Ensuring Compaction and Zone Cleaning Efficiency through Same-Zone Compaction in ZNS Key-Value Store

Sungjin Byeon[1], Joseph Ro[1], Jun Young Han[2], Jeong-Uk Kang[2], and Youngjae Kim [1,†]
[1]Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
[2]Samsung Electronics Co.
{sjbyeon, josephro12, youkim}@sogang.ac.kr, {jy0.han, ju.kang}@samsung.com

***Abstract*—In this paper, we address inefficiencies in compaction algorithms within ROCKSDB for ZNS SSD, with a focus on the performance and lifetime issues arising from excessive copying of value data. To overcome these challenges, we propose an innovative compaction algorithm that considers the SSTable layout, termed *Same-Zone Compaction*. This algorithm aims to selectively choose SSTables predominantly located within the same zone during the compaction process, maximizing the generation of deleted SSTables in that zone after the merge-sort operation. Following this, newly created SSTables are strategically allocated to zones expected to have high deletion rates in future compactions, enhancing the frequency of Same-Zone Compaction. We introduce two algorithms: the Zone-Aware Compaction Victim Selection Algorithm (ZACA) and the Compaction-Aware Zone Allocation Algorithm (CAZA). Implemented in ROCKSDB v7.4 with ZenFS v2.1, our comprehensive evaluation using micro-benchmarks and YCSB benchmarks shows that ZACA and CAZA together reduce Zone Cleaning overhead by 80%, improve PUT(k,v) performance by 1.16x, and extend the lifespan of ZNS SSDs by 8.6%.**

I. INTRODUCTION

The Zone Namespace SSD (ZNS SSD) [1]–[3] divides the logical address space into fixed-size zones, each only allowing sequential writes to optimize NAND flash memory utilization [4]. This coarse-grained management at the zone level eliminates the need for over-provisioning space and in-device garbage collection [3], [5]. However, to create free space on ZNS SSDs, the ZNS file system operated by the host (e.g., ZenFS [6]–[8]) must perform Zone Cleaning (ZC) [9]. ZC involves replicating valid data from one zone to another and then using the zone-reset command to delete the source zone, turning it into an empty zone [1], [7], [10], [11]. This process of copying valid data during ZC introduces the Write Amplification (WA) issue [12].

Meanwhile, LSM-tree-based key-value stores like ROCKSDB [13] are considered optimal for ZNS SSDs due to their log-structured tree structure, which facilitates sequential writes [12], [14]. LSM-TREE operates on top of the ZNS file system mentioned earlier and performs compaction by merge-sorting SSTable/SST files, deleting invalid data, and creating an ordered tree composed of unique key-value pairs. This compaction process, similar to ZC, involves significant reading and writing of SSTables, which introduces the Write Amplification (WA) issue.

†Y. Kim is the corresponding author.

In addressing WA concerns, ZenFS has implemented the Lifetime-based Zone Allocation (LIZA) algorithm. This algorithm estimates the lifetime of SSTables based on the level of LSM-TREE at which the SSTable is placed and places SSTables with same lifetime into the same zone. However, relying solely on the LSM-TREE level to predict SSTable lifetimes can lead to inaccuracies, resulting in the co-location of SSTables with different actual lifetimes in the same zone. Consequently, these SSTables may not be deleted simultaneously during compaction, leading to increased WA due to the need to copy valid data during Zone Cleaning.

Additionally, ROCKSDB selects SSTables for compaction based solely on their size, without considering the layout of SSTables (i.e., information about the zones where SSTables are placed). This results in two issues. Firstly, compaction involving multiple SSTables triggers read I/O from various zones. Since SSTables are scattered across multiple zones, this results in random read I/O, which is slower than sequential reads from SSTables within a single zone [15], [16] (Refer to Figure 4). Consequently, the execution speed of compaction is degraded due to this random read I/O. Secondly, after compaction, the SSTables involved in the compaction process need to be deleted. However, the deleted files are scattered across multiple zones. As will be explained later, during ZenFS's Zone Cleaning (ZC), this dispersion of SSTable files lead to the generation of numurous valid data copies, thereby diminishing the efficiency of ZC.

To address these challenges, we propose a compaction technique that takes into account the layout of SSTable in ROCKSDB. The first key idea involves selecting a group of SSTables located in the same zone for merge-sort. This strategy, which we refer to as *Same-Zone Compaction*, aims to maximize the number of SSTables deleted in the zone after the merge-sort operation. The second key idea is to allocate a zone for a newly created SSTable after the merge-sort in compaction, with the anticipation that the zone has a number of SSTables likely to be deleted together in the future. This strategy is designed to facilitate numerous Same-Zone Compactions. To implement these ideas, we propose the following two algorithms.

Firstly, we introduce the Zone-Aware Compaction Victim Selection Algorithm (ZACA), which selects SSTables based on their zone layout and prioritizes those with the highest S_{same} (§V). The S_{same} is a metric that quantifies how many

SSTables participating in a compaction are located in the same zone. However, ZACA presents challenges such as increased merge-sort time and write stall time compared to the baseline algorithm, named the Size-based Compaction Victim Selection Algorithm (SICA). ZACA shows positive effects on ZC Efficiency. However, it can increase the size of compaction victims, leading to an increase in write stalls. To address these challenges, we propose the Adaptive Compaction Controller (ACC). The ACC dynamically switches between invoking SICA and ZACA based on a predefined free space threshold. When free space is under the threshold, ZACA turns on as Zone Cleaning is urgent. It reduces the valid data copy overhead of ZenFS's ZC and RocksDB's write stall, thereby enhancing the performance of PUT(k, v) operations.

Secondly, we present the Compaction-Aware Zone Allocation algorithm (CAZA) (§VI). Initially introduced in our previous study [17], CAZA strategically allocates zones where many SSTables are likely to be deleted simultaneously in the future to newly created SSTables. The core strategy idea is to allocate the zone with the highest overlap of the key-range among SSTables. This paper extends the CAZA algorithm by incorporating the size of SSTable and leveled compaction score (S_{level}), which are detailed in Section VI. For clarity, we will refer to the enhanced version simply as CAZA.

The key contribution of this paper is as follows:

- We closely examined the compaction process in RocksDB for ZNS SSDs, defined the necessity of Same-Zone Compaction, and proposed a compaction algorithm that considers the layout of SSTables (§III).
- To maximize the benefits of Same-Zone Compaction, we introduced ZACA (§V), a compaction victim selection algorithm.
- Furthermore, to mitigate the issue of prolonged compaction execution times associated with ZACA, we proposed an Adaptive Compaction algorithm that can operate in hybrid mode alongside SICA, enhancing overall efficiency (§V-B).
- We proposed CAZA (§VI), a compaction-aware zone allocation algorithm, designed to induce Same-Zone Compaction in the future SSTables created after compaction.
- We implemented both ZACA and CAZA by modifying RocksDB v7.4 and ZenFS v2.1.

Extensive evaluations demonstrate that employing both ZACA and CAZA effectively reduces ZenFS's ZC overhead by 80%, leading to a significant 16% improvement in the performance of PUT(k, v) across both micro-benchmarks and YCSB benchmarks. Furthermore, this approach results in an 8.6% reduction in the zone-reset count, contributing to an extended lifetime of ZNS SSDs.

II. Background

A. LSM-tree-based Key-value Stores

RocksDB's LSM-Tree structure [14] consists of both memory components and storage components. When a user initiates a PUT(k, v) operation, the key-value pair is stored in an in-memory data structure called the Memtable. Once the Memtable reaches full capacity, a flush operation is triggered, which transforms the Memtable into an on-disk data structure known as the Sorted String Table (SSTable).

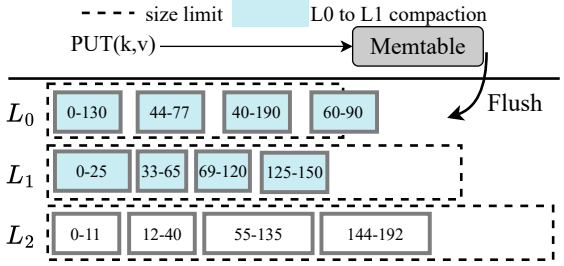


Fig. 1: Description of RocksDB's LSM-tree.

The SSTables written to disk are managed within the hierarchical level structure of the LSM-Tree. SSTables at level 0 (L_0) are unsorted, while those at level 1 and beyond are sorted through the compaction process. Each level in this structure has a defined size limit, and as the levels increase, so do these limits. When the size of SSTables at any level exceeds its predefined size limit, a compaction process is initiated to maintain the structural integrity and ensure the lookup performance of the LSM-Tree.

The compaction process involves merge-sorting the SSTables in L_i with those in L_{i+1} that have overlapping keys. The SSTables involved in this merge-sort are referred to as compaction victims. The result of this merge-sort is the creation of new SSTables in L_{i+1}. Subsequently, the SSTables in both L_i and L_{i+1} that were involved in the compaction are deleted. SSTables at level 1 and beyond have non-overlapping key ranges, forming a disjoint set. Figure 1 illustrates the data structure and its operations.

Expensive L_0-to-L_1 compaction: In the process of L_0-to-L_1 compaction, numerous SSTables at both L_0 and L_1 are involved in a merge-sort operation, since SSTables at L_0 are initially unsorted [18]. If an SSTable at L_0 is chosen as a victim for compaction, this leads to the inclusion of many SSTables at L_1 whose key ranges overlap with the SSTable selected as the victim. Consequently, L_0-to-L_1 compaction takes a substantial amount of time and blocks incoming PUT(k, v) requests, thus emerging as the primary cause of write stalls [19]–[21].

B. Algorithm for Choosing SSTables for Compaction

The selection of SSTables for merge-sort during compaction directly impacts the efficiency of the compaction process. Therefore, an efficient algorithm for choosing SSTables is crucial. RocksDB adopts an algorithm that selects SSTables (compaction victims) for compaction based on the sizes of the SSTables. This algorithm is referred to as the Size-based Compaction Victim Selection Algorithm (SICA).

SICA operates as follows:

- (Step 1) SICA calculates the extent to which the sum of sizes of SSTables at each level exceeds the size limit. We denote this value as the leveled compaction score (S_{level}). Specifically, S_{level} is computed by dividing the sum of sizes

of SSTables at each level by the size limit of that level. Compaction is initiated first at the level with the highest S_{level} with exceeding 1.
- (Step 2) Let L_i be the level with the highest S_{level}. In this scenario, the largest SSTable in L_i, along with SSTables in the next level (L_{i+1}) whose key ranges overlap with it, are selected as compaction victims.
- (Step 2-1) If multiple SSTables are the largest in L_i, the algorithm selects the SSTable in L_i that has the smallest total size of overlapping SSTables in L_{i+1}. This choice aims to minimize the number of key-value pairs participating in the merge-sort, thereby reducing the compaction time.
- (Step 2-2) If there are no SSTables in L_{i+1} whose keys overlap with those in L_i, the metadata and level of the SSTable in L_i are changed to L_{i+1} without performing merge-sort and write I/O [18]. In this manner, SICA can significantly reduce the size of L_i while simultaneously decreasing the WA of L_{i+1}. However, as detailed in Section III, SICA selects SSTables without considering the zone layout of SSTables (the zones where SSTables are arranged), which results in suboptimal compaction efficiency, adversely affecting both system performance and device lifespan.

C. Zone Management Middleware

To operate an LSM-tree-based key-value store on ZNS SSDs, middleware or a file system that supports ZNS is required. For RocksDB, this is achieved through ZenFS [1], [6], [7], which is a user-level file system designed to facilitate this compatibility. ZenFS performs two major functions: Zone Allocation and Free Space Reclamation.

1) Zone Allocation: ZenFS performs Zone Allocation, a feature specifically designed to place SSTables in appropriate zones. When a new SSTable is generated due to a flush or compaction, ZenFS not only allocates a zone for this SSTable but also manages the metadata associated with it.

Specifically, ZenFS implements the Lifetime-based Zone Allocation algorithm (LIZA), which assigns lifetimes to SSTables based on the varying compaction frequency characteristics at different levels. As discussed in Section II-A, RocksDB sets increasing size limits for each level as the level number grows. When the size limit of a level L_i is exceeded, compaction from L_i to L_{i+1} is triggered. This means that lower levels, with smaller size limits, experience more frequent compactions, whereas higher levels, with larger size limits, undergo compactions less frequently.

LIZA allocates zones in the following manner: It uses four different lifetime hint values; Short (1), Medium (2), Long (3), and Extreme (4). For ease of explanation, these are denoted as integer values. Each new SSTable is assigned a lifetime hint based on its destination level. SSTables destined for levels L_0 or L_1 are assigned a Medium (2) hint. For L_2, a Long (3) hint is assigned. SSTables for L_3 and higher levels receive an Extreme (4) hint. Short (1) is assigned to data other than SSTables, such as the Write-Ahead Log (WAL) and Manifest, which contains summary information of the LSM-Tree.

In LIZA, each zone also has a lifetime hint, determined by the hint of the first SSTable written to the zone. Once the lifetime hint h of an SSTable is determined, LIZA searches for the zone with the smallest lifetime hint value that is equal to or greater than h and places the SSTable there. If there is no matching zone, LIZA allocates an empty zone and sets its lifetime hint to h. If there are no empty zones available, SSTables are placed in the zone with the closest lifetime hint.

2) Free Space Reclamation: As zones fill up with invalid SSTables, the availability of free zones in the ZNS SSD diminishes. Consequently, ZenFS initiates a process known as Zone Cleaning (ZC). The ZC process involves the following steps: (1) selection of a victim zone with the fewest valid SSTables; (2) copying of valid SSTables to another zone; and (3) setting all SSTables within the victim zone to a free state by erasing the zone using zone-reset commands.

Copying valid SSTables is a time-consuming process, requiring numerous NAND page read and write I/O operations [11], [22], [23]. Consequently, a substantial delay in providing sufficient free space can occur, especially when there is a significant amount of valid data in the victim zones. In such cases, I/O operations issued by RocksDB's internal processes, such as compaction and flush, may be blocked until an adequate amount of free space is provided, resulting in what is known as *I/O blocking* [5], [10]. Furthermore, copying valid SSTables during ZC can contribute to increased WA [12], which in turn, adversely affects the overall efficiency and lifespan of the ZNS SSD.

III. Motivation

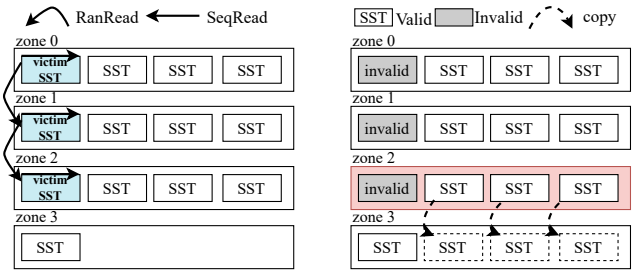
A. Problems with RocksDB's Compaction Algorithm

RocksDB selects SSTables (compaction victims) for merge-sort during compaction without considering their zone placement. Consequently, it grapples with the challenge of suboptimal optimization from both performance and device lifetime perspectives. To provide a more detailed explanation of these issues, first we define two compaction types – *Zone-Across Compaction* and *Same-Zone Compaction*.

> *Definition 1:* ***Zone-Across Compaction*** *refers to the scenario where the SSTables selected as compaction victims are distributed across multiple different zones.*
> *Definition 2:* ***Same-Zone Compaction*** *refers to the scenario where the SSTables selected as compaction victims are all located within a single zone.*

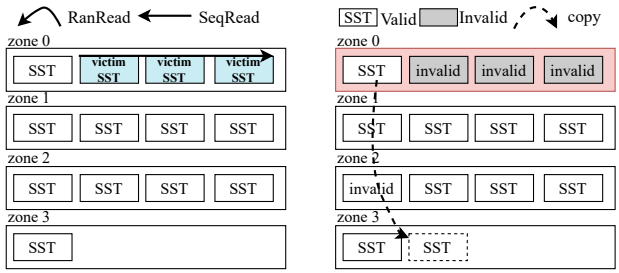
Figure 2 and Figure 3 effectively illustrate the comparison between the operations of Zone-Across Compaction and Same-Zone Compaction, and their respective ZC costs.

Figure 2(a) illustrates an example of Zone-Across Compaction. In Figure 2(a), three victim SSTables are distributed across distinct zones. In such cases, reads must be performed on SSTables belonging to different zones during the compaction process. As zones are not necessarily adjacent in physical space, random reads are necessary to access SSTables



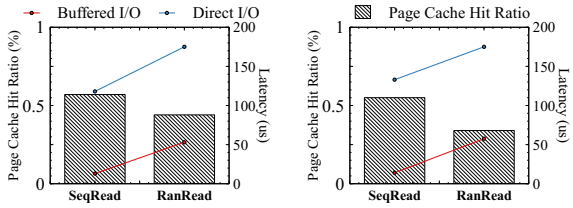
(a) Zone-Across Compaction (b) Zone Cleaning

Fig. 2: Zone-Across Compaction.



(a) Same-Zone Compaction (b) Zone Cleaning

Fig. 3: Same-Zone Compaction.



(a) Queue Depth = 1 (b) Queue Depth = 256

Fig. 4: Comparisons of latency and page cache hit ratios for sequential or random read patterns on ZNS SSDs.

during the merge-sort operation. Following the completion of the merge-sort, all SSTables that participated become invalidated/deleted.

Furthermore, ZenFS manages the space of zones on ZNS SSDs. When the number of free zones falls below a specific threshold, resulting in a space shortage, ZenFS initiates ZC. Figure 2(b) illustrates a scenario where ZC is taking place. Suppose the victim zone selection algorithm for ZC chooses the zone with the fewest valid SSTables. In the given example, zones 0, 1, and 2 each contain an equal number of valid SSTables. Consequently, zone 2 is selected as the victim and undergoes a reset. The cost associated with resetting zone 2 involves copying the three valid SSTables to another zone and performing a single zone reset.

Figure 3(a) provides an example of Same-Across Compaction. In Figure 3(a), the three SSTables chosen as compaction victims are all located in zone 0. Since all SSTables are sequentially arranged within a single zone, the merge-sort can be executed with sequential reads, thus enhancing the efficiency of the process. Subsequently, if Zone Cleaning (ZC) is performed following the previously described victim selection algorithm, zone 0 is designated as the victim. Figure 2(b) illustrates this ZC scenario. The cost of ZC involves copying one SSTable from zone 0 to another zone and performing a single zone reset.

Since the ZNS SSD leverages high I/O parallelism within the device, it achieves higher throughput and lower response times for sequential read access patterns compared to random access patterns [16], [24]. Furthermore, when the Linux kernel detects a sequential I/O pattern, it employs read-ahead techniques to prefetch the subsequent block address into the page cache [15]. Thus, prefetching enhances the page cache hit ratio for sequential read patterns, thereby reducing disk I/O and improving read performance.

To further investigate, we assessed the I/O performance of ZNS SSDs under workloads characterized by sequential read and random access patterns, employing the Flexible I/O Tester (FIO) [25]. The configuration of the ZNS SSD used in our experiments is detailed in SectionVII-A. The outcomes are presented in Figure 4.

In Figure 4, when using Buffered I/O, regardless of the queue depth, the page cache hit ratio for sequential read patterns is 15% higher than that for random read patterns, and the latency is 75% lower. Additionally, even in the case of direct I/O without using the page cache, sequential read patterns exhibit 31% lower latency compared to random read patterns.

In Same-Zone Compaction, since all SSTables are located within the same zone, sequential read patterns can be more effectively utilized for reading SSTables compared to Zone-Across Compaction. Consequently, Zone-Across Compaction exhibits less efficient read performance due to its reliance on random read patterns during compaction, which can lead to reduced compaction processing speeds.

Moreover, Zone-Across Compaction involves deleting SSTables scattered across multiple zones, leading to zones with a mix of valid and invalid data. This situation increases the amount of valid data that must be copied from the selected victim zone during subsequent ZC. As a result, these copies of valid data prolong the execution time of ZC and extend the blocking period of I/O operations. Additionally, the extensive copying of valid data results in write amplification, which adversely impacts the device's lifetime.

> *Observation 1:* Zone-Across Compaction requires a higher number of random reads compared to Same-Zone Compaction.
> *Observation 2:* Zone-Across Compaction results in a greater volume of valid data being copied during Zone Cleaning (ZC) compared to Same-Zone Compaction.

Additionally, the Size-based Compaction Victim Selection Algorithm (SICA) introduced in Section II for RocksDB focuses on minimizing the execution time of compaction through merge-sort and reducing the size of L_i that exceeds the specified size limit. Consequently, this often leads to cases

of Zone-Across Compactions.

> *Observation 3:* RocksDB's Size-based Compaction Victim Selection Algorithm (SICA) frequently results in Zone-Across Compactions due to its failure to account for the placement of compaction victims (Victim SSTables) within zones (SSTable layouts).

B. Inefficient Zone Allocation in ZenFS and Its Limitations

ZenFS implements the Lifetime-based Zone Allocation algorithm (LIZA). Our detailed analysis of LIZA's algorithm and the compaction process has confirmed that it significantly reduces the occurrence of Same-Zone Compaction.

During L_i-to-L_{i+1} compaction, the victims selected for compaction include not only SSTables from L_i but also SSTables from L_{i+1}. However, LIZA organizes SSTables with similar lifetimes based on their levels, placing those with the same lifetime in the same zone. Consequently, if the lifetimes of L_i and L_{i+1} SSTables differ, SSTables from L_i and L_{i+1} will unavoidably be placed in different zones. For example, L_1 is set to 'Medium,' and L_2 is set to 'Long,' and those SSTables are placed in different zones. It implies that L_1-to-L_2 compactions are always Zone-Across Compaction. In other words, LIZA allocates zones based solely on the lifetime of SSTables, disregarding fact that compaction victims are selected in two levels and deleted after compaction later. Consequently, LIZA is unable to place SSTables with levels within the same zone.

> *Observation 4:* LIZA induces Zone-Across Compaction by separating SSTables with varying lifetimes into different zones, even though these SSTables undergo compaction together.

Compaction is a process that performs a merge-sort with key-overlapping SSTables, as discussed in Section II-A. Specifically, during the L_i-to-L_{i+1} compaction process, a SSTable(S) from level L_i is selected as a pivot, and SSTables whose key ranges overlap with SSTable(S) from L_{i+1} are chosen as compaction victims. In other words, there's a higher probability that SSTables with overlapping/close key-ranges will undergo compaction together. However, LIZA places SSTables sorely based on lifetime determined by the level, disregarding key-ranges of each SSTable.

> *Observation 5:* LIZA induces frequent Zone-Across Compaction because it places SSTables in the same zone even though the SSTables have non-overlapping key ranges.

In RocksDB, L_i-to-L_{i+1} compaction selects the largest SSTable from L_i as compaction victims, as discussed in the Section II-B. The probability of an SSTable from L_i being chosen as the compaction victim for L_i-to-L_{i+1} compaction increases with its size. RocksDB sets the default size of an SSTable to 64MB. We have observed that when compaction victims include SSTables with overlapping keys or deleted keys, the newly created SSTables after merge-sort during compaction can be smaller than 64MB. However, LIZA allocates zones without any consideration for the sizes of SSTables. By placing SSTables in the same zone without distinguishing their sizes, LIZA inadvertently mixes large and small SSTables into the same zone. This lowers the probability of invoking Same-Zone Compaction.

TABLE I: Distribution of SSTable size.

SSTable (MB)	[0-1]	[2-32]	[33-63]	[64]
# of SSTables	38	500	960	3714

Table I presents the distribution of SSTable sizes generated when conducting writes of 300GB or more using RocksDB's db_bench [26]. Approximately 3,700 SSTables were created with the default size of 64MB, while around 1,500 SSTables were generated with sizes smaller than 64MB. Details regarding the experimental setup and workload can be found in Section VII-A.

> *Observation 6:* LIZA induces frequent Zone-Across Compaction by co-locating large and small SSTables within the same zone, without discerning their respective probabilities of being chosen as compaction victims.

LIZA assumes that compaction frequency varies with the size limit of levels and predicts SSTable lifetime based on this assumption. Specifically, LIZA assumes that lower levels, having smaller size limits, experience more frequent compactions. Consequently, it assumes that SSTables in lower levels have shorter lifetimes, while those in higher levels have longer lifetimes. However, instead of such rough estimates, a more accurate prediction of compaction frequency at a specific time for a level is based on a score metric (S_{level}), which is the ratio of the cumulative size of SSTables in a level to its size limit.

For instance, suppose that at time T, the S_{level}[3] representing the score (S_{level}) of L_3 is the highest and greater than or equal to 1. The L_3-to-L_4 compaction occurs frequently, prioritized until time $T + N$ when S_{level}[3] becomes lower than or equal to the S_{level} of other levels. Thus, compaction frequency is not consistently determined by the size limit of the level but can vary based on the S_{level} at time T. Despite this, the LIZA algorithm consistently assigns "Extreme" to the SSTables of L_3 regardless of their S_{level}, and at time T, it assigns "Medium" to the SSTables of L_1 with S_{level} lower than L_3.

> *Observation 7:* LIZA inaccurately predicts the lifetime of SSTables due to an imprecise estimation of compaction frequency.

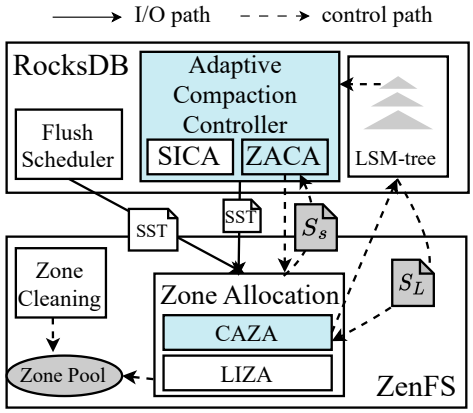


Fig. 5: The software architecture of our proposed system. The areas shaded in blue represent the newly added components.

In summary, LIZA frequently induces Zone-Across Compaction in the LSM-TREE by inaccurately considering parameters such as level, key-range, SSTable size, and S_{level} associated with compaction. The increased occurrence of Zone-Across Compaction due to LIZA results in an elevated random read pattern during compaction. Additionally, the frequent Zone-Across Compaction induced by LIZA leads to an increase in the number of valid SSTable copies and I/O blocking time after the occurrence of Zone-Across Compaction, resulting in decreased performance of compaction and flush operations.

IV. OVERVIEW

ROCKSDB for ZNS SSD is tightly coupled with the file system layer, ZenFS. In this study, we introduce cross-layer collaborative algorithms—specifically, the Zone-Aware Compaction Algorithm (ZACA) at the ROCKSDB layer and the Compaction-Aware Zone Allocation Algorithm (CAZA) at the ZenFS layer. Our aim is to enhance the processing throughput of compaction operations in ROCKSDB and zone cleaning in ZenFS. These collaborative algorithms are designed to extend the lifetime of ZNS SSDs.

ZACA and CAZA collaborate synergistically to facilitate efficient Same-Zone Compaction, as detailed in Section III. Leveraging sequential reads to the maximum extent, these algorithms are designed to minimize compaction time. Furthermore, by strategically reducing the volume of copied valid data during ZC, the algorithms mitigate ZC I/O blocking time in ZenFS, leading to a reduction in write amplification. Ultimately, this collaborative effort contributes to an extended lifespan of ZNS SSDs.

Figure 5 presents a detailed software architecture overview of ROCKSDB and ZenFS with the integration of the ZACA and CAZA algorithms within the system. Firstly, ZACA judiciously selects SSTables close to Same-Zone Compaction as compaction victims, employing a metric called S_{same} (elaborated in Section V). Secondly, CAZA operates within the Zone Allocation module of ZenFS. When a zone allocation request for SSTable files is triggered by ROCKSDB, the Zone Allocation module of ZenFS allocates/assigns zones from the zone pool using the CAZA algorithm. CAZA places SSTables within the zone where they are likely to be selected as compaction victims together during subsequent compaction. The CAZA algorithm considers factors such as S_{level}, SSTable size, and key range when assigning SSTables to zones, thus inducing Same-Zone Compaction.

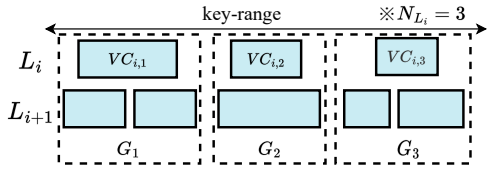


Fig. 6: Description of the participating SSTables (G_j) in the merge-sort process for the victim SSTable ($VC_{i,j}$) when compaction occurs in L_i.

V. ZACA: ZONE-AWARE COMPACTION VICTIM SELECTION ALGORITHM

Same-Zone Compaction involves selecting compaction victims from L_i, where the SSTables from L_i and L_{i+1} are all located in the same zone, and utilizing them for merge-sort. Therefore, the primary objective of ZACA is to select compaction victims during compaction that are as close as possible to Same-Zone Compaction. Consequently, ZACA achieves two key goals: (i) increasing the sequential read I/O pattern during compaction to reduce compaction time and (ii) reducing the amount of the valid SSTable copying, which is the primary overhead in ZC, resulting in a reduction of I/O blocking time.

A. Selecting Compaction Victims using the Same-Zone Score

ZACA selects SSTables as compaction victim for compaction operations using the Same-Zone Score (S_{same}). Now we elucidate the method employed by ZACA of how to calculate the S_{same}. To simplify the explanation, we introduce the following notations. Assume compaction is initiated in L_i, and there are N_{L_i} SSTables in L_i. ZACA selects a SSTable, which is $VC_{i,k}$ (victim candidate), from the N_{L_i} SSTables in L_i using the S_{same} ($k = 1...N_{L_i}$). Consider all SSTables in L_{i+1} whose key-ranges overlap with $VC_{i,k}$, forming a set denoted as G_k ($k = 1...N_{L_i}$). It is noteworthy that k serves as the group index for G_k hereafter.

Figure 6 represents the visualization of N_{L_i}, $VC_{i,k}$, and G_k, where N_{L_i} is 3. ZACA calculates the $S_{same}[k]$ for each G_k. Before delving into the calculation of the $S_{same}[k]$, let's introduce some additional notations. Each SSTable within G_k is associated with at least one zone. It is important to note that ZACA can consult ZenFS to determine the specific zone to which each SSTable belongs. We assume total number of zones on the ZNS SSD is N_{zone}. ZACA computes the sum of sizes ($S_{k,m}$) of SSTables within G_k for each zone m ($m = 1...N_{zone}$). If there are no SSTables from G_k in zone m, then $S_{k,m}$ is set to 0. Otherwise, it takes a value greater than 0.

TABLE II: Notations for Problem Formulation.

Component	Description
N_{zone}	Numbers of zones in ZNS
N_{L_i}	Numbers of SSTables at L_i
k, G_k	Group index, group k
m	Zone index
$VC_{i,k}$	L_i SSTables can be selected as L_i-to-L_{i+1} compaction
$S_{k,m}$	Sum of Group k SSTs' size in zone m
$S_{same}[k]$	Same-Zone Score of G_k
$S_{level}[i]$	L_i-to-L_{i+1} compaction score by size limit
S_{inval}	Score calculated by invalid ratio after compaction

ZACA calculates $S_{k,m}$ for all zones and determines $S_{same}[k]$ for G_k by dividing the sum of the squares of $S_{k,m}$ ($m = 1...N_{zone}$) by the square of the sum.

Equation 1 represents the formulation of $S_{same}[k]$ for G_k.

$$S_{same}[k] = \frac{\sum_{m=1}^{N_{zone}} ({S_{k,m}}^2)}{(\sum_{m=1}^{N_{zone}} S_{k,m})^2} \quad (1)$$

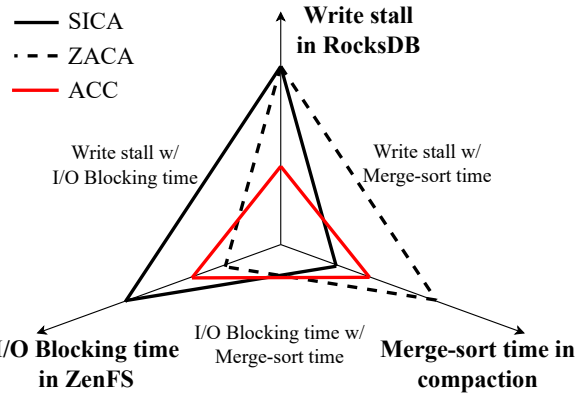
This equation originates from the Standard Deviation, which represents the dispersion of a specific variable. $S_{same}[k]$ takes values between 0 and 1. As the SSTs included in the compaction victim are distributed across multiple zones (Zone-Across Compaction), the score approaches 0. Conversely, if all SSTs are located in a single zone (Same-Zone Compaction), the score is 1. Detailed descriptions of notations are provided in Table II. Note that i is level, k is group index, m is zone index.

ZACA calculates the $S_{same}[k]$ for all G_k in L_i and selects the G_k with the highest S_{same} as the compaction victim. ZACA performs merge-sort for G_k and then generates new SSTables. Finally, ZACA in RocksDB deletes the SSTables in G_k, which means invalidating SSTables located in the corresponding zones by ZenFS.

To simplify the explanation of the algorithm described earlier, we calculate the S_{same} for each case using the examples presented in Figures 2 and 3. Here, we assume a zone size of 4, and each SSTable size is 1.0. In Figure 2, group k, we have $S_{k,0} = 1.0$, $S_{k,1} = 1.0$, $S_{k,2} = 1.0$, $S_{k,3} = 0.0$. Therefore, the $S_{same}[k]$ is calculated as $\frac{(1.0^2+1.0^2+1.0^2)}{(3.0)^2} = 0.33$. On the other hand, in Figure 3, group k', we have $S_{k',0} = 3$, and all other $S_{k',m}$ are 0.0. Since the $S_{same}[k'](= 1.0)$ is greater than $S_{same}[k](= 0.33)$, group k' is selected as the compaction victim. As in the example, ZACA calculates the S_{same}[k] for all G_k ($k = 1...N_{L_i}$) and selects the G_k with the highest S_{same} as the compaction victim.

B. Adaptive Compaction Controller

ZACA enhances the sequential read pattern and ZC efficiency. However, a drawback compared to the traditional SICA is that it performs merge-sort on a larger number of key-value pairs, leading to longer compaction duration. As discussed in Section III, SICA significantly reduces the size of L_i that exceeded the size limit while optimizing the sum of key-value pairs participating in the merge-sort. Consequently, SICA achieves relatively shorter compaction duration. On the other hand, ZACA selects compaction victims based on S_{same} rather than size. This means that in ZACA, the total size, i.e., the sum of key-value pairs participating in the merge-sort, increases during compaction, requiring more read/write I/O in a single compaction operation and potentially causing longer compaction times.



	I/O Blocking in ZenFS	Merge-sort time	Write stall in RocksDB	Performance
SICA	High	Low	High	Low
ZACA	Low	High	High	Low
ACC	Med	Med	Low	High

Fig. 7: The performance tradeoff in RocksDB and ZenFS between SICA and ZACA.

- **Insight 1:** There are two factors contributing to prolonged compaction: (i) an increase in I/O blocking time due to ZC and (ii) an increase in the number of key-value pairs undergoing merge-sort.
- **Insight 2:** While ZACA aims to reduces I/O blocking time, if the number of key-value pairs subjected to merge-sort increases, compaction may take longer compared to SICA. Consequently, there is a concern about an elevated write stall, potentially leading to a decrease in PUT(k,v) performance.

Hence, we introduce the Adaptive Compaction Controller (ACC), a control mechanism that dynamically chooses between SICA and ZACA based on the free space ratio on the ZNS SSD.

Figure 7 visually depicts the design tradeoff according to the compaction victim selection algorithm, highlighting the comparison between SICA, ZACA, and ACC.

Particularly, we have observed that the execution time of ZC increases when the free space ratio on the ZNS SSD is low, causing blocking in I/O operations of LSM-Tree (RocksDB), including Memtable flush operations and compaction I/Os. As explained earlier, SICA does not effectively reduce the I/O blocking time of ZC. Therefore, ACC suggests using ZACA instead of SICA when the free space ratio on the ZNS SSD is lower than threshold(turning_point), as ZACA is known to mitigate I/O blocking. Conversely, when the free space ratio is higher than threshold, ACC opts for SICA, in contrast to the explanation provided earlier. Pseudocode 1

Pseudocode 1: Adaptive Compaction Controller

```
function ACC(free_space_ratio, level)
    if free_space_ratio > turning_point then
        return SICA()
    if level == 1 or level == 0 then
        return SICA()
    return ZACA()
```

outlines the algorithm of ACC.

Additionally, in the case of RocksDB, we observed that L_0-to-L_1 compaction and L_1-to-L_2 compaction are not parallelized (they occur serially) [27], and the compaction duration for these operations is notably high. Specifically, ZACA increases the number of key-value pairs participating in the merge-sort, thereby prolonging the compaction duration. To address this, in ACC, ZACA is designed to be applied only to L_2-to-L_3 compaction and subsequent higher-level compactions, where $i \geq 2$, to mitigate the impact on performance. Consequently, ZACA controlled by ACC reduces write stall, enhancing PUT performance.

VI. CAZA: Compaction-Aware Zone Allocation

As discussed in the previous section, ZACA is an algorithm that directly selects the G_k with the highest Same-Zone Score to induce a compaction most similar to Same-Zone Compaction. On the other hand, CAZA [17] is an algorithm designed to allocate zones for placing SSTables during their creation to increase the likelihood of inducing Same-Zone Compaction in the future. Note that SSTables are generated either when flushing from Memtable to disk or as a result of compaction.

When compaction occurs in L_i, a victim SSTable within L_i, is selected based on the SICA or ZACA algorithms previously discussed. Subsequently, a merge-sort is executed on this victim SSTable and SSTables at level L_{i+1} that have overlapping keys with the key range of the victim SSTable. So, from the viewpoint of an SSTable (S), the compaction process involves either merging with SSTables in L_{i-1} or with those in L_{i+1}.

In the phase of selecting the compaction victim, if an SSTable(S) becomes the victim and undergoes merge-sort with SSTables (regardless of L_{i-1} or L_{i+1}) in the same zone, it represents the ideal scenario known as Same-Zone Compaction. Therefore, strategically placing SSTables in a zone where SSTables from different levels (L_{i-1} or L_{i+1}) with overlapping key ranges are located can enhance the likelihood of Same-Zone Compaction. However, allocating zones for SSTables at the time of creation to increase Same-Zone Compactions presents a challenging problem for the following two reasons.

- **Problem #1:** The newly created SSTable(S) in L_i overlaps its key-range with SSTables in both L_{i-1} and L_{i+1}. However, predicting whether this SSTable(S) will be deleted by L_{i-1}-to-L_i compaction or L_i-to-L_{i+1} compaction during future compactions is challenging. For example, even if an SSTable(S) is assigned to a zone with SSTables in L_{i-1} having overlapping key-ranges, it might be deleted by L_i-to-L_{i+1} compaction, leading to Zone-Across Compaction. Hence, predicting whether an SSTable(S) will be deleted by L_{i-1}-to-L_i compaction or L_i-to-L_{i+1} compaction is crucial for appropriately placing the SSTable(S).
- **Problem #2:** As described in Problem 1, an SSTable(S) overlaps its key-range with SSTables in both L_{i-1} and L_{i+1}. Within each level, there might not be just one but several SSTables with overlapping key-ranges. Moreover, these SSTables can be placed in different zones. For instance, let's assume there are two SSTables in L_{i-1} with overlapping key-ranges with an SSTable (S), and each SSTable is placed in $zone_m$ and $zone_{m+1}$. In such a scenario, determining the appropriate zone to place an SSTable(S) among $zone_m$ and $zone_{m+1}$ requires specific criteria rules.

To tackle Problem #1, when CAZA creates an SSTable(S) in L_i, it compares the S_{level} for L_{i-1} and L_i ($S_{level}[i-1]$, $S_{level}[i]$). If $S_{level}[i-1]$ is greater than $S_{level}[i]$, it indicates that L_{i-1}-to-L_i compaction occurs more frequently than L_i-to-L_{i+1} compaction. This implies a higher likelihood for an SSTable(S) to undergo compaction with L_{i-1} SSTables than with L_{i+1} SSTables. Consequently, an SSTable(S) is placed in one of the zones where L_{i-1} SSTables with overlapping key-ranges are located. Similarly, if $S_{level}[i]$ is greater than $S_{level}[i-1]$, an SSTable(S) is placed in one of the zones where L_{i+1} SSTables with overlapping key-ranges are located.

To address Problem #2, CAZA takes into account the sizes of SSTables. Let's assume CAZA places an L_i SSTable(S) in one of the zones where there are SSTables in L_{i-1} with overlapping key-ranges due to S_{level}. As observed in Observation 6, RocksDB follows the SICA algorithm, selecting the largest SSTable among L_{i-1} SSTables with overlapping key-ranges as the compaction victim for the SSTable(S). This implies that the largest SSTable in L_{i-1} with overlapping key-ranges is more likely to be selected as the compaction victim with the SSTable(S) compared to other SSTables with overlapping key-ranges.

Therefore, CAZA places an SSTable(S) in the zone where the largest SSTable, among those in L_{i-1} with overlapping key-ranges, is located. If there is no space in that zone, it places an SSTable(S) in the zone with the next largest SSTable. Otherwise, when placing SSTables in zones where L_{i+1} SSTables are located, CAZA chooses the zone with the smallest SSTable among those with overlapping key-ranges in L_{i+1}. This is because, during the selection of victim SSTables for L_{i+1}-to-L_{i+2} compaction, the smallest SSTable among those with overlapping key-ranges in L_{i+1} is prioritized.

If there is no overlapping SSTable in L_{i-1} or L_{i+1} or if there is insufficient space in the zone where SSTables from L_{i-1} or L_{i+1} are located, zone allocation may fail. In such cases, SSTables with the most zones having closest key ranges within the same level are allocated. Looking for the closest key range gives a future SSTable in the upper (or lower)

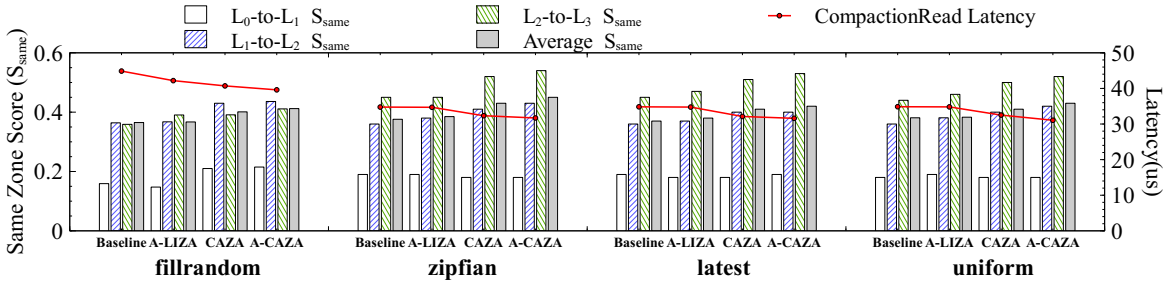


Fig. 8: Comparison of S_{same} and read latency of compaction.

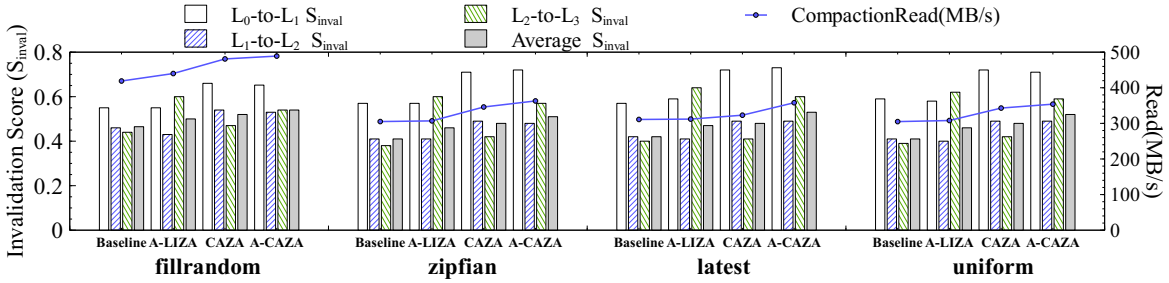


Fig. 9: Comparison of S_{inval} and read throughput of compaction (MB/s).

level an opportunity to bridge closest key ranges. For example, SSTables with key ranges (10-20) and (25-35) can be bridged by SSTable with (15-30) in the upper level.

VII. Evaluation

A. Experimental setup

For the evaluation of ZACA and CAZA, we used Configurable ZNS (ConfZNS) [28], a ZNS SSD emulator based on FEMU [29]. We emulated a 67GB ZNS SSD environment with 32 Intel(R) Xeon(R) Gold 5218R CPUs and 32GB of memory. The Western Digital Ultrastar DC ZN540 [30], a representative real product of ZNS SSDs, features a zone size of 1GB; hence, we set the zone size to 1GB [31]. The emulated ZNS SSD features 8 channels, with 2 dies per channel, resulting in a total of 16 dies. For NAND pages, we set the read latency to 65us, write latency to 450us, and block erase latency to 2ms.

To maintain compatibility with ConfZNS, we modified ZenFS v2.1 on Linux v5.10 and RocksDB v7.4. For the sake of experimentation, we configured RocksDB to utilize 4 compaction threads, 4 flush threads, and 8 subcompaction [32] threads. In the ZenFS, ZC employs a greedy algorithm that zones with the highest levels of invalid data are selected as victims for cleaning.

We conducted evaluations using both synthetic and realistic benchmarks. For the synthetic workload, we used db_bench [26], a tool bundled with RocksDB that offers micro-benchmarks for various workload patterns. Specifically, we utilized the fillrandom workload for our analysis. In addition to synthetic benchmarks, we conducted a realistic evaluation using three workloads from the industry-standard key-value store evaluation tool, the Yahoo Cloud Serving Benchmark (YCSB) [33]. These workloads include *zipfian* (with a zipfian constant of 0.99), *latest*, and *uniform*.

In all above four workloads, the key size is set to 16 bytes, and the value size is set to 1KB. The db_bench's fillrandom workload performs PUT(k,v) operations on a dataset of 72GB, while the YCSB's three workloads operate on a 36GB dataset of key-value pairs. All reported values represent the mean of at least three independent runs.

We compared the following four schemes:

- **Baseline**: Used the LIZA in RocksDB and SICA ZenFS.
- **A-LIZA**: Used ACC in RocksDB and LIZA in ZenFS.
- **CAZA**: Used SICA in RocksDB and CAZA in ZenFS.
- **A-CAZA**: Used ACC in RocksDB and CAZA in ZenFS.

B. Correlation Analysis between S_{same} and Read Performance of Compaction

In this section, we analyze the correlation between S_{same} and the performance of reading during compaction, a process we refer to as *CompactionRead*. S_{same} takes value between 0 and 1, representing the degree of Same-Zone Compaction, where a value close to 1 indicates proximity to Same-Zone Compaction, and a value close to 0 indicates proximity to Zone-Across Compaction. Equation 1 provides a description of S_{same}.

Figure 8 compares S_{same} and CompactionRead latency for the evaluated schemes across four workloads. We calculated the average S_{same} for compactions at each level and obtained the average S_{same} for all levels of compaction. Across all workloads, A-LIZA, CAZA, and A-CAZA show an increasing trend in S_{same} compared to the Baseline. Specifically, A-CAZA exhibits an average increase of 0.05 in S_{same} compared to the Baseline. In the cases of A-LIZA and A-CAZA, where ZACA operates with ACC, the S_{same} for L_2-to-L_3 compaction increases compared to the Baseline and CAZA. This is because ACC, through ZACA, prioritizes compaction victims

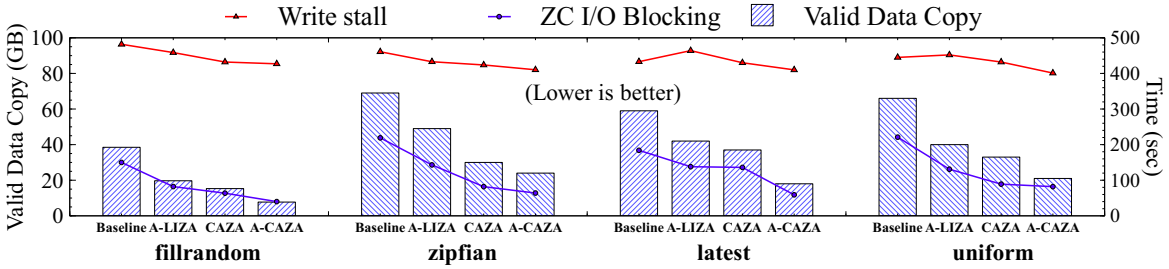


Fig. 10: Comparison of Write stall and ZC efficiency.

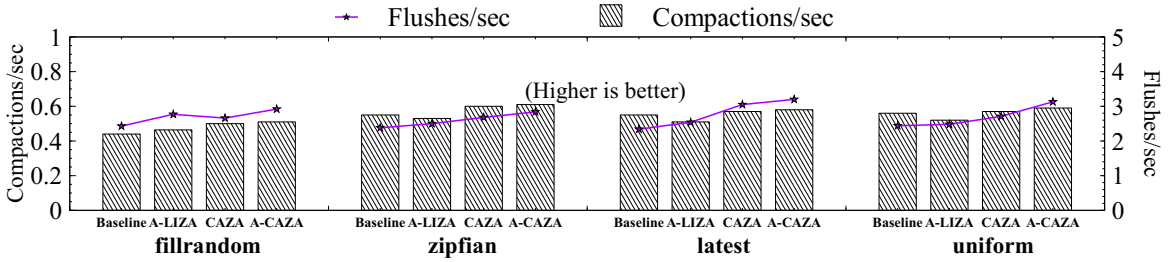


Fig. 11: Comparison of Flush and Compaction throughput.

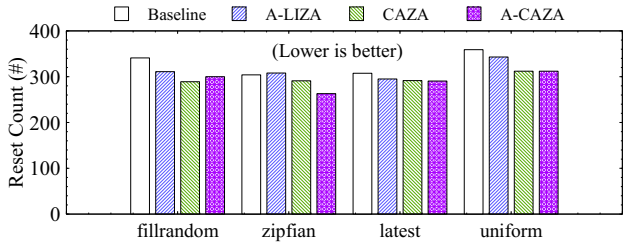


Fig. 12: Comparison of zone-reset count for four schemes across four workloads.

with a higher S_{same} during L_2-to-L_3 compaction. Moreover, the average S_{same} increases across all workloads, indicating a more substantial utilization of the sequential read pattern when compaction reads SSTables. Consequently, CompactionRead latency decreases by approximately 3.82 microseconds, a 10% reduction. The improved CompactionRead latency results in an average 16% increase in CompactionRead (MB/s) in Figure 9.

The comprehensive performance enhancement of compaction due to the improved CompactionRead is further explored in the next section.

We define S_{inval} as a metric to delicately evaluate the impact of S_{same} on ZC efficiency. S_{inval} represents the average invalidation ratio of zones where compaction victims were located after compaction, ranging between 0 and 1. A higher S_{inval} indicates a higher ratio of SSTables invalidated by compaction for a given zone. If the SSTable invalidation ratio is high for a zone due to compaction, it implies a reduced amount of valid SSTables copied by ZC, leading to decreased I/O blocking time. As an example for better understanding, in Figure 2, S_{inval} is $\frac{0.25+0.25+0.25}{3} = 0.25$, and in Figure 3, S_{inval} is $\frac{0.75}{1} = 0.75$.

Due to the increased S_{same} in A-CAZA, there is a tendency for A-CAZA's S_{inval} to increase in all levels of compaction compared to the Baseline. The Average S_{inval} across the four workloads increased by an average of 0.10 compared to the Baseline. Furthermore, in the cases of A-LIZA and A-CAZA, where ACC triggers ZACA, there is a noticeable increase in S_{inval} for L_2-to-L_3 compaction, exceeding 0.14. This is because ACC, through ZACA, selects compaction victims based on S_{same} for L_2-to-L_3 compaction. By increased S_{inval}, it is possible to concentrate the ratio of invalid data within a single, same zone. Consequently, when ZC occurs, the number of valid data copies decreases, leading to an increase in ZC efficiency. In the next section, we will delve into how the increased S_{inval} mitigates ZC overhead in more detail.

C. Compaction Performance and Zone Cleaning Efficiency

In this section, we examine how the increase in S_{same} and S_{inval} affects ZC efficiency, write stall, zone-reset count, compactions/sec, and flushes/sec.

Figure 10 illustrates ZC efficiency and write stall for the four workloads. Since A-CAZA increased S_{inval} by 0.10 compared to the Baseline, the amount of valid data copied by ZC decreased by an average of 70%. In the case of the fillrandom workload, the valid data copy amount decreased by a significant 80%. As a result, ZC's I/O blocking time decreased by an average of 68%.

Figure 12 presents the zone-reset count performed by ZC. As the amount of copied valid SSTables decreases, A-CAZA exhibits a reduction in zone-reset count compared to the Baseline: 12% for fillrandom, 13% for zipfian, 5% for the latest, and 13% for the uniform workload. By reducing the zone-reset operations that erase NAND erase blocks by 8.6%, the lifespan of ZNS SSDs is increased.

Figure 11 illustrates flushes/sec and compactions/sec. Flushes/sec and compactions/sec are obtained by dividing 1 second by the time taken for one flush and one compaction.

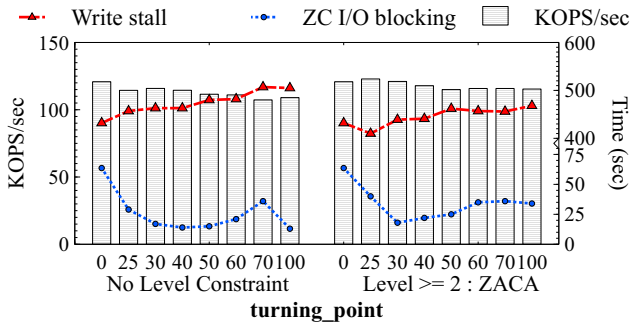


Fig. 13: Comparison of write stall and I/O blocking time based on changes in level constraints and ACC's turning_point.

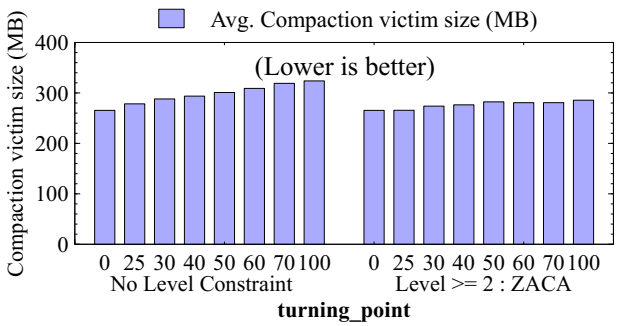


Fig. 14: Comparison of compaction victim size based on changes in level constraints and ACC's turning_point.

Higher values for flushes/sec and compactions/sec indicate a shorter time for each flush or compaction.

With the reduction in ZC's I/O blocking and the improvement in CompactionRead, as discussed in Section VII-B, A-CAZA enhances flushes/sec and compactions/sec by 1.26x and 1.1x, respectively, compared to the Baseline. A-CAZA processes an additional 0.625 flushes and 0.15 compactions per second compared to the Baseline.

However, in the case of the latest and uniform workloads, A-LIZA, which adopts ACC and ZACA, exhibits a decrease in compaction throughput compared to the Baseline, leading to an increase in ROCKSDB's write stall time. This is indirectly indicative of an increase in write stall due to a higher number of key-value pairs selected as compaction victims by ZACA, resulting in extended merge-sort durations and increased writes over an extended period. In the following section, we conduct a detailed analysis of the number of key-value pairs to be merge-sorted and ZC I/O Blocking time, shedding light on their adverse impact on write stall.

D. Analysis of Adaptive Compaction Controller Effect on Write Stall

In this section, we examine the impact of ACC on two factors contributing to write stall: ZC I/O blocking time and the number of key-value pairs to be compacted. We observe the performance trends by modifying the ACC's turning_point and level constraint while using the Zone Allocation algorithm proposed in this paper, CAZA, as ZenFS's Zone Allocation algorithm. For all experiments, we used a fillrandom workload.

Figure 13 illustrates the changes in the PUT(k,v) throughput (KOPS/sec), stall time, and I/O blocking time as the ACC's level constraint, turning_point, and the level where ZACA is applied vary. Note that the x-axis values represent the turning_point. The left bar graph represents the case without a level constraint. Without level constraint, ZACA is performed at every level when free space is under the turning_point. The right bar graph represents ACC executing ZACA for compactions at L_2 and above. Keep in mind that if turning_point set to 0, ZACA is never executed, and if turning_point is set to 50, ZACA is executed when less than 50% free space is available.

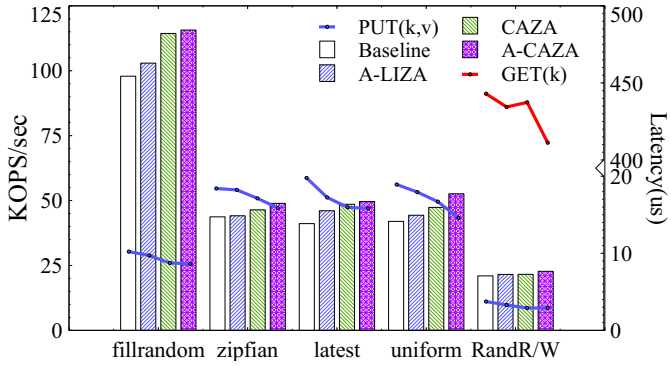


Fig. 15: Overall performance of all schemes.

From Figure 13, in the case of the left bar graph without a level constraint (where ZACA is selected for all levels of compaction), ZenFS experiences a reduction in I/O blocking time when ZACA is used (turning_point > 0) compared to when ZACA is not used (turning_point $= 0$). However, despite this improvement, ROCKSDB's write stall increases, and the throughput of PUT(k,v) measured in KOPS/sec decreases. The reason behind this is that the frequent execution of ZACA leads to an increase in the size of compaction victims, resulting in longer merge-sort operations. Specifically, selecting L_1-to-L_2 compaction victims through ZACA prolongs the merge-sort process. As discussed in Section III, L_1-to-L_2 and L_0-to-L_1 compactions are serialized, amplifying the adverse impact on write stall.

The left bar graph in Figure 14 illustrates the increasing trends of average compaction victim size as number set to turning_point goes up. Moreover, when turning_point is set to $60, 70$, it is observed that as the compaction victim size increases, the WA of RocksDB increases, leading to an increase in I/O blocking time. We consider this result as performance anomaly of ZACA because it behaves the opposite of what is expected.

In the case of the right bar graph in Figure 13, ACC selects compaction victims using ZACA only when $level \geq 2$, based on a tuning_point. Specifically, L_0-to-L_1 and L_1-to-L_2 compaction victims are chosen solely through SICA, and ZACA is employed for L_2-to-L_3 compaction onwards based on the turning_point.

Examining the right bar graph in Figure 13, when ACC

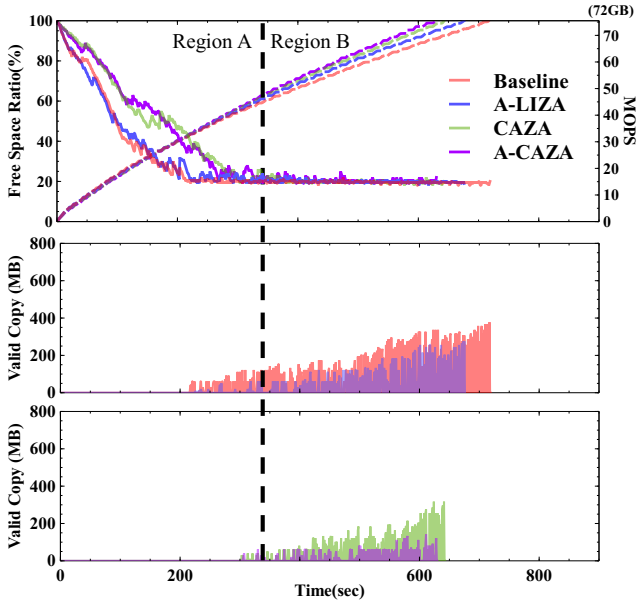


Fig. 16: Microscopic analysis of Baseline, A-LIZA, CAZA, A-CAZA in db_bench fillrandom workload.

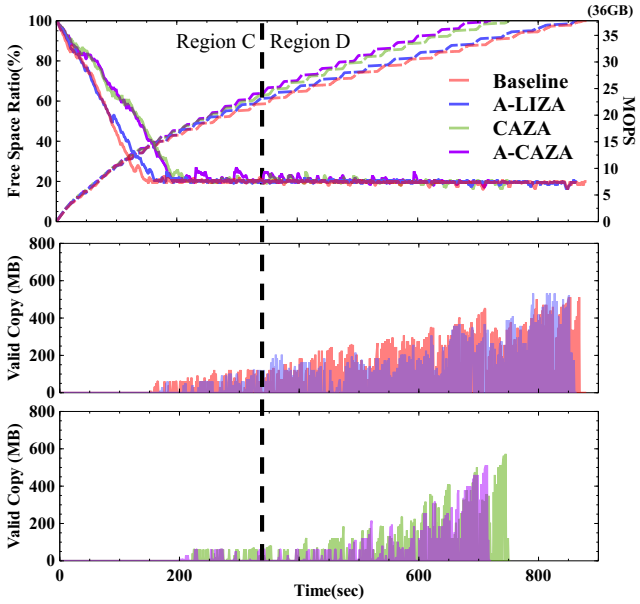


Fig. 17: Microscopic analysis of Baseline, A-LIZA, CAZA, A-CAZA in YCSB-zipfian workload.

is configured with turning_point = 25, meaning that ZACA is applied for levels with $level \geq 2$, significant improvements are observed. Compared to the scenario without ZACA (turning_point = 0), ZenFS experiences a reduction in I/O blocking time by 23.6 seconds and a decrease in stall time by 22 seconds, resulting in a performance increase of 2 KOPS/sec. This is because the reduction in ZC I/O Blocking time outweighs the increase in write stall caused by the growth in compaction victim size. When comparing turning_point = 25 with the scenario without using ZACA (turning_point = 0), the compaction victim size barely increases, while the I/O Blocking time decreases by 23.6 seconds.

When turning_point increases (turning_point $\geq$ 30), more ZACA are executed by ACC. Consequently, the increase in compaction victim size, leading to an increase in write stall, becomes more pronounced than the reduction in ZC's I/O Blocking time that mitigates write stall. Therefore, while ZACA effectively reduces ZC overhead, without appropriate control from ACC, the rise in compaction victim size could amplify the write stall time, resulting in a decrease in PUT(k,v) KOPS/sec. Performance trends related to write stall based on the turning_point value are consistently observed across different YCSB workloads. Hence, it is crucial to use ACC to judiciously adjust ZACA for compactions at levels beyond L_2 and set turning_point = 25.

E. User-Perceived Performance

In this section, we conduct a performance comparison across various workloads. In Figure 15, A-CAZA shows an average increase of approximately 16% in PUT(k,v) ops/sec across all workloads. Moreover, the latency of PUT(k,v) is reduced by 18%.

Although our primary focus is on enhancing write-intensive workloads, we also evaluate the influence of CAZA and ZACA on GET(k). We employ the readrandomwriterandom workload of db_bench, executing 72GB of PUT and 8GB of GET. The results indicate that, compared to the Baseline, A-CAZA reduces the latency of GET(k) by 7%. In other words, CAZA, ZACA, and ACC provide solutions for write-intensive workloads without introducing negative effects on GET(k); instead, GET(k) latency is reduced.

F. Analysis of Algorithmic Overhead

We analyze the algorithmic overhead of four algorithms: SICA, ZACA, LIZA, and CAZA. Table III presents the average execution time for a single invocation when each algorithm is executed 1,000 times with the db_bench fillrandom workload. Comparing the two compaction victim selection algorithms, ZACA takes 823us longer than SICA. Therefore, ZACA exhibits greater algorithmic overhead than SICA.

TABLE III: Average execution time of each algorithm.

Algorithm	SICA	ZACA	LIZA	CAZA
Time(us)	271	1094	10	49

Additionally, for the two Zone Allocation algorithms, LIZA and CAZA, CAZA takes 39us longer. Despite the fact that the proposed ZACA and CAZA algorithms take 4x longer, the performance of flushes/sec, compactions/sec, PUT(k,v) KOPS/sec increased by 1.26x, 1.1x and 1.16x respectively. Thus, the algorithmic overhead on performance is overall negligible.

G. Microscopic Analysis

To analyze the increasing ZC efficiency with the introduction of CAZA and ZACA, we conducted a detailed time-series

analysis of free space and the frequency of valid data copy occurrences during ZC.

Figures 16 and 17 illustrate the results for db_bench and YCSB workloads. Regions 'A' and 'C' represent periods where the free space ratio is above 20%, indicating no occurrence of ZC overhead. On the other hand, regions 'B' and 'D' depict intervals where the free space is close to 20%, leading to the occurrence of ZC overhead.

ZenFS does not trigger valid data copy and I/O blocking when the free space ratio is 20% or higher. In contrast, when the free space ratio drops below 20%, ZenFS blocks compaction and flush operations, inducing valid data copy.

In Region 'A' of Figure 16, CAZA and A-CAZA strategically place SSTables that can be selected as compaction victims in the same zone, allowing them to reclaim more space without triggering valid data copy compared to LIZA and A-LIZA. As a result of reclaiming more free space in Region 'A,' the occurrence of Region 'B,' where valid data copy and I/O blocking typically take place, is delayed.

In comparison to Baseline and A-LIZA, which experience I/O blocking starting from [200-], CAZA and A-CAZA start facing I/O blocking later, specifically from [300-]. Moreover, in Region 'B,' Baseline, A-LIZA, and CAZA induce a maximum of 400MB valid data copy per second compared to A-CAZA (refer to red, blue, and green spikes). This difference arises from A-CAZA's use of the CAZA algorithm for SSTable placement and the ZACA algorithm for compaction victim selection below the turning_point (25%). Consequently, A-CAZA enhances ZC efficiency, resulting in the generation of less than 200MB of copy during ZC (refer to the purple spike) and faster processing of 72GB of PUT(k,v) compared to the other three schemes.

Figure 17 illustrates the results for the YCSB zipfian workload. The zipfian workload, with a size of 36GB, induces more copy and longer I/O blocking time compared to the 72GB size fillrandom, making it a more demanding write-intensive workload. Despite the stronger workload, CAZA and A-CAZA exhibit increased ZC efficiency in Region 'C' by strategically placing SSTables using the CAZA algorithm. As a result, the occurrence of I/O Blocking in Region 'D' is delayed by 50 seconds compared to LIZA and A-LIZA. When comparing Region 'B' for fillrandom and Region 'D' for zipfian, although there is a difference in scale, A-CAZA ensures a faster PUT(k,v) processing rate by inducing fewer copies than the other three schemes (refer to the purple spike).

H. Comparison with Our Prior Work [17]

The basic concept of CAZA was initially introduced in our previous work [17]. In the default version of CAZA, which we denote, the approach incorporates two key attributes of compaction: it selects SSTables that have overlapping key ranges and jointly selects SSTables from both L_i and L_{i+1}. In this manuscript, we extended CAZA algorithm by considering two more key factors, S_{level} and size of SSTables in addition to key-ranges and level.

Table IV shows the comparison between them in fillrandom.

Our enhanced version of CAZA ($\text{CAZA}_{enhanced}$) reduces valid data copy by 3.2GB compared to CAZA baseline ($\text{CAZA}_{baseline}$), consequently increasing user-perceived performance to 6 KOPS/sec. This is because $\text{CAZA}_{enhanced}$ considers SSTables' size and S_{level}. By taking into account the size of the SSTable, $\text{CAZA}_{enhanced}$ predicts the likelihood of SSTables compaction within a single level more accurately. Moreover, $\text{CAZA}_{enhanced}$ predicts frequency of compaction among levels at time T with S_{level}, resulting in enhanced ZC efficiency.

TABLE IV: Comparing CAZA [17] with an enhanced CAZA.

fillrandom	Valid Copy(GB)	Write Stall(sec)	KOPS/sec
$\text{CAZA}_{baseline}$	18.5	446	108
$\text{CAZA}_{enhanced}$	15.3	432	114

VIII. Related Work

There have been various system optimization studies for ZNS SSD [5], [28], [34]–[37]. Waltz [34] enhanced tail-latency in LSM-ZNS storage's PUT(k,v) by incorporating zone-append [35]. ZNSwap [5] investigated the failure of performance isolation in a multi-tenant environment due to Garbage Collection by introducing ZNS SSD into the kernel subsystem swap memory area, diverging from traditional SSDs. ZNS+ [8] accelerated filesystem performance by offloading copy operations to the device in the log-structured file system F2FS [38]. ConfZNS [28] defined Full-Unit zone (FU-zone) and Single-Unit zone (SU-zone) layouts based on internal parallelism, providing a real-latency ZNS SSD emulator for researchers in the ZNS SSD field. H. Bae et al [36] improved read performance and latency in small-zone layout through kernel-level internal parallelism profiling and an interference-aware scheduler. eZNS [37] proposed a method for fully utilizing the internal parallelism of ZNS through Zone Ballooning.

LL-Compaction [22], akin to our research, enhances ZC efficiency through compaction ZNS SSD. They suggests compactions to include SSTables irrespective of key-range considerations when selecting compaction victims in the compaction process for reducing ZC overhead. However, their study does not account for the potential increase in write stall resulting from modification in the compaction victim selection algorithm. This approach raises concerns about the potential increase in the total number of key-value pairs participating in merge-sort, potentially prolonging the merge-sort time. The basic idea of CAZA was initially introduced in our previous work [17]. In this paper, we substantially extend the CAZA with SSTables' size and S_{level} and present ZACA with ACC as a innovative new approach.

IX. Conclusion

LSM-based Key-Value Stores are recognized as suitable for ZNS SSDs. This paper addresses performance and lifetime issues associated with RocksDB's SICA compaction algorithm and ZenFS's LIZA algorithm, known for frequently triggering Zone-Across Compaction. To overcome these issues, the paper

introduces Same-Zone Compaction, aiming to facilitate fast read I/Os through a sequential read pattern during compaction while enhancing ZC efficiency. Specifically, the paper presents ZACA, ACC, and CAZA algorithms designed to promote Same-Zone Compaction. Consequently, our proposed algorithms successfully reduce ZC overhead by 80%, enhance compaction read performance by 14%, and improve flush and compaction speeds by 1.26x and 1.1x, respectively. The performance enhancements attributed to Same-Zone Compaction result in a notable 16% increase in the processing rate of the PUT(k,v) in RocksDB. Furthermore, it contributes to an 8.6% reduction in the zone-reset count, ultimately extending the lifetime of ZNS SSDs.

Acknowledgments

This paper has been extended based on our previous research [17]. In particular, we are deeply grateful to Hee-Rock Lee, Chang-Gyu Lee, and Seungjin Lee for their initial development of the CAZA algorithm. This work was n part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386), and in part by Samsung Electronics Co., Ltd (IO221014-02908-01).

References

[1] H. Holmberg, "Zenfs, zones and rocksdb - who likes to take out the garbage anyway?." https://snia.org/sites/default/files/SDC/2020/074-Holmberg-ZenFS-Zones-and-RocksDB.pdf, 2020.

[2] "NVM Express. NVM Express Workgroup, NVM Express® Zoned Namespace Command Set Specification Revision 1.1a.." https://www.nvmexpress.org/specification, 2022.

[3] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, "Zns: Avoiding the block interface tax for flash-based ssds," in *Proceedings of 2021 USENIX Annual Technical Conference*, pp. 689–703, 2021.

[4] N. Tehrany and A. Trivedi, "Understanding nvme zoned namespace (zns) flash ssd storage devices," *arXiv preprint arXiv:2206.01547*, 2022.

[5] S. Bergman, N. Cassel, M. Bjørling, and M. Silberstein, "Znswap: Unblock your swap," *ACM Transactions on Storage*, vol. 19, no. 2, pp. 1–25, 2023.

[6] W. Digital, "Zenfs." https://github.com/westerndigitalcorporation/zenfs, 2022.

[7] M. Oh, S. Yoo, J. Choi, J. Park, and C.-E. Choi, "Zenfs+: Nurturing performance and isolation to zenfs," *IEEE Access*, vol. 11, pp. 26344–26357, 2023.

[8] K. Han, H. Gwak, D. Shin, and J. Hwang, "Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, 2021.

[9] D. Seo, P.-X. Chen, H. Li, M. Bjørling, and N. Dutt, "Is garbage collection overhead gone? case study of f2fs on zns ssds," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023.

[10] S. Byeon, J. Ro, S. Jamil, J.-U. Kang, and Y. Kim, "A free-space adaptive runtime zone-reset algorithm for enhanced zns efficiency," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023.

[11] D. Huang, D. Feng, Q. Liu, B. Ding, W. Zhao, X. Wei, and W. Tong, "Splitzns: Towards an efficient lsm-tree on zoned namespace ssds," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 3, pp. 1–26, 2023.

[12] G. Choi, K. Lee, M. Oh, J. Choi, J. Jhin, and Y. Oh, "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*, 2020.

[13] RocksDB, "Rocksdb." https://github.com/facebook/rocksdb, 2022.

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[15] A. R. Butt, C. Gniady, and Y. C. Hu, "The performance impact of kernel prefetching on buffer cache replacement algorithms," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2005.

[16] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *Proceedings of 2008 USENIX Annual Technical Conference*, 2008.

[17] H.-R. Lee, C.-G. Lee, S. Lee, and Y. Kim, "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*, 2022.

[18] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[19] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with matrix container in nvm," in *Proceedings of 2020 USENIX Annual Technical Conference*, 2020.

[20] X. Wang, P. Jin, B. Hua, H. Long, and W. Huang, "Reducing write amplification of lsm-tree with block-grained compaction," in *Proceedings of the 38th International Conference on Data Engineering*, IEEE, 2022.

[21] R. Wang, J. Wang, P. Kadam, M. T. Özsu, and W. G. Aref, "dlsm: An lsm-based index for memory disaggregation," in *Proceedings of the 39th International Conference on Data Engineering*, IEEE, 2023.

[22] J. Jung and D. Shin, "Lifetime-leveling lsm-tree compaction for zns ssd," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022.

[23] T. Yao, J. Wan, P. Huang, Y. Zhang, C. Xie, and X. He, "Geardb: A gc-free key-value store on hm-smr drives with gear compaction," in *Proceedings of the ACM Turing Award Celebration ference-China 2023*, 2023.

[24] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based ssds," *IEEE Computer Architecture Letters*, vol. 9, no. 1, pp. 9–12, 2010.

[25] J. Axboe, "Flexible i/o tester." https://github.com/axboe/fio, 2022.

[26] Facebook, "db_bench." https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[27] Facebook, "compaction_picker_level.cc::line201." https://github.com/facebook/rocksdb/blob/main/db/compaction/, 2024.

[28] I. Song, M. Oh, B. S. J. Kim, S. Yoo, J. Lee, and J. Choi, "Confzns: A novel emulator for exploring design space of zns ssds," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, 2023.

[29] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The case of femu: Cheap, accurate, scalable and extensible flash emulator," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, 2018.

[30] W. Digital, "Western digital ultrastar dc zn540." https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/ultrastar-dc-zn540-ssd/data-sheet-ultrastar-dc-zn540.pdf.

[31] Q. Wang and P. P. Lee, "Zapraid: Toward high-performance raid for zns ssds via zone append," in *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2023.

[32] RocksDB, "Rocksdb subcompaction wiki." https://github.com/facebook/rocksdb/wiki/Subcompaction, 2022.

[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[34] J. Lee, D. Kim, and J. W. Lee, "Waltz: Leveraging zone append to tighten the tail latency of lsm tree on zns ssd," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 2884–2896, 2023.

[35] M. Bjørling, "Zone append: A new way of writing to zoned storage," in *Proceedings of Linux Storage and Filesystems Conference*, 2020.

[36] H. Bae, J. Kim, M. Kwon, and M. Jung, "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*, 2022.

[37] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, "ezns: An elastic zoned namespace for commodity zns SSDs," in *Proceedings of 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 461–477, July 2023.

[38] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *Proceedings of 13th USENIX Conference on File and Storage Technologies*, 2015.