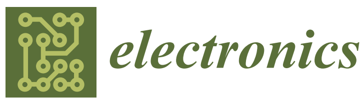

Article

Preemptive Zone Reset Design within Zoned Namespace SSD Firmware

Siu Jung, Seungjin Lee, Jungwook Han and Youngjae Kim *

Department of Computer Science and Engineering, Sogang University, 35 Baekbeom-ro, Mapo-gu, Seoul 04107, Republic of Korea
* Correspondence: youkim@sogang.ac.kr; Tel.: +82-2-705-8933

Abstract: Zoned Namespace (ZNS) SSDs address the disadvantages that come from supporting the block interface within conventional SSDs, granting more control over data management to host systems, while also relieving heavy duties from device firmware. However, with the removal of on-device garbage collection, host systems must explicitly send zone reset requests to free up storage space, which may incur multiple NAND block erase operations according to the configured zone size, resulting in increased tail latency. In this article, we propose a Preemptive Zone Reset scheduling design, which we implemented within the firmware of our ZNS SSD prototype, and compare it to an intuitive Zone Mapping Table method, which we consider as the state-of-the-art. The main idea is to service high priority foreground I/O requests while preempting block erase operations induced by zone resets. Our proposed approach, opposed to the baseline method, as much as halved tail latency for write-only workloads, and reduced read tail latency by up to 1.76 times in a mixed workload.

Keywords: zoned namespace SSD; zone reset; preemptive scheduling

Citation: Jung, S.; Lee, S.; Han, J.; Kim, Y. Preemptive Zone Reset Design within Zoned Namespace SSD Firmware. *Electronics* **2023**, *12*, 798. https://doi.org/10.3390/electronics12040798

Academic Editor: Fabio Grandi

Received: 7 December 2022
Revised: 23 January 2023
Accepted: 3 February 2023
Published: 5 February 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Emerging Zoned Namespace (ZNS) SSD [1] technology provides a new opportunity for NAND flash-based SSDs to communicate with host machines through the zoned block device interface, alleviating issues that came with supporting the conventional block device interface. Traditionally, SSDs were forced to execute intricate firmware, called Flash Translation Layer (FTL), in order to conform to the prevalent block layer abstraction, which inherently conflicts with flash media characteristics [2,3]. The main concern arises due to read/write granularity being physical NAND pages, while expensive erase operations must be performed in larger physical NAND block units that span multiple contiguous physical pages. As pages must be in the erased state to be programmed (written) and direct overwrites are physically impossible, this compelled the FTL to perform out-of-place updates involving DRAM-heavy operations such as dynamic logical-to-physical mapping table management and garbage collection (GC), the latter requiring additional reserved media capacity (overprovisioning) of at least 5% of the total storage space. These operations, most notably GC, generate significant overhead, resulting in write amplification, limited throughput, and overall unpredictable performance, further amplified when the SSD storage utilization approaches its maximum capacity [4–6].

Considering these disadvantages, the recent NVMe Zoned Namespace Command Set Specification [7] proposes an alternative interface standard. Commonly referred to as ZNS, this new standard groups consecutive logical blocks into zones, which reflects the physical media boundaries and sequential page programming characteristic of erase blocks. Accordingly, zones comply to the sequential write constraint and must be erased before rewrites. By exposing a more accurate representation of flash media [8], host software obtains more freedom in data placement and management, and allows ZNS SSDs to relinquish responsibilities, originally fulfilled by the FTL, to the host. With the removal of device-level page-granularity GC, internal DRAM usage can be reduced substantially

and media overprovisioning is no longer necessary, extending the NAND-to-DRAM ratio in SSDs.

Nevertheless, adopting the ZNS interface is still an on-going process and requires diverse research to be accomplished from both device firmware and host software perspectives [4,9–13]. These works mostly propose GC schemes when utilizing ZNS SSDs, mainly focusing on host-side implementations. However, we argue and prove that firmware optimizations within the actual device must also be addressed when it comes to I/O response time. In particular, host-induced explicit zone reset calls are still unavoidable in order to erase obsolete data, and may impact performance considerably if not designed carefully within device firmware. Although ZNS hands over the task of selecting which blocks to erase, the tax of scheduling the actual block erase operations remains. Since zone sizes are generally configured as a multiple of the erase block size, numerous time-consuming block erases must be performed for a single zone reset, potentially impeding foreground I/O requests and resulting in increased tail latencies [14]. To the best of our knowledge, firmware-side zone reset scheduling schemes are yet to be discussed in the literature.

In this article, we propose a Preemptive Zone Reset design, which minimizes foreground I/O disruption. The core concept involves identifying preemptible points in-between block erase operations preformed for zone resets, and giving foreground I/O higher scheduling priority compared to these block erases. We compared our proposed design to an intuitive zone reset approach that manages a zone mapping table, which we considered a baseline. We implemented our design within the firmware of our ZNS SSD prototype, utilizing the Cosmos+ OpenSSD [15] platform. Evaluations show that Preemptive Zone Reset, in its best cases, reduced tail latencies up to twofold for a write-only pattern, and 1.76 times for reads in a mixed workload.

Our main contributions and research flow are as follows:

- As yet, ZNS SSDs are not available for public use. Furthermore, detailed design decisions for the NVMe controller, firmware, flash controller, and physical layout of actual ZNS SSD prototypes have not been released by manufacturers. Thus, in order to conduct our firmware research, we implemented our own ZNS SSD prototype by modifying the Cosmos+ OpenSSD platform;
- Zone reset handling in ZNS SSD firmware rests unexplored in academia and stays hidden by manufacturers. We show that block erases needed for a zone reset request introduce significant overhead as zone sizes increase;
- We first implemented an intuitive zone reset design which manages a zone mapping table, and point out its limitations. Then, we present our proposed Preemptive Zone Reset scheme, which gives foreground I/O higher priority and performs Partial Zone Erases accordingly. We evaluated and compared both designs on our ZNS SSD prototype.

2. Background & Problem Definition

This section provides background knowledge on the NVMe ZNS specification and presents the motivation behind our study.

2.1. NVMe Zoned Namespace Standard

The NVMe Zoned Namespace Command Set Specification [7] defines a new standard of NVMe SSDs for zoned block device support. NVMe zones are each composed of a predetermined number of contiguous Logical Block Addresses (LBAs), typically 4KB in size, which can only be written sequentially. The Write Pointer (WP) of each zone keeps track of the next LBA where data must be appended. Initially indicating the Zone Start LBA (ZSLBA) of its zone, the WP increments with every LBA written, and is relocated to the ZSLBA once the zone is reset. Reads may be performed in random order beneath the WP, but result in undefined behavior beyond it. The specification defines a Zone State Machine to manage Open and Active Resources of zones. Zones may be in the Empty, Full, Closed, or Opened state. The following three extra commands are added for NVMe ZNS.

1. The Zone Append command appends data to the zone matching the designated ZSLBA and returns the lowest LBA of the set of logical blocks written;
2. The Zone Management Receive command returns to the host a data buffer containing information about zones, such as the zone state, zone capacity, WP, etc.;
3. The Zone Management Send command can be used to request an action on one or more zones, including close zone, open zone, reset zone, etc.

Overall, the ZNS interface shifts duties originally attended by the FTL, such as GC and mapping table management, in the forms of explicit zone reset and append-only restraint, granting more privilege regarding data management to the host. As mentioned, the host may send a zone reset request to a specific zone or all zones, through the Zone Management Send command. Zones that are reset transition to the Empty state, and their WPs are relocated to their ZSLBA. Apart from these demands, the NVMe ZNS specification does not enforce implementation details on how zone reset should be handled in the NAND flash controller level, leaving firmware developers the opportunity to be creative. This entails that zone reset performance depends highly on its firmware design. Hence, we argue that the unspecified implementation details of the state-of-the-art method and its limitations are worth researching.

2.2. Problem Definition

We present two simple and intuitive zone reset implementations in this subsection, one of which we consider as the up-to-date baseline. Then, we highlight the obstacles that appear in both methods by analyzing their disadvantages.

2.2.1. Synchronous Method

One straightforward approach is to perform block erases synchronously, that is, right as the host requests a zone reset. When the NVMe controller of the ZNS SSD receives a zone reset request and the firmware FIFO scheduler issues it to the flash controller, all physical blocks of the zone are erased, holding other I/O and flash requests during the operation. Since block erases happen synchronously, address translation from logical to physical pages could be omitted with static one-to-one direct calculation. This eliminates the need for dynamic mapping table management, and thus reduces DRAM utilization.

2.2.2. Logical-to-Physical Zone Mapping Method

Another way to perform zone reset would be to map the logical zone to a new, free physical zone as depicted in Figure 1. Just as the FTL in conventional SSDs keeps a mapping table for logical block address to physical page address translation, this method manages a mapping table in the ZNS firmware. Logical zones conform to the NVMe ZNS specification and are perceived by the host and NVMe controller, while physical zones are an abstraction compatible with the physical layout viewed by the flash controller. When the NVMe controller receives a zone reset request, firmware invalidates the physical zone that was initially allocated to the logical zone, and disconnects the link in the mapping table. Then, the logical zone switches to the Empty State, and its write pointer is reinitialized to the ZSLBA. If a request writes to an empty logical zone, firmware assigns a free physical zone to that logical zone, and updates the mapping table to reflect the connection. We continue this type of free physical zone allocation until there are no more free zones, at which point the flash controller would immediately perform erase requests on one or more invalid physical zones in order to free up storage space. The zone mapping table size would vary according to the zone and device storage size, though we can expect it to be smaller than the page or block mapping tables in conventional SSDs if a zone is comprised of several erase blocks. This method was presented in ZNS+ [11], and we consider it the state-of-the-art design for zone reset handling in ZNS SSD firmware.

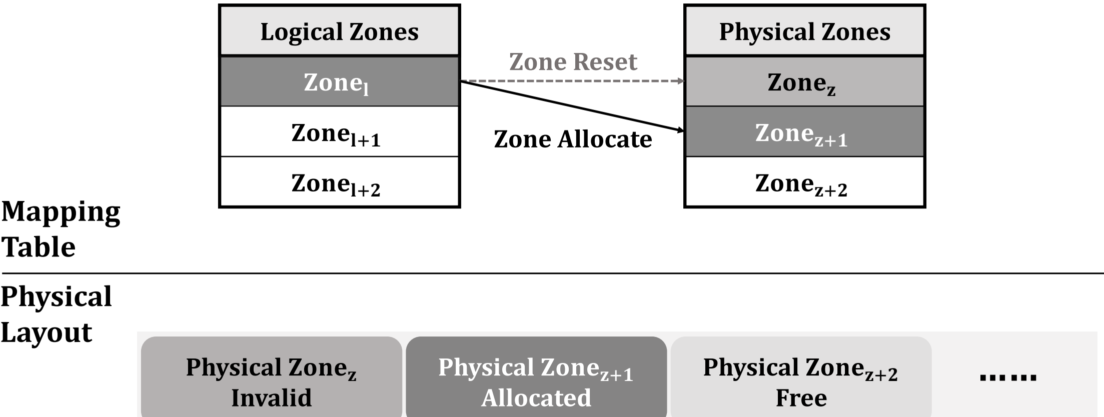

Figure 1. Logical-to-Physical Zone Mapping Method.

2.2.3. Disadvantages and Issues

For the synchronous method, the disadvantage is rather intuitive: all incoming I/O and flash requests are halted right after each zone reset request. This results in high, albeit predictable, latency, since block erases are the most expensive NAND flash operation in terms of execution time, and we erase numerous blocks at once.

The mapping method complements this issue by detaching zone reset requests from its resulting block erase operations, and servicing foreground I/Os promptly when there are free zones to allocate. However, this only defers the inevitable block erases, ultimately converging back to the foreground I/O blocking issue of the synchronous method, when invalid zones must be erased to further handle incoming write requests. That is, although the host is responsible for choosing which zone to reset, the device still bears the burden of actually freeing up invalidated zones. Hence, we expect to experience high latency either when zone resets are called excessively, or when the ZNS SSD approaches its maximum capacity.

Moreover, the number of block erase operations that must be performed to reset a zone is proportionate to the configured zone size. Therefore, we anticipate that each blocking period will last longer with larger zone sizes. Figure 2 shows the average execution time of the necessary block erases for zone resets by zone size, measured inside our ZNS SSD prototype. Detailed characteristics, such as the physical layout and implementation, of the Cosmos+ OpenSSD and our prototype is covered in Section 3.1.

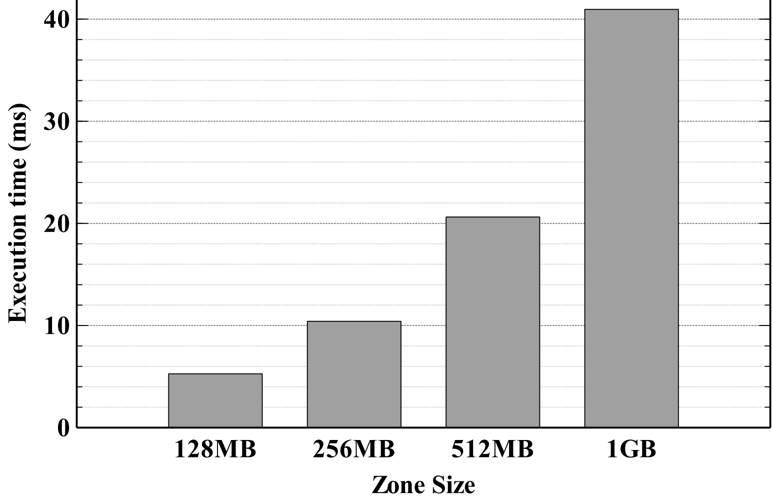

Figure 2. Average block erase execution time by zone size.

We flushed all requests in the NAND request queue before and after measuring in order to segregate the execution time of the performed block erase operations. We averaged 100 executions for zone sizes of 128 MB, 256 MB, 512 MB, and 1 GB. We observe that when zone size is 128 MB, i.e., when a zone is comprised of a single erase block, the execution time is similar to a block erase operation of the original Cosmos+ OpenSSD, which takes 5 ms. Additionally, we see that execution time increases linearly with zone sizes. This experiment demonstrates that performing the required block erases all at once for each zone reset introduces significant latency with larger zone sizes.

Performing block erases only when necessary is an undeniable advantage of the zone mapping method. We pursue our research based on this feature, while seeking a technique to lessen the foreground I/O blocking situation as much as possible. As a solution, we propose a Preemptive Zone Reset design.

3. ZNS SSD Prototype and Preemptive Zone Reset Design

Since ZNS SSDs and their internal specifications are unobtainable for the aforementioned reasons, we started off our implementation by prototyping our own ZNS SSD. Then, we introduced our proposed Preemptive Zone Reset design as an approach to overcome the problems defined in the previous section.

3.1. ZNS SSD Prototype

In this subsection, we provide information about the Cosmos+ OpenSSD platform, and how we modified it into a ZNS SSD.

3.1.1. Cosmos+ OpenSSD

The Cosmos+ OpenSSD [15] is an FPGA-based open-source SSD platform, which allows SSD controller and firmware modification, providing a flexible environment for hardware and software functionality development. Hardware details of the Cosmos+ OpenSSD are shown in Table 1.

Table 1. Hardware specification of the Cosmos+ OpenSSD platform.

Hardware	Specification
FPGA	Xilinx Zynq-7000
CPU	Dual-Core ARM Cortex
DRAM	DDR3 1 GB
Storage Capacity	256 GB
Host Interface	PCIe Gen2 8-lane

The four physical channels and eight physical ways constitute 32 flash dies in total. The FTL manages a page-level mapping table, where each page is 16KB in size, and consecutive pages are interleaved on each die. Each erase block consists of 256 physical pages spread across all 32 dies, which results in 4 MB per die, and 128 MB in total. Commands accessing different dies are executed out of order, guaranteeing way-level parallelism. DMA commands and 16 KB data buffers are used for data transfer between the host and NAND flash memory, and each buffer entry is evicted following the Least Recently Used (LRU) policy.

3.1.2. ZNS Prototype Implementation

The NVMe ZNS kernel driver is supported in Linux kernel versions 5.9 and above. We consulted the NVMe ZNS specification version 1.1a [7] when implementing our ZNS SSD prototype, and avoided any kernel modification. For our prototype, we removed internal GC, overprovisioning, and the page-level mapping table. Without GC, there is no need for free block management and victim selection. Instead, we added a new structure for zone management, keeping zone information such as WP location and zone state in DRAM, requiring 8B for each physical zone. As with zone resets, the NVMe ZNS Specification

does not impose a specific zone size requirement aside from logical block size alignment. Therefore, we left the zone size as a configurable option, so long as it remained a multiple of the erase block size, and the size of logical zones viewed by the host matched that of the physical zones viewed by the device. The physical layout of our prototype is shown in Figure 3.

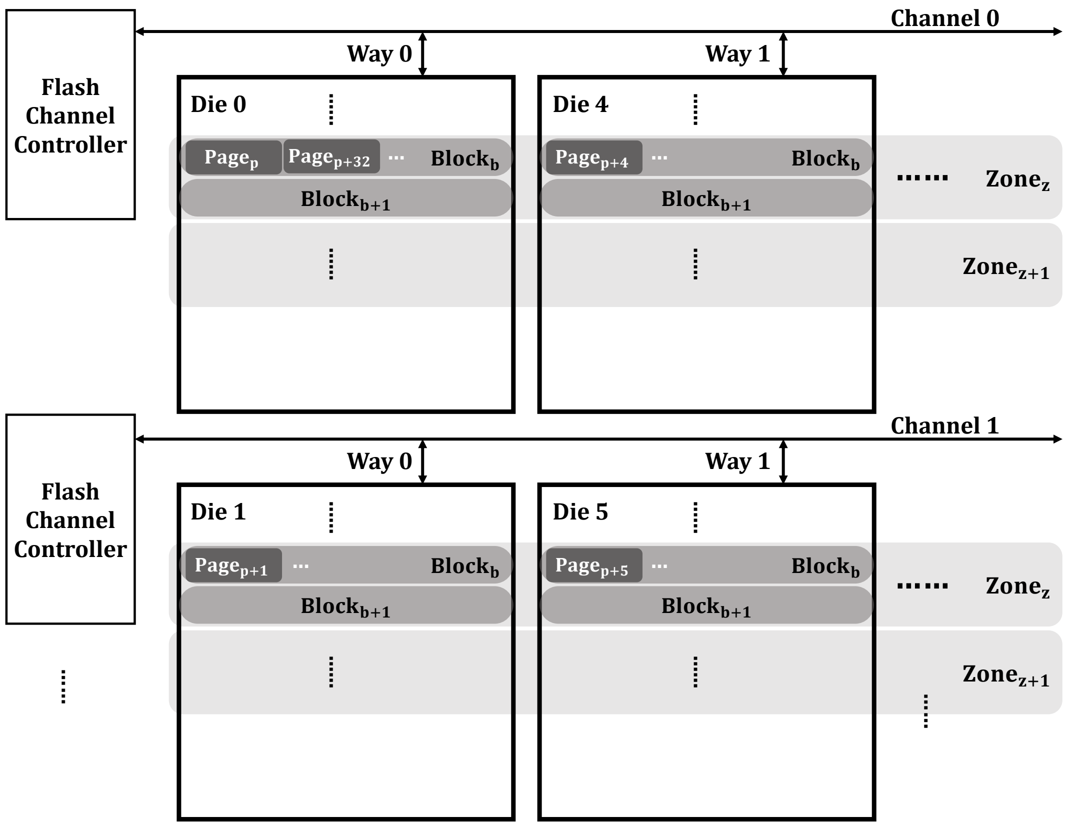

Figure 3. Physical layout of ZNS prototype. Zone size is configured as two erase blocks.

As previously stated, contiguous pages are striped across dies, and every 256 pages on the 32 dies are grouped together to form an erase block. For our physical layout, we chose to associate consecutive erase blocks into zones, retaining the address translation layer of the original Comsos+ OpenSSD. With this design, operations accessing sequential pages can be executed in parallel. Figure 3 shows an example where zone size is configured to 256 MB, each zone consisting of two erase blocks. The number of erase blocks that constitute a zone varies according to the configured zone size. Page offsets within a zone are retrieved by straightforward calculation. Major changes had to be made in firmware buffer management to satisfy the sequential write rule of ZNS, generally concerning buffer eviction situations.

3.2. Preemptive Zone Reset

We present in this subsection the schemes involved in the proposed Preemptive Zone Reset design to tackle the challenges based on our aforestated investigations.

3.2.1. Partial Zone Erase

As mentioned above, we aimed to minimize the foreground I/O blocking phenomenon produced by the multiple block erases that stem conclusively from zone reset calls. Hence, we employed the logical-to-physical zone mapping table from above, and only performed updates to the mapping table when it was permitted, instead of expensive block erases.

However, deferring every block erase until they become the last resort results in lengthy and inevitable blocking periods. On the other hand, from the host system perspec-

tive, there is no noticeable performance degradation if firmware performs other operations when there are no pending I/O requests. In other words, we hoped to reduce I/O tail latency by performing block erases in small portions on invalid zones while there were no enqueued DMA or NAND flash requests. Based on this analysis, we proposed a Partial Zone Erase scheme.

Figure 4 shows an example of an invalid $Zone_z$ and active or blocked operations in the firmware scheduler of our Partial Zone Erase scheme. We began by acknowledging the fact that, as firmware developers, NAND operations cannot be stopped once they are issued and running. Thus, we identified that preemptible points exist in-between the block erase operations of an invalid zone. While I/O_1 was active, we blocked the next erase operation of $Zone_z$, in this case $Block_{b+1}$. Once I/O_1 was complete, we checked both DMA and flash request queues. As we checked that there were no foreground I/Os that needed servicing, we issued the erase $Block_{b+1}$ operation. During the erase operation, we received request I/O_{i+1}, which had to wait until $Block_{b+1}$ was erased. When erase $Block_{b+1}$ was complete, we preempted the zone erase procedure and handled I/O_{i+1} with higher priority. We continued this sequence of single block erases until $Zone_z$ was fully erased and free, at which point we considered the next invalid zone.

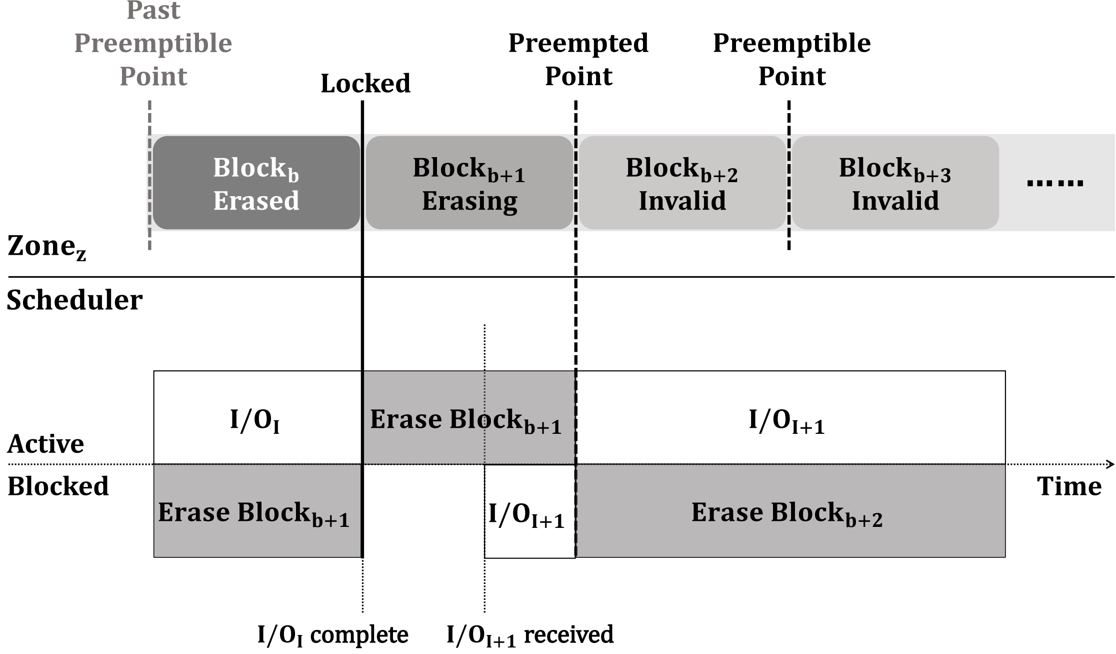

Figure 4. Partial Zone Erase scheme.

3.2.2. States and Thresholds

There might be cases, such as in bursty I/O situations, where performing Partial Zone Erases between I/O requests cannot provide enough free zones. Furthermore, performing Partial Zone Erases also introduces a minimal blocking period, which could be avoided when there are not many invalid zones to erase. To resolve this, we defined three states for our Preemptive Zone Reset design:

- State 0 (S_0): only services foreground I/O requests and does not perform Partial Zone Erases;
- State 1 (S_1): performs Partial Zone Erases only if there are no pending foreground I/Os to service;
- State 2 (S_2): blocks all foreground I/O requests and performs all necessary block erase operations in order to provide a free zone, namely a Full Zone Erase on an invalid zone. In this state, securing a free zone that can be used for allocation is the highest priority.

We defined two thresholds, $T_{invalid}$ and T_{free}, accounting for the number of invalid zones ($Z_{invalid}$) and free zones (Z_{free}) respectively, to transition between states. Note that the total number of zones (Z_{total}) in the ZNS SSD is equal to the sum of $Z_{invalid}$, Z_{free}, and the number of zones that are currently allocated and in use. That is:

$$Z_{total} = Z_{alloc} + Z_{invalid} + Z_{free}, \tag{1}$$

where Z_{alloc} is the number of zones currently in use. Initially, when the device is brand new, Z_{free} would be equal to Z_{total}, whereas $Z_{invalid}$ and Z_{alloc} would both be 0. For each zone allocation while processing I/Os, Z_{free} would decrease by 1, when Z_{alloc} would increment. Contrarily, Z_{alloc} decreases as $Z_{invalid}$ increases for every zone reset request sent by the host system. Finally, when an invalid zone is fully erased and becomes free, $Z_{invalid}$ decreases and Z_{free} increases.

State transitioning is depicted in Figure 5. Originally, the ZNS SSD is in S_0, where $Z_{invalid}$ is 0, and Z_{free} is equal to Z_{total}. If $Z_{invalid}$ becomes greater than or equal to $T_{invalid}$ due to subsequent zone reset calls, the device shifts from S_0 to S_1. By performing Partial Zone Erases, if $Z_{invalid}$ becomes smaller than $T_{invalid}$, the state goes back to S_0 from S_1. While in S_0 or S_1, the ZNS SSD transitions to S_2 when Z_{free} becomes smaller than or equal to T_{free}. When Z_{free} exceeds T_{free} from Full Zone Erases, the device transitions from S_2 to either S_0 or S_1. If $Z_{invalid}$ is still greater than or equal to $T_{invalid}$ it changes to S_1. Otherwise, it goes back to S_0. With these three states and two thresholds, we hoped to disperse the I/O blocking period caused by block erase operations from zone reset requests.

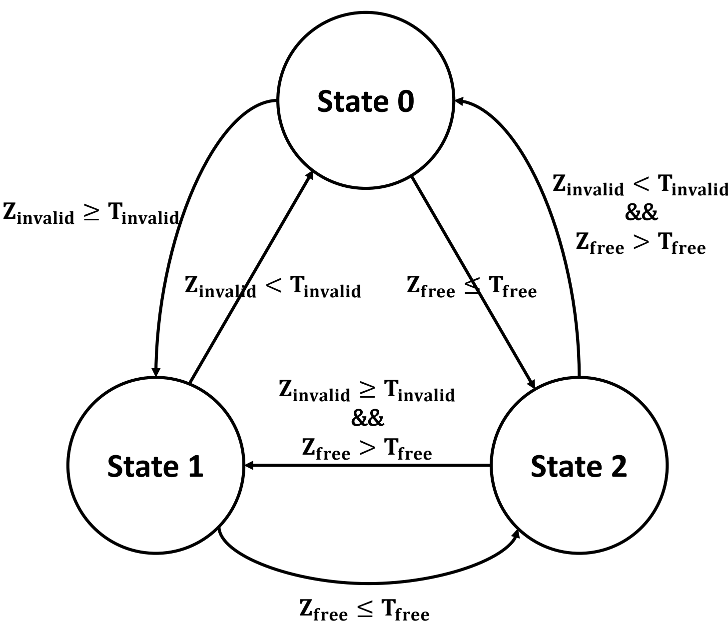

Figure 5. State diagram of Preemptive Zone Reset.

3.2.3. Implementation

The original Cosmos+ OpenSSD already keeps count of the number of pending or running DMA and flash requests in each queue. Therefore, while in S_1 in our prototype, we consulted this number instead of the actual requests in the queues, and the associated overhead remained constant-time. To be more specific, our implementation only needs to consult two variables to check both queues, resulting in negligible overhead regardless of the number of pending requests. In addition, most of our implementation only involves constant variables, only changing according to zone sizes for Full Zone Erases. In order to keep track of invalid and free zones, we managed an invalid zone queue and a free zone queue. Within the next invalid zone in the invalid zone queue, we tracked the next block to be erased for the Partial Zone Erase scheme. When an invalid zone became fully

erased, it was popped from the invalid zone queue and pushed in the free zone queue to be used for allocation in the future. The zone mapping table and invalid/free zone queues in our implementation require 16B of DRAM for each pair of logical/physical zones. In the original Cosmos+ OpenSSD, the page-level mapping table occupied 8B for each logical/physical page pairs, amounting to 128 MB of DRAM for 256 GB of storage. By comparison, our implementation takes up to 4 KB for 256 zones of 1 GB in size, and a maximum of 32 KB for 2048 zones of 128 MB, which is significantly smaller. In addition, $T_{invalid}$ and T_{free} were left as easily configurable options.

Write Pointer Optimization: To reduce the number of block erase operations involved, it is natural to only erase blocks that actually require being freed, which means those that are programmed. Thus, for further optimization, both Partial and Full Zone Erases of our Preemptive Zone Reset scheme exclusively performed erases up to the written block, that is, up to the WP of the invalid zones. This way, we avoided performing erase operations on blocks that were already free and programmable in situations where host-side software data placement policies send untimely reset requests to partially written zones. We acknowledge that this optimization introduces uneven wear-leveling. However, the issue is out of scope for now, and we leave a wear-leveling-aware ZNS SSD design for future work.

4. Evaluation

We compared the performance of the state-of-the-art zone mapping method with our Preemptive Zone Reset design. The host system in our experiment setup consisted of Linux Kernel v5.18.0, Intel(R) Core i7-10700 CPU @ 2.90 GHz with 16 cores and 16 GB memory. The target storage device was our ZNS SSD prototype, where the hardware specification was identical to that of the Cosmos+ OpenSSD platform in Table 1. We evaluated raw device and application performances by testing two distinct benchmarks.

4.1. fio Benchmark

We first compared the two zone reset methods with fio [16], a widely used benchmarking tool for storage devices. The workload synchronously wrote 64 GB of data, each write request sending 2 MB, on the logical zones within the first 16 GB of the 256 GB ZNS SSD prototype. That is, the first logical zones, 16 for 1 GB zone size and 32 for 512 MB zone size, were each reset thrice during the whole workload. Our plan was to simulate a 16 GB ZNS SSD and generate as many zone resets, and consequently block erases, as possible. Accordingly, we set the T_{free} threshold, which triggers the Full Zone Erases for both zone mapping and Preemptive Zone Reset methods, and the $T_{invalid}$ threshold for Partial Zone Erases as shown in Table 2. Note that $T_{invalid}$ was set to activate Partial Zone Erases when there was at least one invalid zone, and that there was no host-side I/O idle time in the configured benchmark. Moreover, the number of block erase operations involved in the workload was the same for the two methods. Overall, these settings were intentionally strict and unfavorable conditions for our Preemptive Zone Reset design. Figure 6 shows the results for sequential writes.

Table 2. Configured free zone and invalid zone thresholds according to zone size.

Zone Size	$T_{invalid}$	T_{free}
512 MB	1	479
1 GB	1	239

The x-axis represents latencies in ms, while the y-axis shows latency percentages, and the Mapping and Preemptive methods are displayed in red and blue, respectively. Graphs on the left show latencies from P50, while those on the right show a more focused view on tail latencies. These same legends are applied for all latency graphs that follow. As observed in the graph, both mapping and Preemptive Zone Reset methods reveal the same latencies of approximately 6 ms from 50 to 99 percent for both 512 MB and 1 GB zone sizes.

Both methods reveal latency spikes starting from P99.9 when block erases are performed. Nonetheless, the tail latency increase is much more accentuated for the mapping method, reaching up to almost 50 ms and 30 ms, for 1 GB and 512 MB zone sizes respectively. By contrast, Preemptive Zone Reset's tail latency settles at about 23 ms for both 1 GB and 512 MB zone sizes. This implies that, while the mapping method performs Full Zone Erases resulting in higher tail latency for larger zone sizes, our Preemptive Zone Reset scheme successfully disperses the block erase operations, and thus shortens each eventual blocking period even in bursty I/O situations.

As writes in ZNS were performed at the WP for each zone, the random write workload selected random zones instead of random LBAs. For this reason, the results for random writes in Figure 7 display similar patterns to sequential writes, with only a slight increase in tail latency values. We skipped analysis for throughput as both methods did not show noticeable differences in bandwidth. Even so, it is worth mentioning that this indicates the overhead for checking the I/O queues in Preemptive Zone Reset is rather imperceptible. Additionally, the WP optimization is unable to reduce the number of block erase operations due to the deliberately harsh workload characteristics explained above.

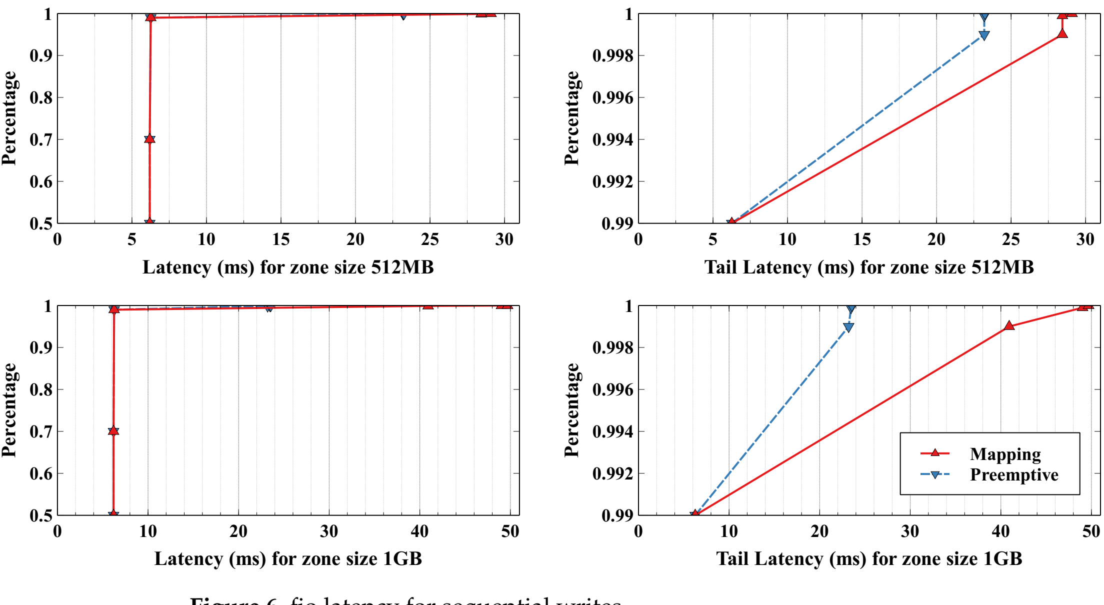

Figure 6. fio latency for sequential writes.

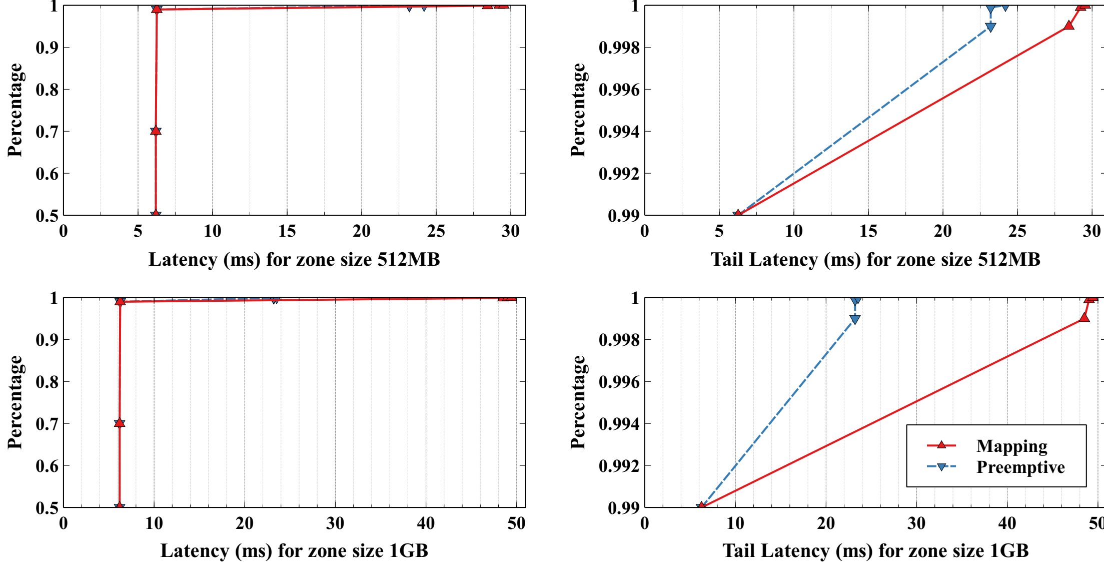

Figure 7. fio latency for random writes.

4.2. RocksDB Benchmark

For the second benchmark, we ran db_bench [17], a benchmarking tool for RocksDB [18], a popular LSM-tree [19] based key-value store, on top of its integrated user space file system implemented for ZNS SSDs, ZenFS [4,20,21]. Again, to intentionally trigger as many zone resets, the thresholds were left unchanged from Table 2. In this software stack, ZenFS carried out on-disk data placement and the management duties of key-value pairs. Hence, unlike the fio workload, ZenFS allocated zones according to its lifetime-based zone allocation algorithm and sent zone reset requests even to zones that were not fully written. The benchmark first filled the database with 50,000,000 random key-value pairs, then performed 50,000,000 overwrites on random entries. The key size was fixed at 48B, and we experimented for value sizes of 80B and 464B. However, we only show results for the fillrandom workload with 464B values, seen in Figure 8, to avoid duplicate analyses. The estimated raw database size was about 24 GB.

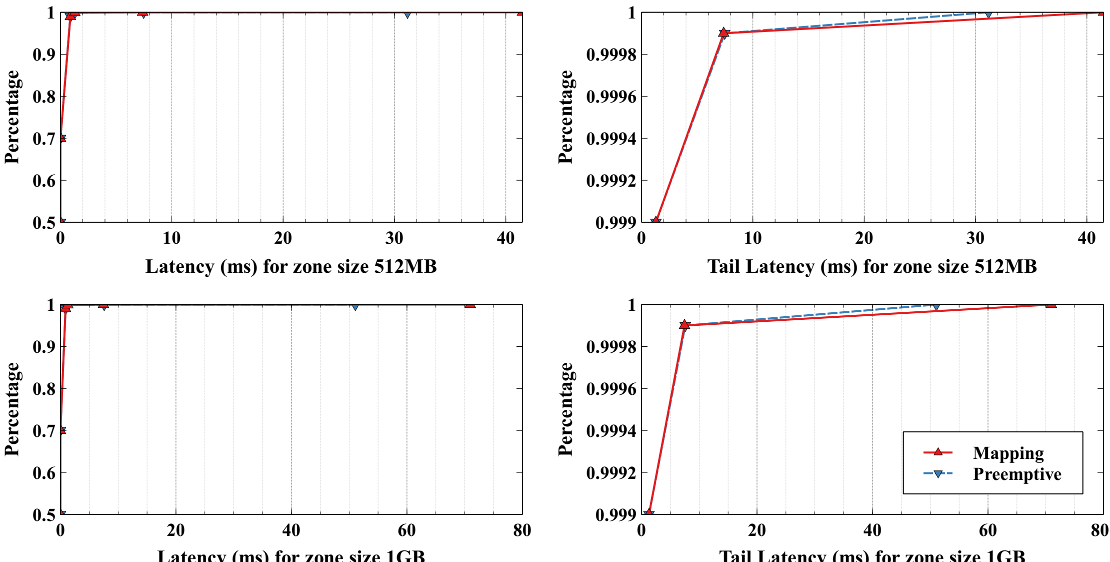

Figure 8. RocksDB write latency of fillrandom with 464B values.

The latencies are almost identical for both zone reset methods and zone sizes up until P99.99. Yet, we notice that tail latency increases significantly after that point, where the gap in response time between mapping and preemptive becomes noteworthy. P100 reaches up to approximately 41.5 ms for mapping, while preemptive arrives at about 31 ms for 512 MB zone sizes. When zone sizes are 1 GB, preemptive attains 51 ms for P100 latency, whereas mapping took nearly 71 ms. The other configurations of db_bench, 80B value size and overwrite, demonstrated similar shapes, though the tail latency of mapping more than doubled compared to preemptive in the overwrite workload.

Next, we analyzed the effects of our WP optimization. Figure 9 visualizes the total number of block erase operations performed during the fillrandom workload for 464B value sizes. The mapping method performs erase operations for all blocks in a zone that is to be reset, erasing over 2000 and 4000 blocks for 512 MB and 1 GB zone sizes, respectively. On the other hand, the WP optimization in Preemptive Zone Reset does not erase unwritten blocks. This largely reduces the number of erase operations required for a zone reset, fixed at slightly under 900 blocks for both zone sizes. The graph also proves that ZenFS does not fully utilize a zone, leading to premature reset requests during its data management strategy, especially for larger zone sizes. Efficient data placement from host-side software targeting ZNS SSDs to diminish zone resets and write amplification, such as [12,13] which targeted LSM-tree based key-value stores, may be worth further research in the future. Though we did notice a modest improvement in bandwidth owing to the reduced number

of block erases, the gain is rather trivial. Therefore, the analysis for the throughput of db_bench is excluded for the same reasons as in fio.

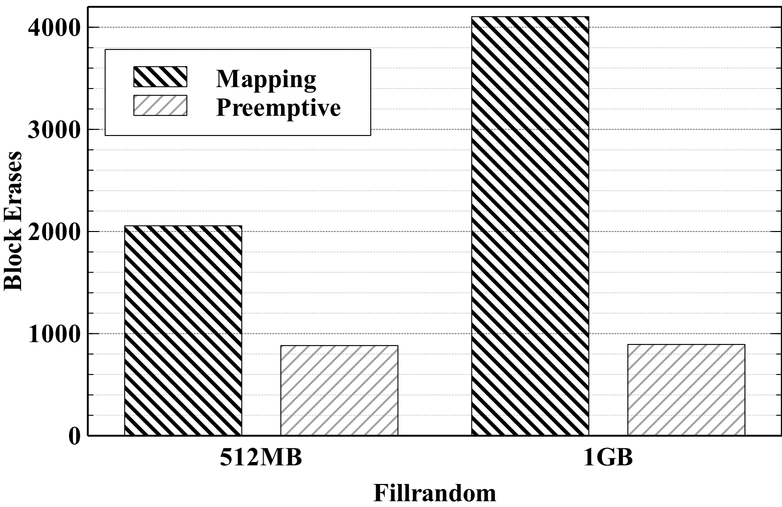

Figure 9. Total number of block erases performed during fillrandom with 464B values.

Lastly, to evaluate read latency, we ran a mixgraph benchmark after consulting [22]. The settings were tweaked to a write-dominant ratio in order to better observe the impact of zone resets on reads. The workload sent 50,000,000 queries, nine puts for each get, to a database of 50,000,000 random entries with 48B keys and 464B values. The results in Figure 10 display similar patterns to the fillrandom experiment, as the distinction in latencies up until P99.99 is insignificant for all zone reset and size configurations. As with fillrandom, the gap becomes distinguishable in P100 tail latency, where mapping took 84.6 ms when preemptive arrived at 48 ms for 512 MB zone sizes, and mapping 117 ms to preemptive 67 ms for 1 GB zone sizes.

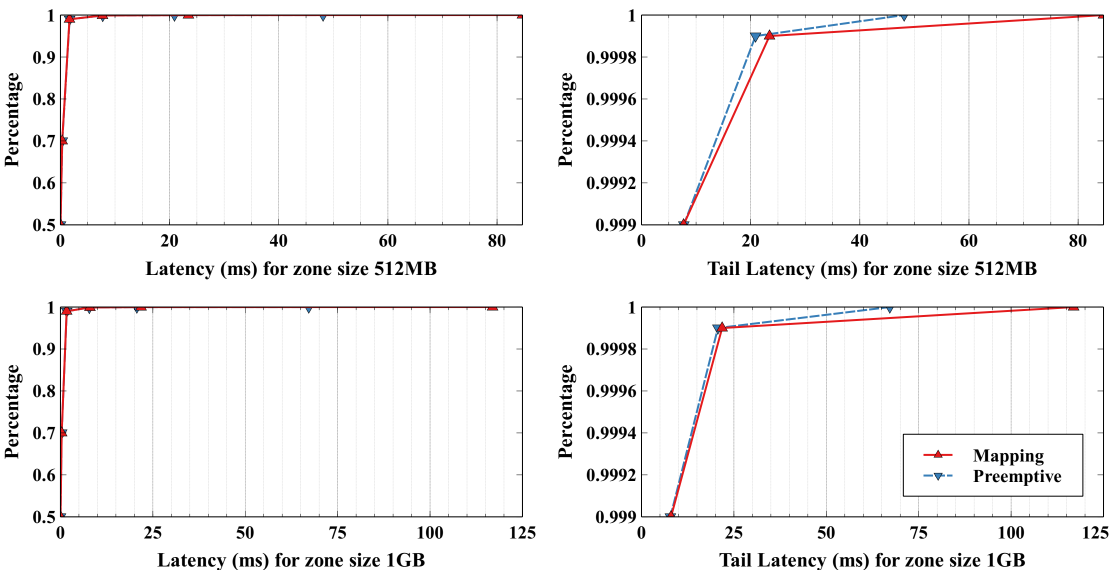

Figure 10. RocksDB read latency of mixgraph with 464B values.

Since the number of requests sent is much larger than the fio workload tested above, the tail latency difference in the db_bench evaluations appear relatively later, above P99.99. In general, we argue that Preemptive Zone Reset managed to reduce P100 tail latency by at least 1.33 times for writes and 1.74 for reads in all experimented benchmarks and environments.

5. Related Works

Since their introduction, reducing internal GC overhead in SSDs has been a hot topic, over which much ink has been spilled [2,3,5,6,23–25]. These studies mainly focus on victim block selection schemes to reduce the amount of valid pages that must be copied out during GC, employing techniques such as caching or dynamic GC, aiming to decrease write amplification. However, these options are only viable for conventional SSDs, where the FTL must keep track of invalid pages produced by out-of-place updates, as these responsibilities are shifted to the host with ZNS SSDs.

Nonetheless, there have been recent studies addressing GC overhead targeted for ZNS SSDs. LSM_ZGC [9] proposes a GC scheme resembling that of Log-Structured Merge (LSM) trees by performing valid data copy in a smaller, fine-grained unit instead of whole zones, with the intention of hot/cold data segregation. Although the paper evaluates their design on a real ZNS SSD prototype, the implementation only involves host-side software, which requires existing applications or file systems to adopt their proposed solution through additional modification.

ZNS+ [11] presents a host-induced internal data copy scheme within ZNS SSDs by adding their own original ZNS commands, mitigating avoidable data movement between the host and device when performing GC. Their design covers both file system and ZNS SSD firmware modifications, and focuses on accelerating segment compaction in F2FS [26], as log-structured file systems' data append behavior naturally complies to the sequential write constraint of ZNS. While ZNS+ does employ a zone mapping table within their design to detach zone reset calls from actual block erase operations, scheduling issues, notably when and how the numerous block erases are performed for each invalid zone, are left unaddressed.

PGC [27–29] introduces a preemptible GC scheme that allows incoming and pending I/O requests to preempt the overall GC process. As foreground I/Os are scheduled with a higher priority, the SSD is able to provide sustainable bandwidth until block erase operations become inescapable. Though the paper targets GC performed by the FTL within conventional SSDs, we took inspiration from this work and attempted to adapt the proposed design to ZNS SSDs as a way to relieve foreground I/O blocking issues when erasing zones.

6. Conclusions

Despite the fact that numerous recent studies targeting ZNS SSD adaptation and performance enhancement focus mostly on host-side software, we took a different approach by considering zone reset optimizations from the firmware perspective. We introduced a Preemptive Zone Reset method, which leverages techniques such as Partial Zone Erase, states and thresholds, and WP optimization. Our suggested preemptive scheduling design of the block erase operations, caused by zone resets, managed to mitigate the tail latency of foreground I/Os by dispersing the ensuing blocking periods. We attempted to deep dive into ZNS SSD firmware details, first with our prototype implementation, and then with our proposed Preemptive Zone Reset design. In the future, we hope to address issues concerning uneven wear-leveling, and data placement and management policies in file systems such as ZenFS and F2FS.

Author Contributions: S.J. made substantial contributions to the original ideas, designed the experiments and wrote the initial manuscript. S.L. and J.H. improved the original ideas and helped with experimental setup. Y.K. provided ideas for the experiments and discussed the results. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded in part by Samsung Electronics Co., Ltd (IO221014-02908-01), and in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386).

Data Availability Statement: Not applicable.

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

References

1. Western Digital Corporation. Zoned Storage. Available online: https://zonedstorage.io/docs/introduction/zoned-storage (accessed on 12 June 2012).
2. Gupta, A.; Kim, Y.; Urgaonkar, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV), Washington, DC, USA, 7–11 March 2009; Association for Computing Machinery: New York, NY, USA, 2009; pp. 229–240. [CrossRef]
3. Kim, Y.; Gupta, A.; Urgaonkar, B. A Temporal Locality-Aware Page-Mapped Flash Translation Layer. *J. Comput. Sci. Technol.* **2013**, *28*, 1025. [CrossRef]
4. Bjørling, M.; Aghayev, A.; Holmberg, H.; Ramesh, A.; Le Moal, D.; Ganger, G.R.; Amvrosiadis, G. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In Proceedings of the USENIX Annual Technical Conference (ATC '21), Santa Clara, CA, USA, 14–16 July 2021; pp. 689–703.
5. Agrawal, N.; Prabhakaran, V.; Wobber, T.; Davis, J.D.; Manasse, M.; Panigrahy, R. Design Tradeoffs for SSD Performance. In Proceedings of the USENIX Annual Technical Conference (ATC '08), Boston, MA, USA, 22–27 June 2008; USENIX Association: Berkeley, CA, USA, 2008; pp. 57–70.
6. Hu, X.Y.; Eleftheriou, E.; Haas, R.; Iliadis, I.; Pletka, R. Write Amplification Analysis in Flash-Based Solid State Drives. In Proceedings of the ACM International Systems and Storage Conference (SYSTOR '09), Haifa, Israel, 4–6 May 2009; Association for Computing Machinery: New York, NY, USA, 2009. [CrossRef]
7. NVM Express. NVM Express Workgroup, NVM Express® Zoned Namespace Command Set Specification Revision 1.1a. Available online: https://www.nvmexpress.org/specifications (accessed on 12 June 2022).
8. Bjørling, M. From Open-channel SSDs to Zoned Namespaces. Vault 2019. Available online: https://www.usenix.org/conference/vault19/presentation/bjorling (accessed on 12 June 2022).
9. Choi, G.; Lee, K.; Oh, M.; Choi, J.; Jhin, J.; Oh, Y. A New LSM-Style Garbage Collection Scheme for ZNS SSDs. In Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '20), Virtual, 13–14 July 2020; USENIX Association: Berkeley, CA, USA, 2020.
10. Stavrinos, T.; Berger, D.S.; Katz-Bassett, E.; Lloyd, W. Don't Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21), Ann Arbor, MI, USA, 1–3 June 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 144–151. [CrossRef]
11. Han, K.; Gwak, H.; Shin, D.; Hwang, J. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, Virtual, 14–16 July 2021; Brown, A.D., Lorch, J.R., Eds.; USENIX Association: Berkeley, CA, USA, 2021; pp. 147–162.
12. Lee, H.R.; Lee, C.G.; Lee, S.; Kim, Y. Compaction-Aware Zone Allocation for LSM Based Key-Value Store on ZNS SSDs. In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), Virtual, 27–28 June 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 93–99. [CrossRef]
13. Jung, J.; Shin, D. Lifetime-Leveling LSM-Tree Compaction for ZNS SSD. In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), Virtual, 27–28 June 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 100–105. [CrossRef]
14. Wu, G.; He, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12), San Jose, CA, USA, 14–17 February 2012; USENIX Association: Berkeley, CA, USA, 2012; p. 10. [CrossRef]
15. Kwak, J.; Lee, S.; Park, K.; Jeong, J.; Song, Y.H. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage* **2020**, *16*, 15. [CrossRef]
16. Axboe, J. fio Benchmark Tool. Available online: https://git.kernel.dk/cgit/fio/ (accessed on 12 June 2022).
17. Facebook. RocksDB Database Benchmark Tool. Available online: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools (accessed on 12 June 2022).
18. Facebook. RocksDB. Available online: https://github.com/facebook/rocksdb (accessed on 12 June 2022).
19. O'Neil, P.; Cheng, E.; Gawlick, D.; O'Neil, E. The Log-Structured Merge-Tree (LSM-tree). *Acta Inform.* **1996**, *33*, 351–385. [CrossRef]
20. Western Digital Corporation. ZenFS. Available online: https://github.com/westerndigitalcorporation/zenfs (accessed on 12 June 2022).
21. Holmberg, H. ZenFS, Zones and RocksDB—Who Likes to Take out the Garbage Anyway? SNIA 2020. Available online: https://www.snia.org/educational-library/zenfs-zones-and-rocksdb-who-likes-take-out-garbage-anyway-2020 (accessed on 12 June 2022).
22. Cao, Z.; Dong, S.; Vemuri, S.; Du, D.H.C. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20), Santa Clara, CA, USA, 24–27 February 2020; USENIX Association: Berkeley, CA, USA, 2020; pp. 209–224.
23. Salkhordeh, R.; Kremer, K.; Nagel, L.; Maisenbacher, D.; Holmberg, H.; Bjørling, M.; Brinkmann, A. Constant Time Garbage Collection in SSDs. In Proceedings of the 2021 IEEE International Conference on Networking, Architecture and Storage (NAS), Riverside, CA, USA, 24–26 October 2021; pp. 1–9. [CrossRef]

24. Lin, M.; Yao, Z. Dynamic garbage collection scheme based on past update times for NAND flash-based consumer electronics. *IEEE Trans. Consum. Electron.* **2015**, *61*, 478–483. [CrossRef]
25. Pan, Y.; Lin, M.; Wu, Z.; Zhang, H.; Xu, Z. Caching-Aware Garbage Collection to Improve Performance and Lifetime for NAND Flash SSDs. *IEEE Trans. Consum. Electron.* **2021**, *67*, 141–148. [CrossRef]
26. Lee, C.; Sim, D.; Hwang, J.Y.; Cho, S. F2FS: A New File System for Flash Storage. In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15), Santa Clara, CA, USA, 16–19 February 2015; USENIX Association: Berkeley, CA, USA, 2015; pp. 273–286.
27. Lee, J.; Kim, Y.; Shipman, G.M.; Oral, S.; Wang, F.; Kim, J. A Semi-Preemptive Garbage Collector for Solid State Drives. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11), Austin, TX, USA, 10–12 April 2011; pp. 12–21. [CrossRef]
28. Lee, J.; Kim, Y.; Shipman, G.M.; Oral, S.; Kim, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2013**, *32*, 247–260. [CrossRef]
29. Kim, Y.; Lee, J.; Oral, S.; Dillow, D.A.; Wang, F.; Shipman, G.M. Coordinating Garbage Collection for Arrays of Solid-State Drives. *IEEE Trans. Comput.* **2014**, *63*, 888–901. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.