SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizng in Device

Hyukjoong Kim[1], Dongkun Shin[1], Yun Ho Jeong[2], and Kyung Ho Kim[2]

[1]Sungkyunkwan University, Korea
[2]Samsung Electronics, Korea

Abstract

Recent advances in flash memory technology have reduced the cost-per-bit of flash storage devices such as solid-state drives (SSDs), thereby enabling the development of large-capacity SSDs for enterprise-scale storage. However, two major concerns arise in designing SSDs. The first concern is the poor performance of random writes in an SSD. Server workloads such as databases generate many random writes; therefore, this problem must be resolved to enable the usage of SSDs in enterprise systems. The second concern is that the size of the internal DRAM of an SSD is proportional to the capacity of the SSD. The peculiarities of flash memory require an address translation layer called flash translation layer (FTL) to be implemented within an SSD. The FTL must maintain the address mapping table in the internal DRAM. Although the previously proposed demand map loading technique can reduce the required DRAM size, the technique aggravates the poor random performance. We propose a novel address reshaping technique called sequentializing in host and randomizing in device (SHRD), which transforms random write requests into sequential write requests in the block device driver by assigning the address space of the reserved log area in the SSD. Unlike previous approaches, SHRD can restore the sequentially written data to the original location without requiring explicit copy operations by utilizing the address mapping scheme of the FTL. We implement SHRD in a real SSD device and demonstrate the improved performance resulting from SHRD for various workloads.

1 Introduction

In recent times, the proliferation of flash-memory-based storage such as solid-state drives (SSDs) and embedded multimedia cards (eMMCs) has been one of the most significant changes in computing systems. The cost-per-bit of flash memory has continued to fall owing to semiconductor technology scaling and 3-D vertical NAND flash technology [32]. As a result, SSD vendors currently provide up to 16 TB of capacity.

Flash memory has several characteristics that must be carefully handled. In particular, its “erase-before-write” constraint does not permit in-place update. In order to handle the peculiarities of flash memory, special software–called a flash translation layer (FTL) [14, 20]–is embedded within flash storage systems. When a page of data must be updated, the FTL writes the new data to a clean physical page and invalidates the old page because in-place overwrite is prohibited in flash memory. Therefore, the logical address and physical address of a flash page will be different. The FTL maintains a mapping table for the logical-to-physical (L2P) address translation. When the SSD has an insufficient number of clean pages, garbage collection (GC) is triggered to reclaim invalid pages. GC selects victim blocks to be recycled, copies all the valid pages in the recycling blocks to another block, and erases the recycling blocks.

Recent flash storage devices adopt page-level address mappings instead of block-level schemes in order to provide higher performance. Page-level mappings permit requests to be serviced from any physical page on flash memory, whereas block-level mappings restrict the physical page location of a request based on its logical address. However, the finer-grained mapping scheme requires a large L2P mapping table. Typically, the size of a page-level mapping table is 0.1% of the total storage capacity because the length of the address translation data for a 4 KB page is 4 bytes.

The mapping table is accessed by every I/O request; therefore, in order to achieve high performance, the entire table must be loaded into an internal DRAM of the SSD. Thus, as an example, 8 TB of SSD requires 8 GB of DRAM for the mapping table. More than 4 GB of DRAM requires a 64-bit processor, which is hardly acceptable to embedded systems. In addition, a larger capacity of DRAM system composed of multiple DRAM modules requires DRAM controller to handle more address lines, which will increase the controller cost and DRAM access latency. Furthermore, a large DRAM-equipped SSD will have a high power consumption and product cost. Therefore, the large mapping table is the most critical hurdle in increasing the capacity of SSDs.

In order to resolve this problem, on-demand map loading schemes such as DFTL [14] were proposed. Instead of maintaining all the address translation entries in DRAM, DFTL dynamically loads/unloads the page-level mapping entries to/from a small DRAM according to the workload access patterns. The entire image of the page-level mapping table is stored in reserved flash memory blocks called *map blocks*. If storage workloads exhibit significant temporal locality, the performance of the demand-loading approach will be similar to the performance of the traditional page-level mapping scheme that loads all the mapping entries in DRAM. In addition, DFTL can utilize spatial locality by loading multiple logically contiguous mapping entries at the miss of a mapping entry. However, real scenarios have many random workloads with low localities. DFTL is vulnerable to these workloads; this limitation is a critical drawback of DFTL in real computing systems.

In this paper, we focus on random write rather than read because writes occupy about 70% at server storage workloads as observed in [28]. In addition, whereas read requests are controllable by several optimization techniques such as page caching or prefetching, the immediate handling of write requests are unavoidable in many cases because they are generated for data durability.

One possible solution for random write workloads is to use log-structured file systems (LFSs) (e.g., NILFS [21] and F2FS [22]) or copy-on-write file systems (e.g., btrfs [33]) because they generate sequential write requests using an out-of-place update scheme. The key-value stores based on LSM-Trees (e.g., LevelDB [7] and RocksDB [9]) and log-structured databases (e.g., RethinkDB [8]) also remove random writes. However, such log-structured file systems or applications suffer from garbage collection or compaction overhead and require many metadata block updates owing to their out-of-place update scheme. Moreover, file system and SSD perform duplicated garbage collections [35].

Another solution is to translate random write requests into sequential write requests in the block device driver. ReSSD [26] and LSDM [37] sequentialize random write requests by providing an additional address translation layer in the storage device driver and temporarily writing the sequentialized data to a reserved area in the storage device. We call this operation *sequentializing*. However, when the reserved area becomes full, the temporarily written data must be moved to the original location; this operation is called *randomizing (restoring)*. The randomizing operation results in additional storage traffic. Further, it eventually sends random writes to storage.

In this paper, we propose a novel scheme, called sequentializing in host and randomizing in device (SHRD), to reshape the storage access pattern. In order to reduce the map-handling overhead in a DFTL-based SSD, the proposed scheme sequentializes random write requests into sequential requests at the block device driver by redirecting random write requests to the reserved storage area. Therefore, it is analogous to the previous device-driver-level sequentializing approach. However, in the previous schemes, randomizing is achieved by performing explicit move operations in the host system; thus, they use the “sequentializing in host and randomizing in host (SHRH)” approach. Our scheme conducts the randomizing in storage device by changing only the logical addresses of the sequentialized data.

SHRD has several advantages. It can improve random write performance by reducing map-loading or command-handling overheads and by increasing the utilization of parallel units in an SSD. It can also improve the lifetime of an SSD. The reduction in the number of map update operations results in a reduction in the number of program and erase operations on flash memory blocks, thus minimizing the write amplification ratio.

This study makes the following specific contributions. (1) We propose and design a novel request reshaping scheme called SHRD, which includes a storage device driver and the FTL of an SSD. The idea of SHRD is to improve the spatial locality for FTL mapping table accesses by logging random requests in the storage and reordering these requests. (2) We implement the proposed SHRD scheme by modifying the firmware of an SSD device and the Linux SCSI device driver. Unlike many other studies based on SSD simulators and I/O traces, our scheme is demonstrated and verified by using a real SSD device. (3) We use several enterprise-scale workloads to demonstrate the improved performance achieved by SHRD. We observe that in comparison with DFTL, the performance of SHRD is 18 times better for a random-write dominant I/O workload and 3.5 times better for a TPC-C workload.

The remainder of this paper is organized as follows: In Section 2, the FTL schemes are introduced; in Section 3, the motivation and main idea are presented; in Section 4, the proposed SHRD scheme is described in detail; the experimental results are presented in Section 5; previous studies on improving random write performance are presented in Section 6; and the conclusion of this study is described in Section 7.

2 Backgrounds

Generally, the page-level mapping FTL maintains the entire L2P mapping table in the internal DRAM of an

SSD. In order to service incoming write requests, FTL allocates active flash blocks, writes incoming data sequentially in the physical pages of the active blocks, and updates the L2P mapping table in DRAM. If the active blocks are full with user data, the FTL flushes dirty mapping entries into the reserved area of flash memory called map blocks and then allocates new active blocks. If a sudden power-off occurs before the map flush operation, the power-off recovery (POR) operation of an SSD scans only the active blocks and rebuilds the L2P mapping table with the logical page numbers (LPNs) stored in the out-of-bound (OOB) area of flash pages; this OOB area is a hidden space reserved for FTL-managed metadata.

In order to reduce the mapping table size, several solutions have been proposed such as hybrid mapping schemes [20, 25] and extent mapping schemes [27, 16, 31]. Although these solutions can reduce the mapping table size significantly, they are vulnerable to random write workloads. Hybrid mapping scheme shows considerable performance degradation when the log blocks and normal data blocks are merged. Extent mapping scheme must split a large extent into multiple smaller extents when the random updates are requested.

Rather than reducing the size of the mapping table, we can use an on-demand map loading scheme such as DFTL [14]. This scheme uses a small amount of DRAM for the cached mapping table (CMT), which has only a subset of the entire page-level mapping table that is maintained in the map blocks of flash chips, as shown in Figure 1. For each I/O request, DFTL determines the physical page number (PPN) to be accessed based on the mapping entries in the CMT. The map blocks must be read and written as page units; therefore, multiple contiguous mapping entries constitute a *map page*, and DFTL loads/unloads mapping entries in map page units. For example, if each mapping entry has a size of 4 bytes, 4 KB of map page can contain 1024 logically contiguous mapping entries. Owing to the page-level loading scheme, DFTL requires spatial locality as well as temporal locality in the storage access workload. In order to handle a map miss, one victim map page must be written in the map block if the map page is dirty and one demanded map page must be read from the map block. Therefore, the demand map loading scheme demonstrates poor performance for a random workload due to additional storage traffic.

The map-miss handling in DFTL decreases the utilization of the parallel units of SSD. In order to provide high I/O bandwidth, multiple flash chips in SSD should be accessed simultaneously via parallel channels and chip interleaving. However, if several requests generate map misses simultaneously and the missed map entries must be read from a same chip, the handling of the requests must be serialized and thus several flash chips will be idle during the map-miss handling.

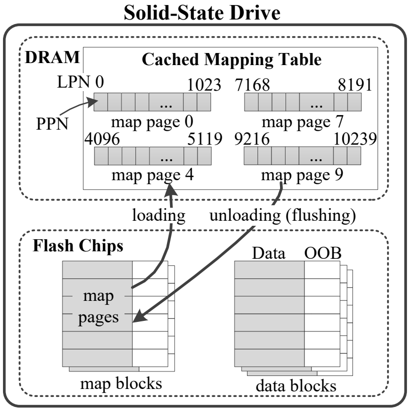


Figure 1: Demand map loading.

3 Main Idea

The poor random write performance of SSD can be attributed to various reasons. The first reason is the mapping-table handling overhead. If the SSD uses a demand map loading scheme such as DFTL, random requests will result in frequent CMT misses, as explained in Section 2. Even if the SSD can load all the mapping entries into the DRAM, the random writes will generate many dirty map pages in the DRAM. When the SSD periodically flushes dirty map pages, many pages will be updated in the map blocks.

The second reason is the request-handling overhead. The occurrence of many small requests increases the request traffic between the host computer and the SSD and increases the interrupt-handling overhead of the host computer. In order to solve this problem, eMMC adopts the packed command from the version 4.5 standard; thus, multiple requests can be merged into one packed command [1]. However, the current SATA/SAS protocol does not support the request merging.

The final reason is the cost of GC. GC selects a victim flash block having a small number of valid pages in order to reduce page-copy operations. While a sequential write workload generates many completely invalid blocks, a random write workload distributes invalid pages among several flash blocks, thus making it difficult to find a low-cost GC victim block. The overhead of GC can be mitigated by using hot and cold separation algorithms [10, 24].

From among the several reasons for poor random performance of flash storage, we focus on the mapping-table handling overhead and the request-handling overhead because they are the major causes and do not have any solutions currently. SHRD can reduce the mapping-table handling overhead by improving the spatial locality of a workload, and can reduce the request-handling overhead

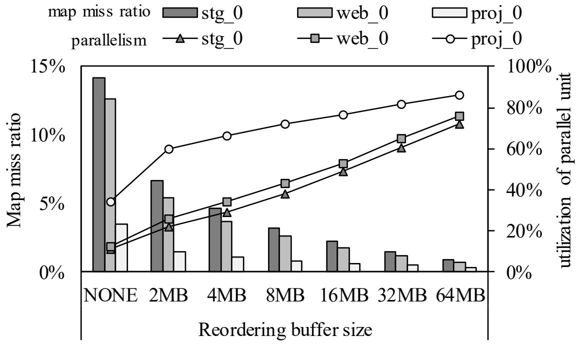


Figure 2: Effect of request reordering.

by packing multiple requests into one large request.

Even if a workload has low spatial locality within a short time interval, it can have high spatial locality within a long time interval. If a memory space is available for request buffering, the map miss ratio can be reduced by reordering requests in the order of LPN to improve spatial locality. Figure 2 shows the simulation results for the map miss ratio of DFTL and the utilization of parallel flash chips in an SSD. The MSR-Cambridge server workloads [4] are used for input traces. We assume that the CMT size of DFTL is 128 KB and a maximum of 64 flash pages can be accessed in parallel via multi-channel, multi-bank, and multi-plane mechanisms. The x-axis shows the reordering buffer size. In each experiment, all the I/O requests are partitioned into several groups in the order of request arrival time such that the total request size of each group is equal to the reordering buffer size. Then, the requests within a group are sorted in the order of LPN. Such transformed input traces are provided to an SSD simulator, which uses the DFTL algorithm. In comparison with the original trace, the map miss ratios significantly decrease as the size of the reordering buffer increases because the spatial locality improves. The utilization of the parallel units of the SSD also improves.

In order to support such a request reordering, we must allocate the reordering buffer in either host system or SSD. The host-level buffering has several problems. First, a large memory space is required for data buffering. Second, applications may invoke synchronous operations such as *fsync*() in order to ensure instant data durability. Most database systems rely on the fsync system call to be assured of immediate data durability. Therefore, host-level data buffering is impractical. The large memory allocation within SSD also does not correspond with our motivation.

Our idea is to buffer only mapping entries instead of request data. The mapping entries are buffered in a small size of host memory, and the data are directly written at SSD without buffering. To obtain the same effect of request reordering, SHRD writes random write requests in a reserved space in the SSD called random write log buffer (RWLB). This step is *sequentializing*. The RWLB is a logical address space; therefore, an SSD can allocate any physical flash blocks for the RWLB. SHRD assigns a temporary logical page number (tLPN) sequentially for each original logical page number (oLPN). tLPNs are allocated from the RWLB address space. The write operations to the RWLB can be performed with large and sequential write requests that invoke little mapping-table handling overhead because the original addresses are sequentially mapped to the addresses of the RWLB. The storage device driver in host computer maintains the mapping information between oLPN and tLPN in order to redirect read requests, and thus SHRD does not require any change on host file systems.

When the logical address space of the RWLB is exhausted, the buffered mapping entries in host system are sent to SSD after being reordered based on their oLPNs. SSD restores the sequentialized data into the original addresses with the mapping entries. This step is *randomizing (restoring)*, which modifies only the L2P mapping table of the SSD instead of moving the sequentialized data. Although the randomizing operation updates many L2P mapping entries, the map-loading overhead is minimized because the randomizing operations of sequentialized pages are performed in the order of oLPN, thus improving the spatial locality during map update operations.

4 SHRD Scheme

4.1 Overall Architecture

The SHRD architecture consists of an SHRD device driver (D/D) in the host system and SHRD-supporting firmware in the SSD, as shown in Figure 3. The file system sends a write request, which consists of the target logical page number (oLPN), the size, and the memory pointer to user data. The SHRD D/D receives the write request and checks whether sequentializing is required based on the write request size. The sequentializing and randomizing involve special operations; therefore, they result in some overhead. Considering the trade-off between performance gain and the overhead caused by SHRD, only small and random requests must be sequentialized. If the request size does not exceed a predefined threshold value called *RW threshold*, the `sequentializer` assigns sequential temporary addresses (tLPNs) to the request.

The sequentialized write requests are sent to the SHRD-supporting SSD via a special command, called *twrite*, which sends both the oLPN and the assigned tLPN. The SSD writes the sequentialized data into the blocks assigned to the RWLB. The

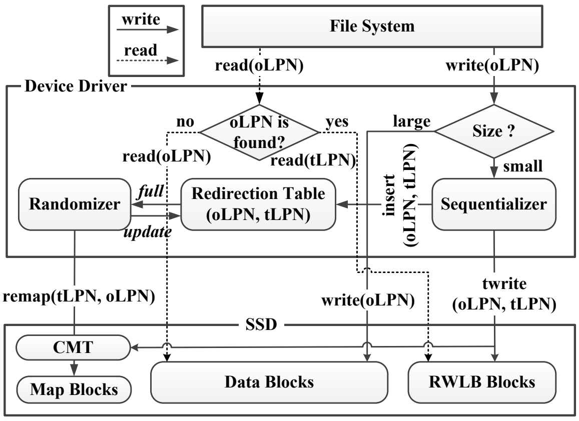


Figure 3: SHRD architecture.

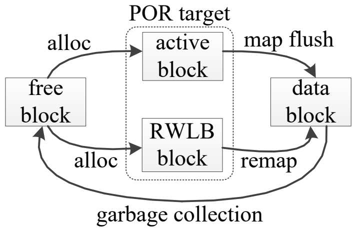


Figure 4: Life cycle of flash memory block.

`sequentializer` also inserts the mapping information into the `redirection table`; read requests use this table to redirect the original address to the sequentialized temporary address.

If the logical address space of the RWLB is exhausted, the `randomizer` restores the original logical addresses of the sequentialized data via the *remap* command. When the SSD receives the *remap* command, it changes only the internal L2P mapping table without changing the physical location of the data.

The SHRD-supporting FTL is similar to DFTL, but it can handle SHRD-supporting special commands such as *twrite* and *remap*. It also manages several different types of flash memory blocks such as data block, active block, RWLB block, and map block. Figure 4 shows the life cycle of a flash memory block. The FTL allocates active blocks and RWLB blocks for the normal data region and RWLB, respectively. The mapping entries of the pages in these regions are updated only in the CMT; therefore, the POR operation must scan the OOB area of these blocks to rebuild the mapping information. When the mapping entries of an active block are flushed into the map blocks, the active block is changed to a data block and it can be a victim of GC. If all the pages in an RWLB block are remapped to its original logical address, the block is changed to a data block. An RWLB block cannot be a victim for garbage collection because the mapping entries of its pages are not fixed yet.

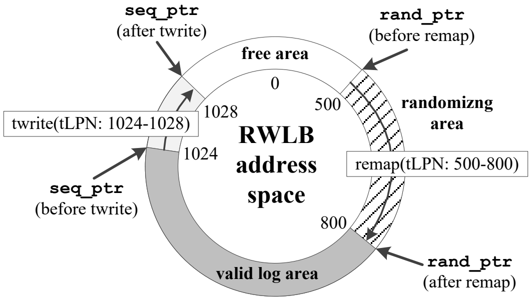


Figure 5: RWLB address management.

4.2 Sequentializing in Host

In order to manage the address space of the RWLB, which is hidden from the file system, SHRD maintains two address pointers to the RWLB address space, as shown in Figure 5. The RWLB is managed in a circular manner. `seq_ptr` is the start location for sequential address allocation, and it is incremented by sequentializing. `rand_ptr` is the start location of sequentialized pages that must be randomized, and it is incremented by randomizing. Therefore, the redirection table in the host has the mapping information of the pages that are written to the address range from `rand_ptr` to `seq_ptr`.

The sequentialized write requests are not immediately sent to the SSD. In order to minimize the command-handling overhead, multiple sequentialized write requests are packed into one write request, as shown in the example in Figure 6. The request packing can also reduce the interrupt handling overhead. The sequentialized write requests have contiguous address values; therefore, the packed write request has only one start address value. The block layer of the Linux operating system supports a maximum of 512 KB of write requests; therefore, the requests can be packed until the total packed request size does not exceed 512 KB. If no request is present in the I/O scheduler queue, the sequentializer sends the packed requests to the storage device immediately instead of waiting for more requests even though the size of packed requests is less than 512 KB. The request packing also halts if a higher priority request such as *flush* arrives. During the random write packing operation, if an examined request is not sequentialized (e.g., a read or large write request), SHRD sends first the normal request to storage and then the packing operation for random write requests continues. As more requests are packed into a single request, the command-handling overhead will be reduced. However, the request packing does not affect the map-handling overhead in SSD because subsequent *twrite* requests will be assigned with sequential tLPNs.

SHRD-supporting SSD must handle the sequentialized and packed requests differently from normal write requests; therefore, we need to implement *twrite* as a special command. The SATA storage interface provides

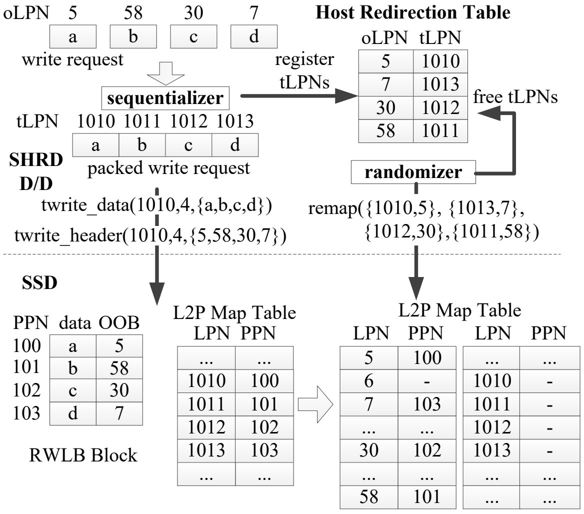


Figure 6: Example of SHRD operations.

vendor commands to support the extension of its command set. However, a vendor command cannot be used in the command queueing mode, and thus, the performance can degrade significantly. In order to overcome such a problem, we implemented several special commands of SHRD operations by utilizing the legacy write command. The SHRD-supporting special commands are exactly same as the normal write command except that their target address values are beyond the logical address space of storage device. Depending on the address value of write command, SSD firmware interprets it as a particular special command. The information of the special command (e.g., oLPN and tLPN) is transferred via data blocks of the write command. Therefore, no hardware or software changes are required in the SATA interface protocol.

Two SHRD commands (i.e., *twrite_header* and *twrite_data*) are used to write the sequentialized data to the RWLB of the SSD. *twrite_header* contains the address mapping information in its 4 KB of data, i.e., start tLPN, page count, and array of oLPNs, as shown in Figure 6. The sequentializing information of a maximum of 128 pages can be transferred via 4 KB of data.

After a *twrite_header* command is sent, the corresponding *twrite_data* command is sent in order to transfer the packed user data (maximum of 512 KB), which will be written to the RWLB blocks. The *twrite_data* command contains the target logical address value of tLPN, which is assigned by the sequentializer. The SSD firmware determines a PPN for each tLPN and inserts the mapping entry (tLPN, PPN) into the L2P mapping table. Each oLPN value transferred by the *twrite_header* command is written into the OOB area of the corresponding flash page. The oLPN will be used for POR when the page is in the RWLB block, and it will be used by GC after its block is changed to a data block.

After the completion of the *twrite* command, the sequentializer inserts the address translation entry between oLPN and tLPN into the redirection table. Because the remap entry will be used by subsequent read requests and remap operations, the redirection table maintains both the oLPN-to-tLPN and tLPN-to-oLPN map entries. If an old mapping entry for the same oLPN exists, the redirection table is updated and a *trim* command for the previous tLPN can be optionally sent to the SSD in order to inform it about the invalidation of the data at the address of the tLPN. The size of the redirection table is determined by the size of the RWLB. For example, in our implementation, when the RWLB size is 64 MB, the size of the redirection table is 256 KB and the table can contain 16K mapping entries.

During the handling of *twrite* commands, normal read or write commands can be transferred to SSD if their target addresses are not related with a pending *twrite* command. However, any dependent read request must wait for the completion of the related *twrite* command. In addition, any dependent write request can be processed only after the redirection map table is updated by *twrite*.

4.3 Read Redirection

For a read request from the file system, the SHRD D/D searches for the target logical page numbers, oLPNs, in the redirection table. If the oLPNs are found, the sequentializer redirects the read request to the RWLB by changing the target address to the corresponding tLPNs. Otherwise, the original addresses are used. The search time in the redirection table can increase read request latencies. In order to minimize the search time, the oLPN-to-tLPN mapping entries in the redirection table are maintained with a red-black (RB) tree; thus, the search time grows at the rate of $O(\log n)$. A complex case of read handling is the scenario in which the target logical address range has been partially sequentialized. In this case, the read request must be split into multiple sub-read requests, and only the sub-reads for sequentialized data must be sent to the RWLB. After the completion of all the sub-read requests, the original read request will be completed.

4.4 Randomizing in Device

When the RWLB address space is exhausted by sequentializing operations, the SHRD D/D performs the randomizing operation to reclaim the allocated addresses of the RWLB by sending a special command called *remap*, which restores the oLPNs of sequentialized pages. First, the randomizer selects the address range to be reclaimed; this address range starts from the `rand_start` pointer of the RWLB. Then, the randomizer searches the redirection table for the mapping entries whose tLPN values are included in the randomizing address range and creates the remapping entries for randomizing. The search operation accesses the tLPN-to-oLPN mapping entries in the redirection table. The generated remapping entries, each of which is represented by (tLPN, oLPN), are sorted. By sending the oLPN-sorted remapping entries to the SSD, the spatial locality of CMT access is improved and the CMT miss ratio can be reduced.

The remapping entries are sent as a 4 KB of data in the *remap* command. The size of one remapping entry is 8 bytes, and one remap command can transfer a maximum of 511 remapping entries. The remaining space is used to store the number of remapping entries. Therefore, the SHRD D/D sends multiple remap commands incrementally during randomizing, and other irrelevant read/write requests can be sent to the SSD between remap commands. However, normal requests can be delayed within the SSD if there are many pending remap requests, because each remap command modifies a large number of address mapping entries of the SSD and normal requests cannot be handled simultaneously with the remap command. To solve this problem, two optimization techniques are used. First, we can reduce the maximum number of remapping entries for a single remap command. Second, the maximum number of remap commands which are pending in the SSD can be limited. In our implementation, these two numbers are limited to 128 and 1, respectively. Using these techniques, the delay of normal request can be limited. If a normal request is relevant to a remap command, it must wait for the completion of the remap command.

When the SSD receives a remap command, for each remapping entry (tLPN, oLPN), it inserts the mapping entry (oLPN, PPN) into the L2P mapping table. The PPN is the physical location of the data to be randomized, and it can be obtained by reading the mapping entry (tLPN, PPN) from the L2P mapping table. Therefore, two map pages must be accessed for randomizing one physical page–the map page that contains (tLPN, PPN) and the map page that contains (oLPN, PPN). However, the sorted remapping entries in a remap command will modify only a small number of map pages because it is quite probable that consecutive remapping entries will access a same map page. After all the pages in an RWLB block are randomized, the block is changed to a normal data block. In order to change the block information, dirty mapping entries of the CMT must be flushed into the map blocks, and then, the block change information must be written to the map blocks. After the completion of the remap command, the randomizer removes the mapping entry (oLPN, tLPN) from the redirection table. Optionally, host can send a *trim* command for tLPN.

The special commands of SHRD have ordering constraints. The *twrite_data*(tLPN) command must be sent after the corresponding *twrite_header*(oLPN, tLPN) command is completed. The *remap*(oLPN, tLPN) command must be sent after the corresponding *twrite_data*(tLPN) command is completed. If the SHRD D/D issues a command before its dependent command is completed, these commands can be reordered by the command queueing scheme of the storage interface, thus potentially violating the consistency of the SSD data. Owing to the ordering constraints, SHRD operations result in a small amount of performance overhead in our current implementation. If the SSD firmware can guarantee the ordering between dependent commands, the overhead can be reduced. We will consider this requirement in future work.

4.5 Power-Off-Recovery and Garbage Collection

The redirection table is maintained in the host DRAM; therefore, a sudden power-off can result in a loss of all the sequentializing mapping information. In order to address this issue, we use the *autoremap* technique during POR. As shown in Figure 6, each flash page of sequentialized data contains the oLPN in its OOB area. Therefore, the mapping between PPN and oLPN can be obtained. The POR operation scans all the RWLB blocks and performs the randomizing operation by using the PPN-to-oLPN mappings. The autoremap operation is similar to the randomizing by the SHRD D/D, except that autoremap is invoked internally by the SSD POR firmware. Therefore, only the SSD mapping table is modified without changing the physical locations. The POR operation must scan all the RWLB blocks; therefore, the RWLB size influences the POR time. However, the increased POR time is not critical because sudden-power-offs are rare and the POR is performed at the device booting time. In addition, in our implementation, the RWLB block scan time is less than 0.7 seconds for 64 MB of RWLB. After the autoremap operation, the blocks in the RWLB are changed to normal data blocks and they can be victims for GC.

For each valid page in a GC victim block, GC must copy the pages to a free block and modify the corresponding mapping entry in the L2P mapping table. GC uses the oLPN in the OOB area in order to access the L2P mapping entry of the copied page. If a selected GC victim block was initially allocated as a RWLB block, there can be many map misses in the CMT during GC, because the valid pages of the victim block were written by random write requests. To handle the map misses during GC, we use a map logging technique. If the mapping entry of a copied page is missed from the CMT, the proposed map logging technique does not load the corresponding map page from a map block immediately. Instead, the new mapping entry is written in the map log table (MLT) in DRAM. For the purpose, a small amount of memory space is reserved in DRAM. After several GC victim blocks are reclaimed, the MLT will have many mapping entries to be written to the map pages. If the MLT is full, GC sorts the mapping entries based on the LPN value to improve the spatial locality when accessing map pages, and flushes all the mapping entries of the MLT into the map pages.

The idea of map-logging technique is similar to that of SHRD technique. Therefore, the map logging may be also applied to normal write request handling. However, SSD must reserve a large amount of memory space for the MLT to support normal write requests. In addition, the map logging technique cannot reduce the request handling overhead. Therefore, it is better to use SHRD technique for normal requests.

5 Experiments

5.1 Experimental Setup

We implemented an SHRD-supporting SSD by modifying the firmware of the Samsung SM843 SSD, which is designed for data center storage. For our experiments, the parallel units in the SSD were partially enabled–4 channels, 4 banks/channel, and 4 planes/bank. We implemented two 4-KB-page-level mapping FTLs, demand-loading FTL (DFTL) and SHRD-supporting FTL (SHRD-FTL). They use the CMT scheme with a DRAM whose size is less than that of the entire mapping table. Although the total storage capacity is 120 GB, the device provides only 110 GB of address space to the host system. The remaining over-provisioning space is used for GC and for the RWLB. The host computer system used a Linux 3.17.4 kernel and was equipped with Intel Core i7-2600 3.4 GHz CPU and 8 GB DRAM. The SHRD device driver was implemented by modifying the SCSI device driver of the Linux kernel. For the simplicity of the system, the trim/discard commands are not enabled.

In order to demonstrate the performance improvement achieved by the SHRD scheme, several server benchmarks were used: fio [3], tpcc [6], YCSB [13], postmark [19], and fileserver/varmail workloads of filebench [2]. In the case of the fio random write workload, four concurrent threads were generated and each thread wrote 8 GB of data with 4 KB of random write requests in 32 GB of address space. The tpcc workload was generated by percona's tpcc-mysql. The DB page size was configured to 4 KB, the number of warehouses was 120, and the number of connections was 20. In the case of the YCSB workload, MySQL system and the update-heavy workload (i.e., Workload A), which has 20% reads and 80% updates, were used. The number of transactions of postmark workload was 100,000. In the cases of the fileserver and varmail workloads, the number of files was configured as 200,000. Other parameters used the default options. The benchmarks were run at EXT4 filesystem by default. Table 1 presents the characteristics of each workload, which includes the logical address space, the average size of write requests, the portion of write requests, and the frequency of *flush* command generated by *fsync* calls (the average number of write requests between two flush commands).

Table 1: Workload Characteristics

	logical	avg. write	write	writes btwn
benchmark	space (GB)	size (KB)	portion	flushes
fio(RW)	32	4.8	100%	61.6
tpcc	16	14.8	60%	18.9
YCSB	24	24.1	63%	5.7
postmark	20	72.0	90%	2743.7
fileserver	24	65.6	65%	2668.0
varmail	10	12.8	44%	1.0

Table 2: Statistics on SHRD Operations

	small req.	requests	pages	updated map pages
benchmark	portion	/twrite	/twrite	/remap
fio(RW)	100%	30.3	31.42	5.74
tpcc	58%	3.95	6.76	5.33
YCSB	57%	2.45	9.71	9.76
postmark	88%	11.54	57.38	1.34
fileserver	33%	9.33	57.3	1.66
varmail	97%	1.19	3.25	1.24

For all the following experiments, the default sizes of RWLB, RW threshold, and CMT are 64 MB, 128 KB, and 1 MB, respectively, unless they are specified.

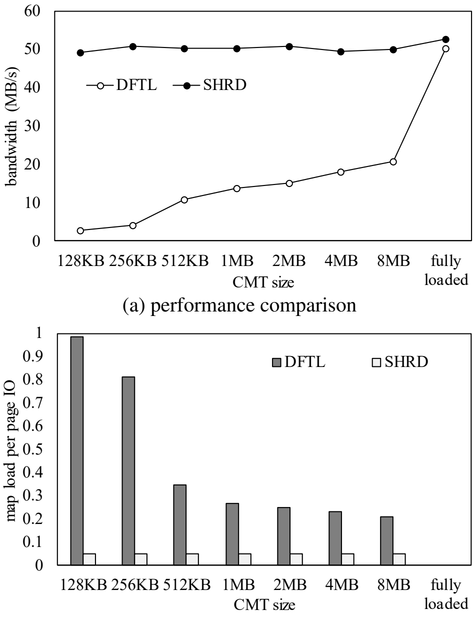
5.2 Experimental Results

Performance improvement with SHRD Table 2 shows several statistics on SHRD operations for each workload, i.e., the portion of sequentialized small write requests, the average number of packed requests/pages per single twrite command, and the average number of updated map pages per single remap command. Although each remap command contains a maximum of 128 remapping entries, it updated less than 10 map pages at all workloads due to the oLPN-sorted map access. Table 3 shows the portions of three reasons for request packing interruption during sequentializing, and the portion of small twrite requests that have less than 32 KB of packed requests.

Figure 7(a) compares the performance of fio random write benchmark under the SHRD and DFTL schemes for different values of the CMT size. Because the logical address space of fio benchmark is 32 GB, if the CMT size is 32 MB, all the mapping entries of the workloads can be fully loaded into the CMT; thus, there is no map-miss handling overhead. If CMT size is less than 32 MB, the CMT can cache the mapping entries for 32 GB of address space partially. For example, 1 MB of CMT can contain only 3.1% of the entire mapping entries of the workload. Figure 7(b) shows the average number of

Table 3: Reasons for packing interruption

	reasons			small twrites
benchmark	no_req	flush	full	(< 32 KB)
fio(RW)	66%	31%	4%	6%
tpcc	50%	50%	0%	53%
YCSB	56%	39%	5%	86%
postmark	81%	0%	19%	1%
fileserver	51%	0%	49%	1%
varmail	21%	79%	0%	83%



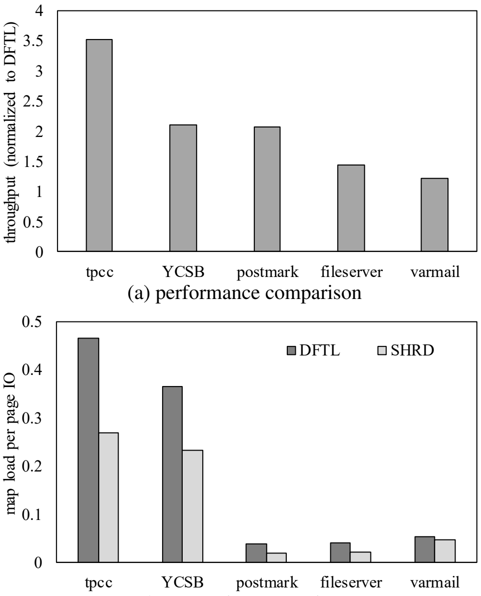
(a) performance comparison

(b) map miss comparison

Figure 7: Fio random write workload results.

map page misses per write operation. A value of 1 indicates that every write operation triggers one additional map page load operation. As the CMT size decreases, DFTL shows worse performance because the number of map page loads increases. However, the performance of SHRD is similar to the performance when map pages are fully loaded into DRAM, irrespective of the CMT size. Even when the mapping table is fully loaded, SHRD shows a better performance than DFTL owing to the reduction on request handling overhead.

Figure 8(a) compares the performance of the SHRD and DFTL schemes for several real workloads under a fixed size of CMT (i.e., 1 MB). For all benchmark workloads, SHRD improved the I/O performance in comparison with DFTL. For the database workloads (i.e., tpcc and YCSB), SHRD demonstrated significantly better performance than DFTL because they are write-dominant workloads. In addition, the average size of write requests is small at these workloads; thus many write requests were sequentialized as shown in Table 2. However, DB workloads generate *fsync* calls frequently; therefore, the request packing at sequentializing was frequently interrupted by the *flush* command, as shown in Table 3, and a smaller number of requests were packed for the DB workloads in comparison with other workloads.



(a) performance comparison

(b) map miss comparison

Figure 8: Real workload results.

The postmark workload is more write-dominant than the DB workloads, and many requests were sequentialized. However, the performance gain was similar to that of YCSB workload because its average write request size is larger. The fileserver workload generates many large requests; therefore, DFTL also show good performance. In addition, the fileserver workload includes many read requests. Because SHRD cannot directly improve read performance, the performance gain was small. The varmail workload generates *fsync* frequently; therefore, a smaller number of requests were packed by the sequentializer as shown in Table 2. Then, many special commands of SHRD must be transferred, thus degrading the performance owing to the ordering constraints explained in Section 4.4. In addition, the varmail workload is read-dominant, thus its performance improvement was small. We used the varmail workload as an adverse workload to check the SHRD overhead. Nevertheless, SHRD showed a better performance than DFTL even for the workload.

Figure 9 shows the map entries accessed by SHRD-FTL during the execution of postmark benchmark. Although the original postmark workload has many small and random write requests, SHRD changed them to sequential write requests to the RWLB address region (blue dots). Therefore, only a small number of accesses occurred on the normal address region by large sequential requests (black dots). During the periodic randomizing operations, the original addresses and the temporary addresses were accessed (read dots). The map entry ac-

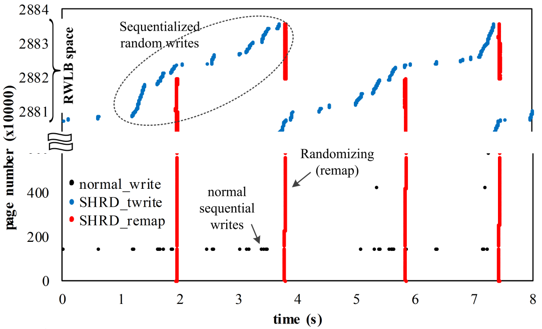
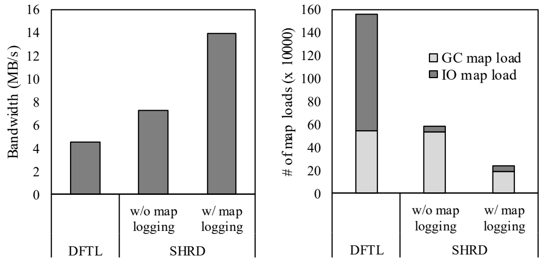


Figure 9: Map entry access pattern (postmark).



(a) performance improvement (b) map page miss reduction

Figure 10: Effect of map logging at GC.

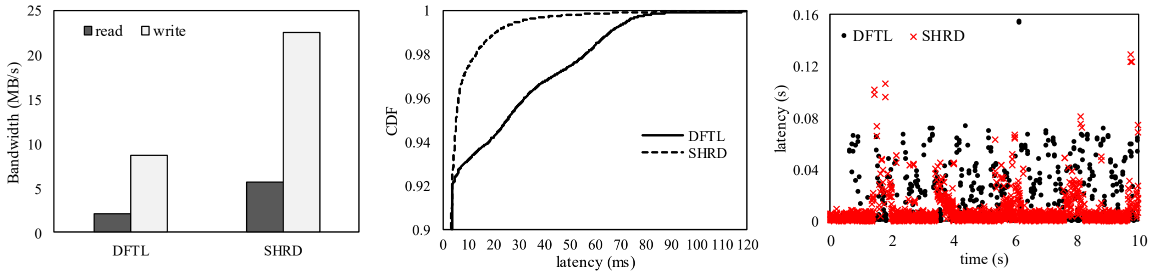
cess pattern during randomizing is sequential owing to the LPN-sorted accesses.

Map logging GC In order to examine the effect of map logging technique explained in Section 4.5, we initialized the SSD with an aging operation. First, 60% of the logical area of SSD was filled with sequential writes. Then, 80% of the sequentially-written data were overwritten with random writes. Finally, we ran the fio workload. The MLT size is 96 KB. Because the aging operation consumes all the physical blocks of SSD, the GC was triggered at the start of the fio test. As shown in Figure 10(a), the performance was further improved by using the map logging technique in addition to SHRD. As shown in Figure 10(b), SHRD can reduce only the number of map misses by normal IO requests. When the map logging technique is used additionally, the number of map misses by GC decreases.

SHRD overhead In the SHRD scheme, the redirection table must be searched for each read request. We measured the read performance degradation due to the redirection table searching overhead. First, we wrote two 4 GB files, one using 4 KB of random write requests and the other using 512 KB of sequential write requests, respectively. Then, the sequentially-written file was read with 4 KB of random read requests. The RWLB size is 64 MB; hence, the redirection table was filled with the mapping entries of the randomly-written file. Although, for each random read request to the sequentially-written file, its mapping entries cannot be found from the redirection table, the read requests must traverse the RB tree of the redirection table until the searching reaches the leaf nodes. In the worst-case scenario, we observed that the read performance degradation is less than 2%.

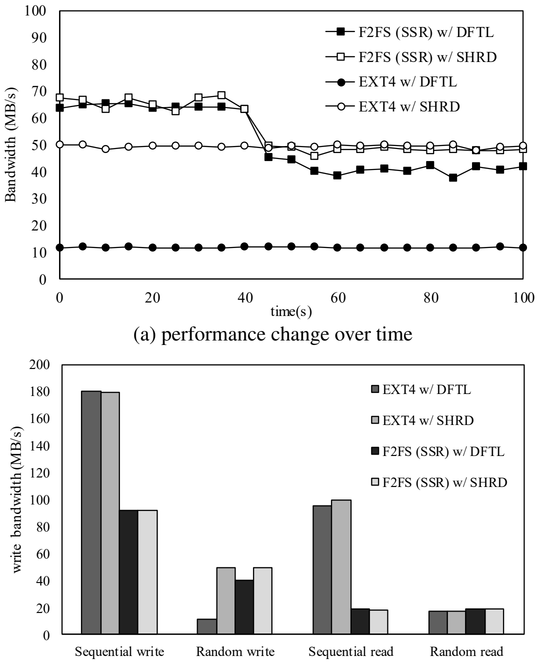
In the case of real workloads, read and write requests are mixed, where SHRD may show a worse read performance due to the sequentializing and randomizing operations on write requests. In particular, the remap command can delay the handling of read request, as explained in Section 4.4. Figure 11 compares the read performance of DFTL and SHRD. The fio mixed workload was used, which has four threads generating 4 KB of random read and write requests. The ratio of write requests is 80%; hence, it is a write-dominant workload. SHRD improved the write performance and thus the read performance was also improved by minimizing the waiting time for write request, as shown in Figures 11(a) and 12(b). SHRD showed long read latencies when SSD handles remap commands. However, the read latencies delayed by remap commands are similar to those delayed by map misses at DFTL, as shown in Figure 11(c).

Performance improvement at EXT4 and F2FS In order to compare the performance gains at different file systems, EXT4 and F2FS were used. F2FS is a log-structured file system, and it supports the slack space recycling (SSR) which permits overwrite operations for invalid blocks [22]. Compared to the garbage collection of LFS, the SSR operation can prevent significant performance degradation when the file system utilization is high. Figure 12(a) shows the performance improvement by SHRD for a random workload under the EXT4 and F2FS file systems. The file system utilizations were initialized to 75% by creating 1,024 files with sequential requests and updating 20% of the data with 4 KB of random write requests. Then, the fio random write workload was run. EXT4 showed a significant performance degradation for the random workload when DFTL was used. SHRD improved the performance of EXT4 significantly by reducing the map handling overhead. For F2FS, many free segments were generated by a background garbage collection before running the fio workload. Therefore, F2FS showed significantly better performance than the original EXT4 file system until 40 seconds elapsed because F2FS generated sequential write requests to the free segments. However, the free segments were exhausted and SSR operations were triggered starting from 40 seconds. The SSR operations generated small random write requests to utilize invalid blocks; thus, the performance of F2FS plummeted. However, by adopting the SHRD scheme, the performance of F2FS was improved even when the SSR operation was triggered. Consequently, EXT4 and F2FS showed similar performance when they adopted the SHRD scheme. As shown in Figure 12(b), F2FS showed worse sequential read/write performance than EXT4 at an aged condition. Although EXT4 showed worse random write performance than



(a) bandwidth (b) CDF of read latency (c) read latency fluctuation

Figure 11: Read performance.



(a) performance change over time

(b) performance comparison at different workloads

Figure 12: Effect of SHRD at EXT4 and F2FS.

F2FS under the DFTL scheme, SHRD removed the random write performance gap between EXT4 and F2FS. Therefore, the combination of EXT4 and SHRD can provide better performance for all types of workloads.

RWLB size and RW threshold When the SHRD scheme is implemented, several factors must be determined by considering the tradeoffs. As a larger size of RWLB is used, more number of remapping entries can share each map page, thus improving the spatial locality on accessing the map pages. In addition, the overwrite operations can invalidate more number of sequentialized pages in RWLB before they are randomized. However, a large RWLB requires a large redirection table and a large amount of table searching overhead. Figure 13 shows the performance changes for various sizes of the RWLB. A large RWLB provides better performance; however, the performance reaches a saturation point. Therefore, the RWLB size must be selected considering the drawbacks of a large RWLB.

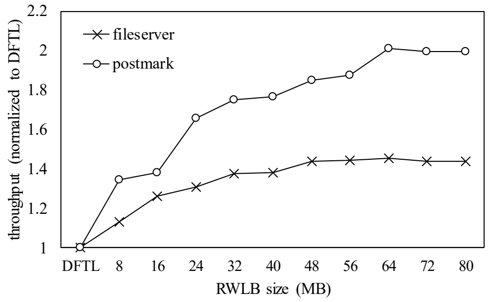


Figure 13: Effect of RWLB sizes.

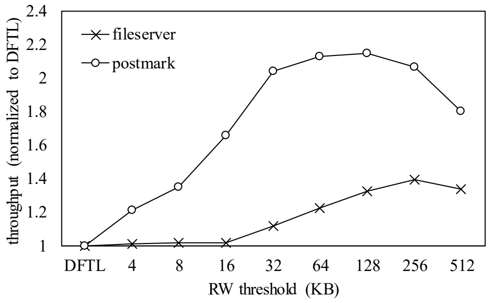


Figure 14: Effect of RW threshold.

The RW threshold determines the amount of data to be sequentialized by SHRD. As we increase the threshold, the performance can be improved by reducing the map handling overhead in the SSD as shown in Figure 14. However, a too large threshold can degrade performance by increasing the overhead of SHRD operations.

6 Related Work

Several studies have investigated approaches to handle the performance gap between sequential writes and random writes at flash storage. LFSs can eliminate random writes at the file system layer; SFS [29] and F2FS [22] are examples of such LFSs. SFS separates hot and cold data into different segments to reduce the cleaning overhead of a traditional LFS. F2FS arranges the on-disk layout from the perspective of the FTL on the SSDs and adopts adaptive logging to limit the maximum latency of segment cleaning. However, despite all the efforts, these flash-based LFSs continue to suffer from write amplification in the segment cleaning phase. Further, LFSs show poor read performance as shown in Figure 12(b).

DFS [17], open-channel SSD [5], and application-managed flash [23] elevate the flash storage software into the OS layer and directly manage flash block allocation according to the flash address space. Therefore, the address mapping table is managed by host computer, and SSD will receive only sequential write requests. However, they can be used only for a specific SSD design and burdens the OS with excessive flash management overhead such as wear-leveling and GC.

Nameless Write [36] permits the storage device to choose the location of a write and inform the OS about the chosen address. Therefore, Nameless Write could eliminate the need for address indirection in SSDs. However, this scheme requires burdensome callback functions to communicate the chosen address to the host OS and necessitates significant changes to the conventional storage interface.

ReSSD [26] and LSDM [37] log random writes sequentially in a pre-reserved storage area and maintain the redirection map table in host memory. However, similar to the previous LFSs, these schemes must copy the data when the log space is reclaimed, thus causing write amplification. Further, they do not consider the POR issue; therefore, when a sudden power-off occurs, the logged data can be lost because the host memory is volatile.

The NVMe standard has a new interface called host memory buffer (HMB) [11], which permits NVMe SSD to utilize the DRAM of the host system via PCIe interface; thus, the vendor can build DRAM-less SSDs by maintaining the entire mapping table in the host DRAM. However, the latency of the host DRAM will be greater than the latency of the internal DRAM for SSD controller. In addition, the volatile mapping table must be flushed periodically to the SSD. On the contrary, SHRD minimizes the flushing overhead of the mapping table and requires only a small size of host memory.

Meanwhile, several studies adopt the FTL-level remap concept, in a manner similar to SHRD. JFTL [12] remaps the addresses of journal data to the addresses of home locations, thus eliminating redundant writes to flash storage. X-FTL [18] supports transactional flash storage for databases by leveraging the address mapping scheme of FTL. ANViL [34] proposes a storage virtualization interface based on FTL-level address remapping by permitting the host system to manipulate the address map using three operations-clone, move, and delete. SHARE [30] also utilizes the address remapping to enable host-side database engines to achieve write atomicity without causing write amplification. Ji *et al.* [15] proposed to use the remap operation for file system defragmentation.

Although the concept of address remapping was introduced by the mentioned studies, it is not trivial to implement the remap operation. SSD maintains two directions of address mappings, i.e., L2P mapping and its reverse P2L mapping. The P2L mapping is used by GC to identify the LPN of a physical page. The remap operation must change both the mappings, and the changed mapping information must be stored in flash memory blocks in order to ensure the durability. The P2L map of each physical page is generally stored in the OOB area of flash page, and thus it cannot be modified without copying the remapped physical page into another flash page. Otherwise, SSD must maintain a separate P2L mapping table. This problem is not easy, and any solution can involve P2L map handling overhead exceeding the benefits of remap operation. In the case of SHRD, we require a "restore" operation to the predetermined address rather than a remap operation to any address because we know the original logical address of a sequentialized page. We can easily implement the restore operation by storing the original logical address in the OOB area at sequentializing without modifying the P2L mapping.

7 Conclusion

We proposed a novel address reshaping technique, SHRD, in order to reduce the performance gap between random writes and sequential writes for SSDs with DRAM resource constraints. The sequentializer of the SHRD technique transforms random write requests into sequential write requests in the block device driver by assigning the address space of a reserved log area in the SSD. Read requests can access the sequentialized data by using a redirection table in the host DRAM. Unlike the previous techniques, SHRD can restore the original logical addresses of the sequentialized data without requiring copy operations, by utilizing the address indirection characteristic of the FTL. We also resolved the POR issue of the redirection table on the host DRAM. We developed a prototype of an SHRD-supporting SSD and a Linux kernel device driver to verify the actual performance benefit from SHRD, and demonstrated the remarkable performance gain. SHRD will be an effective solution for DRAM size reduction in large-capacity enterprise-class SSDs.

Acknowledgements

We thank the anonymous reviewers and our shepherd Sungjin Lee for their valuable comments. We also thank Dong-Gi Lee and Jaeheon Jeong in Samsung Electronics for supporting our work on Samsung SSD device. This research was supported by Samsung Electronics and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2016R1A2B2008672).

References

[1] Embedded Multi-media card (eMMC), Electrical standard (4.5 Device). http://www.jedec.org/standards-documents/results/jesd84-b45.

[2] Filebench. http://filebench.sourceforge.net/.

[3] Flexible I/O Tester. https://github.com/axboe/fio.

[4] MSR Cambridge Traces. http://iotta.snia.org/traces/388.

[5] Open-Channel Solid State Drives. http://lightnvm.io/.

[6] tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql.

[7] LevelDB. http://leveldb.org, 2016.

[8] RethinkDB: The open-source database for the realtime web. https://www.rethinkdb.com/, 2016.

[9] RocksDB: A persistent key-value store for fast storage environments. http://rocksdb.org, 2016.

[10] Chang, L.-P., and Kuo, T.-W. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proc. of 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '02, pp. 187–196.

[11] Chen, M. C. Host Memory Buffer (HMB) based SSD System. http://www.flashmemorysummit.com/, 2015.

[12] Choi, H. J., Lim, S.-H., and Park, K. H. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage 4*, 4 (2009), 14:1–14:22.

[13] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 143–154.

[14] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems*, ASPLOS '09, pp. 229–240.

[15] Ji, C., Chang, L., Shi, L., Wu, C., and Li, Q. An empirical study of file-system fragmentation in mobile storage systems. In *Proc. of 8th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '16.

[16] Jiang, S., Zhang, L., Yuan, X., Hu, H., and Chen, Y. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *Proc. of IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pp. 23–27.

[17] Josephson, W. K., Bongo, L. A., Flynn, D., and Li, K. DFS: A file system for virtualized flash storage. In *Proc. of the 8th USENIX Conference on File and Storage Technologies*, FAST '10.

[18] Kang, W.-H., Lee, S.-W., Moon, B., Oh, G.-H., and Min, C. X-FTL: Transactional FTL for sqlite databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 97–108.

[19] Katcher, J. Postmark: A new file system benchmark. Tech. rep., TR3022, Network Appliance, 1997.

[20] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. A space-efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002), 366–375.

[21] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., and Moriai, S. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review 40*, 3 (2006), 102–107.

[22] Lee, C., Sim, D., Hwang, J.-Y., and Cho, S. F2FS: A new file system for flash storage. In *Proc. of the 13th USENIX Conference on File and Storage Technologies*, FAST '15.

[23] Lee, S., Liu, M., Jun, S., Xu, S., Kim, J., and Arvind. Application-managed flash. In *Proc. of 14th USENIX Conference on File and Storage Technologies*, FAST '16, pp. 339–353.

[24] Lee, S., Shin, D., Kim, Y.-J., and Kim, J. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review 42*, 6 (2008), 36–42.

[25] Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S., and Song, H.-J. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems 6*, 3 (2007).

[26] Lee, Y., Kim, J.-S., and Maeng, S. ReSSD: A software layer for resuscitating SSDs from poor small random write performance. In *Proc. of the ACM Symposium on Applied Computing*, SAC '10, pp. 242–243.

[27] Lee, Y.-G., Jung, D., Kang, D., and Kim, J.-S. μFTL: A memory-efficient flash translation layer supporting multiple mapping granularities. In *Proc. of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pp. 21–30.

[28] Li, Q., Shi, L., Xue, C. J., Wu, K., Ji, C., Zhuge, Q., and Sha, E. H.-M. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *Proc. of 14th USENIX Conference on File and Storage Technologies*, FAST '16, pp. 125–132.

[29] Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y. I. SFS: Random write considered harmful in solid state drives. In *Proc. of the 10th USENIX Conference on File and Storage Technologies*, FAST '12.

[30] Oh, G., Seo, C., Mayuram, R., Kee, Y.-S., and Lee, S.-W. SHARE interface in flash storage for relational and NoSQL databases. In *Proc. of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 343–354.

[31] Park, D., Debnath, B., and Du, D. CFTL: A convertible flash translation layer adaptive to data access patterns. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pp. 365–366.

[32] Park, K.-T., et al. Three-dimensional 128 Gb MLC vertical NAND flash memory with 24-WL stacked layers and 50 MB/s high-speed programming. *IEEE JOURNAL OF SOLID-STATE CIRCUITS 50*, 1 (2015), 204–213.

[33] Rodeh, O., Bacik, J., and Mason, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage 9*, 3 (2013), 9:1–9:32.

[34] Weiss, Z., Subramanian, S., Sundararaman, S., Talagala, N., Arpaci-Dusseau, S. A. C., and Arpaci-Dusseau, R. H. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proc. of the 13th USENIX Conference on File and Storage Technologies*, FAST '15.

[35] Yang, J., Plasson, N., Gillis, G., Talagala, N., and Sundararaman, S. Don't stack your log on my log. In *Proc. of 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, INFLOW '14.

[36] Zhang, Y., Arulraj, L. P., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. De-indirection for flash-based SSDs with nameless writes. In *Proc. of the 10th USENIX Conference on File and Storage Technologies*, FAST '12.

[37] Zuck, A., Kishon, O., and Toledo, S. LSDM: Improving the performance of mobile storage with a log-structured address remapping device driver. In *Proc. of 8th International Conference on Next Generation Mobile Applications, Services and Technologies* (2014), pp. 221–228.