Towards A Unified Garbage Collection Strategy in ZNS Key-Value Store File Systems Using Same-Victim GC

Hamin Hwangbo[1,*], Joseph Ro[1,*], Sungjin Byeon[1], Safdar Jamil[1], Jun Young Han[2]
Jooyoung Hwang[2], and Youngjae Kim[1,†]
[1]Sogang University, Seoul, Republic of Korea, [2]Samsung Electronics Co.
{hbhm0703, josephro12, sjbyeon, youkim}@sogang.ac.kr, {jy0.han, jooyoung.hwang}@samsung.com

***Abstract*—Zoned Namespace (ZNS) SSDs are gaining traction for eliminating in-device GC and enabling application-aware data management. BlobDB, an enhanced RocksDB with key-value separation, reduces compaction overhead but suffers from the GC over GC (GoG) problem, causing redundant data copying during BlobDB's GC and Zone Cleaning (ZC). To address this, this paper proposes Same-Victim GC, aligning the victims and sizes of both GCs. Specifically, we introduce the BlobDB-Aware Zone Allocation (BAZA) algorithm to allocate blob files by creation order, eliminate victim file mismatch between two GCs, and Z_Cutoff to minimize BlobDB's GC size without additional overhead. Implemented in ZenFS v2.14 and RocksDB v7.4, our solution eliminates valid data copying, doubles compaction performance, and improves space utilization by 1.28×.**

***Index Terms*—Log-Structured Merge-Tree, Key-Value Store, Zoned Namespaces Solid-State Drive**

I. Introduction

Zoned Namespace (ZNS) SSDs [1], [2] are an emerging storage technology that has garnered significant interest from data centers and enterprise storage providers [3]. ZNS SSDs partition their storage space into fixed-size zones that must be written sequentially and erased in one operation using a zone-reset command [1]. Unlike traditional SSDs, ZNS SSDs shift the responsibility for data management and garbage collection (GC) from the device to the host system [4], [5]. This allows the host system to utilize application-specific knowledge to manage these tasks, reducing write amplification (WA) and minimizing I/O interference [6], [7], making ZNS SSDs ideal for applications requiring a log-structured data layout [8].

Meanwhile, Log Structured Merge(LSM)-tree-based key-value stores, such as RocksDB [9], are considered well suited for ZNS SSDs because LSM-tree shows sequential write pattern, so-called append-only. LSM's append-only is a manner which modifies data by compaction, rather than by overwriting the data. The compaction process includes merge-sort and deleting the obsolete files. However, the compaction process induces high WA and write stalls. To relieve the overheads, WiscKey [10] proposed key-value (KV) separation design by decoupling values from keys and storing them separately. BlobDB [11], the state-of-art RocksDB variant, adopts Wisckey's KV separation design, and stores values in separated file named, blob files.

To operate BlobDB on ZNS SSDs, middleware is required to manage file operations for the ZNS interface. BlobDB adopts ZenFS [12], [13], an user-level file system designed to bridge RocksDB with ZNS. When using BlobDB over ZenFS, two garbage collection processes operate at both the application and file system levels: BlobDB's GC and Zone Cleaning (ZC). We refer to this architecture, where two GCs operate simultaneously, as GC over GC (GoG). Currently, since BlobGC and ZC operate independently without awareness of each other, GoG leads to unnecessary copying of valid data during each GC process, negatively impacting overall performance at both levels.

Several works have targeted data copy overheads and the resulting performance degradation occurring in GoG within RocksDB and LevelDB running on ZenFS [4], [5], [8], [14]. However, these methods do not address the management of blob files, whose GC policies differ significantly from those of Sorted String Table (SST) files, making these solutions inapplicable. The key limitation of existing research is that they primarily focus on the compaction process of LSM-trees without considering the unique requirements of key-value separation in LSM-trees. In key-value separation, the LSM-tree size is reduced, making the management and cleaning of invalid values crucial.

To solve these issues, we conducted a thorough case study on the relationship between the BlobDB's GC and ZenFS's ZC. We found that the aforementioned problems are caused by a mismatch between the victim files and the size of GCs. To address this, we proposed a new method called Same-Victim GC, which aligns both the victim files and size of GCs for BlobDB and ZenFS (§III). To achieve Same-Victim GC, we introduce the BlobDB-Aware Zone Allocation (BAZA) algorithm (§IV-A), which allocates blob files sequentially to zones based on their creation order. BAZA is designed based on the observation that changing the zone allocation algorithm to align the lifetime of blob files, which is decided by "Oldest first" policy of BlobDB's GC, reduces the mismatch in victim files for both GCs. Additionally, for further improvement, we proposed Z_Cutoff, which adjusts the size of BlobDB's GC to match the size of ZC (§IV-C). However, challenges such as the Write Pointer (WP) locking problem, where WPs must be locked to ensure successful sequential writes, complicate the BAZA algorithm. BAZA addresses this by monitoring and adjusting the allocation information of ZenFS to achieve optimal Same-Victim GC.

The key contribution of this paper is as follows:

[*]Both are first co-authors and have contributed equally.
[†]Y. Kim is the corresponding author.

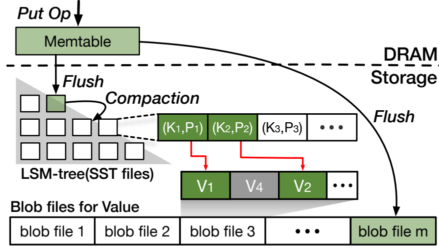


Fig. 1: Description of KV separation in BlobDB.

- Identified that mismatches in the victim files and sizes of GCs lead to inefficiencies, thus proposing Same-Victim GC, which matches victims and size of both GCs.
- Proposed the BAZA algorithm to eliminate mismatch between both GC victims. Moreover, Z_Cutoff minimize mismatch between size of GCs.
- Addressed the WP locking problem and overcome the problem by implementing a monitoring and adjustment mechanism within the BAZA algorithm, achieving optimal Same-Victim GC while ensuring successful sequential writes.

Extensive evaluations, tested on ConfZNS [15] and a ZN540 [16], demonstrate that employing BAZA effectively eliminate valid data copy during ZC, and improve compaction performance by 2× across both db_bench and YCSB.

II. Background

A. WiscKey-Based Key-value Store

To minimize write and space amplification of traditional LSM-tree, WiscKey [10] proposed a key-value (KV) separation design. This design decouples values from keys, storing values in append-only log files. The state-of-the-art RocksDB [9] adopts the traditional LSM-tree architecture, whereas its variant, BlobDB [11], utilizes the KV separation design, as illustrated in Figure 1. Both KV stores follow the common interface for PUT operations where KV pairs are stored in MemTable and later written to persistent storage by flush operation. However, in BlobDB, the flush operation first writes the values to append-only log files, known as blob files, and then directs the keys and value pointers to SST files. Therefore, with the KV separation design, updating or deleting KV pairs results in obsolete values in the corresponding blob file. These values must be reclaimed by a GC process called BlobDB Garbage Collection (BlobGC), which is incorporated the BlobGC in the compaction.

B. Compaction and BlobGC in BlobDB

compaction in BlobDB is triggered based on the total size of SST files in each level of LSM-tree as RocksDB does. The compaction in BlobDB not only merge-sort the SST files, based on overlapping key-range, to reclaim the obsolete KV pointer but it also marks the corresponding values in blob files as obsolete. These obsolete values are reclaimed by the BlobGC. Figure 2 illustrates the compaction and BlobGC in BlobDB. The compaction in Figure 2 selects victim SST files and perform merge-sort on them. In Figure 2, the compaction encounters an updated entry of (K_1, P_4), therefore marking the

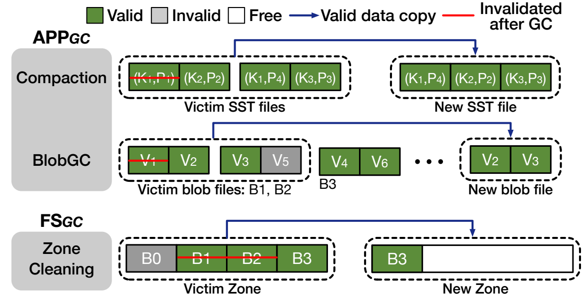


Fig. 2: Description of App_{GC} and FS_{GC}: App_{GC} involves the compaction process and BlobGC, while FS_{GC} refers to zone cleaning.

older entry (K_1, P_1) as invalid which will later be reclaimed by BlobGC.

While BlobDB performs compaction on the LSM-tree, BlobGC is executed concurrently. The BlobGC reclaims values for deleted or invalidated key-value pointers, retaining only the valid values corresponding to unique keys. As shown in Figure 2, during the compaction, the obsolete key-value pointer (K_1, P_1) is invalidated. BlobGC then invalidates the value V_1 that P_1 was pointing to. Additionally, valid values V_2 and V_3, pointed to by the copied value pointers P_2 and P_3, are copied. The reason V_4 is not copied is that the blob file storing V_4 is not included among the victim blob files. Victim blob files are selected based on their creation index, following the "Oldest First Policy", which selects the oldest blob files up to a threshold defined by $number\ of\ blob\ files \times cut_off\ ratio$. In the Figure 2, since cut_off ratio is set as 0.25 and there are a total of 8 blob files, the number of victim blob files is set to 2 $(= 8 \times 0.25)$. After the compaction and BlobGC processes copy the unique key-value pointers and their corresponding values to the new SST and blob files, BlobDB deletes the victim SST and blob files. Since compaction is completed only after BlobGC finishes, a prolonged BlobGC process negatively impacts the overall performance of compaction.

C. Zone Allocation and Zone Cleaning in ZenFS

To adopt BlobDB over ZNS SSDs, ZenFS performs two major functions: Zone Allocation and Zone Cleaning (ZC) [5]. Zone Allocation involves assigning zones for file placement. Since WiscKey separates keys and values, ZenFS allocates SST files and blob files into different zones. Specifically, ZenFS uses the Lifetime-Based Zone Allocation (LIZA) algorithm for SST files, assigning unique lifetime hints based on the SST file's level in the LSM-tree [4], [5]. Using LIZA, ZenFS ensures that SST files with the same lifetime hints are allocated within the same zone. However, there has been no previous work addressing the allocation of blob files. The current ZenFS approach is to allocate zones to blob files using LIZA. Thus, in this paper, we assume the default allocation algorithm for blob files in ZenFS as LIZA. With LIZA, ZenFS assigns blob files the lifetime hint of the referencing SST file. As a result, blob files with the same lifetime hint are allocated to the same zone.

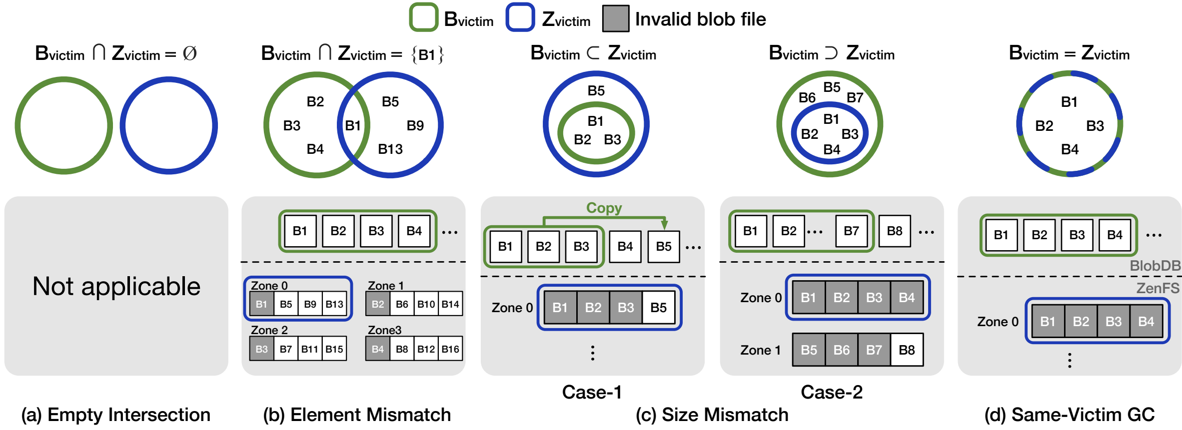


Fig. 3: Comprehensive explanation of multiple cases of GoG in the BlobDB-ZenFS system.

After SST files and blob files are deleted by compaction and BlobGC, the space in the storage they occupied becomes invalid. ZC reclaims this invalid space and converts it into free space. ZC is triggered when the device's free space falls below 20%. During ZC, ZenFS copies valid data from the victim zone to target zones and performs a zone reset on the victim zone. Figure 2 illustrates the ZC process in ZenFS. As shown, since B0 was already invalidated, B2 and B3 are invalidated during BlobGC, only B3 is copied into a new zone. ZC is triggered when the device's free space is limited, preventing foreground I/O operations. This blocking of foreground I/O negatively impacts performance, making it crucial to minimize the adverse effects of ZC. To mitigate such effects, ZC issues a zone-reset command whenever all data within a zone is invalid, even if the free space is not below the trigger point.

III. Analysis of Overhead Due to GC over GC

As shown in Figure 2, BlobDB and ZenFS reclaim invalidated space through BlobGC and ZC. We define this situation where GC is executed at both layers as GC over GC (GoG). In GoG, BlobGC and ZC operate independently without awareness of each other, leading to unnecessary copying of valid data during each GC process. GoG can also occur between the compaction of the LSM-tree and ZC. However, due to the KV separation design, the LSM-tree remains small, resulting in minimal compaction overhead. Therefore, in this paper, we focus on the GoG issue arising between BlobGC and ZC. In this section, we categorize various cases resulting from GoG and analyze their associated problems in current system.

A. Analysis of Multiple Cases in GoG

To facilitate explanation, let's denote the set of blob files selected as victims during BlobGC as B_{victim} and the set of blob files contained within a zone selected as a victim during ZC as Z_{victim}. We will refer to the blob files within these sets as "elements". The notation B_i indicates the i-th indexed blob file. Additionally, $S(B_{victim})$ represents the total size of the blob files in B_{victim}, and $S(Z_{victim})$ represents the total size of the blob files in Z_{victim}. Since the two GCs operate independently and are unaware of each other's execution, the intersection of the elements in each set can vary. Moreover, while the zone size is fixed, making $S(Z_{victim})$ constant, $S(B_{victim})$ varies depending on the number of blob files at the time BlobGC occurs. Due to the potential differences in the intersection of elements and the varying sizes of the two sets, there are theoretically four types of relationships that can exist between B_{victim} and Z_{victim}. Figure 3 illustrates the four possible relationships between the two GC sets:

- **Empty Intersection:** Figure 3(a) shows a case where the elements of the two sets do not overlap at all, resulting in an empty intersection. In this case, the sizes of the two sets can be either the same or different.
- **Element Mismatch:** Figure 3(b) depicts a case where the sets have one or more overlapping elements but do not completely overlap. The sizes of the two sets can vary.
- **Size Mismatch:** Figure 3(c) represents a case where the elements match such that the intersection is equivalent to one of the sets, but the sizes of the two sets differ.
- **Same-Victim GC:** Figure 3(d) shows the case where both the elements and sizes of the two sets match perfectly.

Although there are theoretically four possible relationships between BlobGC and ZC, in practice, the Empty Intersection does not occur. As previously mentioned, ZC selects a zone as Z_{victim} based on the blob files that have been deleted by BlobGC. If no blob file in a zone is selected as B_{victim}, that zone will not be selected as Z_{victim}. Therefore, in a zone, BlobGC always occurs first, followed by ZC. Consequently, cases where every elements of B_{victim} and Z_{victim} mismatches do not arise, and we do not consider the Empty Intersection in our analysis.

B. Element Mismatch

Figure 3(b) illustrates the Element Mismatch scenario. BlobDB selects B1 through B4 as B_{victim}. Meanwhile, ZenFS selects Zone 0 as the victim, choosing B1, B5, B9, and B13 as Z_{victim}. Since their intersection includes only B1, after BlobGC occurs, only B1 from the Z_{victim} set will be deleted. Consequently, ZC has to copy three other valid blob files, which is 75% of the total zone size. This means ZC can only reclaim 25% of the zone's space, which is inefficient. Even if different zones were selected as victims, the result would

be similar. If $S(B_{victim})$ changes, the intersection between B_{victim} and Z_{victim} remains small, leading to ZC reclaiming only a low percentage of the zone's space. Thus, Element Mismatch results in the copying of valid data during ZC, leading to inefficient ZC.

C. Size Mismatch

Size Mismatch refers to a scenario where the intersection of the two sets is equivalent to one of the sets, but the sizes of the two sets differ. In the case of Size Mismatch, we can consider two possibilities: $S(B_{victim})$ being smaller or larger than the fixed size $S(Z_{victim})$. Figure 3(c) Case-1 illustrates the situation where $S(B_{victim}) < S(Z_{victim})$. B_{victim} includes blob files B1 through B3. As a result of BlobGC, a new blob file, B5, is created. If B5 is allocated to Zone 0 or any zone with a high invalid data ratio, it may be copied again during ZC. Thus, when $S(B_{victim}) < S(Z_{victim})$, the same data often gets copied twice, leading to an increase in unnecessary data copying during ZC.

In contrast, Figure 3 (c) Case-2 illustrates the scenario where $S(B_{victim}) > S(Z_{victim})$. B_{victim} includes blob files B1 through B7. After BlobGC completes, ZC selects Zone 0, which contains only invalid data, as Z_{victim} and reclaims it without any additional valid data copying. However, if we consider the elements of Z_{victim}, B_{victim} could be set to include only B1 through B4 to reclaim Zone 0. In this case, since B_{victim} is larger than necessary compared to the $S(Z_{victim})$, it unnecessarily extends the BlobGC process. As mentioned in Section II-A, BlobGC is tightly integrated with compaction. Therefore, when $S(B_{victim}) > S(Z_{victim})$, the time for BlobGC is prolonged, leading to an increase in compaction duration.

D. Same-Victim GC

In this section, we analyze the scenario where Element Mismatch and Size Mismatch are resolved. We define **Same-Victim GC** where the elements and size of both sets match. Figure 3(d) illustrates Same-Victim GC. B1 to B4 are selected as B_{victim}, and Z_{victim} also includes B1 to B4. First, BlobGC copies the valid data from B1 to B4 to create the B5 blob file in another zone, then invalidates B1 to B4. After BlobGC, ZC can immediately reclaim the space occupied by B1 to B4 without copying any valid blob files. Compared to the previously mentioned cases of Element Mismatch and Size Mismatch, Same-Victim GC eliminates unnecessary data copying during ZC. Additionally, compared to Figure 3(c) Case-2, Same-Victim GC reduces the $S(B_{victim})$, allowing BlobGC to complete in a shorter time.

However, the current BlobDB-ZenFS system rarely achieves Same-Victim GC and frequently encounters Element Mismatch and Size Mismatch. This is because the two GCs operate independently at different layers without awareness of each other. In the next section, we analyze why the current BlobDB-ZenFS system rarely achieves Same-Victim GC. We then propose our novel algorithms, BAZA and Z_Cutoff, which are designed to enable Same-Victim GC.

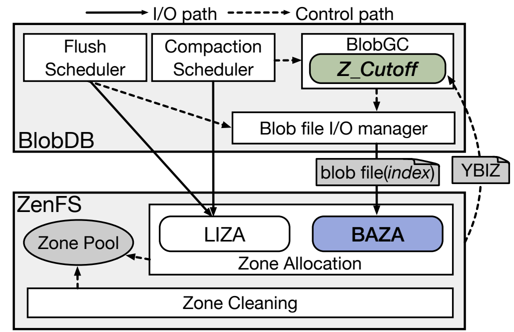


Fig. 4: The overview of BAZA and Z_Cutoff.

IV. Design towards Same-Victim GC

In this section, we propose two novel techniques, BAZA and Z_Cutoff, which is collaboration between the BlobDB and the ZenFS. The design goal of both techniques is to resolve Element Mismatch and Size Mismatch, thus induce Same-Victim GC. Figure 4 presents a software architecture overview of BlobDB and ZenFS with BAZA and Z_Cutoff. In the ZenFS layer, BAZA replaces LIZA for blob file allocation. BAZA is a Zone Allocation technique in ZenFS that resolves the Element Mismatch (§ IV-A). By resolving the Element Mismatch in the two victim sets, BAZA reduces unnecessary valid data copying during ZC.

However, ZNS has a limitation known as the WP-locking problem, which prevents BAZA from placing blob files as intended (§IV-B). To address the WP-locking problem, which is a challenge in our design, we propose Z_Cutoff, a technique in BlobDB layer and resolves additional Element Mismatch and Size Mismatch. To do this, Z_Cutoff uses the Youngest Blob Index in Zone (YBIZ) for each zone (§IV-C). As a result, Z_Cutoff reduces the compaction overhead caused by the increased $S(B_{victim})$ and decreases unnecessary valid data copying during ZC.

A. BAZA : BlobDB-Aware Zone Allocation algorithm

In this section, we analyze the root causes of Element Mismatch induced by LIZA, the current blob file allocation algorithm in ZenFS. Furthermore, we explain the detailed workings of BAZA, a novel approach aimed at addressing Element Mismatch. As discussed earlier in Section II-C, the current ZenFS assigns the lifetimes of blob files based on the lifetimes of the SST files they reference, without considering the "Oldest First" policy of BlobGC. Specifically, the lifetime of an SSTable is assigned according to its level; the lower the level, the smaller the size limit, and the more frequent the compactions. Thus, ZenFS estimates that SSTables at lower levels contain hot data [4].

However, blob files are not deleted when their corresponding SST files are removed. Instead, they follow an "Oldest First" policy, where the oldest blob files are deleted first. Consequently, the timing of deleting SST files and their associated blob files differs, requiring separate lifetime estimation algorithms for each. For example, consider a scenario where a level 0 SST file (S) and its corresponding blob file (B) are

created due to a flush. Since this SST file (S) is generated at level 0, ZenFS classifies it as hot data. In contrast, the blob file (B), being the most recently created, will be the last to be deleted under the "Oldest First" policy. Thus, while (S) is hot data, (B) would likely be relatively cold data, reflecting the different lifetimes of the referencing SST file and the blob file. However, LIZA results in blob files with different lifetimes being placed in the same zone, leading to an Element Mismatch.

> *Observation 1:* Currently, ZenFS assigns the lifetime of a blob file based on the lifetime of its referencing SST file. However, because this assignment does not accurately reflect the actual lifetime of the blob file, it results in an Element Mismatch.

Therefore, to resolve the Element Mismatch, we propose BAZA, which allocates blob files in ascending order based on their indices. The ascending order placement method in BAZA involves sequentially placing files according to their creation order (index). Since the "Oldest First" policy also deletes files in sequence from oldest to newest, BAZA can select blob files allocated in ascending order within a single zone as B_{victim}, which can then be selected as Z_{victim} by the ZC.

B. Design Challenges : WP-locking problem

However, due to the WP-locking problem, the ascending allocation method of BAZA is not always feasible. The WP-locking problem arises from the characteristic that writing to a zone can only occur sequentially at the WP [2], leading multiple threads cannot write to a single zone concurrently. For instance, if there are two write threads and thread 1 is writing to a specific zone, the WP of that zone is locked, preventing thread 2 from writing to the same zone and necessitating writing to a different zone. Additionally, LSM-trees frequently involve multiple threads executing write I/O operations to enhance write performance [17], [18].

Figure 5 illustrates two scenarios where multiple blob file writing threads execute write I/O in BlobDB. In the case of writing B3 and B4, a single thread requests writes to Zone 0, successfully achieving sequential ascending allocation. Conversely, for case of writing B7 and B8, two threads concurrently request writes. While thread 1 allocates and writes B7 to Zone 1, thread 2 cannot write B8 to Zone 1 and must allocate it to Zone 2. If a write request for a blob file arrives after the completion of B7's write, the new blob file (B9) will again be sequentially allocated to the existing zone (Zone 1). This situation prevents BAZA from achieving the desired ascending allocation of blob files. The next section will describe Z_Cutoff, an algorithm for setting $S(B_{victim})$ that takes WP-locking problem into account.

C. Z_Cutoff: Optimizing $S(B_{victim})$ Using Zone Allocation Information

In this section, we analyze the root causes behind the current BlobGC inducing Size Mismatch, and propose Z_Cutoff, an algorithm that addresses the WP-locking problem while resolving Size Mismatch. As discussed earlier in Section II-B, the current BlobGC selects B_{victim} as the oldest blob files amounting to $number\ of\ blob\ files \times cut_off\ ratio$. This policy causes the $S(B_{victim})$ to vary dynamically based on the total number of blob files, often resulting in $S(B_{victim})$ being either larger or smaller than $Z(B_{victim})$, thereby leading to Size Mismatch.

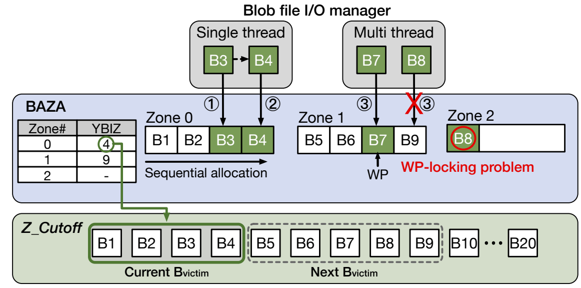


Fig. 5: Description of two allocation scenarios in BAZA and the process of Z_Cutoff setting B_{victim}.

To resolve Size Mismatch, a simple approach would be to set the size of $S(B_{victim})$ equal to the zone size. However, due to the WP-locking problem, this approach can lead to Element Mismatch again. For example, as shown in Figure 5, if BlobGC sets B_{victim} for a zone affected by WP-locking problem (Zone 1) to match the zone size with files (B5, B6, B7, B8), it results in Element Mismatch for blob files B8 and B9. To overcome this, we need a more sophisticated approach that addresses the WP-locking problem while minimizing $S(B_{victim})$, since increased $S(B_{victim})$ leads to prolonged compaction times.

> *Observation 2:* Due to the WP-locking problem in ZNS, BAZA may fail to achieve perfect index-based ascending sequential allocation. This can, in turn, lead to Element Mismatch again.

To address the Element Mismatch problem caused by the WP-locking problem, we propose Z_Cutoff, which adjusts $S(B_{victim})$ by setting the $cut_off\ ratio$ based on the Zone Allocation information. Below are how Z_Cutoff determines $cut_off\ ratio$: Z_Cutoff references the Youngest Blob Index in Zone (YBIZ), which is the index of the most recently created blob file in each zone. When a zone changes to a full state, BAZA assigns the YBIZ value to that zone, since full zone only can be considered as Z_{victim}. Z_Cutoff uses the smallest YBIZ value among all zones to adjust the $S(B_{victim})$ from the oldest blob file index up to this minimum YBIZ value. By calculating the number of blob files that BlobGC will delete based on this minimum YBIZ value, Z_Cutoff minimizes the compaction time.

$$cut_off\ ratio = \frac{\min_Z(YBIZ_Z) - \min(blob\ file\ index)}{N} \quad (1)$$

Equation 1 shows the equation for the $cut_off\ ratio$ derived from the YBIZ value. Here, N represents the to-

tal number of blob files, and Z denotes the zone index. Z_Cutoff finds the minimum YBIZ value, calculates the number of blob files up to this value, and then dynamically computes the $cut_off\ ratio$ by dividing this result by N.

For example, in Figure 5, the YBIZ for Zone 0 is 4, and the YBIZ for Zone 1 is 9. Since Zone 2 is not in a full state, it is not considered for Z_{victim}, thus B8 in Zone 2 is excluded from the YBIZ. When BlobGC happens, Z_Cutoff uses the smallest YBIZ value, which is 4, to derive the $cut_off\ ratio$. In this example, with a total of 20 blob files, the $cut_off\ ratio$ is calculated as $4/20 = 0.2$. Consequently, in Zone 0, a Same Victim-GC occurs since B1, B2, B3, and B4 become the victim. However, as previously mentioned, the WP-locking problem can cause BAZA to fail in achieving allocating files in ascending order, like B8 and B9 in the Zone 1 and Zone 2. When the next BlobGC happens, Z_Cutoff derives the $cut_off\ ratio$ based on the smallest YBIZ value, which is 9, and the current total number of blob files, which is now 16 (since B1, B2, B3, and B4 have already been deleted). This results in a $cut_off\ ratio$ of $5/16 = 0.3125$. This approach minimizes Size Mismatch between B_{victim} and Z_{victim} and reduces the amount of valid data copied during ZC, offering a suboptimal solution for Same-Victim GC.

V. Evaluation

A. Experimental setup

For the evaluation of BAZA and Z_Cutoff, we used Configurable ZNS (ConfZNS) [15], a ZNS SSD emulator based on FEMU [19]. We emulated a 64GB ZNS SSD environment with 16 Intel(R) Xeon(R) Gold 5218R CPUs and 16GB of memory. To experiment on the large zone ZNS SSDs [16], we set the zone size to 1024MB. To maintain compatibility with ZNS SSD, we modified ZenFS v2.14 and RocksDB v7.4. For the sake of experiment, we configured BlobDB to utilize 2 compaction threads, 2 flush threads, 4 subcompaction [18] threads, and 4 max open files.

We compared the following three schemes:

- **Baseline**: Current BlobDB-ZenFS system.
- **BAZA**: Only BAZA algorithm is adopted.
- **BAZA+**: Both BAZA and Z_Cutoff are adopted.

Workloads: We conducted evaluations using both synthetic and realistic benchmarks. For the synthetic benchmark, we used db_bench, a tool bundled with RocksDB that offers micro-benchmarks for various workload patterns. Specifically, we utilized the fillrandom workload with size of 36GB for our analysis. For realistic benchmarks, we used three workloads from the Yahoo Cloud Serving Benchmark (YCSB) with a *zipfian distribution* (zipfian constant of 0.99). We loaded 24GB of data first and performed operations on a 24GB dataset based on the workload type. Below are explanation of three types workload with YCSB:

- **WL(A)**: 80% update & 20% insert (Write-only)
- **WL(B)**: 80% update & 20% read (Update-intensive)
- **WL(C)**: 50% update & 50% read (Mixed Workload)

In all workloads, the key size is set to 16 bytes, and the value size is set to 128KB. We set both SST and blob file sizes to 64MB. Every experiment's results are the mean of at least three independent runs.

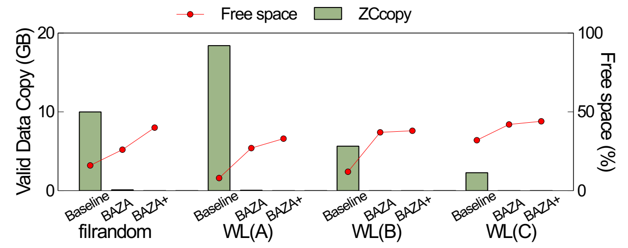


Fig. 6: File system level overhead.

B. Analysis of Same-Victim GC Effect in File System

To compare the efficiency of three schemes in ZenFS, we evaluated the amount of valid data copied during ZC and the free space remaining after completing the workload. Since our research focuses on the interaction between BlobGC and ZC, the valid data copy was measured exclusively for blob files. The left y-axis and bar graph in Figure 6 illustrate the amount of valid data copied during ZC. In the Baseline, valid data copies occurred during ZC across all workloads. For instance, in the case of WL(A), approximately 19GB of valid data were copied, which, given the total write amount of 24GB, indicates that 80% of the data were redundantly written due to ZC. This redundancy arises because the blob files within the zones are not sequentially allocated, leading to Element Mismatch. In contrast, BAZA showed significantly reduced data copy during ZC, with only 105MB and 53MB of data copied during the fillrandom and WL(A) workloads, respectively, and no data copy occurring in other workloads. Moreover, BAZA+ completely eliminated valid data copy during ZC. The elimination of data copying in these schemes is attributed by BAZA algorithm, which is designed considering the BlobGC's "Oldest First" policy, effectively resolving Element Mismatch. As a result, when BlobGC occurs, all blob files within the zone become invalidated, allowing ZC to simply issue a zone-reset command to reclaim space. The slight data copy observed in BAZA is due to Size Mismatch, which causes valid data copy when $S(B_{victim}) < S(Z_{victim})$.

The right y-axis and line graph in Figure 6 illustrate the free space remaining on the device after performing the workloads. In the Baseline scheme, the workloads occupy an average of 83% of the space across the four workloads. In contrast, BAZA uses only 67% of the space, and BAZA+ uses just 61% of the space to perform the same workloads. This difference arises due to the Element Mismatch in the Baseline scheme, leading to the failure of assigning blob files with similar lifetimes to the same zone. Consequently, invalid blob files are partially mixed within the zones, causing space amplification and resulting in 22% more zone usage compared to BAZA+. On the other hand, both BAZA and BAZA+ resolve the Element Mismatch, using 16% and 22% less device capacity, respectively, to complete the workloads. As a result, BAZA algorithm ensures that files with similar lifetimes are allocated within the zones, resolving Element Mismatch, leading to (1) no data copy during ZC, and (2) improved space utilization.

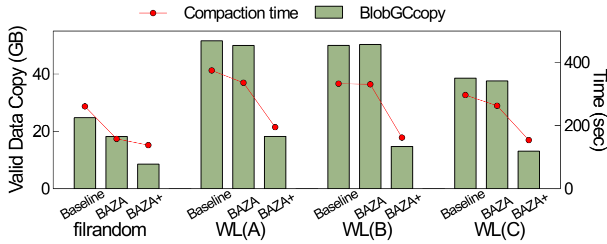


Fig. 7: Application level overhead.

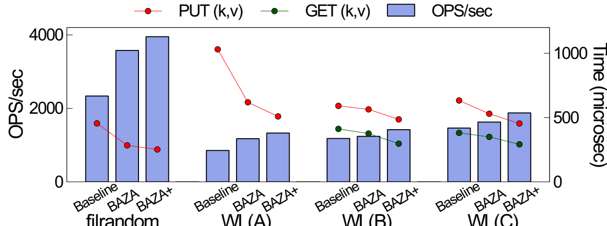


Fig. 8: Overall system performance.

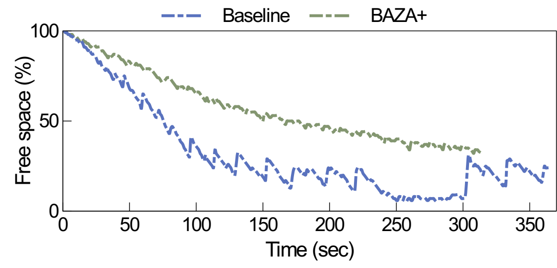
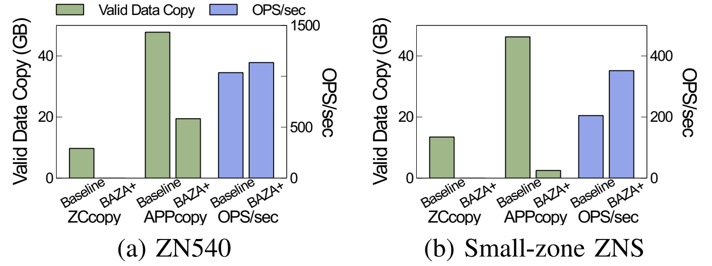


Fig. 9: Microscopic analysis of Baseline and BAZA+.



(a) ZN540 (b) Small-zone ZNS

Fig. 10: Experiments on different settings of ZNS devices.

C. Analysis of Same-Victim GC Effect in Application

To compare the efficiency of three schemes in BlobDB, we evaluated the amount of valid data copied during BlobGC and the compaction time. Figure 7 illustrates the valid data copy during BlobGC and the cumulative compaction time for the four workloads. In BAZA, the valid data copy during BlobGC was similar to the Baseline. This similarity is because both the Baseline and BAZA use the same algorithm to adjust the size of $S(B_{victim})$, resulting in comparable amounts of valid data copy during BlobGC. However, the compaction time for BAZA was reduced by an average of 17% compared to the Baseline. This reduction is because write stalls during compaction, caused by valid data copying during ZC, are minimized in BAZA compared to the baseline. For BAZA+, the amount of valid data copy during BlobGC decreased by an average of 33% across all workloads compared to the Baseline. This improvement is because BAZA+ adjusts $S(B_{victim})$ to closely match the zone size through Z_Cutoff, minimizing the effect of Size Mismatch. Additionally, the compaction time in BAZA+ was reduced by 50%. This reduction is attributed to the elimination of valid data copying during ZC, which reduces write stalls during compaction, and the smaller $S(B_{victim})$, which decreases BlobGC time.

D. Analysis of Same-Victim GC Effect in Overall Performance

In this section, we conduct an overall performance comparison across four workloads. We evaluate throughput, average PUT performance, and average GET performance. Since the fillrandom and WL(A) workloads do not include any GET operations, their GET performance results are not included. As shown in Figure 8, throughput improves by 1.27× in BAZA and by 1.44× in BAZA+ compared to the Baseline. The increase in throughput for BAZA is due to the reduction in valid data copying during ZC, which prevents write stalls. BAZA+ further benefits from reduced valid data copying during BlobGC, enhancing overall performance. In the previous section, we observed that BAZA reduces compaction time by 17% and BAZA+ by 50% compared to the Baseline. Consequently, for all four workloads, the average PUT performance improves by 1.37× in BAZA and by 1.6× in BAZA+. For GET performance, BAZA shows an average improvement of 1.1×, and BAZA+ shows an improvement of 1.34× for the WL(B) and WL(C). This improvement is also attributed by the reduction in compaction time.

E. Microscopic Analysis

Figure 9 shows a time-series analysis comparing the Baseline and BAZA+ in terms of remaining free space and execution time for WL(A). As shown, the Baseline takes 48 seconds longer than BAZA+. The Baseline experiences write stalls and increased compaction time when free space drops below 20% due to valid data copying during ZC, caused by Element Mismatch. As we observed in previous sections, the Baseline involves significant valid data copying during both GC processes, resulting in poorer space utilization compared to BAZA+ over the same time period. In contrast, BAZA+ effectively reclaims free space quickly during ZC without valid data copying, with the Same-Victim GC approach. This leads to relatively better space utilization compared to the Baseline.

F. Experiment on Real ZNS Device

To validate our previous experiments, we conducted tests using a real device. We used the ZN540 Western Digital NVMe ZNS SSD, which consists of 904 zones, each with a capacity of 1077MB, and a total size of 1TB [16]. For ease of experimentation, we mounted only 64 zones in ZenFS and conducted experiments on WL(A). Figure 10(a) shows the comparison results of valid data copy during ZC, valid data copy during BlobGC, and throughput between the Baseline and BAZA+. As illustrated, the Baseline induces approximately 9GB of valid data copy during ZC, whereas BAZA+ eliminates valid data copy. This improvement is due to resolving Element Mismatch. For BlobDB, BAZA+ reduces valid data copy

during BlobGC by approximately 60% by also addressing Size Mismatch. In terms of throughput, BAZA+ outperforms the Baseline by about 1.09×. Therefore, we confirmed that our BAZA+ also enhances performance on a real ZNS device, demonstrating the practical benefits of our approach.

G. Experiment on Small-Sized Zone ZNS

For ZNS SSDs, the zone size varies depending on the manufacturer [16], [20]. To observe the effects in cases with smaller zone sizes, we used ConfZNS to set the zone size to 64MB, and used it as a small-sized zone ZNS. To ensure that multiple blob files are allocated within a single zone, we set the blob file size to 32MB. Figure 10(b) compares the Baseline and BAZA+ in terms of valid data copy during ZC, valid data copy during BlobGC, and throughput in WL(A). As shown, during ZC, the Baseline results in approximately 13GB of valid data copy, whereas BAZA+ eliminates valid data copy entirely. For BlobGC, BAZA+ reduces valid data copy to just 0.05% of that in the Baseline. This significant effect is more pronounced than in large-sized zone ZNS. The reason for these results is that as the zone size decreases, the size of $S(B_{victim})$ also decreases, leading to decreased BlobGC overhead. In terms of throughput, BAZA+ improves performance by 1.72× over the Baseline even in small-sized zone ZNS. Therefore, we confirmed that our BAZA+ enhances performance in ZNS devices with small-sized zones, demonstrating its effectiveness across different zone configurations.

VI. Related Work

Efforts to implement Same-Victim GC for the LSM-tree-based KV Store (RocksDB) and ZNS-supported file system (ZenFS) have been made before. Lee et al. [5] proposed the CAZA algorithm, which allocates SST files into zones based on lifetimes derived from RocksDB's compaction. Jung et al. [14] modified the compaction to align with ZenFS's ZC policy, enabling compaction of SSTables within the same zone. However, these solutions do not extend to blob files, whose management and GC policies differ from SST files. Additionally, these approaches address either the RocksDB level or the ZenFS level individually. Effective Same-Victim GC requires coordinated GC between both levels. Our solution focuses on coordinating between the application and the file system, leveraging the characteristics of BlobDB and the advanced functionalities of ZenFS for efficient data management.

Several works have addressed inefficiencies between log-structured file systems like F2FS [21] and FTL in SSD. Yang et al. [22] analyzed the interactions between logs in multiple layers, highlighting issues caused by unaligned segment sizes and uncoordinated GC. Yoo et al. [23] aimed to eliminate redundancies in GC by leaving most FTL functions to the log-structured file system. However, these approaches are based on interactions between F2FS and traditional SSDs, not ZNS SSDs. While ZNS systems benefit from an interface that matches the physical data layout on flash drives and hides flash hardware complexities, further coordination with the application layer is still necessary.

VII. Conclusion

This paper proposes Same-Victim GC to address the GC over GC (GoG) problem between BlobDB and ZenFS. Specifically, the BlobDB-aware Zone Allocation for Same-Victim GC (BAZA) and Z_Cutoff eliminate mismatches in victim files and the size of the GC process in BlobDB and ZenFS. The Same-Victim GC reduces valid data copying overhead and enhances compaction performance. Evaluations show a 33% reduction in data copying during BlobGC, a 1.5× improvement in compaction time, and the elimination of data copying in ZC.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386 and RS-2024-00416666), and in part by Samsung Electronics Co., Ltd. (IO221014-02908-01).

References

[1] 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 *USENIX Annual Technical Conference*, 2021.

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

[3] J. Y. Ha and H. Y. Yeom, "zceph: Achieving high performance on storage system using small zoned zns ssd," in *38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, 2023.

[4] S. Byeon, J. Ro, J. Y. Han, J.-U. Kang, and Y. Kim, "Ensuring Compaction and Zone Cleaning Efficiency through Same-Zone Compaction in ZNS Key-Value Store," in *38th International Conference on Massive Storage Systems and Technology*, MSST '24, 2024.

[5] 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 *14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022.

[6] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and W. Lloyd, "Don't be a blockhead: Zoned namespaces make work on conventional ssds obsolete," in *13th USENIX Workshop on Hot Topics in Operating Systems*, 2021.

[7] M. K. M. J. Hanyeoreum Bae, Jiseon Kim, "What you can't forget: exploiting parallelism for zoned namespaces," in *14th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, 2022.

[8] Q. Y. C. G. W. X. Gaoji Liu, Chongzhou Yang and Z. Cao, "Prophet: Optimizing lsm-based key-value store on zns ssds with file lifetime prediction and compaction compensation," in *38th International Conference on Massive Storage Systems and Technology*, MSST '24, 2024.

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

[10] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *14th USENIX Conference on File and Storage Technologies*, 2016.

[11] Facebook, "Blobdb." https://github.com/facebook/rocksdb/wiki/BlobDB.

[12] 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 *15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023.

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

[14] J. Jung and D. Shin, "Lifetime-Leveling LSM-Tree Compaction for ZNS SSD," in *14th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, 2022.

[15] 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 *16th ACM International Conference on Systems and Storage*, SYSTOR '23, 2023.

[16] W. Digital, "Western digital ultrastar dc ZN540." https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-zn540-nvme-ssd, 2021.

[17] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *USENIX Annual Technical Conference*, ATC '19, 2019.

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

[19] 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 *16th USENIX Conference on File and Storage Technologies*, FAST '18, 2018.

[20] K. K. Minwoo Im and H. Yeom, "Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism," in *23rd International Middleware Conference Industrial Track*, 2022.

[21] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *13th USENIX Conference on File and Storage Technologies*, FAST '15, 2015.

[22] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, 2014.

[23] J. Yoo, J. Oh, S. Lee, Y. Won, J.-Y. Ha, J. Lee, and J. Shim, "Orcfs: Orchestrated file system for flash storage," *ACM Trans. Storage*, 2018.