). Finally, the remapping tables are updated by inserting pairs of original and transformed LBAs recorded in the transaction (❹). Thus, our scheme can successfully guarantee the crash consistency and recovery of remapping tables in the case of system failure and restart.

The transaction processing of log data requires managing additional remapping information besides that in the remapping table and persisting them in the reserved log. This additional step can cause write amplification. For transaction processing of I/O requests in our scheme, we use a similar policy as transaction processing of EXT4. The transaction can be processed and persisted based on predefined threshold values such as transaction size and time interval between transactions. By using the policies, our scheme persists 4KB of transaction data per 2MB of data in the storage. Therefore, the overheads by transaction processing can be reduced.

In the case of a system restart, the remapping table must be reconstructed from log data to allow incoming I/O requests to access previously written data. To recover the remapping tables from the logs, we need to check if the remapping table and the data are in a consistent state or not. That is, transactions in the logs have been either committed or ended unexpectedly by the system crash. Therefore, when recovering remapping tables, we need to first determine if the logs are valid by checking transaction start (Begin Tx) and commit (Commit Tx) marks. If the successful commit of a transaction before the crash is confirmed, we can rebuild a remapping table by creating LBA pairs recorded in the corresponding log and inserting them into the remapping tables. On the other hand, if the log does not contain the commit mark, we cannot use the log to rebuild the remapping tables. This is because the transaction without a commit mark is considered to be incomplete and some of the remapping information might be inconsistent with actual data written in the storage. Therefore, we collect and use only valid log data to recover the remapping tables. After reconstructing the remapping tables, we remove all the logs of transaction data and store restored remapping tables in reserved log space. The logs persisted in the future will be applied to previously stored tables to restore remapping information in the next system restart. In the case of an invalid transaction, there can be data written in transformed LBAs but not recorded

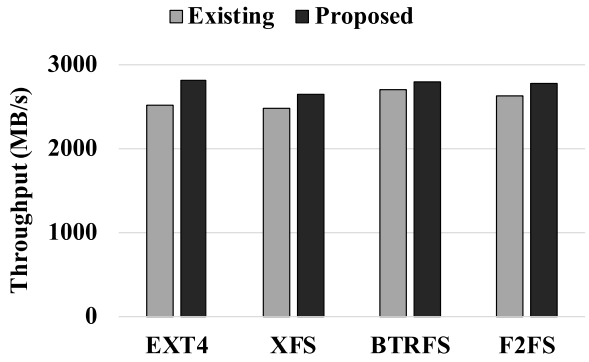


FIGURE 8. FIO benchmark performance of existing and proposed schemes in case of sequential writes.

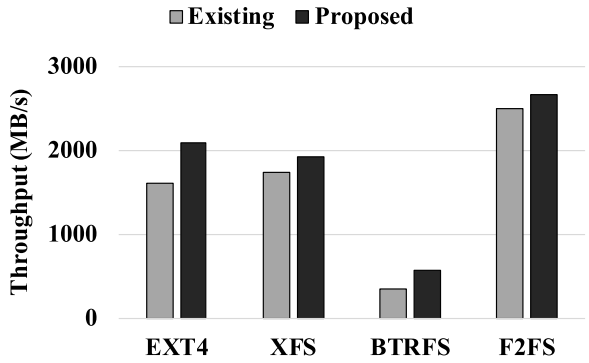


FIGURE 9. FIO benchmark performance of existing and proposed schemes in case of random writes.

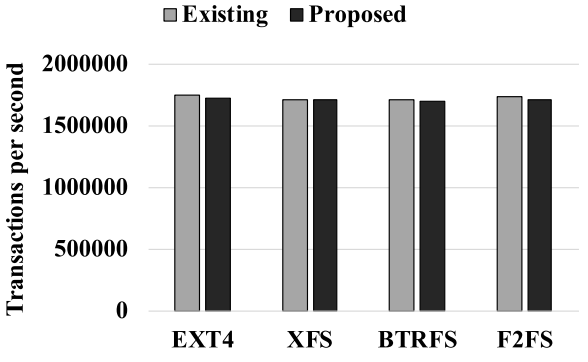


FIGURE 10. FFSB benchmark transactions per second of existing and proposed schemes in case of read operations.

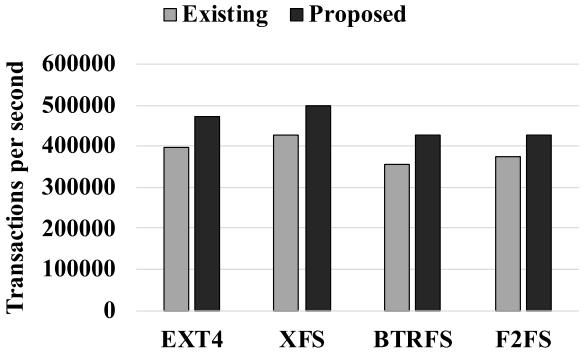


FIGURE 11. FFSB benchmark transactions per second of existing and proposed schemes in case of write operations.

in the transaction. Since we consider them to be invalid, the transformed LBAs in such transactions can simply be reused by future write requests. Thus, our scheme can successfully recover remapping information while providing consistency in the remapping operations.

IV. EVALUATION

A. EXPERIMENTAL SETUP

For the experiment setup, we used the system with Intel Core i9-9900K CPU (up to 5.00GHz) of 8 physical cores and 16GB of memory. For the storage devices, we used Samsung PM1725b NVMe SSD (MZPLL3T2HAJQ-00005). We used Ubuntu 20.04.4 LTS for the operating system. We implemented our scheme in Linux kernel 5.16.12 and compared the performance of the existing and proposed schemes. We used EXT4 [20], XFS [21], BTRFS [22], and F2FS [17] as file systems for evaluating performance. For the benchmarks, we used FIO [25] as a micro benchmark and used Flexible Filesystem Benchmark [33] as a macro benchmark. All experimental results are an average of five runs.

B. MICRO BENCHMARK

For the micro benchmark, we ran FIO benchmark with 8 threads, 8 GiB file per thread, 4 KiB block size, I/O queue depth of 16, and buffered I/O mode. We show the performance of sequential and random write using FIO in order to show performance improvement by our proposed scheme compared with the existing scheme.

Figure 8 shows the performance of the existing and proposed schemes in the case of sequential writes. As shown in the figure, our proposed scheme improves the performance by 11.4%, 7.0%, 3.1%, and 7.3% for EXT4, XFS, BTRFS, and F2FS, respectively. It shows small improvements even in the sequential writes due to the increased opportunities for storing disjoint, sequential data into consecutive block addresses.

Figure 9 shows the performance of the existing and proposed scheme in the case of random writes. The performance differences between the existing and proposed schemes of random writes are higher than those of sequential writes because the proposed scheme transforms the random access pattern into the sequential access pattern. As shown in the figure, our proposed scheme improves the performance by 30.4%, 11.2%, 64.1%, and 7.1% for EXT4, XFS, BTRFS, and F2FS, respectively. This shows that the address remapping scheme achieves higher I/O performance of NVMe SSD by transforming random writes into sequential writes.

C. MACRO BENCHMARK

For the macro benchmark, we used Flexible Filesystem Benchmark (FFSB) to evaluate the performance of the existing and proposed schemes when file I/O operations are executed. Using FFSB, we can simulate file operations such as

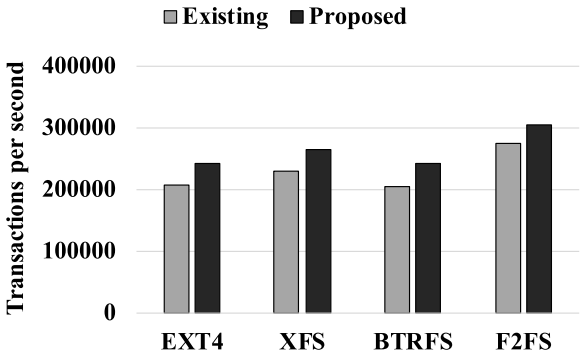


FIGURE 12. FFSB benchmark transactions per second of existing and proposed schemes in case of mixed operations.

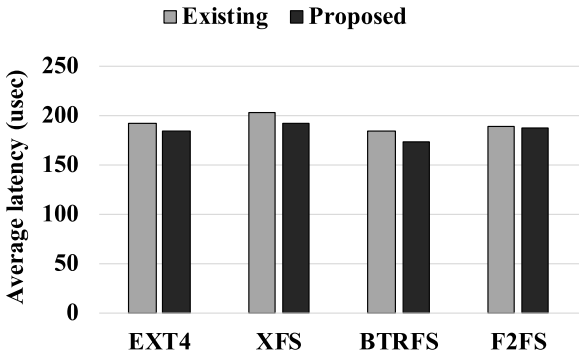


FIGURE 13. FIO benchmark average latency of existing and proposed schemes in case of sequential writes.

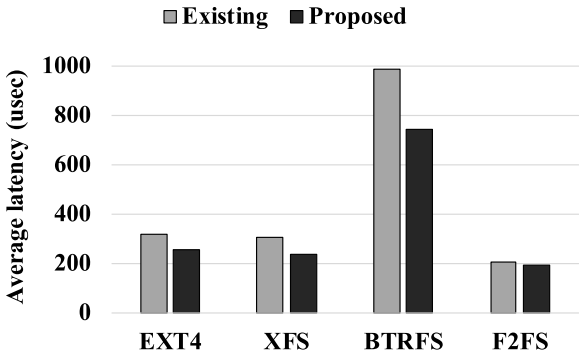


FIGURE 14. FIO benchmark average latency of existing and proposed schemes in case of random writes.

create, delete, read, write and append. We ran FFSB benchmark with the following configurations: 100 files/directories, 4GiB file size, 4GiB append size, 4KiB read/write size, and 4KiB read/write block size.

To show the impact of our scheme on the read performance, we ran FFSB with read only configuration. Figure 10 shows the performance of existing and proposed schemes in the case of file read operations. As shown in the figure, the proposed scheme performs 1.4%, 0.2%, 0.3%, and 1.1% lower numbers of transactions per second for EXT4, XFS, BTRFS, and F2FS, respectively. Our proposed scheme experiences degraded read performance with all file systems. This is because all read requests need to search for transformed LBAs that are mapped with the original LBAs of the requests in the remapping tables. Therefore, the proposed scheme introduces extra overheads in the case of read operations. However, performance loss by read operations is minimal.

To show the performance of file write operations, we ran FFSB with a write-only configuration. In this experiment, file write operations include write operations that overwrite created file sets and append operations that write new data at the end of files. Figure 11 shows the performance of existing and proposed schemes in the case of file write operations. As shown in the figure, our proposed scheme improves the transactions per second by up to 19.4%, 16.7%, 21.0%, and 13.7% compared with an existing scheme for EXT4, XFS, BTRFS, and F2FS file systems, respectively. This result shows that our scheme can improve the performance of NVMe SSD even though massive write operations are executed. In our scheme, overwrite requires searching and modification of a table entry in the remapping tables while append only needs inserting a new entry. The experimental results indicate that file write operations can be performed without noticeable latency and provides enhanced I/O performance.

To show the effectiveness of our scheme when file read and write operations are executed simultaneously, we ran FFSB with a 5:5 ratio of read and write. As shown in Figure 12, our proposed scheme improves the transactions per second by up to 16.9%, 15.7%, 18.2%, and 11.0% compared with an existing scheme for EXT4, XFS, BTRFS, and F2FS file systems, respectively. This indicates that our proposed scheme can successfully improve the I/O performance of NVMe SSD in case of massive file operations including simultaneous read and write requests.

D. OVERHEADS

To evaluate the overheads of the proposed scheme, we measured the latency and performance when the scheme performs address remapping, cleaning operation, and system restart.

1) ADDRESS REMAPPING LATENCY

To evaluate the overhead of address remapping, we measured the average latency of processing I/O, including I/O submission and completion. We ran FIO with the same configuration as the section above.

Figure 13 shows the latency of the existing and proposed schemes when sequential writes are performed. As shown in the figure, our proposed scheme shows similar latency to the existing scheme and the lowest latency of the proposed scheme is lower by up to 5.83% in the case of BTRFS. This demonstrates that the proposed scheme introduces minimal overheads of managing remapping tables even when performing sequential writes.

Figure 14 shows the latency of existing and proposed schemes when random writes are performed As shown in the figure, our proposed scheme experiences 19.8%, 20.9%, 24.5%, and 7.5% lower latency for EXT4, XFS, BTRFS, and F2FS, respectively. This indicates that no significant

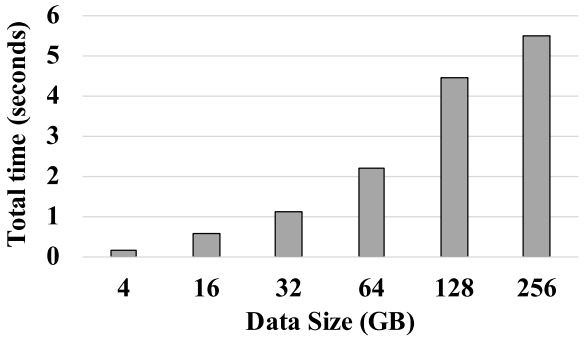


FIGURE 15. Reconstruction latency for various data sizes.

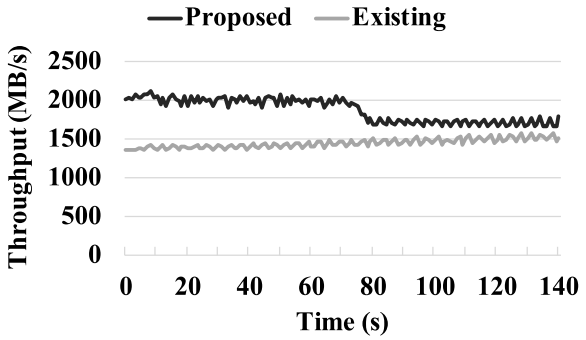


FIGURE 16. Cleaning overhead of the proposed scheme.

overhead occurs from remapping operations and overall I/O processing benefits from performing sequential writes transformed from random writes in our proposed scheme.

2) RECONSTRUCTION LATENCY

To evaluate the latency incurred when the system restarts, we measured the reconstruction time with data of various sizes.

Figure 15 shows the latency of the reconstruction required for various sizes of data in the storage, up to 256GB. As shown in the figure, our proposed scheme requires up to 5.49s to reconstruct remapping tables. This demonstrates that the proposed scheme introduces little overheads to reload address remapping information when the system restarts.

3) CLEANING OVERHEADS

To show the overheads of the cleaning operation, we ran FIO benchmark using the proposed scheme and EXT4 file system. We used the same configuration as IV-B but different data sizes. In this experiment, 256GB of data were previously written in the storage and we used FIO to overwrite the data to trigger the cleaning operation. In the process of overwriting, old data pages become invalid and the cleaning operation is performed.

Figure 16 shows the performance of the existing and proposed schemes when overwriting the data on EXT4. As shown in the figure, the cleaning operation causes the performance of the proposed scheme degrades up to 22.1%. However, the proposed scheme performs better compared to the existing scheme even in the case of cleaning operations. This demonstrates that our proposed scheme can suffer from cleaning overheads but can improve performance in general. We only included the experimental result with EXT4 since experimental results on other file systems show similar trends.

V. RELATED WORKS

A. OPTIMIZING I/O PERFORMANCE OF NVMe SSDs

Many previous studies have proposed optimization of I/O performance for NVMe SSDs. Kim et al. [34] provides a user-level I/O framework that allows users to select I/O policies such as allocating a dedicated queue and using a polling mechanism. D2FQ [35] presents I/O scheduling with fair-queueing that leverages the NVMe WRR feature. It sends I/O requests to one of the three queues which process the requests at different speeds while guaranteeing fairness. Lee et al. [36] proposes a new design that asynchronously performs operations in the I/O stack to overlap CPU operations with device I/O. This design improves I/O performance by reducing time spent in the block I/O layer. H-NVME [37] is a hybrid framework for virtual machines that allow fully utilizing the capabilities of NVMe SSDs. By providing two different deployment modes, H-NVME allows users to use lightweight parallel queues or directly access the NVMe devices to improve performance based on priorities.

Our study is in line with these studies [34], [35], [36], [37] in terms of improving the I/O performance of the systems using NVMe SSDs. In contrast, our work focuses on improving the performance by sequentializing random writes in the NVMe device driver.

B. OPTIMIZATION BASED ON CHARACTERISTICS OF SSDs

There have been several studies on improving I/O performance by mitigating overheads from the internal mechanism of flash-based SSDs. LAST [38] identifies the locality of write requests to improve I/O performance. It places data into separate log buffers by distinguishing sequential and random writes. By clustering data separately according to the temporal and sequential locality, LAST can reduce the merge cost. Park et al. [39], [40] designs a hot data identification scheme that helps place hot data in the same block. By classifying hot and cold data with low memory usage, it can reduce overheads from garbage collection. Sun et al. [31] presents a scheme that writes data on fragmented pages of blocks in SSDs. This reduces valid pages in the blocks to be erased and thus, decreasing page copy latency.

Our study is in line with these studies [31], [38], [39], [40] in terms of minimizing the negative impact of garbage collection and improving the I/O performance. However, we focus on improving I/O performance by transforming the access pattern of write operations from sequential to random writes.

C. LOG-STRUCTURED SCHEME ON SSDs

There have been several studies on sequentially writing data blocks in order to reduce performance degradation from random writes. Log-structured file system (LFS) [41] is a storage management scheme to store data in a structure called a

log and write all modifications sequentially in the storage device. Since it is originally designed for HDDs, it improves write time by eliminating seeks. With the advance of SSDs, many file systems are designed for flash memory [13], [17]. F2FS [17] is a file system that supports a flash-friendly disk layout. It writes data in a sequential manner by adopting append-only logging and separates hot and cold data to improve the performance of SSDs.

Other works have investigated approaches to performing sequential writes in levels other than the file system. Z-MAP [42] is a space management scheme that maintains buffers in the granularity of zone which is a group of multiple flash blocks. By buffering random writes in the partition called a streaming buffer zone, it can log random writes sequentially. SHRD [18] improves spatial locality by logging random writes into reserved log space. It sequentializes random writes from the SCSI device driver and SHRD-supporting FTL manages mapping information. ReSSD [19] exposes a virtual block device that identifies small random writes and transforms them into sequential and ordered-sequential writes. Kim et al [43] reshapes random writes into sequential writes in distributed file systems.

Our study is in line with these studies [13], [17], [18], [19], [41], [42] in terms of improving the performance by sequentializing random writes. In contrast, we focus on the performance improvement of NVMe SSD by implementation in the NVMe device driver. In addition, since our scheme is based on a device driver, our scheme can be applied to the individual device without affecting other devices in a storage system.

VI. CONCLUSION

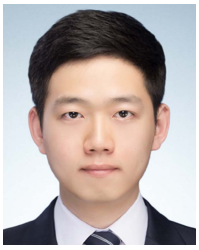
In this paper, we propose an address remapping scheme for NVMe SSDs to improve the I/O performance. To do this, we design and implement the address remapping scheme that intercepts I/O requests and transforms random writes into sequential writes in the NVMe device driver. Also, we provide methods to manage information on transformed addresses, perform cleaning operations, and guarantee consistency of address remapping operations. The experimental results show that our proposed scheme can improve the I/O performance by up to 64.1% compared with the existing scheme.

In our future works, we plan to add more functionalities such as caching hot data before accessing remapping tables. In addition, we will evaluate the effect of on-demand reconstruction of remapping tables to examine the trade-off between reconstruction latency and interference with I/O performance. The scheme can be applied to storage devices that have a performance gap between sequential and random writes. We also plan to consider applying the scheme to other types of devices with a performance gap.

REFERENCES

[1] H. Cai, B. Xu, L. Jiang, and A. V. Vasilakos, "IoT-based big data storage systems in cloud computing: Perspectives and challenges," *IEEE Internet Things J.*, vol. 4, no. 1, pp. 75–87, Jan. 2016.
[2] W. Zhou, D. Feng, Z. Tan, and Y. Zheng, "Improving big data storage performance in hybrid environment," *J. Comput. Sci.*, vol. 26, pp. 409–418, May 2018.
[3] S. W. D. Chien, A. Podobas, I. B. Peng, and S. Markidis, "tf-Darshan: Understanding fine-grained I/O performance in machine learning workloads," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2020, pp. 359–370.
[4] B. K. Debnath, S. Sengupta, and J. Li, "FlashStore: High throughput persistent key-value store," in *Proc. PVLDB*, 2010, vol. 3. no. 2, pp. 1414–1425.
[5] Y. Zou, A. Awad, and M. Lin, "DirectNVM: Hardware-accelerated NVMe SSDs for high-performance embedded computing," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 1, pp. 1–24, Jan. 2022.
[6] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proc. 9th USENIX Conf. File Storage Technol. (FAST)*, 2011, p. 20.
[7] K. R. Krish, A. Anwar, and A. R. Butt, "HatS: A heterogeneity-aware tiered storage for Hadoop," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2014, pp. 502–511.
[8] X. Zhao, Z. Li, X. Zhang, and L. Zeng, "Block-level data migration in tiered storage system," in *Proc. 2nd Int. Conf. Comput. Netw. Technol.*, 2010, pp. 181–185.
[9] C. Li, H. Chen, C. Ruan, X. Ma, and Y. Xu, "Leveraging NVMe SSDs for building a fast, cost-effective, LSM-tree-based KV store," *ACM Trans. Storage*, vol. 17, no. 4, pp. 1–29, Nov. 2021.
[10] J. Chu, Y. Tu, Y. Zhang, and C. Weng, "LATTE: A native table engine on NVMe storage," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 1225–1236.
[11] D. Shankar, X. Lu, and D. K. Panda, "Boldio: A hybrid and resilient burst-buffer over lustre for accelerating big data I/O," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 404–409.
[12] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An interference-aware dynamic I/O scheduler for shared non-volatile burst buffers," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2018, pp. 290–301.
[13] 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.
[14] J. Guo, Y. Hu, B. Mao, and S. Wu, "Parallelism and garbage collection aware I/O scheduler with improved SSD performance," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 1184–1193.
[15] C. Matsui, A. Arakawa, C. Sun, and K. Takeuchi, "Write order-based garbage collection scheme for an LBA scrambler integrated SSD," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 2, pp. 510–519, Feb. 2017.
[16] K. Smith, "Garbage collection," SandForce, Flash Memory Summit, Santa Clara, CA, USA, 2011, pp. 1–9.
[17] 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.
[18] 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.
[19] Y. Lee, J.-S. Kim, and S. Maeng, "ReSSD: A software layer for improving the small random write performance of SSDs," *J. Inf. Sci. Eng.*, vol. 28, no. 6, pp. 999–1018, 2012.
[20] 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.
[21] 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)*, 1993, pp. 71–78.
[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] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 17–32.
[24] J. Hines, "Stepping up to summit," *Comput. Sci. Eng.*, vol. 20, no. 2, pp. 78–82, Mar. 2018.
[25] *FIO Benchmark*. Accessed: Sep. 27, 2022. [Online]. Available: http://freecode.com/projects/fio
[26] K. Eshghi and R. Micheloni, "SSD architecture and PCI express interface," in *Inside Solid State Drives (SSDs)*. Cham, Switzerland: Springer, 2013, pp. 19–45.

[27] K. Takeuchi, "Novel co-design of NAND flash memory and NAND flash controller circuits for sub-30 nm low-power high-speed solid-state drives (SSD)," *IEEE J. Solid-State Circuits*, vol. 44, no. 4, pp. 1227–1234, Apr. 2009.
[28] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. Israeli Experim. Syst. Conf. (SYSTOR)*, 2009, pp. 1–9.
[29] 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.
[30] 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 ATC)*, 2019, pp. 799–812.
[31] C. Sun, A. Soga, C. Matsui, A. Arakawa, and K. Takeuchi, "LBA scrambler: A NAND flash aware data management scheme for high-performance solid-state drives," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 1, pp. 115–128, Jan. 2015.
[32] 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.*, 2008, pp. 1–8.
[33] *The Flexible Filesystem Benchmark*. Accessed: Sep. 27, 2022. [Online]. Available: https://github.com/FFSB-Prime/ffsb
[34] H.-J. Kim and J.-S. Kim, "A user-space storage I/O framework for NVMe SSDs in mobile smart devices," *IEEE Trans. Consum. Electron.*, vol. 63, no. 1, pp. 28–35, Feb. 2017.
[35] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ: Device-direct fair queueing for NVMe SSDs," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 403–415.
[36] 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.
[37] Z. Yang, M. Hoseinzadeh, P. Wong, J. Artoux, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "H-NVMe: A hybrid framework of NVMe-based storage system in cloud computing environment," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.
[38] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operat. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, Oct. 2008.
[39] 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.
[40] D. Park, B. Debnath, Y. Nam, D. H. C. Du, Y. Kim, and Y. Kim, "HotDataTrap: A sampling-based hot data identification scheme for flash memory," in *Proc. 27th Annu. ACM Symp. Appl. Comput.*, 2012, pp. 1610–1617.
[41] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.
[42] Q. Wei, C. Chen, M. Xue, and J. Yang, "Z-MAP: A zone-based flash translation layer with workload classification for solid-state drive," *ACM Trans. Storage*, vol. 11, no. 1, pp. 1–33, Feb. 2015.
[43] 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.



DONG KYU SUNG received the B.S. degree in computer science from the University of Minnesota Twin Cities, Minneapolis, USA, in 2018. He is currently pursuing the Ph.D. degree in computer science and engineering with Seoul National University. His research interests include distributed file systems, key-value stores, burst buffer, and operating systems.

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

HYEONSANG EOM received the B.S. degree in computer science and statistics from Seoul National University (SNU), Seoul, South Korea, in 1992, and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, MD, USA, in 1996 and 2003, respectively. He is currently a Professor with the Department of Computer Science and Engineering, SNU, where he has been a Faculty Member, since 2005. His research interests include high performance storage systems, operating systems, distributed systems, cloud computing, energy efficient systems, fault-tolerant systems, security, and information dynamics.

SUNGGON KIM received the B.S. degree in computer science from the University of Wisconsin–Madison, Madison, USA, in 2015, and the Ph.D. degree in computer science and engineering from Seoul National University, in 2021. He was an Intern at the Lawrence Berkeley National Laboratory, CA, USA, in 2018, 2019, and 2020. He was a Postdoctoral Research Associate at Seoul National University. Currently, he is an Assistant Professor with the Department of Computer Science and Engineering, Seoul National University of Science and Technology. His research interests include file systems, cloud computing, big data, distributed systems, and operating systems.