computers

MDPI

Article

Sequentialized Virtual File System: A Virtual File System Enabling Address Sequentialization for Flash-Based Solid State Drives

Inhwi Hwang [1], Sunggon Kim [2], Hyeonsang Eom [1] and Yongseok Son [3,*]

[1] Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea
[2] Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea
[3] Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, Republic of Korea
* Correspondence: sysganda@cau.ac.kr

Abstract: Solid-state drives (SSDs) are widely adopted in mobile devices, desktop PCs, and data centers since they offer higher throughput, lower latency, and lower power consumption to modern computing systems and applications compared with hard disk drives (HDDs). However, the performance of the SSDs can be degraded depending on the I/O access pattern due to the unique characteristics of SSDs. For example, random I/O operation degrades the SSD performance since it reduces the spatial locality and induces garbage collection (GC) overhead. In this paper, we present an address reshaping scheme in a virtual file system (VFS) called sVFS for improving performance and easy deployment. To do this, it first sequentializes a random access pattern in the VFS layer which is an abstract layer on top of a more concrete file system. Thus, our scheme is independent and easily deployed on any concrete file systems, block layer configuration (e.g., RAID), and devices. Second, we adopt a mapping table for managing sequentialized addresses, which guarantees correct read operations. Third, we support transaction processing for updating the mapping table to avoid sacrificing the consistency. We implement our scheme at the VFS layer in Linux kernel 5.15.34. The evaluation results show that our scheme improve the random write throughput by up to 27%, 36%, 34%, and 2.35× using the microbenchmark and 25%, 22%, 20%, and 3.51× using the macrobenchmark compared with the existing scheme in the case of EXT4, F2FS, XFS, and BTRFS, respectively.

Keywords: operating system; file system; virtual file system; solid-state drive

Citation: Hwang, I.; Kim, S.; Eom, H.; Son, Y. Sequentialized Virtual File System: A Virtual File System Enabling Address Sequentialization for Flash-Based Solid State Drives. *Computers* **2024**, *13*, 284. https://doi.org/10.3390/computers13110284

Academic Editor: Paolo Bellavista

Received: 26 September 2024
Revised: 22 October 2024
Accepted: 30 October 2024
Published: 2 November 2024

1. Introduction

In modern computing, the need for faster computation resources such as CPU, main memory, and storage devices increases. The rise of deep learning and big data applications, characterized by massive data generation and processing [1,2], especially underscores the critical need for improved I/O performance. To improve I/O performance, solid-state drives (SSDs) are widely utilized for data storage in various systems such as mobile devices, desktop PCs, and data centers [3–5] due to high throughput, low latency, and low power consumption compared with hard disk drives (HDD).

Despite the advantages of SSDs, their performance can highly depend on the I/O access pattern from applications [6,7]. For instance, the sequential write requests have a high possibility of storing pages that have a similar lifetime in the same block in SSDs so that the pages are invalidated at a similar time [8]. This reduces the overhead of copying valid pages to free blocks and hence write amplification and GC overhead. In contrast, the spatial locality is low in the case of random write; thus, it introduces higher GC overhead and increases write amplification compared to sequential write [6,9].

We perform a preliminary experiment to show the performance degradation of the random write on various file systems. We use a Samsung 860 pro 256GB SSD, which has

530 MB/s of maximum sequential write throughput and up to 43,000 KIOPS of random write performance [10]. We evaluate the performance of sequential and random write on the EXT4, BTRFS, F2FS, and XFS file systems. Figure 1 shows the results of the experiment. The throughput of random write is about 25%, 28%, and 27% lower than sequential write on EXT4, F2FS, and XFS, respectively. Even worse on BTRFS, the throughput of random write is 3.6× slower than sequential write. This slowdown results from the fact that the I/O access pattern induces the write amplification and GC overhead, and, finally, it affects the performance of SSDs.

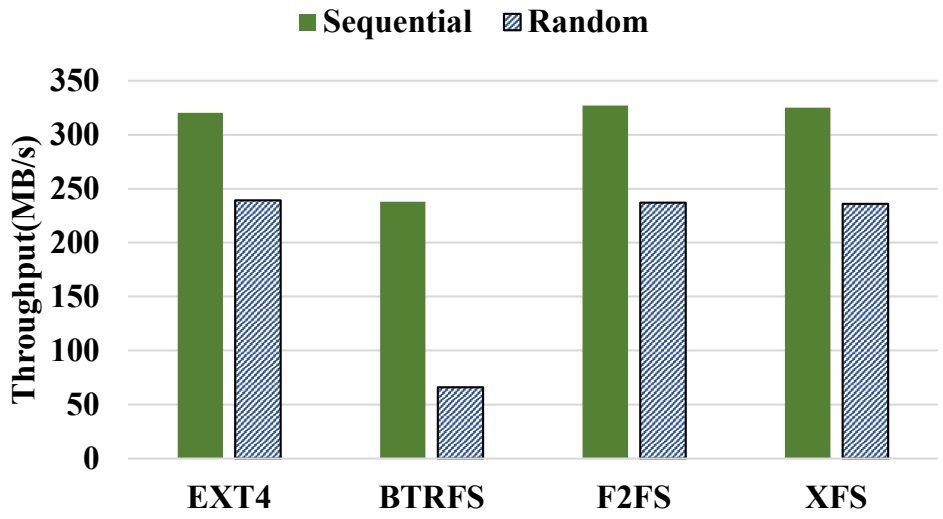


Figure 1. Performance of sequential and random write on SSD.

In previous studies, proposed schemes on various layers handle the performance issue by a random I/O access pattern. SHRD [11] transforms random write requests to sequential write requests in the block device to reduce the performance degradation of SSDs by the random access pattern. F2FS [7] is a flash-friendly file system that writes data sequentially in a log-structured manner. Our work is in line with these previous studies [7,11] in terms of sequentializing write requests to improve the SSD performance. In contrast, we focus on sequentializing the write requests at a virtual file system (VFS) layer. Because VFS is a common interface on top of various file systems, our scheme can improve the I/O performance with any type of file system, block layer configurations, and devices.

In this paper, we propose a sequentialized VFS, called sVFS, which transforms random access patterns into sequential access patterns to reduce write amplification and GC overhead. We design a sequentialization scheme on top of VFS and sequentialize write requests at the file level since the layer is an abstract layer on top of the file systems and all I/O operations in the existing software stack pass through the VFS layer. Hence, sVFS is independent and easily deployed on any file systems, block layer configurations (e.g., RAID), and devices.

To sequentialize the access pattern, after the VFS layer receives a write request for a file, sVFS transforms the logical address on the request to the last position of the corresponding file. For read consistency, sVFS stores the mapping information of the original and transformed logical address in a per-file managed mapping table. Then, when reading a file, sVFS searches the mapping entry of the original and transformed logical address from the mapping table and re-allocates the original logical address to the read request. To maintain consistency, we also support transaction processing to update the mapping table. sVFS improves the random write performance by up to 3.51× while maintaining sequential write performance without sacrificing consistency.

The contributions of our scheme are as follows:

- We investigate the VFS layer to enable sequentialization to improve the SSD performance.

- We design and implement a scheme that sequentializes write requests called sVFS.
- We show that sVFS can improve the random write performance up to 3.51× compared with existing VFSs.

In the rest of the paper, we first introduce the background and motivation of our study in Section 2. Related works are described in Section 3. Then, we explain our sequentialization scheme on VFS to convert random write to sequential write and a detailed implementation in Section 4. After we present the evaluation results of our scheme to verify its validity in Section 5, we discuss the results in Section 6 and conclude this paper in Section 7.

2. Background and Motivation

2.1. Characteristics of SSDs

SSDs have different internal structures and characteristics from conventional HDDs. We focus on two main differences in this paper: out-of-place update and garbage collection.
Out-of-place update: Due to the way data are stored in flash media, SSDs read or write data in a flash page unit and erase data in a flash block unit. Since the size of flash blocks (e.g., a few MB) is typically larger than that of flash pages (e.g., tens of KB), to modify data in a flash page, SSDs erase the entire block where the page is located before writing data. However, because the erase operation is costly, erasing a flash block every time a flash page is modified can introduce significant overhead. Thus, SSDs do not overwrite flash pages directly to update data in specific physical locations.

Instead, SSDs write the data to pages in different physical locations to update them (i.e., out-of-place update). Since data modification in SSDs changes the physical location of data with the same LBA from the upper layer in the storage stack, the flash translation layer (FTL) is introduced to map LBA to the physical location of the data. FTL translates logical block addresses from hosts to physical pages or block addresses in SSDs.
Garbage collection: SSD internally performs garbage collection (GC) to reclaim free space, due to the difference in the units of write and erase operations. After flash pages in a block are updated or invalidated, some pages in a block can remain valid since the pages have a different lifetime from the invalidated pages. For GC, SSD migrates the valid pages in a victim block and erases the victim block to reclaim the space occupied by invalid pages. Since the number of valid pages in a block directly affects the GC overhead, storing pages with a similar lifespan can reduce write amplification from flash page migration and GC overhead.

2.2. I/O Access Pattern on SSDs

The I/O access pattern can significantly impact the performance of SSDs [6–8]. Small random writes, especially, can reduce the overall throughput and aggravate latency compared to sequential writes [12,13]. Previous studies have reported that the random write access pattern with a small request size can degrade the performance of SSDs since it can harm write amplification from GC [6] and underutilize the parallelism of SSDs [14].

Various sequentialization schemes have been adopted for the diverse storage stack layers to overcome the performance gap between random and sequential write. For example, Rocksdb [15] and LevelDB [16] adopted random write sequentialization to the application layer. SFS [6] and F2FS [7] sequentialized random write in the file system layer. SHRD [11] modified the block device driver and device to sequentialize random write. However, they still have a problem that is dependent on the application [15,16] and file systems [6,7], or lacks user information [11].

2.3. Virtual File System

The virtual file system (VFS) layer abstracts various file systems and provides the same interface to the user. The VFS layer operates the file I/O requests from the user level. Applications and libraries at the user level first open files with flags to indicate various options for file operations. Then, they issue system calls with the parameters

including the file descriptor, data buffer, count, and file offset to read or write. In the VFS layer, the I/O requests are processed through the page cache. Under certain conditions (e.g., when requested pages to be read do not exist in the page cache or the dirty page threshold is exceeded), the I/O requests are sent to an underlying file system. In a file system, the I/O requests are transferred with LBA and forwarded to lower layers. The requests pass through lower layers such as the block layer and device drivers, eventually reaching the SSDs.

Existing sequentialization in the application level, file system layer, block layer, device driver, and device can affect only the specific range where the schemes are targeted. Meanwhile, since the VFS layer provides a common and general interface in a storage system, the effect of sequentialization in the VFS layer can be widely applied to the storage system in a more non-specific manner. Furthermore, since the VFS layer is the closest layer to the user in the kernel storage stack, sufficient user information can be provided and exploited for optimization. As described in Section 6, we plan to design and evaluate the optimization exploring hints from the user.

3. Related Works

3.1. SSD Optimization

There are several approaches to optimize SSDs considering the unique characteristics of SSDs. One approach is to exploit the internal parallelism of SSDs [17–19]. Works using this approach analyze the parallelism level of SSDs and/or optimize the existing software stack to process multiple I/O requests efficiently using SSDs. Ref. [17] analyzed the performance of online transaction processing systems using scientific data depending on the parallelism level of SSDs. Ref. [18] proposed a write request rescheduling scheme and [19] modified the block scheduler to utilize the internal parallelism of SSDs.

Another approach is to reduce GC overhead [20–22]. Since GC reads victim blocks and copies valid pages in the blocks in SSDs, it interferes with the user write requests from the host machine. Therefore, reducing GC overhead is one way to improve the performance of SSDs. SWAN [20] presented a spatial separation approach to alleviating the performance interference caused by GC. This partitions the storage devices into the front-end devices dedicated to serving write requests and the back-end devices where GC is performed. Ref. [21] reduced request tail latency using reinforcement learning to exploit the idle time of SSDs. Ref. [22] proposed a data hotness prediction scheme to reduce the WAF and tail latency of write requests in SSDs. Our work is in line with these works [17–22] in terms of optimizing the target performance considering or utilizing the characteristics of SSDs or reducing GC overheads. In contrast, our work focuses on transforming the I/O access pattern using the common interface and virtual file system.

3.2. Write Sequentialization

There are studies of sequentializing access I/O patterns under various layers. LSM-tree [23] is a data structure to store data using append operations. It buffers data in memory and flushes the buffered data in the storage device. Data in the storage device are sorted and compacted for read performance and space utilization. RocksDB [15] and LevelDB [16] are key-value stores using the LSM-tree and adopt write sequentialization in the application layer. LFS [24] is a file system which stores data in a log-structured manner. It collects file system changes in memory in logs and writes the logs to the storage device sequentially. F2FS [7] is an LFS-based file system which also sequentializes random access to sequential access. It is designed to utilize SSDs considering its unique characteristics. SHRD [11] also adopts random write sequentialization on the block device driver layer. It remaps the address of write requests from a file system to reduce GC overhead and increase spatial locality.

These works are in line with our work in terms of transforming the random access pattern to a sequential access pattern to improve the write performance of SSDs. In contrast, our work focuses on the virtual file system by transforming file positions for more

independence. Thus, it can be applied to the system regardless of any file systems, block layer configurations, device drivers, and devices.

4. Design and Implementation

4.1. System Overview

In this paper, we propose sVFS which enables VFS to sequentialize write requests to improve the performance of SSDs. To do this, sVFS transforms the file position of incoming write requests to a per-file managed file position which continuously increases until cleaning and we refer to it as the next file position (NFP). Since sVFS maintains the file position in a per-file manner, multiple threads can write their own files independently with their own reshaped file positions.

Figure 2 shows the architecture overview of sVFS. As shown in the figure, sVFS obtains and increases the next file position (NFP) of a file to sequentialize an incoming write request of the file (①–②). The next file position is the last written position of the file by the previous write request and it increases by the byte size to write. sVFS sequentializes the write request by transforming the original file position of the write request into the value of NFP (③). After sequentialization, the mapping information of the original and sequentialized position is stored in the file position mapping table (FP mapping table) (④). The table is needed to find the file position where the data is located when reading a file. Then, sVFS requests the sequentialized write request to the file system layer (⑤). As a result, the sequentialized file positions allow the file system to generate logical block address (LBA) sequentially.

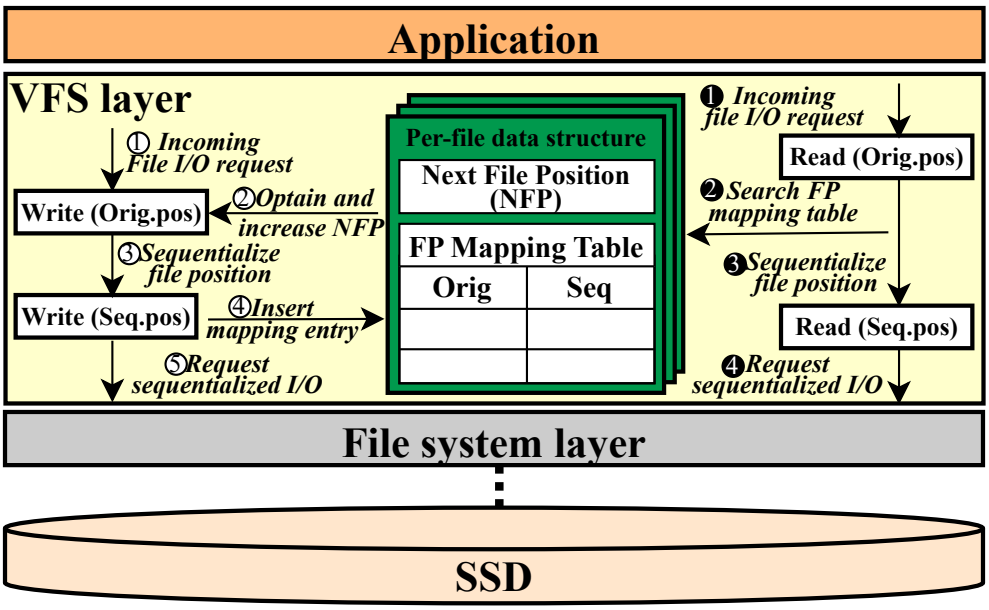


Figure 2. Architecture overview of the proposed scheme (sVFS). (FP: file position, Orig.: original position, Seq.: sequentialized position.)

For read requests, sVFS needs to transform the file position of requests since sVFS has already modified the positions that users request. To do this, sVFS searches the sequentialized position in the FP mapping table using the original position of the request as a key (❶–❷). Then, it modifies the file position of the request into the sequentialized file position (❸) and issues the read request to the underlying file system (❹).

4.2. I/O Operations

4.2.1. Write Operation

When a write request is issued, sVFS atomically obtains and increments the next file position (NFP) of the corresponding file. sVFS transforms the file position of the write

request to the NFP value. Since the NFP value continuously increases until the cleaning operation is performed, write requests to a file can be sequentialized at the VFS level. If the original file position of the request is already contained in the file position mapping table, the existing mapping is updated. Otherwise, the mapping of the original file position and sequentialized position is inserted into the file position mapping table. The transformed write request is issued to the lower layer in the kernel storage stack such as the file system layer. As well as overwrite or append, sVFS can support a write on a non-existing area of a file (e.g., sparse file), since the write procedure of sVFS transforms an arbitrary file position from an I/O request to a sequentialized position using NFP. Accordingly, in the file position mapping table (FP mapping table), sVFS only stores and manages the mapping table entries for issued write requests. As a result, even if spare file I/Os with arbitrary position are requested, sVFS inserts the entries only for the actual issued requests in the mapping table sequentially and keeps the mapping table as compact as possible.

Figure 3 shows an example of file position sequentialization of sVFS in the write operation. As shown in the figure, there are three write operations to a file randomly and the states of the next file position and mapping table from time *T1* to *T4*. At time *T1*, before write requests are issued, an initial value of the next file position is 0. At time *T2*, the first write request is issued with a file position (12 K). First, sVFS obtains the current value of the next file position of the file and increases it by the size of the request (4KB) for the next write request. In this case, the next file position increases from 0 to 4 K (①). sVFS transforms the request using the next file position as a sequentialized position (②). Then, the mapping information between the original position (12 K) and the sequentialized position (0) is inserted into the mapping table (③). This allows sVFS to find the sequentialized position where the data is stored during read operations. Finally, sVFS issues sequentialized write requests to a file system layer (④). At times *T3* and *T4*, write requests are sequentialized in the same manner as the previous one. For example, at time *T3*, a write request on 100K file position is transformed to 4 K position, and the mapping information (<100 K, 4 K>) is inserted into the mapping table. Similarly, at time *T4*, the write request on 0 file position is transformed to 8 K position, and the mapping information (<0, 8 K>) is inserted into the mapping table.

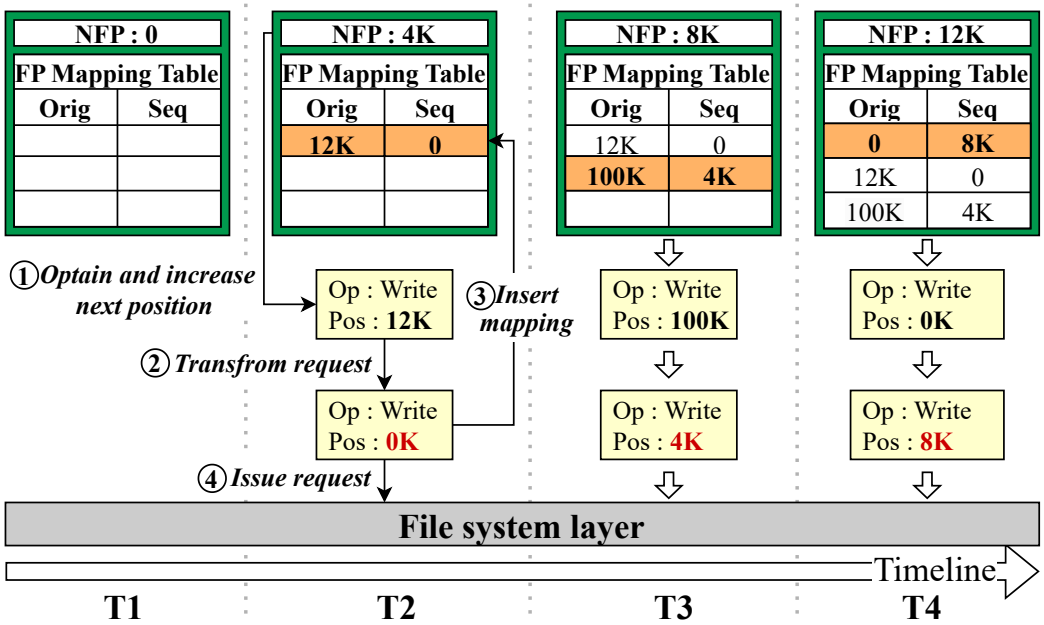


Figure 3. An example of file position sequentialization of sVFS in write operations. When an entry of the original and sequentialized position is inserted in the mapping table, the entries are sorted by the original position. (NFP: next file position, FP: file position, Orig.: original position, Seq.: sequentialized position.)

4.2.2. Read Operation

Since the file positions are sequentialized when write operations occur, it is necessary to support correct read operations by finding the sequentialized file position. To do this, sVFS converts the original position of the read request to the sequentialized position using the FP mapping table. As we described, sVFS inserts an entry of original and sequentialized file positions to the mapping table whenever new write and overwrite operations are performed. Therefore, it allows searching the entry of the mapping table using the original position as a key. After finding the sequentialized position, sVFS starts the read operation by calling a specific file system with the sequentialized position.

Figure 4 shows read operations in sVFS. As shown in the figure, three requests are issued from time *T1* to *T4*. Before read requests are issued, as shown at time *T1*, sVFS maintains the next file position and mapping table including original and sequentialized file positions. At time *T2*, a read request on 12 K position in the file is issued. sVFS uses the requested position as a key and searches the corresponding entry (①). In this case, sVFS finds the mapping of <12 K, 0> in the table (②). After obtaining the sequentialized position (0), sVFS issues a read request using the position to file system layer (③). At times *T3* and *T4*, read requests are processed in the same manner as the previous one. For example, at time *T3*, sVFS finds <100 K, 4 K> entry from the mapping table and issues a read request on the 4 K file position. Then, at time *T4*, sVFS finds <0, 8 K> entry and issues a read request on the 8 K file position. As a result, we support read consistency via the mapping table and its operations even if we transform the file positions sequentially.

In the scenario where a read request corresponds to multiple entries in the FP mapping table, sVFS splits the request into multiple sub-requests. For each sub-request, sVFS performs the read operation in the same manner as in a normal scenario. Then, sVFS merges the responses of the sub-requests into one and forwards it to the user.

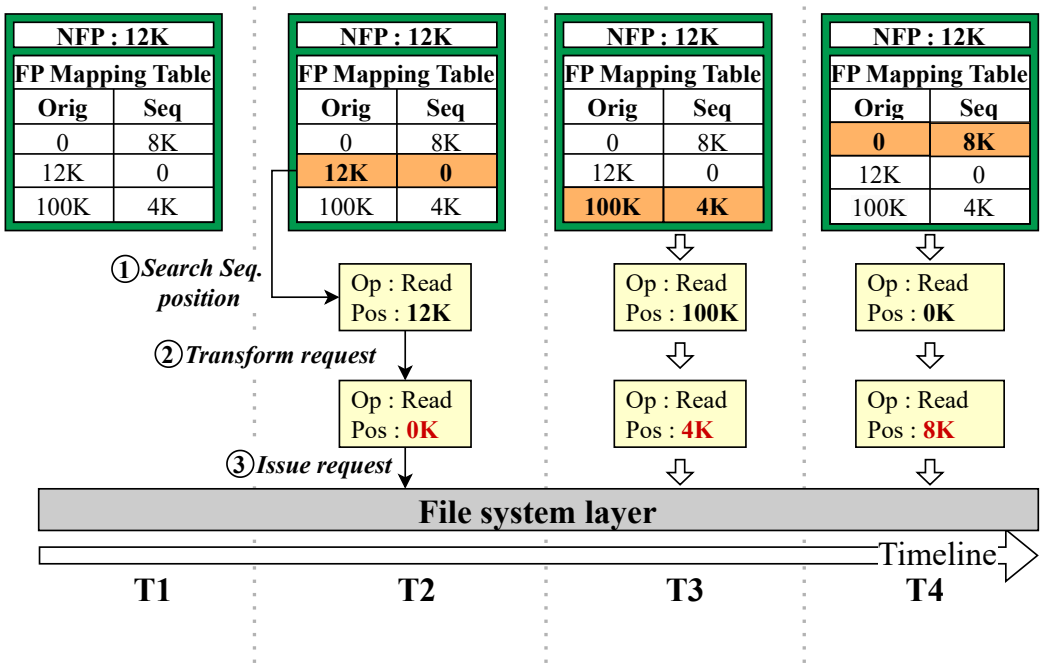


Figure 4. An example of file position sequentialization of sVFS in read operations. (NFP: next file position, FP: file position, Orig.: original position, Seq.: sequentialized position.)

4.2.3. Transaction Processing

As we described, in sVFS, the file positions requested by users and the transformed file positions are different due to their file position sequentialization. Thus, sVFS maintains a mapping table for retrieving actual file positions for supporting correct read operations. However, in the case of a crash or a system shutdown, after reboot, sVFS cannot find

the mapping information since it is stored in the main memory. Thus, sVFS supports the transaction processing of the file position mapping table to guarantee its crash consistency.

For transaction processing, we use a write-ahead logging (WAL) scheme to persist the file position mapping in the storage device. For example, sVFS sequentializes the position of a write request and sends the write request with the modified position to the file system. After the request is processed, sVFS begins a transaction with a beginning mark and writes the mapping table in a transaction log file. Then, sVFS commits the transaction by writing a commit mark in the transaction log file. A built-in cache in an SSD can impact the crash consistency of sVFS since the data in the cache can be lost after a system crash. To guarantee crash consistency whether SSD includes the built-in cache or not, sVFS utilizes the flush command and a force unit access (FUA) flag. Specifically, after writing a log file during transaction processing, sVFS issues a flush command to retain the log file (i.e., updates of the mapping table) in the flash media of the SSD. Then, it writes the transaction commit block with the FUA flag to bypass the built-in cache and directly retains the block in the flash media. After the transaction is successfully committed, sVFS can start a new transaction for the next I/O requests.

To recover the file position mapping table from a crash or a system shutdown, sVFS uses the transaction log file. sVFS reads the log file first and reorganizes the file position mapping table for each file in memory. In this process, sVFS ignores all the transaction logs that do not have the commit mark. sVFS inserts file position mapping of only successfully committed transactions into the file position mapping table. With the transaction and recovery processing, sVFS ensures the consistency of the file position mapping table.

4.2.4. Cleaning

sVFS continuously writes upcoming data sequentially (i.e., out of place) using the next file position even if the overwrite operations are performed. In this process, sVFS generates invalid data that are not used but occupy the space of the storage device. Thus, sVFS performs cleaning operations to remove the invalid data and reclaim free space in a block-aligned manner when there is insufficient free space for write operations. In detail, sVFS triggers the cleaning operation when the free space becomes less than 0.2% of total SSD space based on the garbage collection threshold of the representative LFS (i.e., F2FS). sVFS checks the position of invalid data in a file to perform a cleaning operation. sVFS maintains a bitmap that indicates the validity of data in each file position for a file to reduce the search overhead of invalid pages. After identifying the position of invalid data, sVFS moves valid data to the position. It finds target valid data to move from tail to head of the file and finds the target position which has invalid data from head to tail. Once the valid data is moved to the target position, sVFS updates the mapping information of the data in the file position mapping table. This process is repeated until every position of invalid data is filled with valid data. Then, sVFS reduces the next file position of the file and the size of the file by a trim command.

Figure 5 shows an example of the cleaning operation of sVFS. As shown in the figure, there are five data blocks of 4 KB size in a file. For cleaning operations, sVFS checks the validity of data using a validity bitmap and finds out the location of invalid data. In this case, there are two invalid data blocks in the 4 K and 8 K positions. First, sVFS moves the valid data block in the 16 K position, which is the last valid data block in the file, to the 4 K position, which is the position of the first invalid data block. After sVFS successfully moves the valid data block, it then updates the mapping of <4 K, 16 K> to <4 K, 4 K> in the file position mapping table to reflect the data movement. The validity bitmap is also updated (i.e., marking the 4 K position as valid and the 16 K position as invalid). Second, sVFS moves the valid data block in the 12 K position to the 8 K position, updates the corresponding entry of the mapping table, and updates the bitmap. When all of the data movement is finished, sVFS completes the cleaning operation with trimming. For example, sVFS trims the invalid data block after the last valid data block (i.e., trimming the

data in 12 K and 16 K positions) so that it can reduce the size of the file to 12 KB. Thus, this process reclaims the invalid data to make a free space for the upcoming write operations.

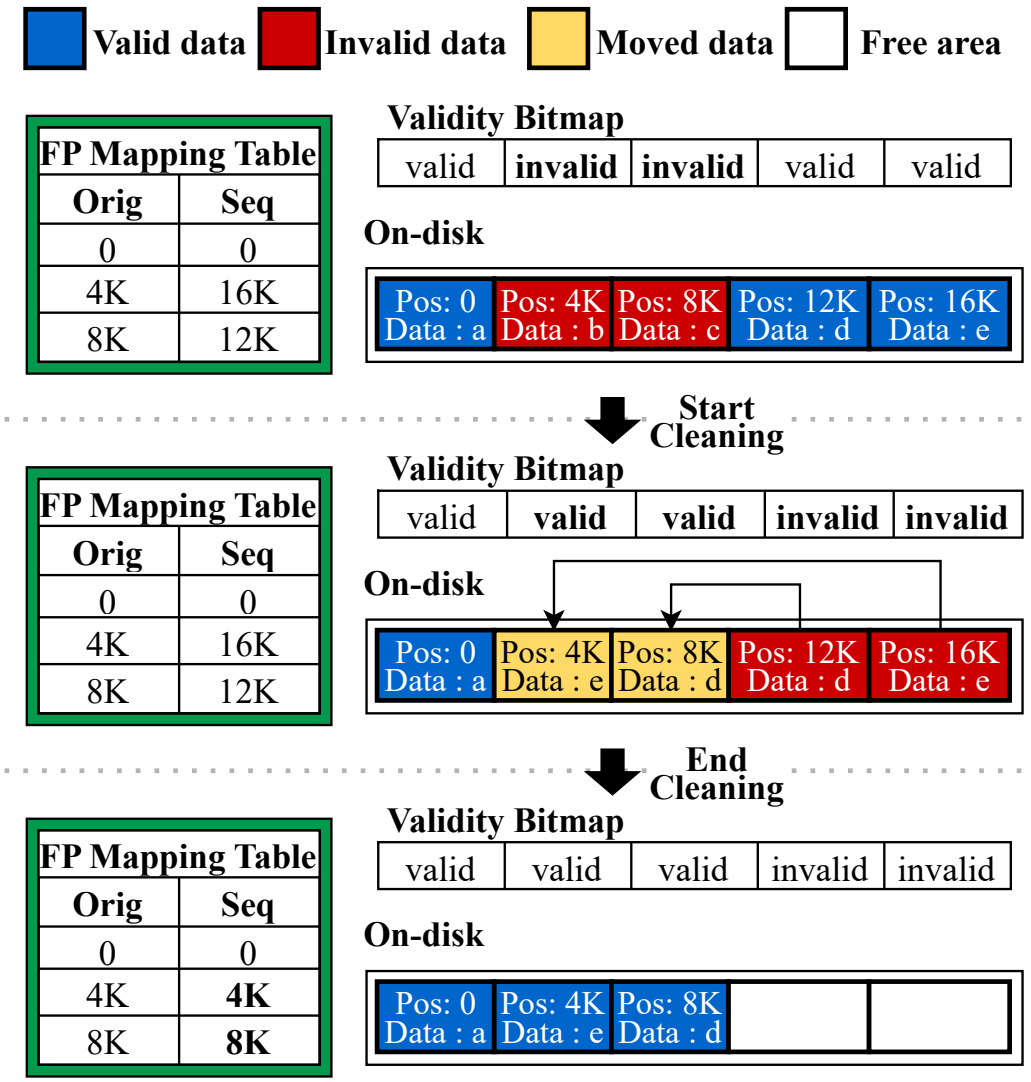


Figure 5. An example of the cleaning operation of sVFS. (FP: file position, Orig.: original position, Seq.: sequentialized position.)

4.3. Implementation

To minimize lock contention overhead, which is introduced by sharing the next file position of a file across multiple I/O threads, we implement sVFS using an atomic value and atomic operations (i.e., `__sync_fetch_and_add()`).

For the file position mapping table, we used red–black tree (RB tree) [25] which is a widely used data structure due to its fast insert and search. This is because it can perform insert and search operations fast, which has `O(log N)` time complexity, and also guarantees the same time complexity in the worst case. An entry of the mapping table is 8 B, which consists of an original file position (4 B) and a sequentialized file position (4 B). Since the mapping table is updated whenever write requests are issued, the overhead to update the mapping table can be 8 B per write request.

To validate our implementation, we check whether data are written in the intended file positions (i.e., sequentialized file position) and sVFS can read proper data. To do this, we insert test code in our implementation and dump file data and the mapping table.

5. Evaluation

5.1. Evaluation Setup

For the evaluation of sVFS, we use a system with an Intel Xeon 2-way E5-4650v3 CPU, which has 24 physical cores and 48 threads, 8 GB main memory, and a Samsung 860 Pro SSD 256 GB for the storage device. In terms of software, we implement our scheme (sVFS) on a Linux kernel v5.15.34 on Ubuntu 20.04.4 LTS. We compare the performance of the existing virtual file system on the EXT4, BTRFS, F2FS, and XFS file systems with that of sVFS on those file systems. We use FIO [26] as a microbenchmark and Flexible Filesystem Benchmark (FFSB) [27] as a macrobenchmark.

5.2. Microbenchmark

We use the FIO benchmark to measure how much sVFS can improve the performance of the existing software stack. We evaluate the random write/read and sequential write/read performance of the proposed scheme and existing software stack with the EXT4, BTRFS, F2FS, and XFS file systems with 48 threads. Each thread performs buffered writes or reads on a 1GB file with a 4 KB I/O request size.

5.2.1. Write Operation

Figure 6 shows the random write throughput of the existing VFS and sVFS on the existing file systems. As shown in the figure, sVFS achieves a greater improvement compared with the virtual file system. Except for BTRFS, sVFS improves the random write throughput by about 28%, 36%, and 34% on EXT4, F2FS, and XFS, respectively. This performance improvement in the random write workload results from the changes in the access pattern of I/O requests in the underlying layers. sVFS has a larger performance impact (3.35×) on the BTRFS than other file systems. Due to the huge performance gap between sequential and random write operations on BTRFS as can be seen in Figure 1, the effect of address sequentialization is magnified on BTRFS.

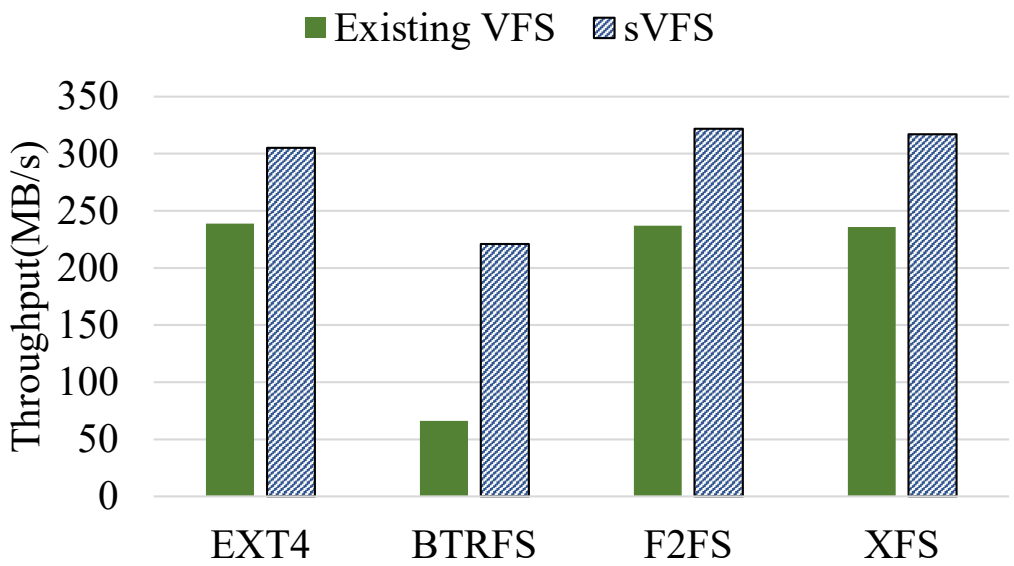


Figure 6. Random write throughput of existing VFS and sVFS with file systems in FIO benchmark.

We note that sVFS can improve the overall performance even if the underlying file system are copy-on-write (e.g., BTRFS) and log-structured (e.g., F2FS) file systems. Specifically, in F2FS, a mapping between a file position of a data block and its logical block address is maintained in a node block (4 KB) which can address up to 4 MB of a file. The file position numbers correspond sequentially to the entry numbers. For example, when data at a file position (10) is updated, the corresponding 10th mapping entry in the node block is updated. Thus, if an application writes 4 MB of a file with sequential file positions, F2FS

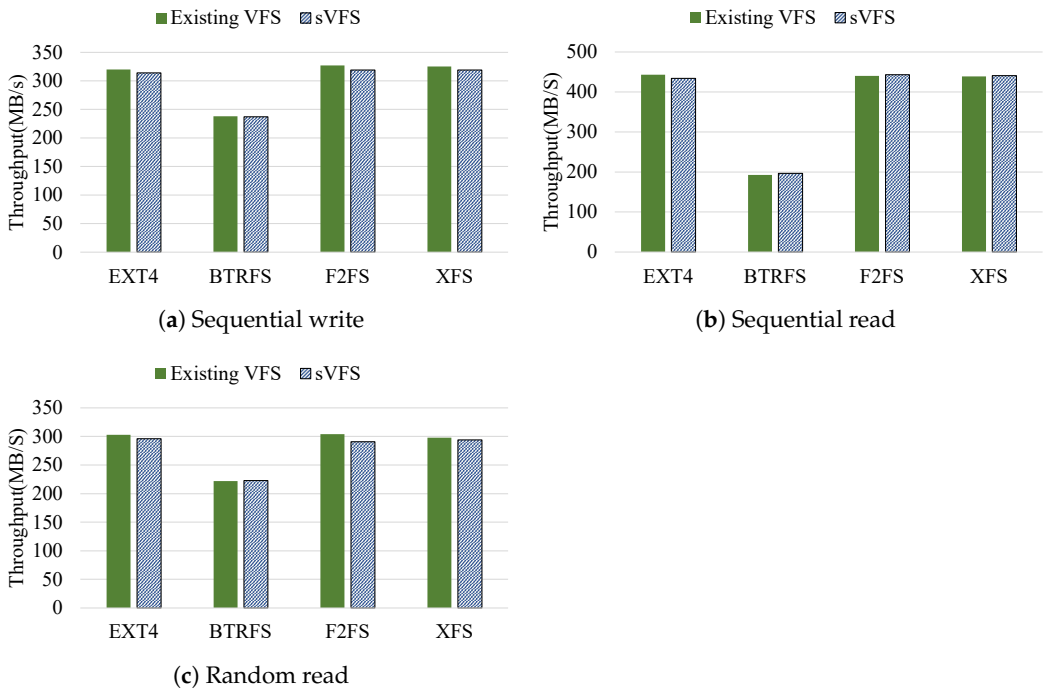
can store the mappings using a node block (4 KB). On the other hand, if an application writes 4 MB of a file with random file positions, in the worst case, F2FS may need up to 1000 node blocks to store the mappings since a node block may point out only a data block. As a result, by sequentializing file positions, sVFS can minimize the number of node blocks to be written to SSD, thereby improving the random write performance of F2FS.

In terms of latency, sVFS shows better performance than the existing scheme. Table 1 shows the average latency of random write on the existing VFS and sVFS with file systems using the FIO benchmark. The latency of random write with sVFS takes 2%, 7%, and 8% less time than the existing VFS on EXT4, F2FS, and XFS, respectively. Similar to the throughput, the latency improvement is noticeable on BTRFS, where the latency decreased by 71%. As a result, sVFS achieves the goal of optimizing write performance on SSDs in terms of latency as well as throughput.

Table 1. Average random write latency of existing VFS and sVFS with file systems in FIO benchmark.

	EXT4	**BTRFS**	**F2FS**	**XFS**
Existing VFS	807 us	2954 us	632 us	419 us
sVFS	613 us	807 us	589 us	385 us

While sVFS optimizes random write performance, it can induce an additional overhead in the sequential write operation. For example, there can be overheads of sequentializing file position and managing the position using the mapping table. Figure 7a shows the performance and the overhead in the sequential write operation. As shown in the figure, throughput degradation of sVFS in the sequential write is about 2% on EXT4, F2FS, and XFS, and under 1% on BTRFS. However, this result demonstrates that sVFS shows a little overhead on the performance.



(**a**) Sequential write

(**b**) Sequential read

(**c**) Random read

Figure 7. Sequential write, sequential read, and random read throughput of existing VFS and sVFS with file systems in FIO benchmark.

5.2.2. Read Operation

As we described, we show that sVFS improves the random write performance while showing a little degradation in the sequential write performance. As well as sequential write, sVFS introduces an additional overhead in the read operation. For example, sVFS

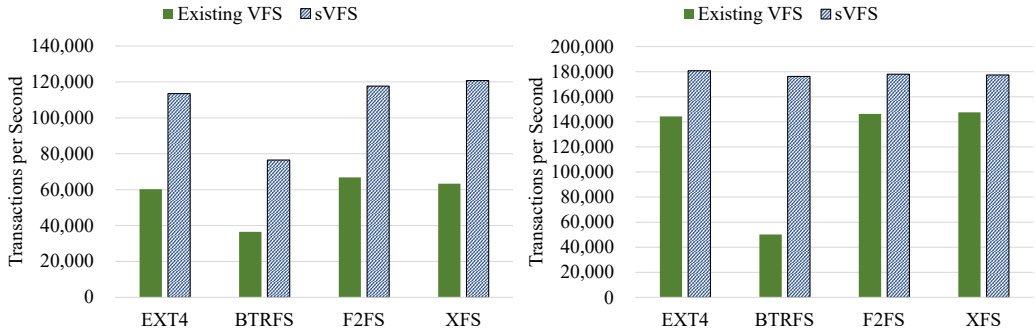
searches the file position mapping table to find the positions of users. Thus, we perform sequential and random read operations using the FIO benchmark to evaluate how much sVFS degrades the read performance.

Figure 7b,c show the throughput in the case of sequential and random read operations, respectively. As shown in the figure, the read performance of sVFS is almost the same or decreases slightly by up to 2% and 4% compared with the existing VFS in sequential and random reads, respectively. As a result, the overhead of managing the file position in the mapping table can be acceptable when read operations are performed.

5.3. Macrobenchmark

We perform the macrobenchmark to evaluate the performance of sVFS for more realistic I/O operations. To do this, we use the flexible file system benchmark [27] (FFSB) which simulates file operations including create, overwrite, append, read, and mixed I/O in file systems. We run two types of workloads for the macrobenchmark: write-intensive and mixed workloads. The write-intensive workload consists of random overwrite and append operations with a 1:1 ratio. The mixed workload consists of random overwrites and sequential reads with a 1:1 ratio. For FFSB configuration, we use a 4 KB block size, 48 I/O threads, and 64 files of 1 GB.

Figure 8a shows the performance of the existing VFS and sVFS on the write-intensive workload. sVFS improves the file system performance by up to 88%, 76%, 91%, and 2.1× in the case of EXT4, F2FS, XFS, and BTRFS, respectively, compared with the existing VFS. This result shows that sVFS can improve the performance of SSDs even in massive write workloads (including append and overwrite operations) for all the evaluated file systems. In our scheme, for the append operation, sVFS inserts the file position mapping information into the mapping table. Also, for overwrite operations, sVFS needs to search and modify the old mapping entry in the mapping table. Thus, even though there are management operations for the mapping table, the results demonstrate that sVFS improves the write performance without noticeable latency.



(**a**) Write-intensive workload (**b**) Mixed workload

Figure 8. Throughput of existing VFS and sVFS with file systems in FFSB.

We run a mixed workload in FFSB to evaluate the performance impact of sVFS when file read and write operations are executed simultaneously. Figure 8b shows the performance of the existing VFS and sVFS in the case of the mixed workload. As shown in the figure, sVFS improves the performance by up to 25%, 22%, 20%, and 3.51× compared with the existing VFS on EXT4, F2FS, XFS, and BTRFS, respectively. The performance gain of sVFS is mainly from the performance increase of write operations while maintaining the similar performance of read operations. This result indicates that sVFS can successfully improve the I/O performance while a huge number of file read and write operations are executed simultaneously.

6. Discussion

Skewed write: sVFS sequentializes the random write pattern to a sequential write pattern by transforming the file position in write requests. While this approach can benefit

workloads with randomly distributed data access, sVFS can have the disadvantage of unnecessary disk write in a workload with a skewed data access pattern. Specifically, multiple write requests with skewed access patterns can be merged and flushed to the storage devices, since the requests have equal file positions and are managed by equal page cache entry. However, sVFS allocates a new file position for each write request and stores the data in different page cache entries, even if the requests have equal file positions originally. Thus, sVFS can perform an additional disk flush and introduce overhead from the flush.

The overhead can be mitigated by removing invalid page cache entries. In detail, when an update request is issued, sVFS can compare the original file position of the request with the original file positions of other dirty page cache entries. If there is a page cache entry which has the same original file position, sVFS can remove the page cache entry since it is invalidated by the request. This can make sVFS avoid flushing unnecessary data. We plan to address this part in future work.

Increase of random read: sVFS may increase the number of random accesses, particularly in scenarios involving sequential reads after random writes. If the sequential and random read performances of the SSD are not significantly different, the increased random reads do not impact the performance, as shown in Figure 8b. However, on low-end SSDs with little internal cache, where there can be a difference between the sequential and random read performance, the sequentialization of sVFS may reduce the read performance.

We plan to address and reduce the impact on read performance for low-end SSDs in future work. For example, to mitigate the performance overhead from increased random access, sVFS can exploit a hint from users. Since the increase in random access mainly occurs in specific I/O patterns, particularly in scenarios involving sequential reads after random writes, users can notify sVFS that a file will be accessed in a specific pattern. Then, sVFS can selectively transform the write access patterns of a file to mitigate the overhead of random access patterns.

Cleaning optimization: sVFS aims to improve the random write performance by sequentializing write patterns in clean-state scenarios instead of improving the GC procedure. There are existing techniques to reduce the cleaning overhead such as data separation [7,20,22] and copy offloading [28–30], and we can adopt these techniques to optimize the cleaning operation.

For instance, sVFS can mark an actively updated file as hot. Since data in a hot file are likely invalidated, delaying cleaning the hot file can decrease the overhead of moving data in cleaning operations. Thus, sVFS can prioritize cleaning cold files and delay hot files to mitigate the cleaning overhead. Meanwhile, as an SSD supporting copy command arises [28,31], sVFS can utilize the command to offload the cleaning operation. sVFS can request copy commands to the SSD to offload data movement during cleaning operations. This can reduce CPU and memory usage, and save SSD interface bandwidth. We plan to implement and evaluate these techniques in sVFS to reduce the cleaning overhead in future work. Furthermore, we also plan to address write amplification and disk wear which can be caused by the cleaning operations of sVFS.

7. Conclusions

Depending on the I/O access pattern, the performance of SSDs can be degraded. To be specific, a random write pattern can reduce the spatial locality, thus introducing a higher GC overhead and increasing the write amplification compared with a sequential write pattern. In this paper, we proposed sVFS which is a file position sequentialization scheme in the virtual file system. It transforms random access patterns into sequential write patterns so that it increases the spatial locality and reduces the GC overhead. We design and implement sVFS on the kernel VFS layer without depending on the types of file systems, block layer configurations, device drivers, and devices. The experiment results show that the proposed scheme improves the random write performance up to 2.35× in the

microbenchmark and 3.51× in the macrobenchmark, while maintaining almost the same performance of sequential write and read operations.

Author Contributions: Conceptualization, I.H. and Y.S.; methodology, I.H. and Y.S.; software, I.H.; investigation, I.H.; resources, S.K., H.E. and Y.S.; writing—original draft preparation, I.H.; writing—review and editing, S.K., H.E. and Y.S.; visualization, I.H.; supervision, Y.S.; project administration, I.H.; funding acquisition, Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Research Foundation of Korea (NRF) (No. NRF-2022R1A4A5034130), Korea Institute for Advancement of Technology (KIAT) (No. KIAT-P0012724) grant funded by the Korea Government. This work was also supported by BK21 FOUR Intelligence Computing funded by NRF (No. NRF-4199990214639) (Corresponding Author: Yongseok Son).

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Stoica, I.; Song, D.; Popa, R.A.; Patterson, D.; Mahoney, M.W.; Katz, R.; Joseph, A.D.; Jordan, M.; Hellerstein, J.M.; Gonzalez, J.E.; et al. A berkeley view of systems challenges for AI. *arXiv* **2017**, arXiv:1712.05855.
2. Roh, Y.; Heo, G.; Whang, S.E. A Survey on Data Collection for Machine Learning: A Big Data—AI Integration Perspective. *IEEE Trans. Knowl. Data Eng.* **2021**, *33*, 1328–1347. [CrossRef]
3. Wong, G. SSD market overview. In *Inside Solid State Drives (SSDs)*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 1–17.
4. Joshi, S. Solid State Drives (SSD) Market Size, $143,557 Million by 2029 Led by SLC Technology, 15% CAGR—Exclusive Research Report by The Insight Partners. Available online: https://www.globenewswire.com/en/news-release/2022/01/21/2370787/0/en/Solid-State-Drives-SSD-Market-Size-143-557-Million-by-2029-Led-by-SLC-Technology-15-CAGR-Exclusive-Research-Report-by-The-Insight-Partners.html (accessed on 12 October 2024).
5. Hard Disc Drive Market Overview. Available online: https://www.futuremarketinsights.com/reports/hard-disk-drive-market (accessed on 12 October 2024).
6. Min, C.; Kim, K.; Cho, H.; Lee, S.W.; Eom, Y.I. SFS: Random write considered harmful in solid state drives. In Proceedings of the FAST, San Jose, CA, USA, 14–17 February 2012; Volume 12, pp. 1–16.
7. Lee, C.; Sim, D.; Hwang, J.; Cho, S. F2FS: A New File System for Flash Storage. In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15), Santa Clara, CA, USA, 16–19 February 2015; pp. 273–286.
8. Kim, S.; Han, J.; Eom, H.; Son, Y. Improving I/O performance in distributed file systems for flash-based SSDs by access pattern reshaping. *Future Gener. Comput. Syst.* **2021**, *115*, 365–373. [CrossRef]
9. Bouganim, L.; Jónsson, B.Þ.; Bonnet, P. uFLIP: Understanding Flash IO Patterns. *arXiv* **2009**, arXi:0909.1780. http://arxiv.org/abs/0909.1780.
10. Samsung SSD 860 PRO. Available online: https://semiconductor.samsung.com/consumer-storage/internal-ssd/860pro/ (accessed on 12 October 2024).
11. Kim, H.; Shin, D.; Jeong, Y.H.; Kim, K.H. SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17), Santa Clara, CA, USA, 27 February–2 March 2017; pp. 271–284.
12. Lee, Y.; Kim, J.S.; Maeng, S. ReSSD: A software layer for improving the small random write performance of SSDs. *J. Inf. Sci. Eng.* **2012**, *28*, 999–1018.
13. Xie, T.; Koshia, J. Boosting random write performance for enterprise flash storage systems. In Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), Denver, CO, USA, 23–27 May 2011; pp. 1–10.
14. Chen, F.; Koufaty, D.A.; Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Perform. Eval. Rev.* **2009**, *37*, 181–192. [CrossRef]
15. RocksDB. Available online: http://rocksdb.org/ (accessed on 12 October 2024).
16. LevelDB. Available online: https://github.com/google/leveldb (accessed on 12 October 2024).
17. Zertal, S. Exploiting the Fine Grain SSD Internal Parallelism for OLTP and Scientific Workloads. In Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), Paris, France, 20–22 August 2014; pp. 990–997. [CrossRef]
18. Park, S.Y.; Seo, E.; Shin, J.Y.; Maeng, S.; Lee, J. Exploiting Internal Parallelism of Flash-based SSDs. *IEEE Comput. Archit. Lett.* **2010**, *9*, 9–12. [CrossRef]

19. Wang, H.; Huang, P.; He, S.; Zhou, K.; Li, C.; He, X. A novel I/O scheduler for SSD with improved performance and lifetime. In Proceedings of the 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST), Long Beach, CA, USA, 6–10 May 2013; pp. 1–5. [CrossRef]
20. Kim, J.; Lim, K.; Jung, Y.; Lee, S.; Min, C.; Noh, S.H. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA, USA, 10–12 July 2019; pp. 799–812.
21. Kang, W.; Shin, D.; Yoo, S. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*. [CrossRef]
22. Yang, P.; Xue, N.; Zhang, Y.; Zhou, Y.; Sun, L.; Chen, W.; Chen, Z.; Xia, W.; Li, J.; Kwon, K. Reducing Garbage Collection Overhead in SSD Based on Workload Prediction. In Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19), Renton, WA, USA, 8–9 July 2019.
23. O'Neil Patrick, Cheng, E.; Gawlick, D.; O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Inform.* **1996**, *33*, 351–385. [CrossRef]
24. Rosenblum, M.; Ousterhout, J.K. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.* **1992**, *10*, 26–52. [CrossRef]
25. Wong, C.; Tan, I.; Kumari, R.; Lam, J.; Fun, W. Fairness and interactive performance of O (1) and CFS Linux kernel schedulers. In Proceedings of the 2008 International Symposium on Information Technology, Kuala Lumpur, Malaysia, 26–28 August 2008; Volume 4, pp. 1–8.
26. FIO. Available online: https://github.com/axboe/fio (accessed on 12 October 2024).
27. Flexible FileSystem Benchmark. Available online: https://github.com/FFSB-Prime/ffsb (accessed on 12 October 2024).
28. NVM Command Set Specificiation. Available online: https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0e-2024.07.29-Ratified.pdf (accessed on 12 October 2024).
29. Han, K.; Gwak, H.; Shin, D.; Hwang, J. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), Virtual, 14–16 July 2021; pp. 147–162.
30. Hwang, J.Y.; Kim, S.; Park, D.; Song, Y.G.; Han, J.; Choi, S.; Cho, S.; Won, Y. ZMS: Zone Abstraction for Mobile Flash Storage. In Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24), Santa Clara, CA, USA, 10–12 July 2024; pp. 173–189.
31. Joshi, K.; Gupta, A.; González, J.; Kumar, A.; Reddy, K.K.; George, A.; Lund, S.; Axboe, J. {I/O} Passthru: Upstreaming a flexible and efficient {I/O} Path in Linux. In Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST 24), Santa Clara, CA, USA, 27–29 February 2024; pp. 107–121.