SplitZNS: Towards an Efficient LSM-Tree on Zoned Namespace SSDs

DONG HUANG, DAN FENG, QIANKUN LIU, BO DING, WEI ZHAO, XUELIANG WEI, and WEI TONG, WNLO, Huazhong University of Science and Technology, China

The Zoned Namespace (ZNS) Solid State Drive (SSD) is a nascent form of storage device that offers novel prospects for the Log Structured Merge Tree (LSM-tree). ZNS exposes erase blocks in SSD as append-only zones, enabling the LSM-tree to gain awareness of the physical layout of data. Nevertheless, LSM-tree on ZNS SSDs necessitates Garbage Collection (GC) owing to the mismatch between the gigantic zones and relatively small Sorted String Tables (SSTables). Through extensive experiments, we observe that a smaller zone size can reduce data migration in GC at the cost of a significant performance decline owing to inadequate parallelism exploitation. In this article, we present SplitZNS, which introduces small zones by tweaking the zone-to-chip mapping to maximize GC efficiency for LSM-tree on ZNS SSDs. Following the multi-level peculiarity of LSM-tree and the inherent parallel architecture of ZNS SSDs, we propose a number of techniques to leverage and accelerate small zones to alleviate the performance impact due to underutilized parallelism. (1) First, we use small zones selectively to prevent exacerbating write slowdowns and stalls due to their suboptimal performance. (2) Second, to enhance parallelism utilization, we propose SubZone Ring, which employs a per-chip FIFO buffer to imitate a large zone writing style; (3) Read Prefetcher, which prefetches data concurrently through multiple chips during compactions; (4) and Read Scheduler, which assigns query requests the highest priority. We build a prototype integrated with SplitZNS to validate its efficiency and efficacy. Experimental results demonstrate that SplitZNS achieves up to 2.77× performance and reduces data migration considerably compared to the lifetime-based data placement.[1]

CCS Concepts: • **Hardware → Emerging interfaces**; • **Information systems → Key-value stores**; **Flash memory;**

Additional Key Words and Phrases: Zoned Namespace, LSM-tree, garbage collection

ACM Reference format:
Dong Huang, Dan Feng, Qiankun Liu, Bo Ding, Wei Zhao, Xueliang Wei, and Wei Tong. 2023. SplitZNS: Towards an Efficient LSM-Tree on Zoned Namespace SSDs. *ACM Trans. Arch. Code Optim.* 20, 3, Article 45 (July 2023), 26 pages.
https://doi.org/10.1145/3608476

[1]This article is a New Paper, Not an Extension of a Conference Paper.

This work is supported by the National Natural Science Foundation of China under Grants No. 61821003 and No. 62172178.
Authors' address: D. Huang, D. Feng (corresponding author), Q. Liu, B. Ding, W. Zhao, X. Wei, and W. Tong (corresponding author), WNLO, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, China; emails: {haltz, dfeng, qkliu, boding, weiz, xueliang_wei, weitong}@hust.edu.cn.

1544-3566/2023/07-ART45
https://doi.org/10.1145/3608476

1 INTRODUCTION

With exponential growth of data, modern web-scale systems and applications set higher demands on capacity and performance from their storage systems. Meanwhile, modern data centers widely deploy legacy block flash-based SSDs accompanied by over-provisioned capacity and cumbersome **Flash Translation Layer** (**FTL**) [31, 58], which is adverse to meeting the performance or capacity demands and **service-level agreements** (**SLA**) [36]. For example, **Log-Structured Merge Tree** (**LSM-tree**) based Key-Value Store such as RocksDB [10], LevelDB [34], BigTable [13], which have become one of the most significant services in data centers, are inevitably affected by the internal activities of legacy block SSDs [5, 6, 52, 53, 55, 56]. In the worst case, the internal activities of legacy block SSDs may result in an order of magnitude higher latency and lower throughput [31, 58] compared with the block SSDs without internal activities.

The emerging storage device, **Zoned Namespace** (**ZNS**) [5, 16] SSD affords LSM-tree with new opportunities. ZNS effectively bridges the semantic gap between flash memory and applications via exposing the internal flash blocks as zones. Unlike the legacy block interface, ZNS divides the logical address space into a collection of fixed-sized zones. Each zone must be sequentially written and reset before being rewritten. This characteristic perfectly tallies with the sequential IO patten of LSM-tree. RocksDB, one of the most popular LSM-tree based key-value store, delivers remarkable throughput and an order of magnitude lower tail latency via integration with ZNS SSDs [5], drawing both academic and industrial interests.

However, from our perspective, LSM-tree on ZNS SSDs is still faced with a formidable challenge. Modern ZNS SSDs typically maintain a gigantic zone size [41, 50], whereas **Sorted String Tables** (**SSTables**), the most files of LSM-tree, tend to have a significantly smaller size [26, 34]. To remove duplicate key-value pairs, LSM-tree performs compactions which merge-sorts multiple SSTables, then deletes them and generates new ones. Each zone usually stores multiple SSTables and thus is fragmented due to compactions, necessitating **Garbage Collection** (**GC**) for defragmentation. GC migrates valid data and resets fully invalidated zones to guarantee sufficient free space for subsequent writes. Because data migration entails excessive I/O, GC is seen as a performance killer [29, 33, 59, 61].

To tame the GC challenge, there have been many efforts yet the currently available approaches have their own set of limitations. Lifetime-based data placement [5] places SSTables with similar lifetimes in the same zone to alleviate zone fragmentation. However, it is challenging to predict the SSTables' lifetimes accurately in the high levels of LSM-tree owing to their broad lifetime spans. To ensure that a zone can be reset without data migration, GearDB [59, 61] and Lifetime Leveling Compaction [29] (LLCompaction) present their respective zone-specific compaction policies. Unfortunately, their compaction policies both aggravate compaction overheads, which in turn impairs the overall performance. Besides, since both of them mandate specific compaction policies, incorporating some existing optimizations on compactions such as LDC [11], dCompaction [47], priority-driven compaction policy [8, 19], Seek Compaction [34] into them is challenging.

With extensive experiments, we get to the following insights which present new opportunities for taming the GC challenge. ① Lifetime-based data placement is not a panacea. The ideal case, where SSTables stored in the same zone can be reset simultaneously, is impractical. ② Novel compaction policies reduce the data migration indeed, yet introduce more compaction overhead, diminishing the performance particularly under write-intensive workloads. ③ With extensive experiments, we find the root cause of GC is the mismatch between the gigantic zone size and small SSTable size. Fortunately, the internal architecture of ZNS SSD enables a smaller zone size, which is a potential solution. ④ While a smaller zone size is promising, the suboptimal performance due to under-utilized parallelism is its Achilles' heel. Our experiments demonstrate that simply

employing small zones is insufficient. **To tackle the GC challenge, we must propose a novel design to efficiently expose, leverage and accelerate zones with various sizes.**

Hence, we propose SplitZNS, an hardware/software co-designed approach to tackle the GC challenge via combining the inherent parallel architecture of ZNS SSDs and the multi-level peculiarity of LSM-tree. The keys to SplitZNS are enabling zones of varying sizes, employing them appropriately, and maximizing parallelism utilization for small zones. First, SplitZNS introduces *splitzones* and *subzones*. Each splitzone is converted from a widezone via tweaking its zone-to-chip mapping. A splitzone is composed of multiple much smaller subzones managed by independent chips to guarantee that the subzones within the same splitzone never interfere with each other. Second, SplitZNS merely employs splitzones for high-level SSTables to alleviate the performance impact due to under-utilized parallelism of subzones. Third, following the peculiarity of LSM-tree, SplitZNS proposes SubZone Ring, Read Scheduler and Read Prefetcher to accelerate the performance of subzones. SubZone Ring employs a per-chip FIFO buffer to imitate a large zone writing style; Read Prefetcher enhances parallelism utilization via prefetching data concurrently through multiple chips during compactions; Read Scheduler assigns query requests the highest priority to prevent compactions from blocking queries on subzones. To demonstrate the efficacy and efficiency of SplitZNS, we build a LSM-tree prototype based on RocksDB [10] with ZenFS [5] and evaluate it using synthetic and real-world workloads. The results show SplitZNS achieves a great performance advantage and significantly reduces I/O stress compared to its competitors [5, 29, 59]. In conclusion, we make the following contributions:

— We analyze the impact of gigantic zones for LSM-tree on ZNS SSDs and the limitations of existing works. We reveal that a novel design which is both compatible with other optimizations and GC-efficient is necessary. On the basis of the analysis, we offer our own distinct insights (Section 3).
— We design and implement SplitZNS based on our insights. First, we introduce smaller zones via the concepts of splitzone and subzone. Second, we employ them appropriately to mitigate the performance impact of low parallelism. Third, we leverage the peculiarity of LSM-tree and the architecture of ZNS SSDs to accelerate subzones (Section 4).
— We conduct extensive experiments that demonstrate the performance advantages of SplitZNS under different workloads. Specifically, SplitZNS achieves up to 2.77× performance in terms of write-only workloads and 1.79× performance in terms of write-intensive workloads compared to lifetime-based data placement (Section 5).

The rest of this article is structured into four sections. In Section 2, we brief on ZNS and LSM-tree. In Section 3, we analyze the existing works and give our insights. Section 4 elaborates on the design and implementation of SplitZNS. In Section 5, we evaluate and analyze the efficiency and efficacy of SplitZNS. Finally, we list related works and summarize our work in Section 6 and Section 8.

2 BACKGROUND

2.1 LSM-Tree

We provide a high-level overview of the LSM-tree [46], as shown in Figure 1. LSM-tree is organized as $n+1$ levels. Each level consists of multiple immutable files (called SSTables) composed of multiple data blocks and index blocks. To support efficient queries, each SSTable and each level (except $L0$) is sorted.

Read and Write. When writing a key-value pair, LSM-tree first appends it to WAL (Write Ahead Log) for crash consistency and then inserts it into a memtable in the memory (usually an

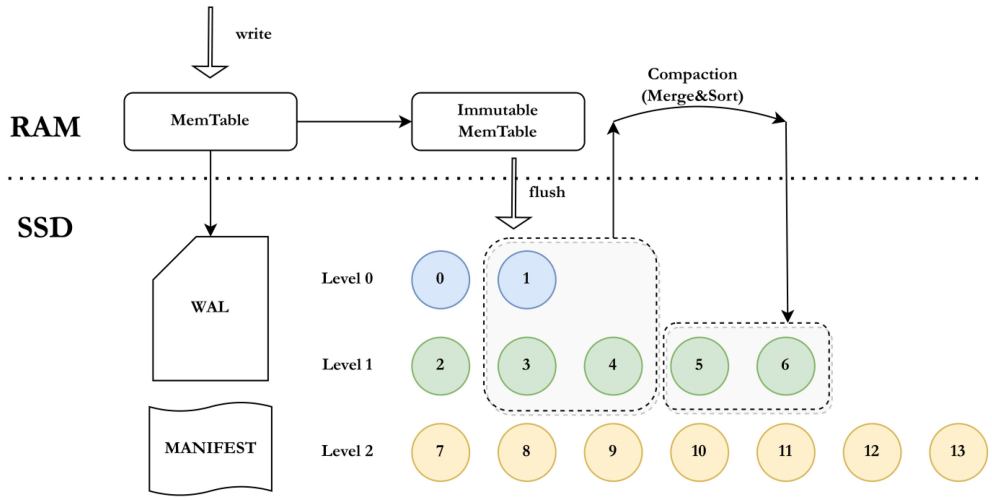


Fig. 1. A high-level overview of LSM-tree.

ordered skip-list) for batched writes to the underlying storage device. When the memtable grows over the threshold size, LSM-tree marks it as immutable, flushes it as an SSTable into $L0$, and deletes the flushed key-value pairs from WAL. To read a key-value pair, LSM-tree first searches the memtables. If the key is not found, LSM-tree will search SSTables level by level. Since each level (except $L0$) and their SSTables are sorted, the LSM-tree can easily find a key-value pair by searching binarily.

Flush and Compaction. The SSTables of the base level (i.e., $L0$) are flushed straight from the in-memory memtables. The SSTables of the subsequent levels are formed by LSM-tree compactions that migrate valid key-value pairs from one level to the next. SSTables at the same level have non-overlapping key ranges except $L0$. To remove duplicate key-value pairs to cut down on read and space amplification, each level has a threshold size that limits the number of SSTables and grows exponentially with the depth of the level. If the total size of SSTables in L_i exceeds the threshold, an SSTable in L_i is selected and sort-merged with SSTables in L_{i+1}, which have overlapping key ranges. Such a process is called a compaction. By compactions, the stale SSTables in L_i and L_{i+1} are deleted and new merged SSTables are created.

2.2 Zoned Namespace SSD

NAND Flash-based SSDs are sophisticated electronics equipped with multi-level parallelism [24, 25], as illustrated in Figure 2. NAND flash memory allows for read and write operations at the flash page level. Multiple flash pages are grouped together within the NAND flash chips to form an erase block, which is the granularity at which erase operations occur. Flash writes are only permitted on pages that have been erased. Typically, each NAND flash chip is divided into multiple dies, each of which can independently execute memory commands. All dies on each chip have a common communication interface. Each die consists of one or more arrays of flash cells called planes. Each plane is composed of multiple erase blocks, each of which necessiates writing sequentially at the unit of flash pages. To maximize the utilization of parallel chips and deliver the greatest possible throughput, SSD manufacturers usually aggregate erase blocks with the same block ID across all planes into a superblock [7, 22]. Since the block interface allows in-place update on the address space, legacy block SSDs maintain a FTL to bridge the semantic gap between flash memory and the block interface, which has a serious impact on performance [31, 58]. ZNS SSDs

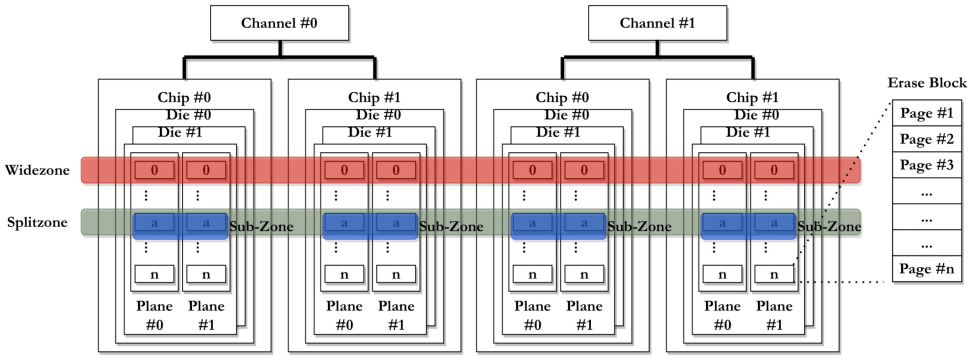


Fig. 2. The internal architecture of a ZNS SSD and widezones/splitzones/subzones.

expose SSD internals (i.e., superblocks) in the form of zones to mitigate the negative effects of FTL. The address space is partitioned into *zones*, which are usually mapped to superblocks and must be written sequentially. The current write position is tracked by a write pointer. While ZNS SSDs eliminate device-side GC, they necessitate zone-aware data placement and possibly host-side GC [5].

2.3 Garbage Collection for LSM-Tree on ZNS SSDs

ZNS SSDs require a log-structured write approach, which can lead to fragmentation and the need for costly garbage collection. Specifically, compactions remove obsolete key-value pairs but leave invalidated SSTables in zones. To purge these invalidated SSTables and maintain sufficient free space, LSM-tree must perform garbage collection by migrating valid data from fragmented zones and resetting them. Without GC, an SSD would only be able to store a limited amount of valid data. We evaluated the space amplification using fillrandom workload (a write-only and uniform workload) from RocksDB micro-benchmark with 32 MiB SSTs on an 80 GiB ZNS SSD. The empirical observation is that there is only 27.2 GiB valid data when RocksDB corrupts due to shortage of space (2.94× space amplification), confirming the necessity of GC.

3 MOTIVATION AND CONTRIBUTIONS

Designing a high performance LSM-tree on ZNS SSDs poses unique challenges. In this section, we first analyze GC for LSM-tree on ZNS SSDs with various zone sizes, which motivates SplitZNS's design. Then, we revisit two kinds of exisiting approaches, lifetime-based data placement and compaction schemes tailored for zones, and reveal their respective issues.

3.1 Widezones

Modern ZNS SSDs typically feature gigantic zones. The zone capacity of INSPUR NS8600 and ZN540, for instance, are 1440 MiB and 1077 MiB [41, 50], respectively. The reason is that SSDs are equipped with multi-level parallelism [25], which accounts for their superior performance. As shown in Figure 2, an SSD usually consists of multiple channels, each of which is consist of multiple chips. The chip is consist of a number of dies, and each die is comprised of multiple planes. Each plane comprises numerous erase blocks. The independent execution of read, write, and erase operations is feasible across the chips and dies. In order to achieve optimal performance by simultaneously accessing multiple chips, commercial ZNS SSDs form a zone by combining multiple erase blocks from each chip [6, 7, 22, 23, 35].

Specifically, the SSD controller picks an erase block from each plane (i.e., a few erase blocks from each chip) to form a zone, as the widezone illustrated in Figure 2. Once an IO request arrives, it can be divided into multiple sub-requests at the unit of flash page. These sub-requests can be dispatched to separate chips so that they can be processed concurrently. With the advancement of flash memory technologies, such as **Multi-Level Cell** (**MLC**) [1] and 3D stacking [42], a flash cell can store more bits, making the erase block size grow. Consequently, the zone size is expanding as well.

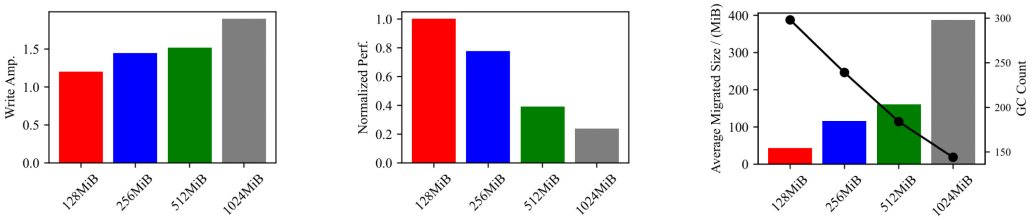
Unfortunately, gigantic zones have a significant negative effect on the performance of LSM-tree. Larger zones result in more severe fragmentation, necessitating more data migration to reclaim zones. In addition to wasting I/O bandwidth, data migration might stall foreground writes since free space is not available until data migration is finished. We conduct a quantitative analysis of the performance and the migrated data size for various zone sizes. For a fair comparison, zones with various sizes share identical I/O performance, which is achieved by adjusting the erase block size. To begin, we use FEMU to emulate an 80 GiB ZNS SSD. 50 GiB data is loaded into the database. Then, the overwrite workload (uniform and write-only) from RocksDB's dbbench is used to write another 10 million key value pairs (16B key and 1 KiB value size) into the database. The overwrite performances of various zone sizes are evaluated. Figure 3(a) and 3(b) shows the results normalized to 128 MiB zone size. The results show that as the zone size increases from 128 MiB to 1,024 MiB, the write amplification from data migration increases by up to 1.9× and the throughput declines by 73%, which confirms the negative effect of significant zone size.

Although manually increasing the SSTable size can help alleviate the impact of significant zone size, the SSTable size is not fixed and usually small under many compaction policies [48, 51]. Simply adjusting the SSTable size is not feasible for these compaction policies. Moreover, to the best of our knowledge, oversized SSTable sizes usually result in high compaction overhead, as is also noted in earlier studies (GearDB [59], LLCompaction [29]) and practical production experience. Specifically, TiKV can use up to 128 MiB SSTables only when compaction-guard (one of its features to reduce compaction overhead based on the characteristic of its upper application) is enabled [51].

The performance decline associated with larger zones can be attributed to the increased amount of data that needs to be migrated before resetting. Figure 3(c) displays the average size of migrated data for different zone sizes. When increasing the zone size from 128 MiB to 1,024 MiB, the average size of migrated data increases from 43 MiB to 386 MiB while the GC count decreases from 298 to 144. This ultimately results in more data migration and slower GC. Sluggish data migration can compete for the IO bandwidth with foreground operations, while slow GC can cause foreground operations to stall, thus resulting a significant decrease in performance.

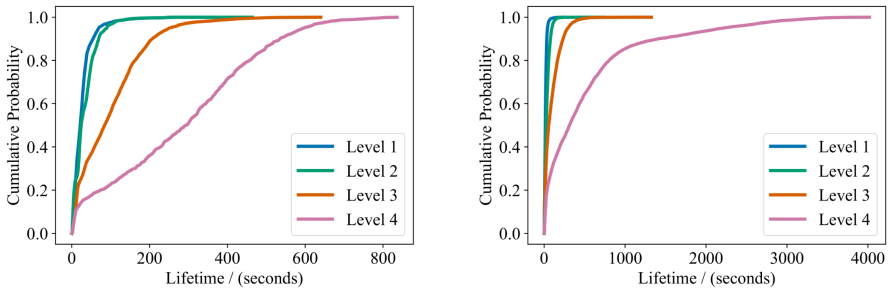
3.2 Existing Approaches

There have been several attempts to address the problem yet existing approaches have their own set of limitations. Simply reducing zone size cannot tame the challenge. Unlike widezones, smaller zones align to fewer erase blocks [2]. Consequently, the number of chips that process I/O requests is limited, causing a severe performance degradation. Our experimental results (Section 5.4) further confirm this fact. Notice we also achieves least internal mutual interferences as Reference [2] and even more accurately in this experiment from our perspective since we expose the internal interferences of zones and directly allocating chips (Section 4.3) instead of using profiling-based allocation. Lifetime-based data placement [5] places SSTables with analogous lifetimes in the same zone. However, the SSTable's lifetime is predicted by its level in the LSM-tree, which, from our



(a) GC write amplification. (b) Normalized performance. (c) Average size of migrated data.

Fig. 3. Impact of zone size on peformance and data migration.



(a) Uniform workload. (b) Skewed workload.

Fig. 4. CDF of SSTable lifetimes.

perspective, is inaccurate, particularly for high-level SSTables. As seen in Figure 4, we track the lifetime of SSTables from various levels in the uniform and skewed (skewness = 0.99) write-only workloads. The experimental setting is same as above. The lifetimes of most *L*4 SSTables range from approximately 10s to 800s for the uniform workload and range from approximately 10s to 4000s for the skewed workload. We observe that (1) with a higher skewness, the lifetime difference is larger since the SSTables with hot key ranges are involved in compactions frequently; (2) however, in terms of both workloads, the lifetimes of high-level SSTables can be rather short or rather long, confirming that with or without the skewness, the lifetime prediction based on the level of SSTable is difficult to achieve a high accuracy. We also evaluate the workloads with other different skewness (0.8, 0.9, 0.95) and their experimental results show similar trends.

GearDB [59, 61] and **Lifetime-Leveling Compaction (LLC)** [29] use zone-specific compaction policies to eliminate GC. Nevertheless, their compaction policies result in a greater write amplification and a longer compaction latency (Section 5.3), which hurts the performance. Moreover, both of them prohibit compaction optimizations when selecting input SSTables for compactions since they mandate a certain order of invalidating SSTables. Examples are LDC [11], which take into account compaction granularity; LevelDB [34], which takes into account the number of times an SSTable's seek operation failed; and RocksDB [8, 10, 19], which takes into account the SSTable's age and overlapping ratio. These optimizations cannot co-exist with GearDB or LLC.

3.3 Our Insights

The aforementioned analysis yields the following insights, which motivate the design. First, a smaller zone size can alleviate GC. By assigning a zone to a small number of erase blocks managed by a single chip, the zone size can be drastically decreased. However, it leads to underutilized parallelism since I/O requests to these small zones are executed one at a time by a single chip,

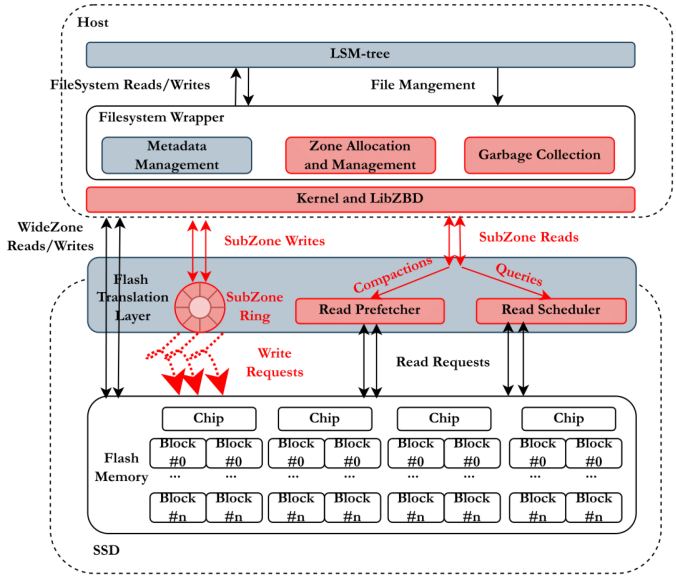


Fig. 5. Overview of splitZNS design.

resulting in a significant decline on I/O performance. Consequently, the second insight is that by using the multi-level structure of LSM-tree, we can efficiently mitigate the negative effect of reduced zone size. Moreover, we can accelerate the performance of these small zones by enhancing parallelism utilization. Based on these insights, we propose SplitZNS, which incorporates different zone sizes into LSM-tree on ZNS SSDs.

4 DESIGN AND IMPLEMENTATION

4.1 Overview

Figure 5 provides a high-level overview of SplitZNS (red parts are modules modified compared to conventional RocksDB and ZenFS). We modify the zone management and GC module to support subzones and splitzones. In addition, we adapt Linux Kernel [39] and LibZBD [18] to support splitzone and subzone state transitions. We reuse and extend the original FTL to deliver I/O requests at the device level and implement SubZone Ring, Read Prefetcher, and Read Scheduler to enhance the performance.

4.2 Splitzone and Subzone

In this section, we present two new concepts, splitzone and subzone. The essential difference between splitzone and widezone is that splitzone modifies the zone to chip mapping of widezone, as shown in Figure 2. Within a widezone, zone is mapped to a superblock composed of all chips. Within a splitzone, however, zone is divided into a number of subzones, each of which is mapped to a single chip.

In order to support the management of splitzones and subzones, the zone state machine is modified and some new commands are introduced, as shown in Figure 6. *SPLIT* command is introduced to convert an empty widezone to a splitzone. The newly converted splitzone is in the open state and contains N_{chip} empty subzones. Our zone allocator, which will be described in Section 4.3, assigns these free subzones to incoming SSTables. Like conventional widezones, each open splitzone consumes an active token and an open token. When all subzones of a splitzone are inactive

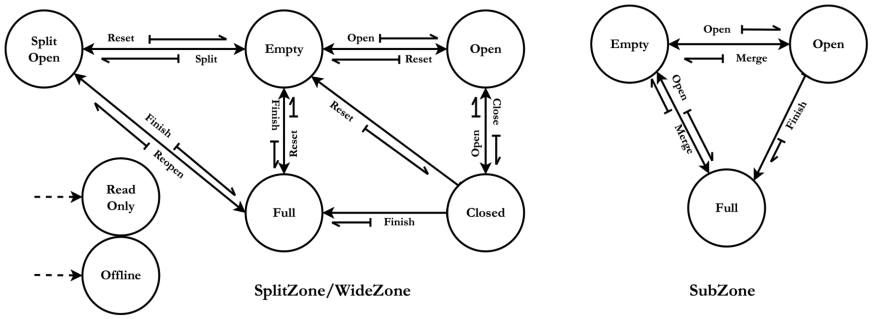


Fig. 6. Zone state transitions.

(i.e., either empty or full), the splitzone can be deactivated with the *FINISH* command. In particular, when all subzones of a splitzone are empty, the splitzone can be converted into an empty widezone by the *RESET* command. Upon deactivating an open splitzone, the consumed active and open token can be released. The state transitions of splitzones and subzones are only initiated by direct host commands, which is a little different from widezones. Once a write request goes to a subzone, this subzone enters automatically into the open state from the empty state; when the host finished writing an SSTable, the subzone is finished; when SSTables are deleted, the subzones are reset with *MERGE* command.

Despite that partial zone reset (i.e., subzone reset) is not supported in the current ZNS specification, it is feasible from the internal view of SSD. On the device side, zones are superblocks, which are aggregated of erase blocks from multiple chips. A zone reset is built of numerous erase block erasures, as the erase block is the smallest unit of erasure. It is feasible to erase simply a single erase block. Several prior block-interface SSD studies, for instance, employ chip-level GC, where erase blocks are erased at the unit of N_{chip} flash blocks [21, 36, 57]. The only involved hardware overhead is the additional memory capacity to store the mapping table from subzones to erase blocks. The overhead is acceptable since the mapping table is at the unit of subzones. For a 1 TiB ZNS SSD with 32 MiB subzones, the number of mapping table entries is 32,768 and each entry occupies 8B (4B for the subzone ID and 4B for the erase block ID), which means the total memory capacity overhead is 256 KiB. Hence, the hardware overhead introduced by partial erasure use is acceptable.

4.3 How to Leverage Splitzones/Subzones?

There is an apparent tradeoff between data migration and utlization of parallelism for splitzones. Splitzones can decrease data migration owing to their smaller subzones, however, their low parallelism may cause a considerable performance decline. In this section, a policy selectively use various zones is introduced to tackle the dilemma. In short, for low-level ($L0, L1, L2, L3$) SSTables, widezones (i.e., large zones, as shown in Figure 2) and lifetime-based data placement are combined to provide high performance and alleviate GC; for high-level SSTables ($L4$ and above), smaller subzones are used to minimize the cost of GC. Notice Li means the ith level of the LSM-tree, as shown in Figure 1.

This technique is founded on two observations. (1) First, lifetime-based data placement is only inefficient for high-level SSTables. We collect statistics in terms of the lifetime of SSTables from various levels. The result demonstrates that the lifetime varies from nearly 0s to 100s for the low-level SSTables and from 10s to 800s for the high-level SSTables ($L4$), as is shown in Figure 4. The observation is that the lifetime prediction for low-level, short-lived SSTables is accurate. Thus,

lifetime-based data placement is effective for their purposes. However, the lifetime prediction for SSTables at higher levels is inaccurate, making lifetime-based data placement inefficient. (2) Second, small zones may result in severe write stalls because their inferior performance makes compactions slower, which is confirmed by experiments in Section 5.4.2. As mentioned before, to write new key-value pairs, LSM-tree must maintain sizeable free space in memory by flushing memtables. If memtables are flushed slowly, writes may stall, leading to poor performance. Flushing a memtable requires *L*0 to reserve sufficient free space, thus *L*0 should also be compacted into the next level as quickly as possible. The lower the level, the greater the impact. Therefore, low-level SSTables should be stored in widezones to avoid write stalls for the same reason. For high-level SSTables which have a gentler impact on foreground writes, subzones are superior choices. Combining these considerations, SplitZNS uses splitzones and subzones merely for high-level SSTables.

Three points should be taken into care to implement a zone allocator which selectively allocates various zones. (1) First, the allocator should allocate the same widezone for SSTables with the equal lifetime (i.e., lifetime-based data placement). Since a splitzone cannot be used mixed with widezones, it must be guaranteed that there are sufficient widezones to avoid shortage of free space for low-level SSTables. Fortunately, the widezones are merely for low-level SSTables, which occupies limited storage space. Hence, a hard limit is set on the maximum number of splitzones (60% of the number of total zones in our case). If the limit is reached, like lifetime-based data placement, the allocator simply allocates widezones for SSTables based on their lifetime. (2) Second, the allocator should be capable of exploiting the parallelism. Remember that subzones within the same splitzone are mapped to independent chips. To leverage this characteristic, a greedy policy is employed to design the zone allocator. When allocating a subzone, SplitZNS always picks a subzone from the splitzone whose capacity usage is the highest (i.e., the number of the full subzones is the highest). Such a policy is favorable for balancing the number of subzones managed by each chip, thus making IO requests to subzones experience the least IO interferences. (3) Besides, in order to further consolidate this optimization, we propose a chip allocator at the device-side, which allocates chips in a round-robin manner. This device-side chip allocator coordinates with the host-side zone allocator to enable the host to be aware and capable of exploiting the SSD internal parallelism. To implement such a round-robin chip allocator, an atomic counter is introduced. The counter iterates from 0 to N_{chips} (the number of chips), and starts from 0 again. When a write request to an empty subzone arrive at the device, this chip allocator assigns a chip, whose number corresponds to the counter value, to this subzone. Then, this counter is incremented atomically. If this allocated chip has been allocated before to some other subzone within the same splitzone, the allocator just repeats the procedure until it finds an appropriate chip. It is ensured that a chip can be allocated for this subzone since a splitzone is divided into N_{chips} subzones so each subzone within the same splitzone can be allocated a unique chip.

4.4 Accelerating Subzones

4.4.1 SubZone Ring. Subzones suffer from poor performance because they are unable to fully exploit the SSD's rich parallelism. In this section, we propose SubZone Ring to improve the parallelism utilization of subzones for writes. SubZone Ring uses a per-chip FIFO buffer to put the data written to the subzones in the buffer and then use as many chips as possible to flush them simultaneously. Using such an approach, writing a subzone is comparable to writing a widezone.

However, we must take read efficiency into consideration before implementing such a subzone write mechanism. Once the data of a subzone has been stored in the buffer, we must search the whole buffer to fetch these data. Unfortunately, FIFO buffer queries are inefficient (complexity O(N)). To overcome the issue, we maintain a small ring array for each subzone whose data has

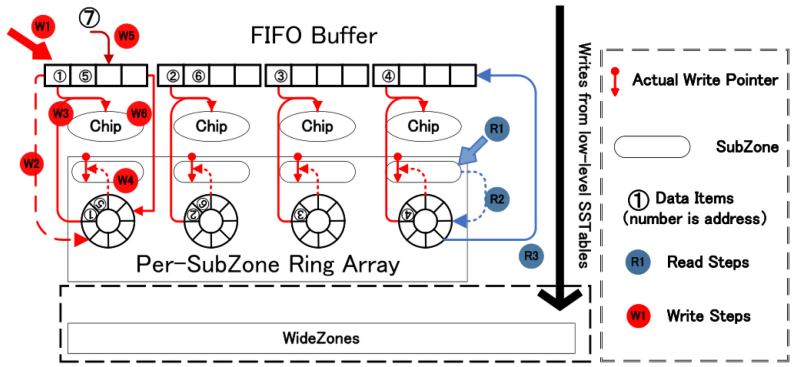


Fig. 7. Subzone ring.

not been completely flushed to the flash memory. Each element in the ring array keeps track of the memory address of a flash page in the FIFO buffer. When a flash page is written to the FIFO buffer, an element is appended to the end of the corresponding subzone's ring array. When the data is flushed from the FIFO buffer to the flash memory, the element at the head of the corresponding ring array is removed. Then, we update the actual write pointer of this subzone. By using such an approach, if a read request to the subzone needs to obtain data from the FIFO buffer, the read address offset, which means the offset of the request address from the target zone's start address, can be used to obtain the memory address of the data in the ring array (i.e., $RingArray[RingArrayHead + ReadPageAddress - ActualWritePointer]$).

We take two examples to describe the read and write steps within the context of the SubZone Ring, as shown in Figure 7. For read and write requests to widezones, we add them directly to the request queues of the chips. Hence, we illustrate here only the procedures performed for read and write requests to the subzones. For brevity, we assume that there are only four chips in the SSD. First, we explain how writing a flash page works. Assuming that we are writing the flash page⑦. (1) W1: First, we must check whether the FIFO buffer is full. (2) W2: If full, select a flash page of data from each chip's FIFO buffer. (3) W3: We then generate a separate write request for each selected flash page of data and add it to the request queue of the corresponding chip. (4) W4: When the chip completes the write request, we update the subzone's actual write pointer position. (5) W5: After the update is done, we write the flash page⑦ to the FIFO buffer. (6) W6: Finally, we complete this write request by adding the memory address of the flash page⑦ to the end of the ring array of the corresponding subzone.

The second example is used to how to read a flash page. (1) R1: First, we need to compare the read request address with the actual write pointer to check whether the required data is in the FIFO buffer. (2) R2: If the requested data is in the FIFO buffer (i.e., $ReadAddress > ActualWritePointer$), we obtain the memory address of the requested data in the ring array based on the offset of the request address relative to the actual write pointer (i.e., $RingArray[RingArrayHead + ReadPageAddress - ActualWritePointer]$). If the requested data is not in the FIFO buffer, we follow the original read procedures to access the flash memory and fetch the data. (3) R3: Finally, we return the data to complete this read request.

The prerequisite for SubZone Ring to work efficiently is that it can fully utilize all the chips each time it flushes data to the flash medium, which means that there are sufficient data items managed by different chips. Fortunately, this prerequisite is usually met owing to the addition of our chip allocator at the device level. The efficiency of SubZone Ring is further confirmed by our experiments, which demonstrate significant improvements on the efficiency of compaction after

employing SubZone Ring. In addition, DRAM protected by supercapacitors is used for power loss protection.

4.4.2 Read Prefetcher and Read Scheduler. This section describes the read improvements we've implemented for subzones. There are two issues with subzone read requests: (1) First, similar to write requests, SSD can only execute one subzone read requests at a time, hence failing to maximize the utilization of rich parallelism inherent to SSD. (2) Second, the read requests to subzones can be divided into read requests originating from queries and read requests originating from compactions. Queries are user foreground read operations that retrieve key-value pairs and thus have a high priority. Compactions are background operations that are insensitive to latency and thus have a low priority. Compactions may block queries and cause excessive tail latency of queries, which is detrimental to the key-value store's quality of service. Therefore, we optimize both the utilization of parallelism and query tail latency.

To begin, we introduce our optimization on queries. We must first differentiate read requests of queries from those of compactions. The obvious contrast between them is that compactions read data sequentially, whereas queries read data randomly. On the basis of this, we construct a Read Scheduler. Since a subzone only stores one SSTable and SSTables are read in their whole and in sequential order only once during compactions, reading a subzone during compactions is quite similar to writing it, thus we borrow the idea of the zone write pointer and introduce a read pointer for each subzone to keep track of the amount of data that has been read for compactions. The read pointer initially resides at the start address of the subzone and is reset when the subzone is reset. When a read request arrives at the read pointer, the read pointer is advanced by the read request's data size. When the read request address is identical to the read pointer, it means the read request is sequential and comes from compactions; otherwise, the read request is random and comes from queries. All requests from queries are placed into the request queue prior to any other read or write request except for those requests which are from queries as well.

Then, we present the optimization to improve parallelism utilization. Typically, queries randomly query multiple SSTables to retrieve data, and since multiple chips will be used, parallelism has already been exploited for them. Comparatively, compactions underutilize parallelism since they typically read a subzone sequentially, and so can only use a single chip. Consequently, in this context, we only consider compactions to optimize parallelism. The way to distinguish between read requests from compactions and queries remains the same as earlier. For read requests from compactions, we preread data at the read pointer for the subzones managed by other chips to simultaneously use multiple chips. We keep a per-subzone preread buffer with a size equal to four flash pages to store the prefetched data. There are three considerations when selecting a subzone for data prereading. First, the subzone should be managed by a different chip. Second, the SSTable stored in this subzone should be one of the SSTables used as input in current compactions. (i.e., the subzone read pointer is not at the start address of the subzone). Third, the preread buffer of this subzone should be empty to avoid unnecessary preread requests. We iterate over all managed subzones for each chip in search of a subzone that meets the requirements. Then, for each selected subzone, we construct a read request to preread the data and add it to the request queue of the corresponding chip. If there are no such subzones on a particular chip, we just skip this chip since it is unnecessary to preread it.

We use an example to demonstrate how data is read if it is not in the SubZone Ring, as shown in Figure 8. For the sake of brevity, we also assume that the SSD has only four chips. For read requests to widezones, we simply append them to the request queues of the corresponding chips. We focus primarily on the procedures performed for subzone read requests. (1) R1: First, the read address is compared to the read pointer. (2) R2: If the read address differs from the read pointer, we assign

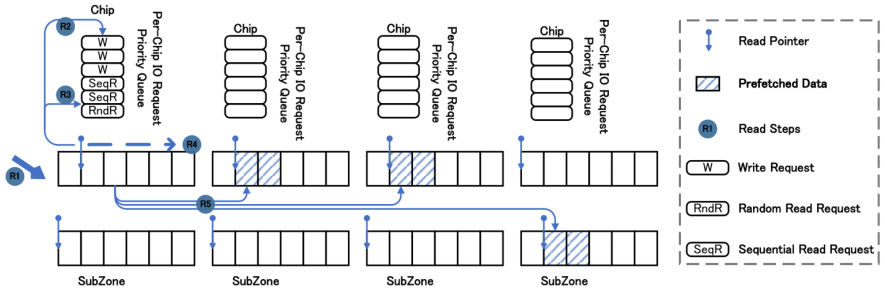


Fig. 8. Read prefetcher and read scheduler.

the read request the highest priority since it is considered from queries. Then, we place it before other types of requests in the request queue, wait for its completion, and return the retrieved data to finish this request. (3) R3: Otherwise, it is appended to the end of the priority queue. (4) R4: The read pointer is then advanced by the request's data size. If the data can be retrieved from the preread buffer, we return the retrieved data and do not preread. (5) R5: If accessing the flash memory is required, we iterate managed subzones for each chip, choose appropriate subzones, and prefetch data to maximize parallelism utilization. When all of these steps have been completed, we return the retrieved data and finish this read request.

4.5 Implementation

4.5.1 Hardware Overhead. Since both SubZone Ring and Read Prefetcher use fast memory to expedite I/O requests, we discuss the memory overhead in this section. First, the memory overhead of SubZone Ring is acceptable. To ensure that the SubZone Ring contains data corresponding to each chip upon flushing, its buffer size is set to at least that of a widezone. Notice that the buffer overhead is proportional to zone size instead of device capacity. For a 1TiB ZNS SSD with a 1GiB zone size, the buffer occupies 0.1% of capacity, which is comparable to the page mapping table cache size in legacy block SSDs. To date, the average price of 1GiB DDR4 2,666 MHz DRAM module is approximately 35.4% of the price of 64 GiB 3D TLC NAND flash memory [20], making the medium cost increase by about 2.2% for the above case. If the SSD is an 8TiB ZNS SSD, the cost increases by only 0.27%. Second, the overhead of Read Prefetcher is acceptable as well. For the Read Prefetcher, we can get the memory overhead by computing $N_{max\_ssts\_in\_a\_compaction} * N_{max\_compactions} * N_{per\_subzone\_buffer\_size}$. $N_{max\_ssts\_in\_a\_compaction}$ is the maximum number of SSTables involved in a compaction (typically 8 in RocksDB); $N_{max\_compactions}$ is the maximum number of compactions (the number of concurrently running compactions will not exceed 16 in our experiments); $N_{per\_subzone\_buffer\_size}$ is the preread buffer size for each subzone (512 KiB in our case). Hence, the total memory overhead is 64 MiB in our case. Another 1 GiB DDR4 DRAM module can be used to support it (2.2% increased medium cost). In conclusion, the memory capacity overhead is limited to 1088 MiB; the device cost increases by 4.4% for a 1 TiB ZNS SSD and by 0.54% for a 8 TiB ZNS SSD.

Apart from the battery-backed RAM, we can achieve a higher cost-efficiency by other choices. First, similar to the existing study [30], we can use SLC mode flash blocks as the low-cost memory. Second, we can use **Host Memory Buffer** (**HMB**) [16] technology to delegate the crash consistency guarantee responsibility to the host. Third, we can use a redo-log to eliminate the additional hardware overhead. Specifically, for a compaction, the input SSTable deletions can be postponed until the issued writes are totally flushed back to the flash medium. If a crash happens, the host can redo the compactions based on the input SSTables.

Table 1. Hardware Specifications

Host Machine			
CPU		Intel Xeon Gold 6230R * 104	
Memory		128 GiB, 2,666 MHz	
Guest Virtual Machine			
CPU		Intel Xeon Gold 6230R * 32	
Memory		32 GiB, 2,666 MHz	
KVM		Enabled	
SSD Parameters			
# of channels	8	page read latency	35 µs
# of chips per channel	2	page write latency	960 µs
# of planes per chip	4	block erase latency	3 ms
flash page size	16 KiB	erase block size	8 MiB

4.5.2 Enforce the Order of Read/Write Requests. This section provides an elaboration on the enforcement of read/write order in the context of optimizations. On the host-side, as in the original ZenFS, SplitZNS employs the mq-deadline scheduler to guarantee the order of read and write requests. On the device side, SplitZNS employs its SubZone Ring as a First-In-First-Out buffer to fulfill the sequential write requirement of zones in the case of write requests. Hence, the sequential write order is guaranteed as the first written data received for each subzone is always flushed first. In the case of read requests, SplitZNS first ascertains the target zone based on the provided read address. Subsequently, SplitZNS examines if the address surpasses the actual write pointer of the target subzone. If the response is affirmative, it indicates that the data has not been written to the flash memory, and SplitZNS retrieves the data from the SubZone Ring. Otherwise, SplitZNS retrieves the data directly from the flash memory. By means of this approach, SplitZNS can guarantee that read requests are fulfilled with correct data.

4.5.3 Accuracy of the Automatic Detection for Compaction and Query Read Requests. Requests are tagged at the host-side and subsequently verified for accuracy of detection. The O_DIRECT bit-flag was applied to all tagged requests to enable them to circumvent the page cache's format check, albeit at the expense of performance. An experiment was conducted using the YCSB-A workload and the experimental setting outlined in Section 5.2 to validate the accuracy of the detection. We conducted surveillance of the detection accuracy at intervals of 10,000 read requests to subzones. The accuracy was calculated as the ratio of requests whose tag matched the outcome of the detection. Experimental results indicate that the precision vary between 78.93% and 99.17%, with an average of 94.16%. Thus, in our perspective, the detection is accurate.

5 EVALUATION

In this section, we evaluate SplitZNS to answer the following questions:

— How does SplitZNS perform under different workloads compared to its competitors (Section 5.2)?
— Why is SplitZNS better compared to its competitors (Section 5.3)?
— How do SplitZNS's techniques improve the performance? (Section 5.4)?
— How scalable is SplitZNS (Section 5.5)?

5.1 Methodology

5.1.1 Hardware Platform. We evaluate SplitZNS and its competitors using a QEMU-based SSD emulator FEMU [35], a widely-used emulator for real data-intensive applications/systems and preferred by researchers [23, 28, 36, 62], to emulate a real ZNS SSD. We refer to prior works [23, 36]

using FEMU and set the configuration of the target system as in Table 1. The emulated ZNS SSD has 8 channels, each of which is connected to 2 chips and each chip has 4 planes. The flash page size is 16 KiB and erase block size is 8 MiB. The flash page read, write, and erase latency are 35 µs, 960 µs and 3 ms, respectively. The Subzone Ring size is twice as large as a widezone. The Read Prefetcher reserves 4 flash pages for each subzone. The emulated ZNS SSD has 160 512 MiB zones. The maximum number of active zones and open zones are both 24. We reserve 32 vCPUs and 32 GiB memory for the guest virtual machine. The real host's DRAM is utilized to emulate the SSD's memory. Each splitzone contains 16 equivalent subzones, which are managed by independent chips.

5.1.2 Software Platform. We implement an LSM-tree integrated with SplitZNS. The implementation is forked from RocksDB v7.6.0 and ZenFS v2.0.0. To support new commands introduced in SplitZNS, we also modify Linux kernel based on Linux v5.13.0 and LibZBD based on LibZBD v2.0.3. We use direct IO requests to bypass the page cache, which may disrupts the write order of IO requests and cause unexpected corruptions in kernel for ZNS. The SSTable size is set equal to the subzone size to avoid data migration (i.e., 32 MiB). The maximum number of concurrent background compactions is 16.

5.1.3 Compared Approaches.

— Lifetime-based Data Placement (LDP). LDP is our main competitor since it allows other optimizations on compactions such as LDC. LDP allocates the same zone for SSTables with identical lifetimes and the lifetimes are predicted based on the level of SSTables. Besides, LDP employs Min-Overlapping-Ratio compaction policy [19] (the default compaction policy of RocksDB) and a greedy GC scheme, which migrates valid data of the zone with least valid ratio to other zones when the free space falls under the threshold (20%).
— Gear Compaction, the compaction policy adopted in GearDB. It is a LevelDB variant optimized for **Shingled Magnetic Record** [9] (**SMR**) disks. Since GearDB is forked from LevelDB, which is not optimized for SSDs like RocksDB, we reproduce it based on RocksDB for a fair comparison. We allocate dedicated zones for SSTables from different levels and employ the Gear Compaction policy. Gear consumes more space than LDP because it triggers more compactions, which lock more SSTables; hence, we enable GC for Gear just as we do for LDP. Notice GC for Gear is not as severe as LDP and yields a negligible performance decline.
— LLC, another compaction policy for LSM-tree on ZNS SSDs. LLC picks SSTables sequentially based on the key range in each level as input SSTables of compactions. Since LLC is forked from LevelDB and not open-source, we also implement it from scratch on RocksDB. We also enable GC for it as Gear and LDP.

5.1.4 Workloads. We explore different aspects of the systems by using synthetic and realistic (YCSB [17]) workloads, as shown in Table 2. There are two types of key popularity: uniform and skewed. In uniform workloads, all keys have the same probability of being accessed. Skewed workloads follow a zipfian or latest key access distribution (skewness = 0.99), which is common in production environments. For each experiment, we first load the database with approximately 50 GiB data, then perform specified workloads with 4 threads, each of which executes 1 million operations. The reason we don't load too much data is the extra space usage caused by the space spike during compactions. The results are the average of three or more runs. If not specified particularly, the performance refers to the throughput.

5.2 Throughput and Latency

Since we focus on optimizations on GC caused by compactions, we mainly discuss the performance (throughput and latency) affected by background activities (i.e., compactions and GC).

Table 2. Workloads

Workload Name	Description	Key Distribution
FillRandom	Insert:100%	Uniform
OverWrite	Write:100%	Uniform
ReadRandom	Read:100%	Uniform
YCSB-L	Insert:100%	Zipfian
YCSB-A	Read:50%, Write:50%	Zipfian
YCSB-B	Read:95%, Write:5%	Zipfian
YCSB-C	Read:100%	Zipfian
YCSB-D	Read:95%, Insert:5%	Latest
YCSB-E	Scan:95%, Insert:5%	Zipfian
YCSB-F	Read:50%, Read Modify Write:50%	Zipfian

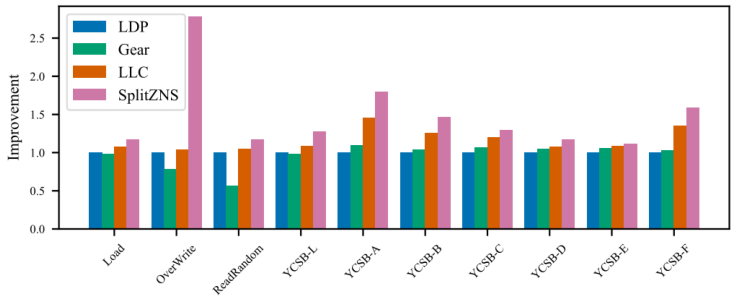
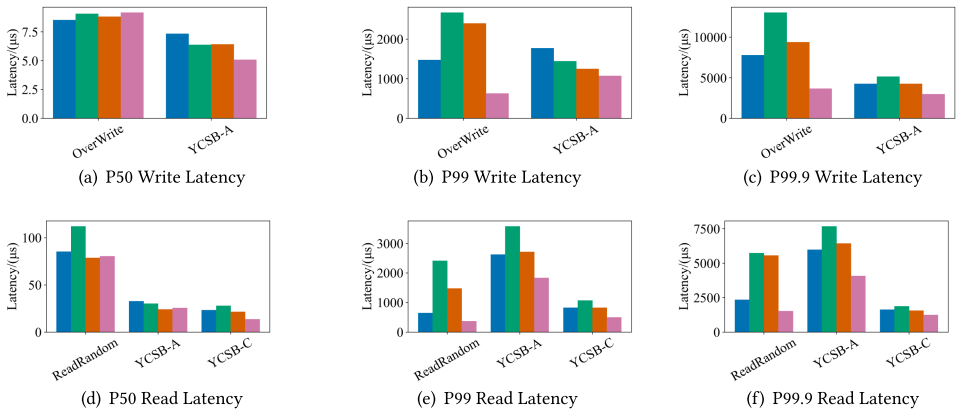


Fig. 9. Overall performance.



(a) P50 Write Latency (b) P99 Write Latency (c) P99.9 Write Latency

(d) P50 Read Latency (e) P99 Read Latency (f) P99.9 Read Latency

Fig. 10. Read and write latency of representive workloads.

The results include the throughput and latency in the write-only, write-intensive workloads and the read-intensive workloads after the write bursts (i.e., right after loading), as shown in Figure 9 (normalized to LDP) and Figure 10, which leads us to the following conclusions. (1) First, SplitZNS effectively mitigates the effects of GC. Compared to LDP, SplitZNS delivers 2.77× the overwrite throughput and 1.79× the YCSB-A throughput. This is because SplitZNS reduces background data migration because it stores SSTables in subzones and resets subzones without data

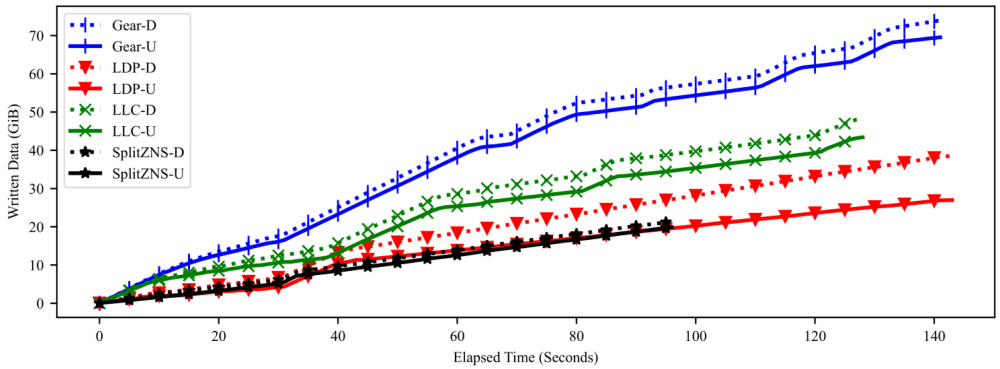
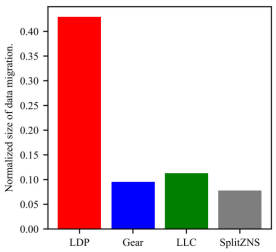


Fig. 11. Device writes and user writes.

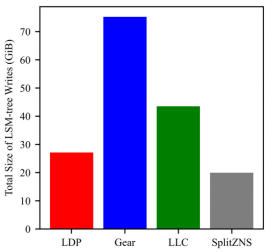
migration when SSTables are invalidated. (2) Second, compared to the zone-specific compaction policies Gear and LLC, SplitZNS obtains 3.55×, 2.68× throughput for overwrite workload and 1.63×, 1.23× throughput for YCSB-A workload. This is because Gear and LLC triggers many more compactions, which consumes more I/O bandwidth and hence impairs the performance. (3) Third, SplitZNS also improves the performance for workloads dominated by read operations. For instance, SplitZNS shows 1.16×, 2.05×, and 1.11× better throughput for readrandom and 1.29×, 1.21×, and 1.07× better throughput for YCSB-C. This is because background GC for LDP and compactions for Gear and LLC have quite an impact on foreground queries. (4) Fourth, we notice that SplitZNS obtains a greater performance advantage over Gear and LLC in terms of uniform workloads. This is due to the fact that Gear's and LLC's compaction policies remove duplicate key-value pairs more efficiently than the Min-Overlapping-Ratio rule employed by LDP and SplitZNS. After loading the database with a skewed zipfian key distribution, SplitZNS and LDP has a greater number of SSTables, resulting in a lower performance. However, SplitZNS stills shows performance advantage over them. More importantly, SplitZNS is compatible with other optimizations on compaction policy, whereas Gear and LLC cannot be integrated with them. (5) Finally, SplitZNS achieves much lower tail latency in terms of both reading and writing. For example, SplitZNS's reduces 57.2% P99, 52.8% P99.9 overwrite latency and 43.1% P99, 43% P99.9 readrandom latency compared to LDP. In terms of skewed workloads, SplitZNS reduces 39.2% P99 read latency for YCSB-C workload, 39.6% P99 write latency and 24.9% P99 read latency for YCSB-A workload. Compared to Gear and LLC, the experiments show similar trends. These results reveal that by minimizing data migration and employing a more appropriate compaction policy, SplitZNS not only provides higher throughput but also more stable service (i.e., lower tail latency).

5.3 Performance Gain Analysis

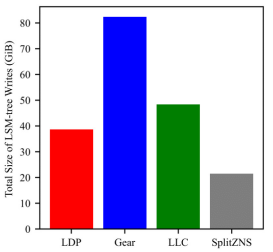
In this section, we analyze data written from LSM-tree and GC, respectively. We choose YCSB-A as the analyzed workload. Figures 11 and 12 show the results. -D means device writes and -U means user writes (i.e., writes from compactions and flushing memtables). We can make the following observations. First, the size of data written by user for LDP and SplitZNS are equal. This is because LDP and SplitZNS share the same compaction policy. LSM-tree writes more data in Gear and LLC since their compaction policies incur a greater write amplification. Their background compactions consume much more I/O bandwidth and thus affect the overall performance, which explains why they exhibit longer tail read latency. The interesting point is that Gear and LLC still show better performance in terms of skewed workloads (i.e., YCSB workloads). The reason is that their compaction policies clean duplicate key-value pairs more efficiently, particularly for skewed



(a) Size of Data Migration normalized to the Size of LSM-tree Writes.



(b) Total Size of User Writes.



(c) Total Size of Device Writes.

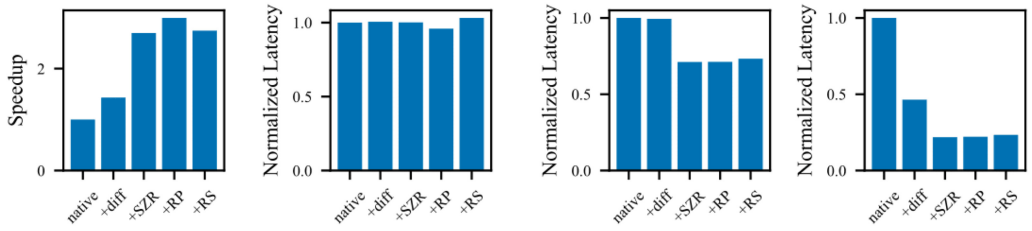
Fig. 12. Data migration.

workloads which have fewer unique keys compared to uniform workloads. Compared to SplitZNS and LLCompaction, their LSM-tree are smaller and have fewer SSTables after YCSB-L workload (37.38 GiB and 2509 files for Gear, 36.41 GiB and 2451 files for LLC, 41.95 GiB and 2,953 files for LDP, 41.97 GiB, and 2,873 files, for SplitZNS). With identical parameters, a smaller LSM-tree usually exhibits higher performance since it triggers fewer compactions when writing and searches fewer SSTables when processing queries.

5.4 Standalone Designs

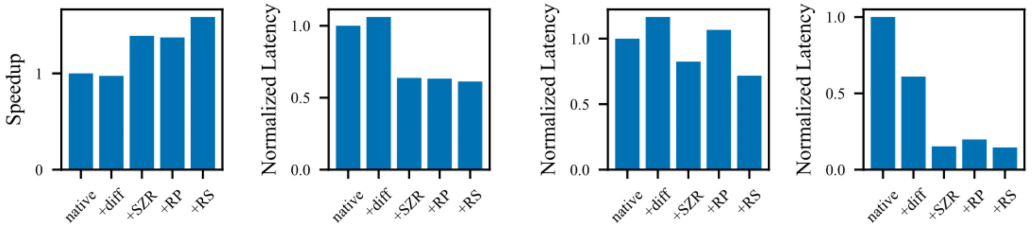
In order to unveil the detailed information of SplitZNS, we break down the performance gain through applying each technique one by one and collect statistics for various internal metrics, including the amount of compaction stalls, the IO performance during compactions and the flash chip utilization. We choose the overwrite and read random workload, which are severely affected by more compactions and GC compared to other workloads, to illustrate the performance gain breakdown clearly. The results under other workloads have similar trends. *native* corresponds to employing small zones for all SSTables. Notice we also use the subzone allocator and chip allocator to alleviate inter-zone inference. *+diff* corresponds to employing small zones only for high-level SSTables (level > 3). *+SZR* corresponds to leveraging SubZone Ring to accelerate write operations on subzones. *+RP* corresponds to leveraging Read Prefetcher to exploit parallelism for read operations. *+RS* corresponds to leveraging Read Scheduler to prevent query being blocked by compactions.

5.4.1 Effectiveness of Standalone Designs. Here, we evaluate each approach in terms of the overwrite and readrandom workloads, as these workloads include just of write or read operations and can more clearly highlight the performance improvement. The results normalized to *native* are shown in Figures 13 and 14. We can make the following observations from the results. First, *native* shows lowest performance. The underlying reason is that the compaction process experiences significant deceleration for ZNS SSDs with only small zones, owing to the inferior performance of small zones compared to large zones. Despite the LSM-tree's ability to execute compactions concurrently, the duration of a single compaction remains significantly prolonged, potentially resulting in significant stalls. Hence, the utilization of small zones can result in a significant decrease in performance, particularly for low-level compactions, which are essential for the performance. Even for high-level compactions, simply using small zones is not a good choice since the LSM-tree does not consistently execute numerous compactions concurrently. Utilizing small zones may lead to a decrease in performance when the quantity of concurrent compactions is limited, as the SSD parallelism is not leveraged. Second, all techniques contribute to the high efficiency of SplitZNS. ① *+diff* increases the write performance by 1.43×, reduces P99.9 latency by 54%. For the read



(a) Normalized Throughput (b) Normalized P50 Latency (c) Normalized P99 Latency (d) Normalized P99.9 Latency

Fig. 13. Performance breakdown with overwrite workload.



(a) Normalized Throughput (b) Normalized P50 Latency (c) Normalized P99 Latency (d) Normalized P99.9 Latency

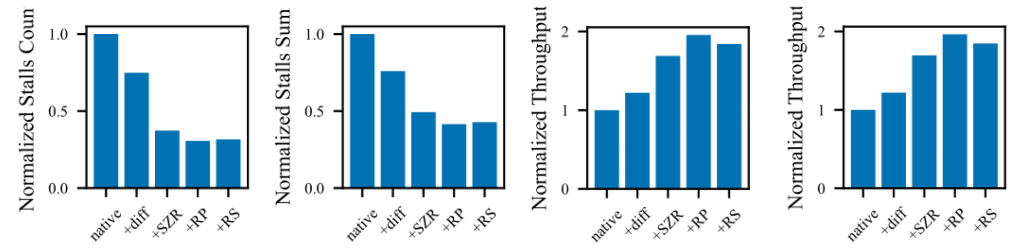
Fig. 14. Performance breakdown with readrandom workload.

workload, *+diff* reduces the P99.9 latency by 39%. ② *+SZR* increases the throughput by 1.88×, reduces P99 latency by 29% and P99.9 latency by 53% under the write workload. Besides, *+SZR* also increases the throughput by 42.7%, reduces the P50 latency by 39%, P99 latency by 30% and P99.9 latency by 75% under the read workload. ③ *+RP* increases the throughput by 1.11× under the write workload. However, it also increases the P99 latency by 29% and P99.9 latency by 29.5% under the read workload. ④ *+RS* eliminates the side effects induced from *+RP*. It further reduces the P99 latency by 33% and P99.9 latency by 27%, making the overall performance even better compared to *+SZR*. We will give a detailed explanation after combining the internal metrics (Section 5.4.2) into considerations.

5.4.2 Internal Metrics. We collect statistics in terms of various internal metrics, including write stalls and compaction throughput when employing different techniques, as shown in Figure 15. The results are also normalized to *native*. Combining these internal metrics into considerations, we give a detailed explanation on the performance advantage from each proposed technique. First, *+diff* reduces write stalls and improves compaction throughput. This is because *+diff* makes low level compactions read and write widezones, which are much faster than subzones. Second, *+SZR* and *+RP* further reduce write stalls and improves compaction throughput. This is because *+SZR* and *+RP* efficiently accelerate the I/O performance of subzones by exploiting the rich parallelism in ZNS SSDs. At last, it seems that *+RS* slows down compactions. This is because *+RS* delays I/O requests from compactions. While *+RS* does not show any improvements on write stalls and compaction throughput, it can reduce query tail latency greatly since it assigns the highest priority to read requests from queries and only sacrifices little throughput.

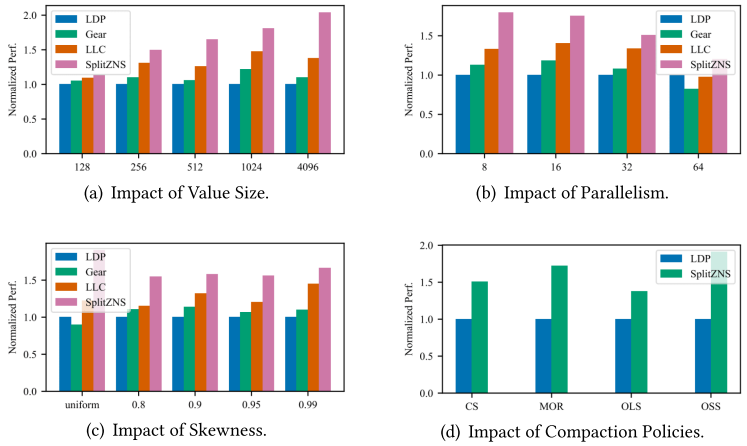
5.5 Sensitivity Study

5.5.1 Value Size. In this section, we study how does SplitZNS perform with various value sizes. We vary value size from 128 bytes to 4,096 bytes and use YCSB-A workload to study the



(a) Norm Count of Write Stalls (b) Norm Time of Write Stalls (c) Norm Read Throughput (d) Norm Write Throughput

Fig. 15. Write stalls and compaction throughput.



(a) Impact of Value Size. (b) Impact of Parallelism.

(c) Impact of Skewness. (d) Impact of Compaction Policies.

Fig. 16. Sensitivity study.

performance advantages. Figure 16(a) shows the results normalized to LDP. We make the following two observations. First, SplitZNS exhibits performance advantages varying from 1.13× to 2.03× compared to LDP, which verifies SplitZNS is effective for various value sizes. Second, SplitZNS performs better with a larger value size. This is because a smaller value size incurs more CPU overhead, making the LSM-tree less I/O intensive. According to our experiments, the percentage of time consumed by merge-sorting keys using CPU during compactions (i.e., $\frac{CompMergeCPU}{Comp}$ in RocksDB) increases by 2.72× (from 25% with 4KiB value size to 68% with 128B value size), making CPU become the performance bottleneck.

5.5.2 Parallelism. To investigate how parallelism affects the performance of SplitZNS, we conduct an experiment with YCSB-A and vary the number of flash chips from 8 to 64. To keep the equal zone size across various numbers of flash chips, we adjust the erase block size. For instance, the erase block size of a 16-chips SSD is double that of a 32-chips SSD. Figure 16(b) demonstrates the results normalized to LDP. We can make the following observation that SplitZNS delivers a greater performance advantage with fewer flash chips (1.79×, 1.75×, 1.50×, and 1.20× compared to LDP for SSDs with 8, 16, 32, and 64 chips, respectively). The reason is that more flash chips bring higher SSD performance, accelerating data migration and mitigating the slowdowns and stalls caused by GC. With the evolution of flash technology, flash chips tend to have a very high density, allowing SSDs with huge capacities to have merely a small number of chips [40]. Thus, SplitZNS's optimizations are applicable on contemporary ZNS SSDs. Besides, SplitZNS also exhibits a

considerable performance advantage on the 64-chips SSD. Therefore, we believe that SplitZNS is scalable in terms of SSD parallelism.

5.5.3 Skewness. To investigate how scalable is SplitZNS in terms of skewness of workloads, we conduct an experiment with YCSB-A. The skewness varies in range (uniform, 0.8, 0.9, 0.95, and 0.99). The number is zipfian constant. The uniform workload is preceded with uniform loading while others are preceded with loading with the corresponding skewness. Figure 16(c) demonstrates the results normalized to LDP. The observation is that SplitZNS outperforms its competitors by from 1.55× to 1.9× under the uniform workload. Under the skewed workloads, SplitZNS achieves up to 1.66×, 1.51×, and 1.34× performance compared to LDP, Gear, and LLC, respectively. The results demonstrate that SplitZNS is scalable in terms of the skewness of workloads.

5.5.4 Compaction Policy. In this section, we evaluate the impact of compaction policies from RocksDB. The evaluated compaction policies are respectively **MinOverlappingRatio** (**MOR**) (MOR, first compact SSTables whose ratio between overlapping size in next level and its size is the smallest), **Compensated Size** (**CS**) (CS, prioritize larger files by size compensated by deletes), **OldestLargestSequence** (**OLS**) (OLS, first compact SSTables whose data's latest update time is oldest), **OldestSmallestSequence** (**OSS**) (OSS, first compact SSTables whose range has not been compacted to the next level for the longest). Figure 16(d) shows the results normalized to LDP. The experimental result shows that SplitZNS increases performance by 1.50× with CS, 1.72× with MOR, 1.37× with OLS, 1.91× with OSS. Different policies invalidate SSTables with different priorities, thus leading to different compaction overhead and GC overhead. Since compaction policy may have a great impact on the performance [11], SplitZNS can alleviate GC overhead without changing the compaction policy, which makes it possible for LSM-tree to enjoy the benefits of both the ZNS SSD and low-cost compaction policy.

5.6 Extended Experiments

Since LSM-tree is designed for HDD at the very beginning, WiscKey [43] proposes key value separation to reduce the write amplification of LSM-tree based on the observation that the random read performance of SSD is much better than that of HDD. WiscKey stores only keys in LSM-tree and stores values in append-only logs called vLogs to reduce the size of LSM-tree and alleviate compaction overhead. To validate SplitZNS's advantage over WiscKey, we conduct an experiment with YCSB-A to compare them. SplitZNS uses the legacy LSM-tree. The WiscKey implementation is from RocksDB's integrated blobdb, which is believed to outperform the original WiscKey. KV separation is evaluated on both ZNS SSD and block SSD. The block SSD is emulated by FEMU with the equivalent hardware configuration and its over-provision space for GC is 20%. In the experiment, the key size is 64B and the value size vary from 200 B to 1 KB. The rest of experimental setting is equivalent as Section 5.2. Figure 17 shows the results normalized to KV separation on the block SSD. Compared to KV separation on block SSD and ZNS SSD, SplitZNS achieves up to 1.7× and 1.3× performance, which demonstrates SplitZNS's effectiveness. The observation is that smaller the value size, larger the benefit of SplitZNS. The reason is that smaller value size results in more write overhead [12] and GC overhead of value logs. Besides, since the number of key value pairs increases, the size of LSM-tree (stores only keys) also increases, which increases the IO overhead as well.

6 RELATED WORKS

To the best of our knowledge, SplitZNS is the first work on combining the architecture of ZNS SSDs and the peculiarity of LSM-tree to build an efficient LSM-tree based key-value store on ZNS SSDs. We discuss related works based on our knowledge from two aspects: LSM-tree and ZNS.

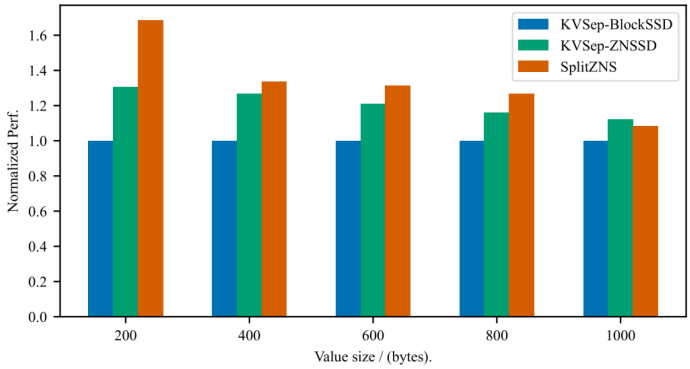


Fig. 17. The effectiveness of splitZNS over KV separation.

6.1 LSM-Tree

LSM-tree has been one of the cornerstones of key-value stores. Since it is first proposed decades ago, there have been many efforts to optimize its performance from a set of aspects. dCompaction [47] delays a compaction until multiple SSTables can be merged together. PebblesDB [48] use partitioned tiering design with vertical grouping to reduce write amplification. MatrixKV [60] exploit emerging non-volatile memory to boost the LSM-tree performance. Vigil-KV [32] achieves consistent ultra-low latency via integration with SSDs with **Predictable Latency Mode** (**PLM**) [16] interface. Some works [27, 55, 56] aggressively implement an LSM-tree at the firmware level. These works are orthogonal to SplitZNS and complement it.

6.2 ZNS

Fast ZNS SSDs spur researchers to build new ZNS-based storage systems. ZenFS [4, 5] provides a filesystem plugin for RocksDB to enable LSM-tree with the ability to leverage the advantages of ZNS SSDs. To alleviate GC for LSM-tree on ZNS SSDs, LLC [29] proposes a novel compaction policy and CAZA [33] proposes a zone allocation policy, both of which similarly leverage the difference of SSTable lifetime based on key ranges. For ZNS SSDs with only small zones, ZNS-MQ [2] detects the internal interferences of zones through profiling and allocates suitable zones for upper applications. In Section 5.4, we prove that SplitZNS achieves higher performance by tighter integration with LSM-tree. ZNSwap [3] optimized Linux swap memory subsystem by exploiting the advantages of ZNS SSDs. Tehrany et al. [49, 50] systematically evaluate various systems on ZNS SSDs. ZonedStore [44] uses ZNS SSDs to build a concurrent cache system for cloud data storage. Choi, Gunhee et al. [15] borrow the idea from LSM-tree and propose a novel and general-purpose garbage collection scheme. Li, Jinhong et al. [37] proposes a hybrid zoned storage key-value system (i.e., using ZNS SSDs as the performance layer and SMR disks as the capacity layer). ZNS+ [23] proposes a new log-structured-filesystem-aware ZNS interface which offloads the data copy operations to the SSD to accelerate filesystem performance. These works reveal the future directions and possible opportunities for ZNS study, which are in-depth and thought-provoking.

7 DISCUSSION

7.1 Key Value Separation on ZNS

Key-value separation, which stores only keys in LSM-tree and stores values in append-only value logs (vLogs) is proposed as a means of accommodating the increased performance of random reads for SSDs [12, 38, 43]. The introduction of ZNS poses novel challenges for key-value separation. The size of vLog, akin to zone size, has the potential to exert a noteworthy influence on the performance,

which is possible to be addressed through SplitZNS. However, the GC process of vLogs differs from the compactions and the lifetime of vLogs is different from those of SSTables. Consequently, the acceleration techniques that were closely associated with the traditional LSM-tree may be rendered obsolete. Moreover, it has been observed that the efficacy of key-value separation is comparatively lower when dealing with workloads comprising of values of small sizes [12] and workloads that are predominantly dominated by range queries [38]. In order to address these challenges, we should pursue a meticulous redesign of SplitZNS, which is part of our future work.

7.2 Wear Leveling

Although SplitZNS erases blocks separately, the existing **wear-leveling** (**WL**) techniques [45] are compliant with SplitZNS to guarantee the SSD's the lifetime. Similar to SplitZNS, recent studies on wear leveling [14, 54] also erase blocks separately, and a real ZNS SSD also erases blocks separately when resetting zones, which confirms that SplitZNS imposes an insignificant impact on wear leveling. Since the comprehensive parameters and statistics such as the initial P/E cycles for erase blocks, the P/E cycles threshold to enable wear leveling, and the P/E cycles distribution pattern are not available for us, conducting a systematic evaluation is currently challenging. It will be incorporated into our future efforts.

8 CONCLUSION

The ZNS storage interface shows potential as a developing technology, as it aligns with the LSM-tree approach based on their comparable sequential write behaviors. We investigate the issue of GC for LSM-trees on ZNS SSDs, along with the limitations of current approaches to address this issue. SplitZNS is proposed to address the challenge through reducing the zone size through the incorporation of splitzone and subzone concepts. SplitZNS appropriately leverage splitzones and widezones, and enhance their effectiveness by taking advantage of the unique characteristics of LSM-tree. The results of experiments confirm the efficacy and effectiveness of SplitZNS.

REFERENCES

[1] Ahmed Izzat Alsalibi, Sparsh Mittal, Mohammed Azmi Al-Betar, and Putra Bin Sumari. 2018. A survey of techniques for architecting SLC/MLC/TLC hybrid Flash memory–based SSDs. *Concurrency and Computation: Practice and Experience* 30, 13 (2018), e4420.

[2] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. 2022. What you can't forget: Exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 79–85.

[3] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. 2022. {ZNSwap}:{un-Block} your swap. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1–18.

[4] Matias Bjørling. 2022. *ZenFS*. Retrieved December 22, 2022 from https://github.com/westerndigitalcorporation/zenfs.

[5] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 689–703.

[6] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. {LightNVM}: The linux {Open-Channel}{SSD} subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*. 359–374.

[7] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE* 105, 9 (2017), 1666–1704.

[8] Mark Callaghan. 2016. *Compaction Priority in RocksDB*. Retrieved December 22, 2022 from http://smalldatum.blogspot.com/2016/02/compaction-priority-in-rocksdb.html.

[9] Jorge Campello. 2015. SMR: The next generation of storage technology. In *Proceedings of the Storage Development Conference (SDC 15)*, Vol. 21.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[11] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. 2019. LDC: A lower-level driven compaction method to optimize SSD-oriented key-value stores. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 722–733.

[12] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. {HashKV}: Enabling efficient updates in {KV} storage via hashing. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[14] Li-Pin Chang, Tung-Yang Chou, and Li-Chun Huang. 2013. An adaptive, low-cost wear-leveling algorithm for multi-channel solid-state disks. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 3 (2013), 1–26.

[15] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. 2020. A new {LSM-style} garbage collection scheme for {ZNS}{SSDs}. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[16] NVMe Committe. 2019. *NVM Express Base Specification 1.4*. Retrieved December 22, 2022 from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.

[18] Ting Yao Damien Le Moal. 2022. *LibZBD*. Retrieved December 22, 2022 from https://github.com/westerndigitalcorporation/libzbd.

[19] Siying Dong. 2016. *Option of Compaction Priority*. Retrieved December 22, 2022 from http://rocksdb.org/blog/2016/01/29/compaction_pri.html.

[20] DRAMeXchange. 2014. *DRAMeXchange*. Retrieved December 22, 2022 from https://dramexchange.com/.

[21] Congming Gao, Liang Shi, Chun Jason Xue, Cheng Ji, Jun Yang, and Youtao Zhang. 2019. Parallel all the time: Plane level parallelism exploration for high performance SSDs. In *Proceedings of the 2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. 172–184. DOI: https://doi.org/10.1109/MSST.2019.000-5

[22] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. 2018. Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 469–481.

[23] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of the 15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 147–162.

[24] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 2012. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *IEEE Transactions on Computers* 62, 6 (2012), 1141–1155.

[25] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing*. 96–107.

[26] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: A raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[27] Junsu Im, Jinwook Bae, Chanwoo Chung, and Arvind and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 173–187.

[28] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. 2022. Improving the reliability of next generation {SSDs} using {WOM-v} Codes. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST 22)*. 117–132.

[29] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-leveling lsm-tree compaction for ZNS SSD. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 100–105.

[30] Ali Khakifirooz, Sriram Balasubrahmanyam, Richard Fastow, Kristopher H. Gaewsky, Chang Wan Ha, Rezaul Haque, Owen W. Jungroth, Steven Law, Aliasgar S. Madraswala, Binh Ngo, V Naveen Prabhu, Shantanu Rajwade, Karthikeyan Ramamurthi, Rohit S. Shenoy, Jacqueline Snyder, Cindy Sun, Deepak Thimmegowda, Bharat M. Pathak, and Pranav Kalavade. 2021. 30.2 A 1Tb 4b/Cell 144-Tier Floating-Gate 3D-NAND flash memory with 40MB/s program throughput and 13.8Gb/mm2 bit density. In *2021 IEEE International Solid- State Circuits Conference (ISSCC'21)*. 424–426. DOI: https://doi.org/10.1109/ISSCC42613.2021.9365777

[31] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards {SLO} complying {SSDs} through {OPS} isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*. 183–189.

[32] Miryeong Kwon, Seungjun Lee, Hyunkyu Choi, Jooyoung Hwang, and Myoungsoo Jung. 2022. {Vigil-KV}:{Hardware-Software}{Co-Design} to integrate strong latency determinism into {Log-Structured} merge {Key-Value} stores. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 755–772.

[33] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. 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*. 93–99.

[34] Sanjay Ghemawat and Jeff Dean. 2014. Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values.

[35] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The case of femu: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, CA.

[36] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. lODA: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 263–279.

[37] Jinhong Li, Qiuping Wang, and Patrick P. C. Lee. 2022. Efficient LSM-tree Key-value data management on hybrid SSD/HDD zoned storage. *arXiv preprint arXiv:2205.11753* (2022).

[38] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated {Key-Value} storage management for balanced {I/O} performance. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.

[39] Torvalds Linus. 2022. *GNU Linux Kernel*. Retrieved December 22, 2022 from https://github.com/torvalds/linux.

[40] Chun-Yi Liu, Yunju Lee, Wonil Choi, Myoungsoo Jung, Mahmut Taylan Kandemir, and Chita Das. 2021. GSSA: A resource allocation scheme customized for 3D NAND SSDs. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 426–439.

[41] Fiona Liu. 2022. *Inspur Information Launches a New Generation of Enterprise NVMe SSD*. Retrieved December 22, 2022 from https://en.inspur.com/en/news-2022/2614704/index.html.

[42] Chih-Yuan Lu. 2012. Future prospects of NAND flash memory technology–the evolution from floating gate to charge trapping to 3D stacking. *Journal of Nanoscience and Nanotechnology* 12, 10 (2012), 7604–7618.

[43] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[44] Yanqi Lv, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, Liming Fang, Yuanjin Lin, and Kuankuan Guo. 2022. ZonedStore: A concurrent zns-aware cache system for cloud data storage. In *Proceedings of the 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1322–1325.

[45] Inc. Micron Technology. 2019. *Wear Leveling in NAND Flash Memory*. Retrieved December 22, 2022 from https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2961_wear_leveling_in_nand.pdf.

[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[47] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[49] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. 2020. Exploring performance characteristics of ZNS SSDs: Observation and implication. In *Proceedings of the 2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–5.

[50] Nick Tehrany and Animesh Trivedi. 2022. Understanding NVMe zoned namespace (ZNS) flash SSD storage devices. *arXiv preprint arXiv:2206.01547* (2022).

[51] TiKV. 2023. *TiKV Configuration*. Retrieved April 1, 2023 from https://docs.pingcap.com/tidb/dev/tikv-configuration-file.

[52] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. Noftl-kv: Tackling write-amplification on kv-stores with native storage management. In *Proceedings of the Advances in Database Technology-EDBT 2018: 21st International Conference on Extending Database Technology*. University of Konstanz, University Library, 457–460.

[53] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems*. 1–14.

[54] Jiaojiao Wu, Jun Li, Zhibing Sha, Zhigang Cai, and Jianwei Liao. 2022. Adaptive switch on wear leveling for enhancing I/O latency and lifetime of high-density SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4040–4051.

[55] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2018. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 563–568.

[56] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2020. Integrating LSM Trees with multichip flash translation layer for write-efficient KVSSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 1 (2020), 87–100.

[57] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail flash: Near-Perfect Elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 15–28. Retrieved from https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan.

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

[59] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. 2019. {GearDB}: A {GC-free}{Key-Value} Store on {HM-SMR} drives with gear compaction. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*. 159–171.

[60] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. {MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} Based {KV} stores with matrix container in {NVM}. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.

[61] Yiwen Zhang, Ting Yao, Jiguang Wan, and Changsheng Xie. 2022. Building GC-free key-value store on HM-SMR drives with ZoneFS. *ACM Transactions on Storage (TOS)* 18, 3 (2022), 1–23.

[62] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. 2021. {Remap-SSD}: Safely and efficiently exploiting {SSD} address remapping to eliminate duplicate writes. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21)*. 187–202.

Received 9 January 2023; revised 26 June 2023; accepted 2 July 2023