ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction

Kyuhwa Han, *Sungkyunkwan University and Samsung Electronics;*
Hyunho Gwak and Dongkun Shin, *Sungkyunkwan University;*
Joo-Young Hwang, *Samsung Electronics*

https://www.usenix.org/conference/osdi21/presentation/han

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction

Kyuhwa Han[1,2], Hyunho Gwak[1], Dongkun Shin[1*], and Joo-Young Hwang[2]

[1]Sungkyunkwan University, [2]Samsung Electronics

Abstract

The NVMe zoned namespace (ZNS) is emerging as a new storage interface, where the logical address space is divided into fixed-sized zones, and each zone must be written sequentially for flash-memory-friendly access. Owing to the sequential write-only zone scheme of the ZNS, the log-structured file system (LFS) is required to access ZNS solid-state drives (SSDs). Although SSDs can be simplified under the current ZNS interface, its counterpart LFS must bear segment compaction overhead. To resolve the problem, we propose a new LFS-aware ZNS interface, called ZNS+, and its implementation, where the host can offload data copy operations to the SSD to accelerate segment compaction. The ZNS+ also allows each zone to be overwritten with sparse sequential write requests, which enables the LFS to use threaded logging-based block reclamation instead of segment compaction. We also propose two file system techniques for ZNS+-aware LFS. The copyback-aware block allocation considers different copy costs at different copy paths within the SSD. The hybrid segment recycling chooses a proper block reclaiming policy between segment compaction and threaded logging based on their costs. We implemented the ZNS+ SSD at an SSD emulator and a real SSD. The file system performance of the proposed ZNS+ storage system was 1.33–2.91 times better than that of the normal ZNS-based storage system.

1 Introduction

In the NVMe zoned namespace (ZNS) [9] interface, the logical address space is divided into fixed-sized zones. Each zone must be written sequentially and reset explicitly for reuse. The ZNS SSD has several benefits over legacy SSDs. First, performance isolation between different IO streams can be provided by allocating separate zones to each IO stream, which is useful for multi-tenant systems. Second, if the zone size becomes a multiple of the flash erase block size (64–1,024 KB), the ZNS SSD can maintain a zone-level logical-to-physical address mapping (i.e., mapping between zone and flash blocks) because each zone is sequentially written. The coarse-grained mapping requires a small internal DRAM of SSD. Compared with legacy SSDs, which require a large DRAM equivalent to 0.1% of storage capacity (e.g., 1 GB DRAM for 1 TB SSD) for a fine-grained mapping, the DRAM usage of the ZNS SSD is significantly reduced. In particular, because the mapped flash blocks of a zone will be fully invalidated at zone reset, the SSD-internal garbage collection (GC) is not required, and thus, the log-on-log [34] problem can be solved by the *GC-less* SSD. The over-provisioned space for GC is not necessary anymore, and the unpredictable long delays by GC can be avoided. The write amplification by GC can also be eliminated, which will allow triple-level cell (TLC) or quad-level cell (QLC) SSDs with low endurance to proliferate.

IO Stack for ZNS. Generally, new storage interfaces require revamping the software stack. For the ZNS, we need to revise two major IO stack components, file system and IO scheduler. First, the in-place updating file systems such as EXT4 must be replaced with append logging file systems such as the log-structured file system (LFS) to eliminate random updates. Because a segment of LFS is written sequentially by append logging, each segment can be mapped to one or more zones. Second, the IO scheduler must guarantee the in-order write request delivery for a zone. For example, an in-order queue for each zone can be used, and the scheduler only needs to determine the order of services between different zones.

Increased Host Overhead. Under the append logging scheme of LFS, the obsolete blocks of a dirty segment must be reclaimed by segment compaction (also called segment cleaning or garbage collection), which moves all valid data in the segment to other segments to make the segment clean. The compaction invokes a large number of copy operations, especially when the file system utilization is high. The *host-side GC* must be performed in exchange for using GC-less ZNS SSD, although the duplicate GCs by the log-on-log situation can be avoided. The overhead of host-side GC is higher than that of device-side GC because the host-level block copy requires IO request handling, host-to-device data transfer, and page allocation for read data [20]. In addition, segment com-

*Corresponding author

paction needs to modify file system metadata to reflect data relocation. Moreover, the data copy operations for a segment compaction are performed in a batch, and thus, the average waiting time of many pending write requests is significant. According to the experiments of F2FS [19] — one of the widely used log-structured file systems, the performance loss by segment compaction is about 20% when the file system utilization is 90%. Therefore, it can be said that the current ZNS focuses on the SSD-side benefit without considering the increased complexity of the host. To simplify the design of SSD, all the complicated things are passed to the host. (Nevertheless, the host can benefit from the ZNS, in terms of performance isolation and predictability.)

In addition, ZNS storage systems will involve *diminishing returns* as we increase the bandwidth of SSD by embedding more flash chips. The zone size of a ZNS device will be determined to be large enough to utilize the SSD-internal flash chip parallelism. Therefore, a higher bandwidth of ZNS SSD will provide a larger zone size, and the file system must use a larger segment size accordingly. Then, the host suffers from segment compaction overhead more seriously because the overhead generally increases in proportion to the segment size [25]. To improve IO performance and overcome diminishing returns, a host-device co-design is required, which places each sub-task of segment compaction in the most appropriate location without harming the benefit of the original ZNS, instead of simply moving the GC overhead from the SSD to the host.

LFS-aware ZNS. We need some device-level support to alleviate the segment compaction overhead of LFS. Two approaches can be considered: *compaction acceleration* and *compaction avoidance*. We propose a new LFS-aware ZNS interface, called **ZNS+**, and its implementation, which supports **internal zone compaction (IZC)** and **sparse sequential overwrite** via two new commands of `zone_compaction` and `TL_open`. A segment compaction requires four sub-tasks: victim segment selection, destination block allocation, valid data copy, and metadata update. Whereas all others must be performed by the host file system, it is better to offload the data copy task to the SSD, because the device-side data copy is faster than the host-side copy. For compaction acceleration, ZNS+ enables the host to offload the data copy task to the SSD via `zone_compaction`.

To avoid segment compaction, LFS can utilize an alternative reclaiming scheme, called *threaded logging* [19, 26, 28], which reclaims invalidated space in existing dirty segments by overwriting new data. It requires no cleaning operations, but generates random overwrites to segments. In the F2FS experiments using a normal SSD [19], threaded logging showed smaller write traffic and higher performance than segment compaction. However, threaded logging is incompatible with the sequential write-only ZNS interface owing to its random writes. Therefore, the current F2FS patch for the ZNS is disabling threaded logging [3]. In this paper, we will use the term *segment recycling* to cover both segment compaction and threaded logging.

The *sparse sequential overwrite* interface of ZNS+ is a relaxed version of the *dense sequential append write* constraint in ZNS. For a zone opened via `TL_open`, the sparse sequential overwrite is permitted for threaded logging. The ZNS+ SSD transforms sparse sequential write requests to dense sequential requests by plugging the holes between requests with untouched valid blocks in the same segment (**internal plugging**) and redirects the merged requests to a newly allocated flash blocks. Similar to IZC, internal plugging internally copies the valid data of a segment without involving any host-side operations. The only requirement of the write pattern for internal plugging is that the block addresses of the consecutive writes must be in the increasing order. Because the internal plugging is handled between write requests, it improves the average response time of write requests compared with the batch-style segment compaction. Although there are significant extensions in ZNS+ compared to the original ZNS, ZNS+ SSD can provide the same merits of the original ZNS SSD such as small mapping table, no duplicate GC, no over-provisioned space, and performance isolation/predictability.

ZNS+-aware File System. The file system also needs to be adapted to utilize the new features of ZNS+. First, an SSD-internal data copy operation will use different copy paths depending on the source and destination logical block addresses (LBAs). For example, when the two LBAs are mapped to the same flash chip, the *copyback* operation of flash memory can be utilized, which moves data within a flash chip without off-chip data transfers, thus reducing the data migration latency. The copyback operation is currently in the standard NAND interface [6], and its usefulness at SSD-internal garbage collection has been demonstrated by many studies [15, 30, 33]. To fully utilize the copyback operations, we propose the **copyback-aware block allocation** for segment compaction, which attempts to allocate the destination LBA of a data copy such that both the source LBA and destination LBA of the target data are mapped to the same flash chip. The technique can be extended to target other fast copy paths of SSDs.

Second, because ZNS+ supports both segment compaction acceleration and threaded logging, the host file system needs to choose one of those segment recycling policies. Although threaded logging can avoid the segment compaction overhead, it has several drawbacks, as will be explained at §3.3.2. Considering both the merits and demerits of threaded logging, we propose the **hybrid segment recycling** technique for ZNS+, which selects either threaded logging or segment compaction based on their reclaiming costs and benefits.

We implemented the ZNS+ SSD at the SSD emulator of FEMU [22] and an OpenSSD device [29]. In the experiments, the file system performance of the storage system composed of our modified F2FS and ZNS+ SSD was 1.33–2.91 times better than that of the storage system based on the original F2FS and ZNS SSD. The source code of ZNS+ is publicly available at `https://github.com/eslab-skku/ZNSplus`.

2 Background

2.1 SSD Architecture

Modern SSDs consist of multiple flash chips and adopt a multi-channel and multi-way architecture for parallelism. There are multiple parallel flash controllers (channels), and each controller can access multiple flash chips (ways) in an interleaved manner. Each flash chip has multiple flash erase blocks, and each block is composed of multiple flash pages. Flash pages cannot be overwritten until the corresponding flash block is erased. Therefore, SSDs adopt an out-of-place update scheme and use a special firmware, called the flash translation layer (FTL), to manage the logical-to-physical mappings that translate logical addresses used by the host into physical addresses indicating the location within flash memory chips. Typically, flash memory manufacturers recommend that the pages inside a flash block be programmed sequentially in the page number order owing to inter-cell interference. Because the flash page size in recent flash products is usually larger than the logical block size (i.e., 4 KB), multiple logically consecutive blocks will be written at a physical flash page in a cluster. In this study, we refer to the logical consecutive blocks mapped to a flash page as a *chunk*.

Flash memory chip generally supports read, program, erase, and copyback commands. The *copyback* command is used to copy data between flash pages within a flash chip without off-chip data transfer[1]. The chunk is the basic unit of the copyback operation. Because the chip-internal data transfer cannot check the error correction code (ECC) of the target page, the bit error propagation problem exists. To cope with the issue, the error checking can be performed by the flash controller at the same time while doing the copyback operation. If an error is detected, the copied page is invalidated and the corrected data is programmed by the flash controller. Another solution is to allow only a limited number of consecutive copybacks using a threshold copyback counts, which is determined based on copyback error characteristics of flash chip [15, 33].

2.2 Zone Mapping in ZNS SSD

The current NVMe standard interface defines ZNS commands and zone types [5]. There are several zone management commands, such as open, close, finish, and reset for a zone, as well as read and write commands. A notable command under discussion is `simple copy`, through which the host can order the SSD to copy data internally from one or more source logical block ranges to a single consecutive destination logical block range. Although it is apparently similar to our `zone_compaction` command, no studies regarding the issues of the copy command are currently available, and `simple copy` does not fit for our ZNS+, as will be explained at §3.

[1] A flash chip consists of multiple flash dies, each of which has multiple flash planes. Specifically, the copyback can be used for a data copy within a flash plane. In this study, we assume that a flash chip has one die and one plane structure for simplicity. Therefore, we use the term "flash chip" instead of "flash plane" to denote the target device for the copyback command.

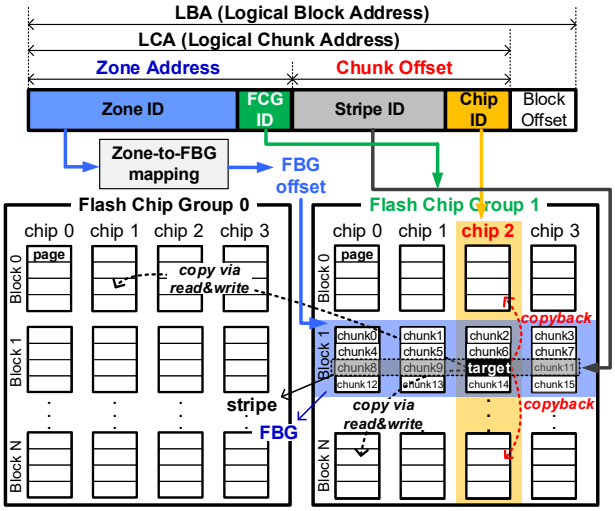


Figure 1: An example of zone and chunk mapping. With the zone address, the second FBG in the FCG 1 is selected. With the chunk offset, the third stripe in the selected FBG and the third flash page (chip 2) in the stripe are targeted.

There are no zone mapping constraints in the ZNS specification. The physical locations of a zone and the chunks in the zone within the storage device are transparent to the host. Device vendors can choose different mapping policies that consider internal design issues. We introduce a general and efficient SSD-internal zone mapping policy, which can minimize the size of required mapping information and maximize the flash chip-level parallelism. Depending on the zone size, one zone can be mapped to one or more physical flash blocks, which are called the **flash block group** (**FBG**) mapped to the zone. The zone size needs to be aligned to the size of the flash block to prevent the creation of partially valid flash blocks. To maximize the flash operation parallelism, the flash blocks from a set of flash chips accessible in parallel will compose the FBG of a zone, and the chunks of a zone need to be interleavingly placed on the parallel flash chips. The number of parallel flash chips for the chunk interleaving is referred to as the *zone interleaving degree* D_{zone}, and the set of logically consecutive chunks across the parallel flash chips is referred to as a **stripe**. For a coarse-grained zone-to-FBG mapping, an FBG has flash blocks at the same block offset in the parallel flash chips, and the chunks of a stripe are located at the same page offset in different flash blocks.

D_{zone} can be smaller than the maximum number of parallel flash chips, D_{max}. Then, D_{zone} needs to be a divisor of D_{max} to partition all the parallel flash chips of an SSD into the same size of **flash chip groups** (**FCGs**). When N_{FCG} denotes the number of FCGs, $N_{FCG} = D_{max}/D_{zone}$. For a given logical chunk address, SSD must determine the mapped flash page of the chunk. Figure 1 presents an example of the zone and chunk mapping, where D_{zone} is 4, and D_{max} is 8. A logical chunk address is divided into the zone address and chunk offset. First, the mapped FBG of the target zone is determined

with its zone address. We assume that the FCG of a zone is statically determined by the least significant $\log_2 N_{FCG}$ bits of the zone address (i.e., FCG ID in Figure 1). Therefore, the i-th zone is mapped to the (i mod N_{FCG})-th FCG. Such a static FCG mapping can reduce the size of zone mapping entry and support our copyback-aware block allocation. The remaining bits of zone address (i.e., zone ID) is used to determine the FBG within the selected FCG. Because the FBG of a zone is dynamically allocated whenever the zone is opened, SSD must maintain the zone-to-FBG mapping table.

The chunk offset is composed of a stripe ID and a chip ID. The former locates a stripe within the target FBG, and the latter determines the target chip index within the target FCG. The bit size of chip ID field is same to $\log_2 D_{zone}$, and the j-th logical chunk of a zone is mapped to the (j mod D_{zone})-th flash chip of the FCG allocated to the zone. Such a direct mapping from chunk offset value can avoid chunk-level fine-grained address mapping. The ZNS SSD only needs to manage the zone-to-FBG mapping. The host can easily calculate the mapped flash chip of a logical chunk by using the FCG ID and chip ID in the logical chunk address without knowing the SSD-internal zone-to-FBG mapping. When a logical chunk needs to be copied to other logical address, the corresponding flash page can be copied within ZNS+ SSD. If the destination logical address is mapped to the same flash chip, the copyback operation can be utilized. Otherwise, a normal copy operation via read and write commands is used.

A zone can be reset via a reset command, which changes the **write pointer (WP)** of the zone to the first block location to overwrite the zone. The WP locates the block address where new data can be written. Owing to the sequential write-only scheme of the ZNS, the WP of a zone is always incremented while the zone space is consumed. Because flash blocks cannot be overwritten, a new FBG is allocated to the zone to be reset, and the zone-to-FBG mapping is modified accordingly. The old FBG can be reused for other zones after it is erased.

2.3 F2FS Segment Management

In this study, we target F2FS [19] as a ZNS-aware file system, which is an actively maintained LFS. As shown in Figure 2, F2FS maintains six types of segments (i.e., hot, warm, and cold segments for each node and data) and uses the multi-head logging policy. Only one segment can be open for each type at a time. Separating hot and cold data into different segments can reduce segment compaction cost. A node block contains an inode or indices of data blocks, whereas a data block contains either directory or user file data. Cold blocks in the hot and warm segments are moved into the cold segments during segment compaction. F2FS supports both append logging and threaded logging. In the append logging, blocks are written to clean segments, yielding strictly sequential writes. On the other hand, threaded logging writes blocks to obsolete space in existing dirty segments without cleaning operations. F2FS uses an adaptive logging policy. When the number of free segments is sufficient, append logging is used first. However, if free segments become insufficient, threaded logging is enabled not to consume further free segments, instead of invoking segment compaction. However, threaded logging is disabled in the current F2FS patch for ZNS, and thus segment compactions will be frequently triggered at the F2FS for ZNS.

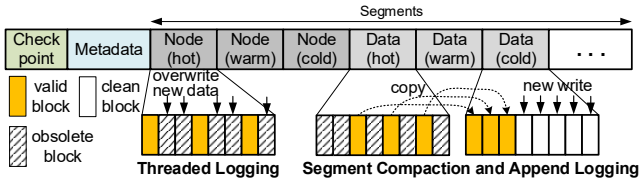


Figure 2: F2FS disk layout and logging schemes.

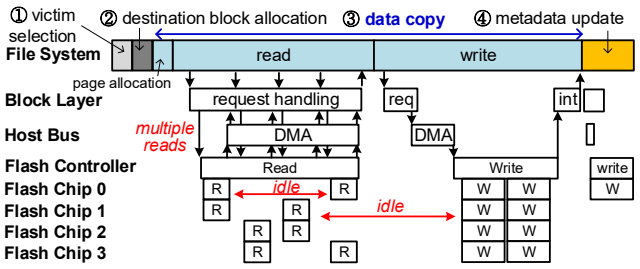
Regarding to segment compaction, F2FS supports both foreground and background operations. The foreground compaction is invoked when there are no sufficient free segments to process incoming write requests. Thus, write requests are delayed during the compaction. The background compaction is triggered only when the file system is idle and the number of free segments is below a threshold. Therefore, the IO delay due to the background compaction is insignificant. Because the threshold is configured small enough so that the compaction does not occur frequently and thus does not harm the lifespan of SSD, the invalidated space cannot be reclaimed in time with only the background compaction, especially when the file system utilization is high and there are a burst of write requests. Therefore, it is important to optimize foreground compaction to improve the overall IO performance. In this paper, we focus on the foreground compaction performance.

3 ZNS+ Interface and File System Support

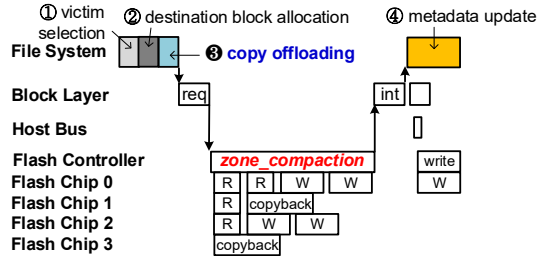
3.1 Motivation

Normal Segment Compaction. The overall process of normal LFS segment compaction consists of four tasks: victim segment selection, destination block allocation, valid data copy, and metadata update, as shown in Figure 3(a). The victim selection finds a segment with the lowest compaction cost (①). The block allocation allocates contiguous free space from destination segments (②). The data copy task moves all valid data in the victim segment to the destination segments via host-initiated read and write requests, which generate significant data transfer traffic between the host and the storage (③). The data copy task has read and write phases.

Read Phase. The host sends read requests for the valid blocks of the victim segment to the SSD if they are not cached at the page cache. Prior to sending the read requests, the corresponding memory pages must be allocated in the page cache, which may invoke write requests to the storage for page frame reclamation. Because the target blocks to be copied are generally scattered at the logical address space, multiple read requests are sent one by one. The SSD reads the target data for a read request with several flash read operations, whose



(a) Normal segment compaction via host-level copy



(b) In-storage zone compaction

Figure 3: Segmentation compaction on (a) ZNS vs. (b) ZNS+ interface (Time goes to the right.)

process can be overlapped at different flash chips. If the size of read requests is small compared to the flash chip parallelism, there will be many idle intervals of flash chips. The data read from the flash chips are transferred to the host via a storage interface such as NVMe. At the experiments using an SSD emulator, we observed that the request submission/completion handling and the device-to-host data transfer accounted for about 7% and 44% of the total read latency, respectively. (Detailed experimental environments are presented in §4.)

Write Phase. This phase can start after all the read requests issued at the read phase are completed. Because the LFS sequentially allocates new blocks for write operations at the destination segment in the append logging scheme, the file system will make one large write request to reduce the request handling overhead. Therefore, the file system waits until all the target blocks are transferred to the page cache, instead of immediately issuing a write request for each block when its read operation is completed. As a result, there is a large idle interval of the SSD, as shown in Figure 3(a).

Metadata Update. F2FS can maintain the consistency of the file system by rolling data back to the most recent checkpoint state when a sudden crash occurs. The file system writes several modified metadata blocks and node blocks to the storage to reflect the change in the data locations and then writes a checkpoint block (④). The metadata must be modified persistently to reclaim the storage space that was occupied by the valid blocks of the victim segment prior to segment compaction. Otherwise, data loss can occur when new data are overwritten in the reclaimed space.

IZC-based Segment Compaction. Figure 3(b) presents the compaction process under our IZC scheme. The data copy task of the normal segment compaction is replaced by the copy offloading (❸), which sends `zone_compaction` commands to transfer the block copy information (i.e., the source and destination LBAs). Because the target data are not loaded into the host page cache, the corresponding page cache allocation is not required. The SSD-internal controller can schedule several read and write operations efficiently while maximizing flash chip utilization. Therefore, the segment compaction latency can be significantly reduced. In addition, the in-storage block copy can utilize copyback operations.

Table 1: Comparison between ZNS and ZNS+

	ZNS	ZNS+
Copy Command	consecutive dest. range (`simple copy`)	dest. LBAs (`zone_compaction`)
Write Constraint	dense seq. write can reuse only after reset	spare seq. overwrite (`TL_open`)
Mapping Transparency	invisible	visible chunk mapping (`identify_mapping`)

3.2 LFS-aware ZNS+ Interface

Table 1 compares the original ZNS and the proposed ZNS+ interface. The ZNS+ supports three new commands, `zone_compaction`, `TL_open`, and `identify_mapping`. `zone_compaction` is used to request an IZC operation. For comparison, the `simple copy` command of the current ZNS standard delivers a single consecutive destination LBA range. Under our ZNS+ interface, the destination range can be non-contiguous; thus, our `zone_compaction` command is designed to specify the destination LBAs rather than a consecutive block range. `TL_open` is used to open zones for threaded logging. The *TL_opened* zones can be overwritten without reset, and the overwrite requests can be sparse sequential. The host can use the `identify_mapping` command to know the address bit fields which determine the mapped flash chip of each chunk.

3.2.1 Internal Zone Compaction

The process of segment compaction in the ZNS+ storage system is as follows.

(1) Cached Page Handling. The first step is to check whether the corresponding page is being cached on the host DRAM for each valid block in the victim segment. If the cached page is dirty, it must be written to the destination segment and must be excluded from IZC operations. If the cached page is clean, it can either be written via a write request or internally copied via `zone_compaction`. If it is transferred from the host via a write request, the corresponding flash read operation can be skipped. Because it is already cached, page allocation is also not needed. Instead, data transfer and write request handling overheads are involved. By comparing the flash read cost and data transfer cost, we can choose a proper scheme to relocate the cached block. In general, high-density flash memories based on TLC or QLC technologies have a relatively higher access latency than the host-to-storage data

transfer cost. However, the recent ZNAND [11] has an extremely short read latency; thus, it may be better to use the in-storage copy for the cached blocks in the ZNAND SSD.

(2) Copy Offloading. Second, to offload the data copy operations to the ZNS+ SSD, `zone_compaction(source LBAs, destination LBAs)` commands are generated. The data in the *i*-th source LBA is copied into the *i*-th destination LBA by the ZNS+ SSD. When threaded logging is enabled at F2FS, the segment compaction can select a TL_opened segment as destination, similarly to the hole-plugging [25,31]. Then, the destination LBAs can be non-contiguous.

(3) Processing IZC. Finally, ZNS+ SSD processes `zone_compaction` commands. It identifies *copybackable* chunks that can be copied with copyback operations by checking the mapped flash chips of their source and destination LBAs. A chunk is copybackable only when all the blocks in it must be copied. The SSD firmware issues flash read and write operations for *non-copybackable* chunks.

Async Interface and Request Scheduling. The handling of `zone_compaction` command is *asynchronous*. The compaction command issued by the host will be enqueued into the command queue, and the host will not wait for the completion of the command. Therefore, the following IO requests can be immediately issued by the host before the completion of the pre-issued zone compaction. The asynchronous handling can improve the performance by eliminating the waiting time of the following requests. Owing to the checkpoint scheme of LFS, the asynchronous command handling does not harm the file system consistency.

Under the asynchronous interface, there will be multiple pending normal requests while handling zone compaction. ZNS+ SSD can reorder normal requests to avoid the convoy effect by the long latency of zone compaction. If the target zone of a normal request is irrelevant to the zone compaction request arrived in advance, it can be processed before the completion of zone compaction. Even for a read request to the destination zone of an on-going zone compaction, the read request can be handled if the WP of the zone has passed through the target block address of the read request.

3.2.2 Sparse Sequential Overwrite

Internal Plugging. To support threaded logging, ZNS+ supports sparse sequential overwrite. Although the write pattern of threaded logging is incompatible with ZNS, there is one consolation; threaded logging accesses the free space of a dirty segment in an increasing order of block address because it consumes the lower address of blocks first. Therefore, its access pattern is sparse sequential (i.e., the WP of a zone will not be decremented). While threaded logging overwrites a segment, if the SSD firmware reads the *skipped blocks* between requests and merges them to the host-sent data blocks, it can make dense sequential write requests to the target zone; this technique is referred to as *internal plugging*. Because the plugging operation is SSD-internal, the latency is shorter than

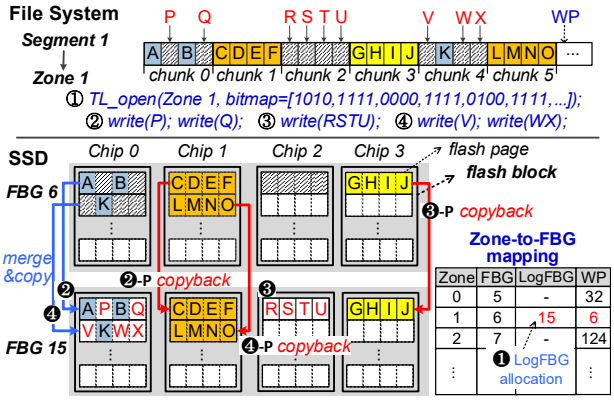


Figure 4: Skipped block plugging for threaded logging

the host-level copy latency. In addition, the SSD can schedule plugging operations efficiently across parallel flash chips to hide their latency.

Figure 4 illustrates an example of internal plugging. Segment 1 is now mapped to Zone 1 in the file system, and FBG 6 is allocated to Zone 1, as shown in the zone-to-FBG mapping. An FBG is composed of four flash blocks, each of which is in different flash chips. The host file system allocates Segment 1 for threaded logging, and sends write requests to invalid blocks to reclaim them while skipping valid blocks. For example, the blocks of A and B in chunk 0 are skipped blocks.

Opening Zone for Threaded Logging. For internal plugging, the SSD must perceive the skipped blocks in the target segment of threaded logging. Owing to the in-order request delivery of ZNS IO stack, the SSD can identify the skipped blocks by comparing the start LBA of an incoming write request with the current WP of the corresponding zone. However, only after a write request arrives, the preceding skipped blocks can be recognized. Therefore, the plugging operation will delay the handling of the write request. To solve this problem, we added a special command, called `TL_open(open zones, valid bitmap)`, that delivers the valid bitmap of the target zones selected for threaded logging (①). Once an allocated segment is informed via `TL_open`, the threaded logging reclaims only the invalid blocks marked at the transferred bitmap. Therefore, the SSD can identify the blocks to be skipped by threaded logging in advance and perform the plugging before the following write request arrives.

LogFBG Allocation. Because a TL_opened zone will be overwritten, the ZNS+ SSD resets the WP of the zone and allocates a new FBG, called **LogFBG**, where new data to the zone are written. For example, in Figure 4, the SSD allocates a LogFBG, FBG 15, for Zone 1 (❶). For the TL_opened zone, the data blocks of the zone are distributed into two FBGs, i.e., the original FBG (FBG 6) and the LogFBG (FBG 15). Therefore, both of them must be maintained as mapped FBGs for the zone. To handle a read request, the SSD identifies the block location of up-to-date data by comparing the target LBA with the WP. If the target LBA is behind the WP (target LBA < WP), the LogFBG is accessed. Otherwise, the original

FBG is accessed for the read request. While the invalid blocks of a TL_opened zone are overwritten by threaded logging, all the valid blocks are copied into the LogFBG. Under a static zone mapping policy, each logical chunk of a zone is always mapped to the same flash chip whenever a new FBG is allocated to the zone. Therefore, fully valid chunks can be copied from the original FBG to the LogFBG via copyback during internal plugging. The number of allocated LogFBGs is determined by the number of TL_opened segments, which is six at maximum in F2FS. Therefore, the space overhead due to LogFBGs is negligible. When the TL_opened zone is finally closed, the LogFBG replaces the original FBG, which is deallocated for future reuse.

LBA-ordered Plugging. When the SSD receives two write requests to chunk 0 in Figure 4 (②), it reads the skipped blocks of A and B from FBG 6, merges them with the host-sent blocks of P and Q, and writes a full chunk at the LogFBG (❷). After handling the write requests to chunk 0, the SSD can perceive that chunk 1 will be skipped by checking the valid bitmap of the zone. To prepare the WP in advance for the write request to chunk 2, the skipped chunk must be copied to the LogFBG. Therefore, after processing a write request, if the following logical chunks are marked as valid in the valid bitmap, the ZNS+ SSD copies them to the LogFBG while adjusting the WP (❷-P, ❸-P, and ❹-P). This type of plugging is called *LBA-ordered plugging* (LP), where each plugging is performed at the current WP of the zone to follow the LBA-ordered write constraint.

PPA-ordered Plugging. Although incoming data must be written at the WP of zone, there is no need to perform the internal plugging operation only at the WP. The internal plugging can be done in advance at the block addresses in ahead of the WP. We only need to consider that the flash pages in a flash block must be programmed sequentially. Therefore, the plugging operation of a fully valid chunk can be scheduled in advance even when the current WP is behind the location where the chunk must be copied to. A fully valid chunk can be copied to a physical page address (PPA) if all the flash pages at lower PPAs within the target flash block have been programmed. For the example in Figure 4, chunk 3 can be copied before the write requests to chunk 0 and chunk 2 arrive because they use different flash chips. If chunk 1 has been copied, chunk 5 can be copied even when the write requests to chunk 2 and chunk 4 have not yet arrived. We call this technique *PPA-ordered plugging* (PP), which considers only the PPA-ordered write constraint. Whenever a flash page is programmed at a LogFBG, the PPA-ordered plugging checks the validity of the chunks mapped to the following flash pages in the corresponding flash block and issues all possible plugging operations for the flash block in advance. However, if excessive plugging operations are issued, they may interfere with the user IO request handling. To resolve this problem, the plugging operations are processed in the background when the target flash chip is idle. If there is no sufficient idle time,

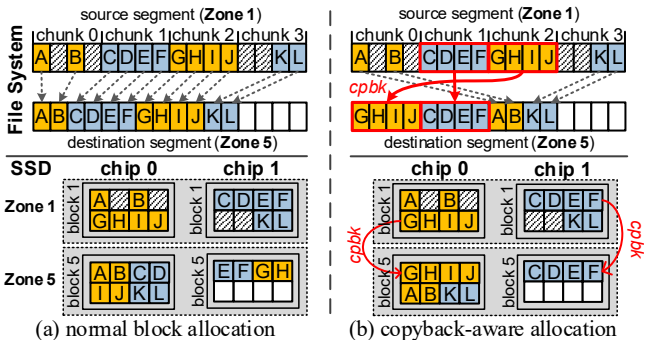


Figure 5: Copyback-aware block allocation

they are handled when the WP of the zone must pass the skipped block locations.

Why Threaded Logging Improves Performance. We can consider that the segment compaction cost restrained by threaded logging is revived in the form of internal plugging in the SSD. This is because the number of blocks to be copied by internal plugging at threaded logging is equal to the number of blocks to be copied at segment compaction for the same segment. However, the metadata modifications to reflect data relocation can be avoided by threaded logging. In addition, the internal plugging cost can be hidden by utilizing idle flash chips. Moreover, the plugging operations are distributed between normal write requests while minimizing the average delay of write requests. On the contrary, the segment compaction is a batch operation. No write requests can be processed until the segment compaction is completely finished. Thus, ZNS+ system can show a better performance when threaded logging is enabled. Because the host knows the amount of skipped blocks within a segment, the internal plugging does not harm the predictability of ZNS. Although the internal plugging amplifies flash write operations, there is no additional endurance degradation compared to segment compaction, which also generates the same amount of flash write operations to reclaim invalidated space.

3.3 ZNS+-aware LFS Optimization

3.3.1 Copyback-aware Block Allocation

Low Utilization of Copyback at LFS. For the valid blocks to be relocated at segment compaction, new block locations are sequentially allocated from a destination segment in the order of their source LBAs in the original LFS. Therefore, the scattered valid blocks in the victim segment are simply compacted without holes between them at the destination segment; in this type of scheme, most of the chunks will not be copybacked. Figure 5(a) illustrates an example of segment compaction under normal block allocation, in which two logically consecutive chunks comprise a stripe over two flash chips. Under normal block allocation, no chunks can be copybacked in this example.

Chunk Mapping Identification. To maximize the usage of copyback, we propose the *copyback-aware block allocation*.

Figure 5(b) illustrates an example of segment compaction under the copyback-aware block allocation. First, the file system reserves contiguous free blocks at the destination segment as the number of valid blocks to be copied by segment compaction. Second, the file system allocates the destination chunk location from the reserved region for each fully valid chunk in the source segment (e.g., chunk 1 and chunk 2 in the example) such that both the source and destination chunks are mapped to the same flash chip. Under a static chunk mapping scheme, the host can easily calculate the mapped flash chip number of a chunk if it knows which bit ranges of logical chunk address determine the chip offset. For the `identify_mapping` command, the ZNS+ SSD returns the bit ranges of FCG ID and chip ID, shown in Figure 1. If two chunk addresses have the same values of FCG ID and chip ID, they are mapped to the same flash chip. The host queries the chunk mapping information only while booting, and no additional inquiry is required at run time. After handling all the fully valid chunks, the destination block locations of the remaining valid blocks are determined to fill all the free space of the reserved region. For example, the new block locations of A, B, K, and L in Figure 5(b) are allocated to the remaining space and can be copied via flash read and write operations.

Maximizing Copyback Usage. If the fully valid chunks in the source segment are not evenly distributed among multiple flash chips, some chunks cannot find the copyback-able chunk locations from the reserved region and must be copied to non-copybackable chunks. One solution is to allocate additional chunks in the destination segment to maximize the usage of copyback, while leaving some unused blocks in the destination segment, which is possible because the `zone_compaction` command can specify non-contiguous destination ranges. Another issue is to use copyback for partially invalid chunks (e.g., chunk 0 and chunk 3 in Figure 5). By copying an entire chunk including invalid blocks via copyback, we can reduce the segment compaction time. Although allocating more space in the destination segment can maximize the usage of copyback, the segment reclaiming efficiency can be degraded. Considering this trade-off, a threshold for the allowed additional space needs to be determined. A more detailed consideration is beyond the scope of this study.

Extensions. The proposed copyback-aware block allocation can be extended for recent multi-core SSDs, where multiple embedded processors exist with each processor running an FTL instance to manage its own partitioned address space and flash chips while utilizing the processor-level parallelism [18, 35]. In the multi-core SSD, a zone can be mapped across multiple partitions, and the inter-partition copy latency will be longer than that of intra-partition copy because communication overhead will be imposed for the inter-partition operation. Therefore, a *partition-aware block allocation* will be beneficial for the multi-partitioned ZNS+ SSDs.

Instead of the file system-level copyback-aware block allocation, we can consider a device-level approach, where the target block location is not specified by the host, and the SSD determines the logical block locations to maximize the usage of copyback and informs the file system of the allocated block locations, similarly to the ideas of the nameless write [36] and the `zone append` command defined in the standard ZNS interface. This approach will enable copyback-aware block allocation even when the host has no knowledge of the SSD-internal chunk mapping.

3.3.2 Hybrid Segment Recycling

Although threaded logging can reduce the block reclamation overhead, its reclaiming efficiency can be lower than that of segment compaction in the ZNS+ owing to two reasons.

Reclaiming Cost Imbalance. First, threaded logging may suffer from unbalanced reclaiming costs among different types of segments. Whereas segment compaction selects a victim segment with the lowest compaction cost (i.e., the smallest number of valid blocks) among all dirty segments, threaded logging can select the target segment for a type of write request only from the same type of dirty segments to prevent different types of data from being mixed in a segment. Even when there are multiple segments whose blocks are mostly invalidated, threaded logging cannot utilize them for a write request if the types of those segments are different from the data type of the write request. Instead, the same type of dirty segment must be selected despite a high internal plugging cost. In addition, unlike segment compaction, threaded logging cannot move cold data into cold segments. The cold data trapped in a segment must be copied by the internal plugging each time the segment is opened for threaded logging.

Pre-Invalid Block Problem. Second, the reclaiming efficiency of threaded logging will be further degraded if threaded logging is used for a long period without checkpointing, which is not mandatory for threaded logging. This is due to *pre-invalid blocks* that are invalid but still referenced by in-storage metadata and thus non-reclaimable. When a logical block is invalidated by a file system operation but a new checkpoint is still not recorded, the logical block becomes pre-invalid and must not be overwritten because a crash recovery will need to restore it. They must be copied by the internal plugging. The pre-invalid blocks accumulate as threaded logging continues without checkpointing, whereas they can be reclaimed by segment compaction because segment compaction accompanies checkpointing. The original F2FS prefers threaded logging because the performance gain by avoiding segment compaction is more significant than the drawback of reclaiming inefficiency in threaded logging, which may be true in legacy SSDs. However, the reclaiming inefficiency results in a high internal plugging cost at the ZNS+ SSD.

Periodic Checkpointing. To solve the pre-invalid block problem, we use the *periodic checkpointing*, which triggers checkpointing whenever the number of accumulated pre-invalid blocks exceeds θ_{PI}. For periodic checkpointing, the file system must monitor the number of pre-invalid blocks. If

checkpointing is invoked too frequently, the write traffic on metadata blocks increases, and the flash endurance of SSD will be harmed. Therefore, an appropriate value of θ_{PI} needs to be determined considering the trade-off. From experiments using several benchmarks, we identified that the overall performance was maximized when θ_{PI} was around 128 MB. Thus, θ_{PI} was configured to the value in the experiments in §4.

Reclaiming Cost Modeling. We propose the *hybrid segment recycling* (HSR) technique, which chooses a reclaiming policy by comparing the reclaiming costs of threaded logging and segment compaction. The reclaiming cost of a segment under threaded logging, C_{TL}, can be formulated as follows:

$$C_{TL} = f_{plugging}(N_{pre\text{-}inv} + N_{valid}) \quad (1)$$

$N_{pre\text{-}inv}$ and N_{valid} indicate the number of pre-invalid blocks and the number of valid blocks, respectively. $f_{plugging}(N)$ is the in-storage plugging cost for N blocks. Because the plugging operation can be performed in the background, $f_{plugging}(N)$ is less than on-demand internal copy cost. We set $f_{plugging}(N)$ to be 90% of the on-demand copy cost based on the experimental results on performance improvement by the internal plugging.

Segment compaction only moves the valid blocks of the victim segment to the free segment. During segment compaction, cold blocks are moved to cold segments, which will decrease the future reclaiming cost of the target segment. However, it must modify node blocks and metadata blocks to reflect the change in block locations at the checkpointing. Therefore, the reclaiming cost of segment compaction, C_{SC}, can be expressed as follows:

$$C_{SC} = f_{copy}(N_{valid}) + f_{write}(N_{node} + N_{meta}) - B_{cold} \quad (2)$$

N_{node} and N_{meta} denote the numbers of the modified node blocks and metadata blocks, respectively. $f_{copy}(N)$ and $f_{write}(N)$ are the copy cost and the write cost of N blocks, respectively. B_{cold} represents the predicted future benefit from cold block migration. When the victim segment of segment compaction is a node segment, no additional node update occurs; thus, N_{node} is 0. To estimate $f_{copy}(N_{valid})$, we can assume that all the chunks are copied via in-storage copy operations, and some portion of the copy operations can be handled by copyback commands.

Using Approximate Cost. Whereas $N_{pre\text{-}inv}$ and N_{valid} can be easily calculated by examining the valid bitmap of each segment, the calculations of N_{node} and N_{meta} are not simple. They are determined by the number of node blocks associated with the data blocks that are relocated at segment compaction. It is highly expensive to precisely calculate each number during the selection process of the reclaiming policy. Therefore, we use approximate values for N_{node} and N_{meta}. Assuming that they become larger in proportion to N_{valid}, $\alpha \times N_{valid}$ can be used instead of the real value of $(N_{node} + N_{meta})$. The value of α depends on workloads, and thus, its average value can be profiled at run time. For our target benchmarks, we observed that α has an average value of 20%.

It is also difficult to predict the exact value of B_{cold}, and thus, it is approximated to $f_{plugging}(\beta \times N_{cold})$, where N_{cold} is the number of blocks unchanged across two consecutive TL-based segment recyclings of the target segment, assuming that β% of N_{cold} in the segment will be still valid when the segment is TL_opened again next time. Therefore, $f_{plugging}(\beta \times N_{cold})$ amount of internal plugging cost at the future threaded logging can be avoided by moving the cold blocks at the current segment recycling. The value of β also depends on workloads, and its appropriate value can be profiled at run time.

By comparing C_{TL} of the threaded logging and C_{SC} of the segment compaction, the HSR chooses one of the recycling policies. Note that threaded logging and segment compaction will select different victims because they examine different candidates. When the reclaiming cost imbalance among different segment types is serious, segment compaction will be chosen because it can find better victim segments.

4 Experiments

We evaluated the performance of ZNS+ SSD using an SSD emulator, which was implemented based on FEMU [22]. In the emulation environment, the host computer system used the Linux 4.10 kernel and was equipped with an Intel Xeon E5-2630 2.4 GHz CPU and 64 GB of DRAM. We allocated four CPU cores, 2 GB of DRAM, 16 GB of NVMe SSD for user workloads, and a 128 GB disk for the OS image to the guest virtual machine. The emulated NVMe SSD consists of 16 parallel flash chips by default, which can be accessed in parallel via eight channels and two ways per channel. The type of flash chip was configured to either TLC, multi-level cell (MLC), or ZNAND, as shown in Table 2. The default flash medium was the MLC in our experiments. The data transmission link between the host and the SSD was configured to a two-lane PCIe Gen2 with the maximum bandwidth of 600 MB/s per lane. The zone interleaving degree, D_{zone}, was set to be the same as the maximum number of parallel flash chips (i.e., $D_{zone} = 16$). Therefore, the default zone size of ZNS+ SSD is 32 MB, which is the total size of 16 flash blocks distributed in 16 flash chips. When the number of parallel flash chips was configured to a different value, the zone size was also changed according to the size of the parallel FBG. To determine the effect of copyback, we modified FEMU to support the copyback operation. Table 2 shows the copyback latency normalized by the latency of the copy operation using the read-and-program commands. The copyback operation is approximately 6–10% faster than the normal copy operation.

We modified F2FS version 4.10 to exploit the ZNS+ interface. The complexity of F2FS was increased by 5.4% to implement ZNS+-aware F2FS (from 20,160 LoC to 21,239 LoC). The F2FS segment size was configured to be equal to the zone size; therefore, its default size is 32 MB. The *copyback-aware block allocation* was applied, which was enabled by default in the experiments. The filebench [4] (fileserver and varmail)

Table 2: Flash memory configurations

	TLC [24]	MLC [23]	ZNAND [11]
Flash page read latency	60 μs	35 μs	3 μs
Flash page program latency	700 μs	390 μs	100 μs
DMA from/to flash controller	24 μs	24 μs	3.4 μs
Normalized copyback latency	0.94	0.90	0.94
Flash page: 16 KB, Pages per flash block: 128, 16 flash chips (default)			
Host interface: PCIe Gen2 2x lanes (max B/W: 1.2 GB/s)			

Table 3: Benchmark configurations

fileserver	112,500 files, file size: 128KB, 14GB fileset, 50 threads
varmail	475,000 files, file size: 28KB, 12.7GB fileset, 16 threads
tpcc	DB size: 12GB, 1GB buffer pool, 16 connections
YCSB	DB size: 12GB, 1GB buffer pool, 32 connections

and OLTP benchmarks (tpcc [8] and YCSB [12] on MySQL) were used for the evaluation. We set the file system size to 16 GB and determined the dataset size of each benchmark such that the file system utilization was 90% by default. The configuration of each benchmark is detailed in Table 3.

In the experiments, the following two different versions of ZNS+ were used: `IZC` and `ZNS+`. While threaded logging is disabled in `IZC`, it is enabled and the hybrid segment recycling is used in `ZNS+`. The PPA-ordered plugging was used by default in the experiments. The ZNS+ schemes were compared with `ZNS`, which uses the original F2FS (ZNS patch version) and ZNS SSD. `ZNS` uses the host-level copy to perform segment compaction and does not utilize threaded logging. Because each workload generates write requests without idle intervals, no background compactions were invoked by F2FS.

4.1 Segment Compaction Performance

Figure 6 presents the average segment compaction latencies of `ZNS` and `IZC` at various benchmarks. The SSD emulator was used for the experiments. The compaction time is divided into four phases (init, read, write, and checkpoint) and three phases (init, IZC, and checkpoint) for `ZNS` and `IZC`, respectively. The init phase reads several metadata blocks of all the files related to the victim segment. Because these metadata are generally being cached in the page cache, the init phase is short.

`IZC` reduced the zone compaction time by about 28.2–51.7%, compared to `ZNS`, by removing the host-level copy overhead and utilizing copyback operations. The ratios of the copyback operations among all the in-storage copy operations were 87%, 74%, 81%, 83%, and 83% during the workloads of the fileserver, varmail, tpcc, YCSB-a, and YCSB-f, respectively. Because the checkpoint phase must wait for the persistent completion of the previous phases, the checkpoint latency includes the waiting time for the completion of the write operations or the IZC operations. Therefore, the checkpoint latency was increased during the OLTP workloads by the `IZC` technique. The segment compaction by host-level copy operations can be disturbed by user IO requests. Whereas the IO traffic of the filebench workloads is intensive, those of the OLTP workloads are small. Therefore, the performance

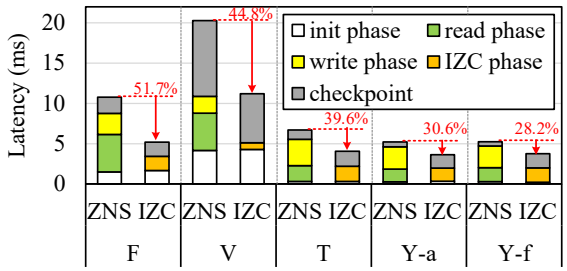


Figure 6: Average compaction time (F: fileserver, V: varmail, T: tpcc, Y-a: YCSB workloada, and Y-f: YCSB workloadf)

Table 4: The bandwidth (MB/s) at fileserver workload in different NAND flash media

	TLC	MLC	ZNAND
`ZNS`	79.5 (1.00x)	84.5 (1.00x)	104.0 (1.00x)
`IZC-H`	113.4 (1.43x)	154.6 (1.83x)	218.9 (2.10x)
`IZC-D`	96.5 (1.12x)	148.0 (1.75x)	242.4 (2.33x)

improvement by IZC is more significant during filebench workloads because the in-storage copy operations mitigate the interference with user IO requests for using the host resource and the host-to-device DMA bus.

We also compared two different policies on fully cached logical chunks, presented in §3.2.1, using different types of NAND media in Table 2. Whereas the fully cached chunks are directly written by the host in `IZC-H`, all the chunks are copied by device in `IZC-D`. Exceptionally, if the latest version of a block only exists in the host DRAM with a dirty flag, the host directly writes the block to the storage.

Table 4 compares the bandwidths of different copy schemes in different flash media. The performance gain by IZC is more significant for a faster flash media because the host IO stack contributes a larger portion of the IO latency for faster flash media. `IZC-H` and `IZC-D` offloaded 89.6% and 99.4% of block copy operations to the ZNS+ SSD, respectively. This indicates that about 10.4% of the total chunks to be copied were cached in the host DRAM (clean: 9.8%, dirty: 0.6%). When the TLC flash memory was used, `IZC-H` outperformed `IZC-D` because the TLC flash read latency is longer than the host-level write request handling overhead. However, the performance difference between `IZC-H` and `IZC-D` is reduced when the MLC flash is used. If the flash access time is further reduced by using ZNAND, `IZC-D` delivers a better performance (i.e., offloading all copy requests to the storage, regardless of whether the target blocks have been cached, achieves better performance). In the following experiments using MLC flash memory, we used `IZC-H` by default.

4.2 Threaded Logging Performance

We compared the overall performances of the benchmarks under `ZNS`, `IZC`, and `ZNS+` to evaluate the effects of in-storage zone compaction and threaded logging support, as shown in

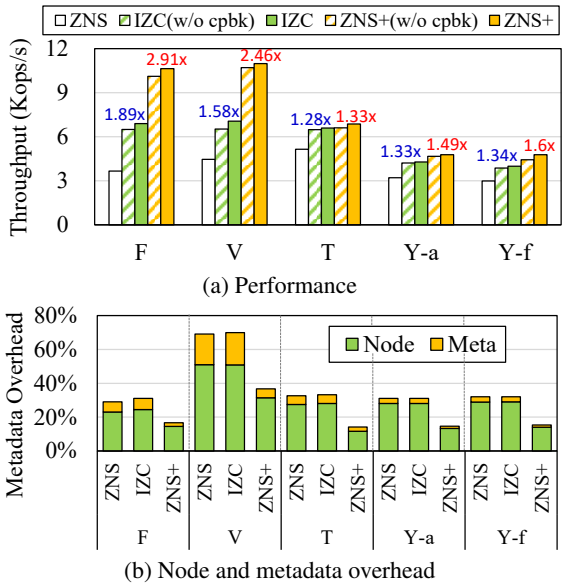


Figure 7: Effect of the threaded logging support

Table 5: Ratios of threaded logging (TL) and background plugging (BP) at ZNS+ (%)

	Fileserver	Varmail	TPCC	YCSB-a	YCSB-f
TL	94.8	92.1	85.8	89.8	89.7
BP	11.3	31.7	24.7	34.1	35.1

Figure 7(a). The copyback-disabled versions of IZC and ZNS+, IZC(w/o cpbk) and ZNS+(w/o cpbk), were also examined. Figure 7(b) presents the file system metadata overhead, which indicates the write traffic on the node blocks and the file system metadata blocks. The overhead values are normalized to the user data traffic. IZC presents about 1.21–1.77 times higher throughputs than that of ZNS because IZC can significantly reduce segment compaction time. ZNS and IZC present similar metadata overheads because they only use segment compaction for invalidated space reclamation. In the OLTP workloads, the segment compaction cost occupies a smaller portion of the total IO latency compared to filebench workloads. Therefore, the overall performance improvements at the OLTP workloads are less than those of the filebench workloads.

ZNS+ outperforms both ZNS and IZC for all benchmarks. ZNS+ presents approximately 1.33–2.91 times higher throughputs than that of ZNS. As shown in Figure 7(b), ZNS+ modifies fewer node blocks and metadata blocks because threaded logging does not invoke checkpointing. ZNS+ reduced the node and metadata write traffic by about 48%, compared to IZC, for the varmail workload. Table 5 presents the ratio of the segments reclaimed by threaded logging in ZNS+. In all the workloads, more than 85.8% of the reclaimed segments are handled by threaded logging in ZNS+, because our periodic checkpoint scheme limits the number of pre-invalid blocks. For the fileserver and varmail workloads, which update file system metadata frequently, ZNS+ presents significant

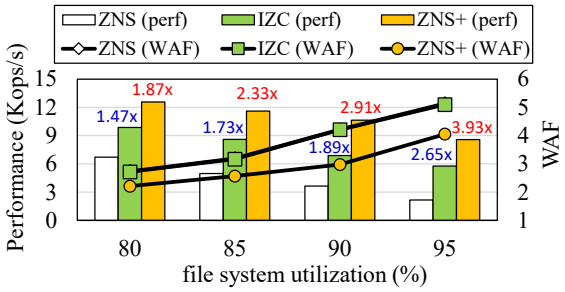


Figure 8: Performance at various file system utilizations (fileserver workload)

improvements over IZC because threaded logging achieved a high reclaiming efficiency during node segment reclaiming. Table 5 also shows the ratio of the blocks copied in the background by the PPA-ordered plugging among all the copied blocks. The ratio of background plugging is high at the fsync-intensive workloads (varmail, tpcc, and YCSB) because the small-sized fsync requests cause frequent idle intervals of the flash chips. Since the average background plugging ratio is about 27%, a significant portion of internal plugging overhead was hidden in the ZNS+ SSD.

We also compared the performances of ZNS schemes while varying the file system utilization. By changing the file-set size of the target workload, we controlled the file system utilization. Figure 8 presents the workload throughput and the write amplification factor (WAF) for each technique for the fileserver workload. The WAF is the total write traffic invoked by the file system (including data block writes, node block writes, metadata updates, segment compaction, and internal plugging) divided by the write traffic generated by the user workload. The WAF values of IZC and ZNS are similar. As the file system utilization increases, the WAF increases because segment compaction must copy a larger number of valid blocks, and segment compaction is invoked more frequently. Because threaded logging reduces the number of node and metadata updates, ZNS+ shows lower WAF values than those of IZC. The performance gain by IZC or ZNS+ over ZNS increases as the file system utilization increases because the segment recycling cost is more significant at a higher file system utilization.

4.3 SSD-internal Chip Utilization

We measured the effects of the proposed techniques on flash chip utilization, as shown in Figure 9. The IZC technique can increase the chip utilization by reducing the idle intervals invoked during the host-level copy operations, as shown in Figure 3. A higher flash chip utilization generally results in a higher IO performance. To measure the effect of the PPA-ordered plugging technique, which utilizes idle flash chips, we observed chip utilization for the following two different plugging schemes of ZNS+: LBA-ordered plugging (LP) and PPA-ordered plugging (PP). Whereas LP copies the following

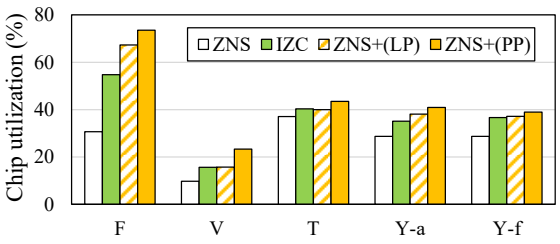


Figure 9: Flash chip utilization in different ZNS schemes

skipped blocks just after handling a write request, PP processes all the possible plugging operations whenever each flash chip is idle.

In Figure 9, IZC and ZNS+ present higher chip utilizations than that of ZNS for all workloads. Whereas ZNS+(LP) can utilize the idle interval between two consecutive write requests, ZNS+(PP) can overlap the plugging operations with normal write request handling by utilizing idle flash chips. Therefore, ZNS+(PP) showed higher chip utilizations compared to ZNS+(LP). The performance improvements achieved by the different plugging techniques were similar to the chip utilization improvements by them.

Figure 10 presents the change in chip utilization for the fileserver workload. We started to measure the utilization after the file system enters a steady state, where segment reclaiming occurs consistently. While a black dot represents the average utilization of a 100-ms interval, the red line indicates the change in the average utilization of a 5-s moving interval. Under the ZNS technique, many low utilization intervals were observed at less than 20% owing to host-level copy operations during segment compaction. IZC eliminated most of the low utilization intervals via in-storage copy operations. ZNS+ increased chip utilization significantly owing to the background plugging operation. A few low utilization intervals at ZNS+ were generated when segment compaction was chosen by the hybrid segment recycling, or the checkpoint was recorded by the periodic checkpointing. However, the periodic checkpointing is indispensable for improving reclaiming efficiency by controlling the maximum number of pre-invalid blocks.

4.4 Copyback-aware Block Allocation

Figure 11 compares the performances under the copyback-*aware* block allocation (CAB) and the copyback-*unaware* block allocation (CUB) for the fileserver workload while varying the number of parallel flash chips in the SSD. Generally, the IO performance is improved as the number of flash chips increases because of the increased IO parallelism. In our experiments, the number of flash chips determined the zone size, and the file system segment size was configured to be equal to the zone size. Therefore, as the number of flash chips increased, the segment size also increased.

Figure 11(a) presents the bandwidth of each technique. When there are only a few parallel flash chips, the performance difference between ZNS and our techniques is insignificant because the maximum internal bandwidth of SSD is extremely low, causing ZNS to fully utilize the parallel flash chips using the host-level copy. In contrast, as the number of flash chips increases, the proposed ZNS+ techniques significantly outperform ZNS.

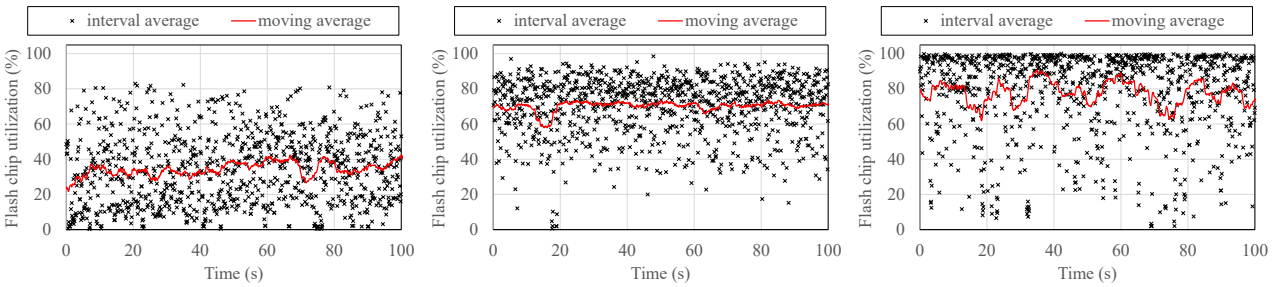
As a larger segment is used, it takes a longer time to reclaim a segment because it will have more valid blocks. The cost of checkpoint operations also increases with a large segment configuration. Therefore, ZNS and IZC present slow increase rates in bandwidth as the chip-level parallelism increases; in contrast, ZNS+ shows a faster increase rate in bandwidth because the increased block copy operations for large segments can be performed in the background by the PPA-ordered plugging. In addition, because threaded logging invokes fewer metadata updates compared to segment compaction, the checkpoint overhead does not increase significantly when the segment size is large. Consequently, the performance gap between IZC and ZNS+ increases as the number of flash chips increases.

Figure 11(b) presents the distribution of two different SSD-internal copy operations — copyback (cpbk) and read-and-program (R/P) — used to copy valid blocks during segment recycling. As the number of flash chips increases, the ratio of cpbk decreases linearly under CUB, because chunks are distributed into a larger number of chips. However, CAB causes more than 80% of the copy requests to be processed by copyback operations. Therefore, the performance of IZC-CAB is about 1.13 times better than that of IZC-CUB when the number of flash chips is 32. In the experiments, the copyback operation was configured to reduce the latency of copy operation by about 10% compared to the read-and-program operation, as shown in Table 2. Thus, a 13% performance improvement by CAB is reasonable.

As shown in Figure 11(b), the copyback ratio of ZNS+ is high despite CAB being disabled, and the number of flash chips is significant. This is because ZNS+ processes approximately 95% of the reclaimed segments with threaded logging. The fully valid chunks to be copied in the internal plugging can be copied using flash copyback operations, as shown in Figure 4. Therefore, the performance difference between ZNS+-CUB and ZNS+-CAB is insignificant.

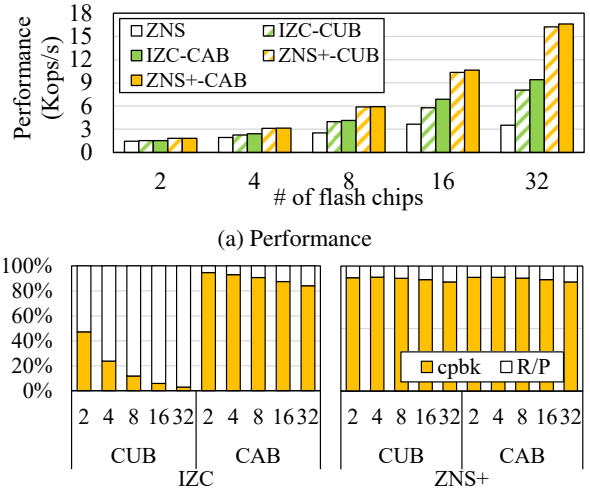
4.5 Performance at High H/W Parallelism

Figure 12(a) shows the performance change while varying the parallelism of host-to-device PCIe communication and flash chips. The PCIe communication parallelism was adjusted by changing the number of lanes, each of which was assumed to provide 600 MB/s of bandwidth. As the IO bandwidth increased, the number of flash chips was also configured to a larger value, because the total flash chip bandwidth must be high enough to utilize the increased IO bandwidth. The zone interleaving degree was configured to the total number of flash chips. Therefore, the zone size and the segment size increased as the number of flash chips increased. Although the data



(a) ZNS (b) IZC (c) ZNS+

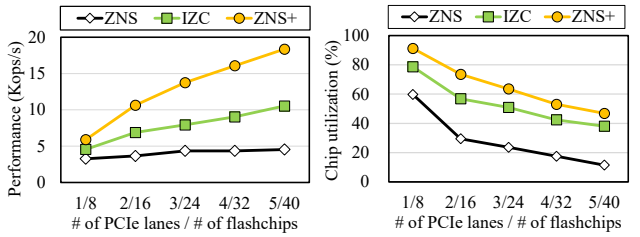
Figure 10: Flash chip utilization for fileserver workload



(a) Performance

(b) Internal copy operation breakdown

Figure 11: Performance comparison for varying chip-level parallelisms (fileserver workload)



(a) Performance (b) Chip Utilization

Figure 12: Performance at different communication and flash chip parallelisms (fileserver workload)

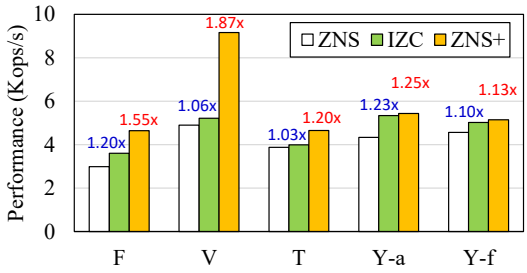


Figure 13: Performance result at a real SSD device.

transfer time between the host and the SSD can be reduced at a higher PCIe bandwidth, the performance of ZNS shows little improvements by the increased parallelism. This is due to the low chip utilization at a high chip parallelism, as shown in Figure 12(b), which is caused by idle intervals in SSD during segment compaction. Consequently, the performance gain by IZC or ZNS+ increases as the H/W parallelism increases.

4.6 Real SSD Performance

We also implemented a prototype of the ZNS+ SSD by modifying the firmware of the OpenSSD Cosmos+ platform [29] to evaluate the effects of the proposed techniques on a real system. The prototype ZNS+ SSD has two limitations compared to the SSD implemented in the FEMU emulator. First, the flash memory controller on Cosmos+ OpenSSD does not support the flash memory copyback operation; therefore, the ZNS+ SSD firmware cannot utilize it. Second, the flash memory controller does not support the partial page read operation; the entire 16 KB of the flash page must be read. Thus, SSD must use several *memcopy* operations to copy partially valid logical chunks, causing significant performance degradation, which can be easily fixed if a new flash controller is designed to support the internal copy. However, because we cannot modify the flash controller of the Cosmos+ platform, the copy requests of only fully valid logical chunks were offloaded to the SSD, and the copyback operations were replaced by the read-and-program operations in the experiments. To implement ZNS+ SSD, the number of firmware code lines was increased only by 6.3% (from 17,242 LoC to 18,334 LoC) compared to the ZNS implementation.

Figure 13 shows the performance improvement achieved by ZNS+ over ZNS for our prototype ZNS+ SSD. Owing to the limitations of Cosmos+ OpenSSD, the performance improvement in the real SSD is less significant compared to the emulation-based experiments. Nevertheless, ZNS+ improves the performance by about 13–87% compared to ZNS by minimizing IO request handling overhead and increasing flash chip utilization. The performance improvements by IZC are about 3–23%. If the flash controller is upgraded considering the internal copy operation, we will achieve higher performance improvements.

5 Related Work

Currently, SSD-based storage systems use a black-box model, where the host has no knowledge of the internal structure and management of the SSD. Such a black box model can decouple the host system and the SSD device while enabling them to communicate with each other via a simple IO interface; however, such a design causes several problems. First, duplicate logging operations and GCs can be performed in both the host and the SSD; this is called the log-on-log [34] problem. Second, it is difficult for the host to predict the IO latency owing to the complex internal operations of SSDs.

To overcome the limitations of the black box model, several studies proposed white- or grey-box models for the SSD, which expose essential knobs to control data placement, IO predictability, and IO isolation. AMF [21] proposed a new block interface that supports append-only IO via read/write/trim commands and an LFS for the interface. It solved the log-on-log problem by a direct mapping between the file system's segment and SSD's parallel flash erase blocks. Then, the SSD-internal GC becomes unnecessary because the segment is written only via the append-only scheme. It can also reduce the SSD-internal resource by using coarse-grained mapping instead of 4KB-level fine-grained mapping.

The open-channel SSD (OC-SSD) [10] exposes its hardware geometry to the host. Therefore, the host can manage flash page allocation and logical-to-physical mapping. Because the host initiates GC to reclaim invalid flash pages, the IO latency can be controlled by the host. Recently, the OC-SSD 2.0 specification [7] defined the `vector chunk copy` command, which is an interface for copying data inside the SSD. Our `zone_compaction` command is similar to the vector chunk copy command of OC-SSD. However, we are targeting the ZNS SSD, which maintains a zone-level address mapping in the device, and we propose a new block allocation technique to utilize the copyback operation of flash memory.

Multi-streamed SSDs (MS-SSDs) [17] can be considered to use a grey-box model. The host can specify the stream ID of each write request, and the MS-SSD guarantees that different streams of data are written into different flash erase blocks. If the stream ID of each write request is assigned based on the lifetime of its data, each flash erase block will have data with similar lifetimes, and thus, the GC cost can be reduced. Whereas the MS-SSD is based on the legacy SSD managed by a fine-grained address mapping, our ZNS+ SSD uses a coarse-grained mapping, and it is GC-less owing to the sequential write-only constraint of the ZNS. Instead, our ZNS+-aware LFS writes different lifetimes of data at different segments to reduce the host-level segment compaction cost.

The ZNS is an industry standardization for the open-channel interface. The ZNS interface is beneficial for flash memory-based SSDs or shingled magnetic recording drives [14, 32] owing to the sequential write-only constraint of the storage media. Compared to the open-channel interface, the ZNS provides a higher level of abstraction. Instead of directly managing the physical flash chips of an SSD, the host accesses the sequential writable zones and uses special commands to change the write pointer and the state of each zone. F2FS [19] and btrfs [27] have been patched to support zoned block devices in Linux kernel 4.10 and 4.14, respectively. In the patched F2FS [3], the file system's segment size is set to be equal to the zone size. It also disables the in-place-update and threaded logging features that can cause non-sequential writes. The patched btrfs [2] modifies the block allocation algorithm and creates a new IO path to ensure sequential access within the zone. In ZoneFS [1], each zone is shown as an append-only file. Thus, user applications can access the zoned block device via a file interface. These ZNS-aware file systems will require zone compaction operations in any form. However, they do not have any optimization techniques to reduce the host-level zone compaction overhead.

One of the recent popular research issues is in-storage computing. By offloading the host-side operations to the SSD, we can reduce the computing load of the host system. When a *reduce* operation, such as filtering and counting, is offloaded, the data traffic between the host and the storage can be reduced significantly [13, 16]. Recently, the SSD-internal bandwidth has exceeded the host IO interface bandwidth as more flash chips are embedded for a large SSD capacity. Therefore, data traffic reduction by in-storage computing is highly beneficial for recent SSDs. Our ZNS+ is also a solution that can reduce data traffic of large-capacity SSDs.

6 Conclusion and Future Work

The current ZNS interface imposes a high storage reclaiming overhead on the host to simplify SSDs. To optimize the overall IO performance, it is important to place each storage management task in the most appropriate location and make the host and the SSD cooperate. To offload block copy operations to the SSD, we designed ZNS+, which supports in-storage zone compaction and sparse sequential overwrite. To utilize the new features of ZNS+, we also proposed ZNS+-aware file system techniques, i.e., the copyback-aware block allocation and the hybrid segment recycling. In future work, we plan to optimize various ZNS-aware file systems and applications to utilize the ZNS+. We will also study the in-storage copyback-aware block allocation and the partition-aware block allocation for multi-core SSDs to minimize the number of block copy operations between different partitions.

Acknowledgements

We thank our shepherd Haryadi S. Gunawi and the anonymous reviewers for their valuable feedback. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. IITP-2017-0-00914, Software Star Lab) and Samsung Electronics.

References

[1] Accessing zoned block devices with ZoneFS. https://lwn.net/Articles/794364/.

[2] Btrfs zoned block device support. https://lwn.net/ml/linux-btrfs/20180809180450.5091-1-naota@elisp.net/.

[3] F2FS-tools: zoned block device support. https://sourceforge.net/p/linux-f2fs/mailman/message/35456357/.

[4] Filebench. https://github.com/filebench/filebench/wiki.

[5] NVMe 1.4a - TP 4053 zoned namespaces. https://nvmexpress.org/wp-content/uploads/NVM-Express-1.4-Ratified-TPs-1.zip.

[6] ONFi: Open NAND Flash Interface. http://www.onfi.org/specifications.

[7] Open-Channel Solid State Drives Specification Revision 2.0. http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.

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

[9] Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/introduction/zns/.

[10] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 359–374, 2017.

[11] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, et al. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *2018 IEEE International Solid-State Circuits Conference (ISSCC'18)*, pages 338–340, 2018.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *1st ACM Symp. Cloud Computing*, pages 143–154, 2010.

[13] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *43rd International Symposium on Computer Architecture (ISCA'16)*, page 153–165, 2016.

[14] Weiping He and David H. C. Du. SMaRT: An Approach to Shingled Magnetic Recording Translation. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, page 121–133, 2017.

[15] Duwon Hong, Myungsuk Kim, Jisung Park, Myoungsoo Jung, and Jihong Kim. Improving SSD Performance Using Adaptive Restricted-Copyback Operations. In *IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA '19)*, 2019.

[16] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging In-Storage Computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, August 2016.

[17] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, 2014.

[18] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. DC-Store: Eliminating noisy neighbor containers using deterministic I/O performance and resource isolation. In *18th USENIX Conference on File and Storage Technology (FAST'20)*, page 183–191, 2020.

[19] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 273–286, 2015.

[20] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: a low-latency kernel I/O stack for ultra-low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 603–616, 2019.

[21] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 339–353, 2016.

[22] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 83–90, 2018.

[23] Yoohyuk Lim, Jaemin Lee, Cassiano Campes, and Euiseong Seo. Parity-stream separation and SLC/MLC convertible programming for life span and performance improvement of SSD RAIDs. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*, 2017.

[24] Hiroshi Maejima, Kazushige Kanda, Susumu Fujimura, Teruo Takagiwa, Susumu Ozawa, Jumpei Sato, Yoshihiko Shindo, Manabu Sato, Naoaki Kanagawa, Junji Musha, et al. A 512GB 3b/cell 3D flash memory on a 96-word-line-layer technology. In *2018 IEEE International Solid-State Circuits Conference (ISSCC'18)*, pages 336–338, 2018.

[25] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM SIGOPS Operating Systems Review*, 31(5):238–251, 1997.

[26] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *3rd Annual Haifa Experimental Systems Conference (SYSTOR'10)*, 2010.

[27] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.

[28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[29] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openSSD: A PCIe-based open source SSD platform. *Proc. Flash Memory Summit*, 2014.

[30] Wei Wang and Tao Xie. PCFTL: A Plane-Centric Flash Translation Layer Utilizing Copy-Back Operations. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3420–3432, 2015.

[31] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[32] F. Wu, Z. Fan, M. Yang, B. Zhang, X. Ge, and D. H. C. Du. Performance Evaluation of Host Aware Shingled Magnetic Recording (HA-SMR) Drives. *IEEE Transactions on Computers*, 66(11):1932–1945, 2017.

[33] Fei Wu, Jiaona Zhou, Shunzhuo Wang, Yajuan Du, Chengmo Yang, and Changsheng Xie. FastGC: Accelerate garbage collection via an efficient copyback-based data migration in SSDs. In *55th Annual Design Automation Conference (DAC'18)*, 2018.

[34] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*, 2014.

[35] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technology (FAST'20)*, page 121–136, 2020.

[36] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-Indirection for Flash-Based SSDs with Nameless Writes. In *10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.