Compaction-Aware Zone Allocation for LSM based Key-Value Store on ZNS SSDs

Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim
Department of Computer Science and Engineering
Sogang University, Seoul, Republic of Korea
{heerock,changgyu,seungjinn,youkim}@sogang.ac.kr

ABSTRACT

Unlike traditional block-based SSDs, Zoned Namespace (ZNS) SSDs expose storage through the zoned block interface, completely eliminating the need for in-device garbage collection (GC) and relinquishing this responsibility to applications. As a result, application-aware data placement decisions give the opportunity for applications on the host to perform efficient GC. Meanwhile, RocksDB for ZNS SSD places data with similar invalidation times (lifetimes) in the same zone through ZenFS (a user-level file system) using the LIfetime-based Zone Allocation algorithm (LIZA), and minimizes the GC overhead of valid data copy when reclaiming a zone. However, LIZA, which allocates zones by predicting the lifetime of each SSTable according to the level of the hierarchical structure of the LSM-tree, is very inefficient in minimizing the write amplification (WA) problem due to inaccurate predictions of SSTable lifetimes. Instead, based on our observation that the deletion time of SSTables in the LSM-tree is solely determined by the compaction process, we propose a novel Compaction-Aware Zone Allocation algorithm (CAZA) that allows the newly created SSTables to be deleted together after merging in the future. CAZA is implemented in RocksDB's ZenFS and our extensive evaluations show that CAZA significantly reduces the WA overhead compared to LIZA.

CCS CONCEPTS

• **Information systems → Storage management**.

KEYWORDS

Key-Value Store, Log-structured Merge-tree, ZNS SSD

HotStorage'22, June 27–28, 2022, Virtual Event, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539743

ACM Reference Format:
Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-Aware Zone Allocation for LSM based Key-Value Store on ZNS SSDs. In *Proceedings of 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3538643.3539743

1 INTRODUCTION

ZNS SSD [7] introduces the concept of a zone that forces sequential writes and disallows overwrites. Due to restrictions on write patterns, ZNS SSDs do not perform Garbage Collection (GC) in the Flash Translation Layer (FTL). The removal of in-device GC improves I/O performance by eliminating write amplification (WA) [1, 12, 16, 19]. However, in order to free up space inside the device, applications using ZNS SSDs must perform zone cleaning, which is an operation that erases a whole zone. If valid data remains in the zone selected as the victim for zone cleaning, the WA problem still exists due to the copying of valid data [4, 16].

Nevertheless, applications can make application-aware data placement decisions, reducing WA by minimizing the amount of data copied during zone cleaning. To reduce this amount, data with the same lifetime should be written in the same erase unit (which is a zone for ZNS SSDs). Applications running on top of ZNS SSDs can directly determine the zone in which data will be placed, thus reducing the amount of valid data present in the zone when deletion is about to be performed. This greatly reduces data copying during zone cleaning, minimizing WA problems [3, 17, 18].

RocksDB for ZNS SSD uses ZenFS [6] (a user-level file system for ZNS-enabled RocksDB) for efficient data placement to minimize zone cleaning overhead. ZNS-enabled RocksDB is a Log-structured Merge tree-based key-value store running on top of a ZNS SSD. Specifically, ZenFS employs the Lifetime-based Zone Allocation algorithm (LIZA) that places data with the same lifetime in the same zone. LIZA predicts the lifetime of each data using the following characteristics of LSM-tree. In the LSM-tree, SSTables to be updated and deleted soon are located at a higher level, and SSTables updated relatively later are located at a lower level. LIZA gives a lifetime to each level of the LSM-tree based on hotness (write frequency), predicts the lifetime of the newly created SSTable according to the lifetime of the level to which the

SSTable belongs, and places SSTables with the same lifetime in the same zone, allowing them to be deleted together when compaction is triggered in the future. However, LIZA fails to minimize the WA problem by not accurately identifying SSTables to be deleted together due to lifetime prediction failures.

A critical limitation of LIZA is that LIZA does not consider how SSTables are invalidated in the LSM tree. Thus, this paper proposes a novel Compaction-Aware Zone Allocation algorithm (CAZA), the design of which is based on the observation that *a group of SSTables deleted/invalidated together is determined by the LSM-tree's compaction algorithm.* The compaction job selects a set of SSTables with overlapping key ranges at adjacent levels, merges them into one or more new SSTables, and delete the SSTables used as compaction input for merging. Considering the compaction procedure, CAZA places the newly created SSTable in the zone with the most SSTables that overlap with its key range. Then, when compaction is triggered in the future, the SSTable selected as the victim for merging is merged with the SSTables already in the same zone and deleted/invalidated together. This minimizes the amount of valid data remaining in the zone to a minimum, thereby minimizing the overhead due to the copying of valid data before erasing zones.

This paper makes the following contributions.

- We present a novel zone placement algorithm that is aware of the LSM-tree's compaction process in ZenFS. CAZA will place SSTables with overlapping key ranges in the same zone, so that the SSTables selected for compaction of the LSM-tree are within the same zone.
- Specifically, CAZA was designed considering the following characteristics of LSM-tree. First, SSTables are deleted and invalidated only during compaction, and new SSTables are created accordingly. Second, during compaction, the SSTable selected as the victim and the SSTables that overlap the key range of this SSTable are deleted and invalidated together.
- For evaluation, we implemented CAZA by modifying RocksDB version 6.14. Evaluations confirm that CAZA reduced copies of valid data incurred during zone cleaning by up to 2 times and WA by up to 7.4% compared to LIZA for write-intensive synthetic workloads.

2 BACKGROUND

2.1 Zone Namespace SSD

The NVMe Zoned Namespace (ZNS) is a new standard extension adopting zone interface for flash-based SSDs. The fundamental building block of ZNS is a zone that exports an erase unit of the NAND flashes directly to the host machine [2]. ZNS forms multiple NAND erase blocks into a fixed-size zone and enforces only sequential write with reset commands for erasing. Due to the concise interface, ZNS eliminated the garbage collection typically residing in conventional SSDs.

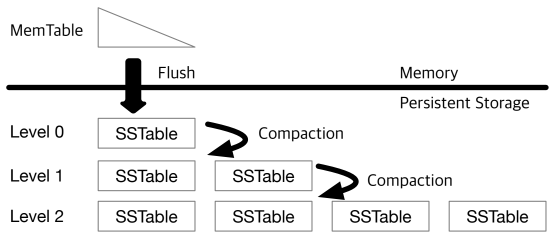


Figure 1: Description of the architecture and operation of RocksDB's LSM-tree

With the change in the hardware landscape, the ZNS requires modifications in the software aspect. First, applications take over free space reclamation [2, 3, 10, 17]. Instead of FTL, applications must explicitly erase zones. This means that copying valid data between erase units should be carried out by the host. Second, applications should perform data placement by choosing which zone to write their data. These two changes imply that write amplification can be eliminated or at least minimized by the host.

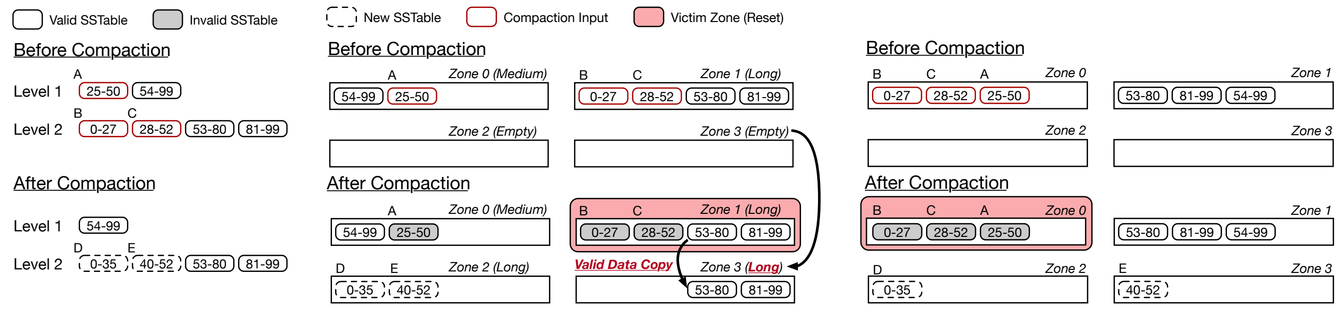
2.2 Log-Structured Merge-Tree

LSM-tree [15] is a data structure optimized for write performance and is used in various key-value stores for RocksDB [8], LevelDB [9], and MongoDB [13]. LSM-tree delivers high write throughput by generating sequential writes through buffering and batching. The LSM-tree consists of multiple levels and MemTable. MemTable temporarily stores key-value pairs in the main memory. Later, MemTable will be flushed to Level 0. Every SSTable in the persistent storage covers some key range in the sorted order. And except for Level 0, all SSTables partition the key range of the level.

Each level has a size limit, and the size limit increases exponentially. When a level grows over its size limit, compaction is triggered on that level and pushes down the data by merging SSTables. Specifically, The compaction triggered at Level i selects a victim SSTable from the Level i. Then the compaction also picks all other SSTables with overlapping key ranges from Level i and Level $i+1$. With all selected SSTables, the compaction merges them and writes one or more new SSTables to the next level. In this case, new SStables will be at Level $i + 1$. Since SSTables are immutable, overwritten or deleted keys are reclaimed during the compaction.

2.3 ZenFS

Ideally, the zone interface should have eradicated the additional write amplification from the underlying I/O stacks by delegating space management to the application. However, we observed that the benefit of the zone interface does not happen smoothly in practice. In particular, when the existing



(a) LSM-tree **(b) LIZA** **(c) CAZA**

Figure 2: Examples showing the efficiency of zone cleaning of CAZA compared to LIZA

application is ported to the zone interface, it requires middleware for zone management such as consolidating data smaller than the zone size because the data structure of the existing application is not designed to fit into the zone interface perfectly.

ZenFS is a representative example of the application-level middleware mentioned above. ZenFS is a backend module in RocksDB responsible for allocating zones for SSTables newly created and reclaiming free space out of zones with invalid data. Among the features of ZenFS, zone cleaning is the root cause of bringing the removed write amplification back to RocksDB. Zone cleaning is similar to that of segment cleaning of the log-structured file system (LFS) [14]. If there is no zone available for writes, the zone cleaning reclaims the free zone as follows. *Step 1:* Select the victim zone to erase. *Step 2:* Copy all valid data in the victim zone to the free zone. *Step 3:* After securing valid data, the zone reset command is sent to the ZNS SSD to erase the victim zone. Since write amplification occurs due to valid data copy, it is crucial for ZenFS to have the minimum or none of the valid data in the victim zone.

3 ZONE ALLOCATION IN ZENFS

To control the amount of valid data remaining in the zone to be erased, ZenFS utilizes the data hotness immanent in the LSM-tree [11]. Essentially, the levels of the LSM-tree are designed to represent the frequency of data modification or hotness. In the LSM-tree, batched updates are flushed to the uppermost level. Then, the compaction moves the data down to the lowermost level over time. As such, data at the uppermost level is most often invalidated, and data at the lowermost level is seldom invalidated. Based on these characteristics of the LSM-tree, ZenFS uses an algorithm (called LIZA) that assigns Write Lifetime Hint Value (lifetime hint) to each level, predicts the lifetime hint of the SSTable according to the level to which the SSTable belongs, and allocates a zone for the SSTable according to the lifetime hint value. In this way, LIZA places SSTables with the same lifetime hints in the same zone. Therefore, since data in the same zone can be invalidated at a similar time, ZenFS can separate valid and invalid data between zones, reducing the overhead of copying valid data during zone cleaning.

The detailed operation process of LIZA is as follows. LIZA has four different lifetime hint values; Short (1), Medium (2), Long (3), and Extreme (4). For ease of explanation, we denote integer values with them. First of all, every new SSTable to be written is assigned a lifetime hint based on its destination level. When the destination level is Level 0 or 1, the lifetime hint is Medium (2). For Level 2 SSTable, Long (3) is assigned. Similarly, Extreme (4) is assigned as the lifetime hint for Level 3 and further. Short (1) is assigned for data other than SSTable such as Write-Ahead Log (WAL) and Manifest (which has summary information of the LSM-tree). In LIZA, each zone also has a lifetime hint. It is determined by the lifetime hint of the first SSTable written to each zone. After the lifetime hint h of the SSTable is determined, LIZA searches for the zone with the smallest lifetime hint value equal to or greater than h, then places the SSTable in that zone. If there is no matching zone, LIZA allocates an empty zone and sets its lifetime hint to h. Note that LIZA may trigger zone cleaning when there is no empty zone remaining.

Inaccurate Hotness Estimation: In a long-term view, LIZA can place SSTables invalidated together in the same zone since new updates will, in the end, push all data from the upper level all the way down. However, LIZA's long-term segregation of invalidated data often fails to reduce the amount of valid data remaining in the victim zones, which creates high pressure for zone cleaning. Specifically, we have identified two situations in which LIZA cannot avoid, which are as follows:

- First, SSTables having overlapping key ranges at adjacent levels of the LSM-tree can be placed in different zones and invalidated at the same time by compaction.
- Second, compaction can invalidate SSTables across zones with different lifetime hint values.

Figure 2(a)&(b) describes the aforementioned two cases that occur in LIZA as an example. Figure 2(a) shows that

SSTables A, B, and C are selected as compaction input, and produce new SSTables D and E after merging. In Figure 2(b), new SSTables are written in Zone 2 and SSTable A, B, and C are invalidated by the compaction. When zone cleaning is triggered right after compaction, Zone 1 is selected as victim zone due to its high invalid ratio. Valid data copy is inevitable since half of Zone 1 must be copied to Zone 3, which was an empty zone previously, and then the lifetime hint for Zone 3 becomes Long as Zone 1.

Another important thing to note is that the lifetime hint of SSTable A (Medium) and lifetime hints of SSTable B and C (Long) are different. This means that LIZA does not avoid compaction that occurs between lifetime hint boundaries. Compaction across zones leads to partial invalidation in each zone after the compaction, which in turn increases data copy overhead during zone cleaning.

4 COMPACTION-AWARE ZONE ALLOCATION

The most fatal limitation of LIZA we found is the lack of design considerations for how SSTables are invalidated in LSM-trees. Therefore, our proposal, CAZA, was designed based on our close observation of how the compaction process selects, merges and invalidates SSTables in the LSM tree. As such, CAZA's design considering the compaction process of LSM-tree maximizes the zone cleaning efficiency of ZenFS by consolidating SSTables with overlapping key ranges located at different levels in the LSM-tree in the same zone and invalidating them together during zone cleaning.

We observe that the data structure of the LSM-tree, describing how SSTables form the LSM-tree, has enough clues to deduce possible combinations of SSTables that will be used as compaction input and be invalidated. For example, let L be the level of some new SSTable S to be placed. Then, from the data structure of the LSM-tree, we can infer (i) a set of candidate SSTables at level $L-1$ to be merged with SSTable S by compaction triggered at level $L-1$, and (ii) a set of SSTables at level $L+1$ to be merged by compaction triggered at level L in the same way, since compaction triggered at level i picks SSTables having overlapping key ranges from level i and $i+1$. When the compaction input includes the SSTable S, we can easily pick all SSTables having overlapping key ranges with S from the adjacent lower level using the data structure of the LSM-tree.

Based on the above observation, CAZA allocates a zone to SSTables that are newly created after merging SSTables with overlapping key ranges at adjacent level by considering how compaction input is selected in the LSM-tree. Thus, CAZA can resolve both limitations of LIZA pointed out in Section 3, which are SSTables having overlapping key ranges in different zones and SSTables being invalidated across different zones due to different lifetime hint levels.

Figure 2(c) shows an example of how the problem of LIZA described in Figure 2(b) is solved. In Figure 2(c), all SSTables (A, B, C) with overlapping key ranges over Level 1 and 2 are placed in Zone 0 before compaction. After compaction, SSTables A, B, and C become invalid in Zone 0 together. Afterwards, when zone cleaning is triggered, it picks Zone 0, which has the highest ratio of invalid data in the zone, as the victim zone for cleaning. Expensive copying of valid data can be avoided because all SSTables in that zone have already been invalidated by previous compaction.

The operation flow of CAZA is as follows.

- ***CAZA 1:*** Start with L, the target level of the new SSTable S. CAZA constructs a set of SSTables ($S_{overlap}$) by searching level $L+1$ for SSTables that overlap the key range of the SSTable S.
- ***CAZA 2:*** CAZA builds a set of all zones Z containing SSTables from $S_{overlap}$.
- ***CAZA 3:*** Then the set Z is sorted in a descending order by the number of SSTables belonging to $S_{overlap}$.
- ***CAZA 4:*** CAZA allocates zones from Z in order.

CAZA may fail to find a zone that satisfies the conditions in ***CAZA 1-4***. We can consider the following two scenarios that possibly trigger zone cleaning. ***ZC 1:*** Some SSTables may have entirely new key ranges. In this case, CAZA allocates an empty zone so that the zone can contain the new key range. Alternatively, ***ZC 2:*** CAZA may not have enough space to store new SSTables in all zones where SSTables with overlapping key ranges reside. In this case, CAZA will allocate a new empty zone rather than zones including disjoint key-ranges. Otherwise, SSTables with non-overlapping key ranges will be located in the same zone and are less likely to be compacted together. We intentionally allocate new empty zones for ***ZC 1-2***, although this may potentially cause zone cleaning. This is to leave an opportunity for the current SSTable to be compacted together with future SSTables that will be placed in the same new empty zone. To limit the cost of zone cleaning by ***ZC***, CAZA only tries to reset zones with no valid data, and the rest of the zone cleaning procedures are covered in ***Fallback***. Finally, if zone cleaning fails, CAZA will fall back to the following scenarios.

- ***Fallback 1:*** CAZA searches for an SSTable in the same level that has the closest key ranges possible, then allocates the zone containing the SSTable. Looking for the closest key range gives a future SSTable in the upper (or lower) level an opportunity to bridge nearby 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.
- ***Fallback 2:*** CAZA falls back to LIZA for the long-term segregation effect.

Fallback 1-2 handle the rest of the allocation that is not covered by the above cases. In particular, zone cleaning on zones with valid data will be at the end of ***Fallback*** cases. We prioritized ***Fallback 1*** to allow more SSTables to participate in compaction while ensuring long-term segregation of invalidated data.

5 EVALUATION

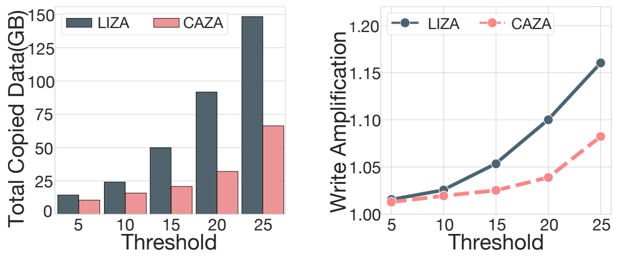
5.1 Experimental Setup

For evaluation, we emulated a 100 GB ZNS SSD with one hundred 1 GB sized zones in DRAM using *null_blk* [5]. All experiments are carried out on a machine with Intel Xeon E5-2640 and Linux 5.10.13. We implemented CAZA by modifying ZenFS in RocksDB v6.14. CAZA adopted a greedy zone cleaning where zones with the most invalid data are selected to be reset. To prevent the zone cleaning from blocking, ten zones are reserved for over-provisioning. We set the threshold (THS) proportional to the total device capacity for the zone cleaning to terminate after reclaiming a certain amount of free space. To inspect the amount of copied data according to the number of zone cleaning occurrences, we vary the threshold from 5% to 25%. The threshold represents the minimum amount of free space zone cleaning must achieve. In other words, the larger the threshold, the higher the pressure for zone cleaning.

We evaluated CAZA against LIZA. To analyze the effect of the two algorithms on zone cleaning, we used db_bench, a micro-benchmark tool released with RocksDB. To consider a situation where compaction is triggered frequently, we ran the db_bench first to load 40 GB of uniformly distributed key-value pairs (16 B key and 128 B value, uniformly distributed) in sequential order by key, then started measuring the write amplification overhead by randomly overwriting another 40 GB of key-value pairs within the same key range as the load step. We set the size of MemTables and L0 SSTables to 64 MB. The size limit of levels starts with 256 MB at L0, and it increases by ten times as level goes deeper.

5.2 Results

5.2.1 Write Amplification. To quantify the reduction of data copy during zone cleaning by CAZA, we measured the write amplification (WA). WA is obtained by dividing the total amount of data written to the ZNS SSD, which includes data copy during zone cleaning, by the total amount of writes by LSM-tree, which only includes writes generated by the compaction. Figure 3 shows a comparison of WA by zone cleaning for LIZA and CAZA varying thresholds. In Figure 3(a), LIZA shows a larger amount of valid data copies than CAZA. The amount of these copies increases with more frequent invocations of zone cleaning as the threshold increases. Importantly, the rate of increase in the amount of valid data copies of LIZA is larger than that of CAZA. Figure 3(b) shows a comparison of WA.



(a) Total Copied Data **(b) Write Amplification**

Figure 3: WA analysis of LIZA and CAZA

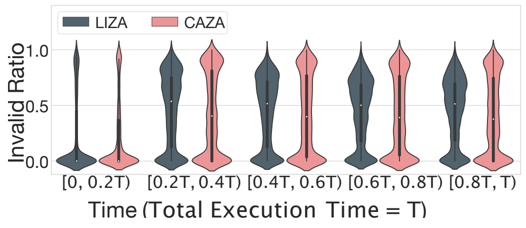


Figure 4: Changes in the distribution of zones according to the ratio of invalid data. "[A, B)" shows the start time and end time of the execution time. 'A' and 'B' are each expressed as a ratio of the total execution time (T).

CAZA shows a lower WA than LIZA for all thresholds, and the gap becomes larger as the threshold increases. The rate in increase of WA of LIZA is larger than that of CAZA.

5.2.2 Effect of Lifetime Segregation. If a zone allocation algorithm perfectly predicts the lifetime of data and segregates them into separate zones, every compaction will ideally only produce fully invalidated zones. However, in practice, valid data may remain in the zone even after the compaction. Consequently, the less accurate the prediction of data lifetime by the zone allocation algorithm, the more valid data is in the zone. The valid data left in the zone entails the overhead of copying valid data during zone cleaning.

To observe the change in the ratio of invalid data in the zone over time, we measured the distribution of the invalid ratio of zones in five equally sized time spans. We fixed the threshold to 15% in this experiment. As shown in Figure 4, LIZA produces a group of high invalid ratio zones to a certain degree, but does not reach invalid ratio 1.0. The group of zones below 1.0 continually becomes an empty zone, which locates at the bottom of a violin plot, by zone cleaning with the data copy. On the other hand, high invalid ratio zones in CAZA are located higher than the LIZA's in all time spans. CAZA has 3.32x more zones with an invalid ratio of 1.0 and 4.2x more zones with an invalid ratio of 0.95 than LIZA. This shows that CAZA places data invalidated at similar timing into the same zone so that less valid data are co-located with invalid data.

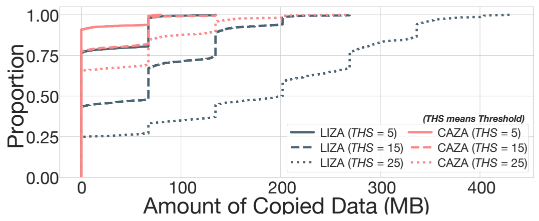
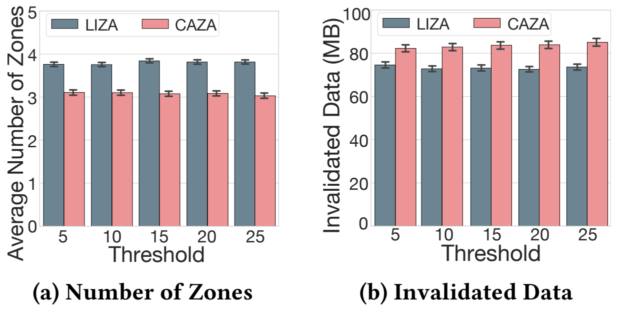


Figure 5: Comparison of amount of data copy per zone reset according to threshold change

To investigate whether CAZA is producing enough zones with only invalid data (invalid ratio of 1.0), we collected the amount of valid data copy from every zone reset with different thresholds (5, 15, 25). As shown in Figure 5, 91% of zone resets in CAZA do not require any valid data copy when the threshold is 5. This is because CAZA was able to precisely segregate data with the same lifetime, and the greedy zone cleaning algorithm mostly selects the zones with an invalid ratio of 1.0 as victims. On the other hand, LIZA selects victim zones with an invalid ratio of 1.0 for only 76% of zone resets. This difference deteriorates as the threshold gets larger. When the threshold is 25, only 25% of victim zones have no valid data in LIZA, compared to 67% in CAZA. This trend in the difference in total data copy is also shown as the write amplification reduction in Figure 3.

5.2.3 Impact of Compaction-Awareness. From Figure 4 and Figure 5, we observe that CAZA can effectively reduce the data copy during zone cleaning by producing zones with a high invalid ratio. To investigate the cause of zones with a higher invalid ratio in CAZA, we evaluated the impact of Compaction-Awareness in two respects. First is the number of zones that the compaction invalidated. Consider the case where a single compaction invalidates a set of SSTables. If CAZA successfully predicted SSTables that belongs to the set, invalidated SSTables would be across a smaller number of zones compared to the LIZA. In other words, CAZA contributed to compaction to produce a small number of zones with a high invalid ratio rather than a large number of Second is the size of invalidated data in a single zone by a single compaction. If more data are simultaneously invalidated in a zone after the compaction, it means that CAZA again successfully predicted and placed the input SSTables of the compaction into the a single zone.

As shown in Figure 6(a), CAZA places the SSTables deleted by a single compaction across three different zones, whereas LIZA places them in 3.8 zones. A lower value means that SSTables are more accurately grouped together in the same zones based on their lifetimes. Figure 6(b) shows the average amount of data invalidated together in a zone by a single compaction. While CAZA leads to 83MB invalidated, LIZA shows lower invalidation per zone as 74MB.



(a) Number of Zones **(b) Invalidated Data**

Figure 6: Impact of compaction-awareness

Table 1: Performance evaluation with db_bench

	Throughput (MB/s)		Latency (μs)	
THS	LIZA	CAZA	LIZA	CAZA
5	43.4	42	12.65	13.03
10	43.1	42.2	12.75	13.00
15	42.4	41.9	12.96	13.12
20	41.2	42	13.36	13.01
25	40	41.4	13.74	13.27

5.2.4 Performance. Lastly, we evaluated the CAZA's performance overhead. To this end, we measured average throughput and latency with db_bench for each experimental setup. For a fair comparison, all experiments started measurements after the initial loading phase when the DB was sufficiently populated. In Table 1, CAZA shows negligible performance difference from LIZA, despite significantly reducing writes during zone cleaning. We suspect that this insignificant performance difference is because (i) we use DRAM-emulated ZNS without considering the NAND latency of ZNS, and (ii) the implementation overhead for CAZA greatly affects DRAM-emulated ZNS. However, if an actual ZNS device is used, it is expected that CAZA will outperform LIZA.

6 CONCLUSION

This paper proposes Compaction-Aware Zone Allocation (CAZA) for the LSM-tree on ZNS SSD. CAZA is carefully designed considering the fact that SSTables are invalidated simultaneously when used as the compaction input together. CAZA effectively segregates SSTables by lifetime, minimizing write amplification overhead during zone cleaning. Extensive evaluation has shown that CAZA reduces write amplification by up to 7.4% compared to ZenFS's state-of-the art zone allocation algorithm, LIZA.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Angelos Bilas, for his feedback. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. NRF-2021R1A2C2014386). Y. Kim is the corresponding author.

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC '08)*. 57–70.

[2] Matias Bjørling. 2019. From Open-channel SSDs to Zoned Namespaces. In *Linux Storage and Filesystems Conference (Vault '19)*, Vol. 1.

[3] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC '21)*. 689–703.

[4] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. 2020. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*.

[5] Western Digital Corporation. 2021. nullbk. https://zonedstorage.io/docs/getting-started/nullblk

[6] Western Digital Corporation. 2022. ZenFS. https://github.com/westerndigitalcorporation/zenfs

[7] Western Digital Corporation. 2022. Zoned Stroage. https://zonedstorage.io/docs/introduction/zoned-storage

[8] Facebook. 2022. RocksDB. https://github.com/facebook/rocksdb

[9] Google. 2021. LevelDB. https://github.com/google/leveldb

[10] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. 147–162.

[11] Hans Holmberg. 2020. 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

[12] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of the ACM International Systems and Storage ConferenceS (SYSTOR '09)*. 1–9.

[13] MongoDB Inc. 2022. MongoDB. https://github.com/mongodb/mongo

[14] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. 273–286.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[16] Reza Salkhordeh, Kevin Kremer, Lars Nagel, Dennis Maisenbacher, Hans Holmberg, Matias Bjørling, and André Brinkmann. 2021. Constant Time Garbage Collection in SSDs. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS '21)*. 1–9.

[17] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Don't be a blockhead: Zoned namespaces make work on conventional SSDs obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. 144–151.

[18] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*. 429–443.

[19] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–26.