Lifetime-Leveling LSM-Tree Compaction for ZNS SSD

Jeeyoon Jung
Sungkyunkwan University, Korea
wjdwldbs1@skku.edu

Dongkun Shin
Sungkyunkwan University, Korea
dongkun@skku.edu

ABSTRACT

The Log-Structured Merge (LSM) tree is considered well-suited to zoned namespace (ZNS) storage devices since the write requests to LSM-tree is sequential. However, zones can be partially invalidated and be fragmented during LSM-tree compaction. The partially-invalid zones cannot be utilized and thus space amplification becomes significant. To reclaim the invalid space, host-managed garbage collection (GC) is required, which increases the write amplification of ZNS storage and degrades I/O performance. We introduce a lifetime-leveling compaction (LL-compaction) tailored for ZNS SSD, which can alleviate space amplification without GC by making the sorted string tables in a zone have similar lifetimes. In our experiments using LevelDB, the LL-compaction achieved 1.7x better performance by removing GCs.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; **Flash memory**.

KEYWORDS

Zoned Namespace, LSM-tree, Compaction, Solid-State Drives

ACM Reference Format:
Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-Leveling LSM-Tree Compaction for ZNS SSD. In *Proceedings of 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3538643.3539741

1 INTRODUCTION

Zoned Namespace Interface. A new NVMe storage interface, Zoned NameSpace (ZNS) [19], provides the logical address space divided into fixed-sized zones. The ZNS storage interface was first introduced for Shingled Magnetic Recording (SMR) [1] HDDs, and it was recently adopted by flash memory Solid-State Drives (SSDs). Considering the sequential-only write constraint of these storage media, ZNS specifies that each zone must be written sequentially and can be reused after reset operation. For ZNS SSD, a zone is generally mapped to multiple flash erase blocks in different flash chips to utilize flash chip-level parallelism. Because each zone is sequentially written, the ZNS SSD can maintain a zone-level logical-to-physical address mapping internally (i.e., mapping between a zone and flash blocks). The coarse-grained mapping requires a small internal DRAM of SSD. Because the mapped flash blocks of a zone will be fully invalidated at zone reset, the SSD-internal garbage collection is not required, and the write amplification by garbage collection can also be eliminated.

LSM-tree KV-Store. The Log-Structured Merge (LSM) tree [16] is considered well-suited to ZNS since the write requests to LSM-tree is sequential. LSM-trees have widely been used in many Key-Value (KV) stores, including BigTable [5], Cassandra [14], LevelDB [10], and RocksDB [9]. LSM-tree is a leveled data structure and each level is composed of several sorted files called Sorted String Tables (SSTs). Each SST has key-value pairs in a sorted form. The first level (L_0) SSTs are directly flushed from in-memory *memtable*. The SSTs of the following levels ($L_1, \cdots, L_n$) are generated by *LSM-tree compactions*, which migrate valid key-value pairs from an upper level to its next level. SSTs in the same level have non-overlapping key ranges each other at all levels except L_0. All levels have thresholds to limit the count or capacity of SSTs in one level to eliminate invalid key-value items for reducing read amplification and space amplification. If the total size of SSTs in a level (L_i) exceeds the threshold for the level, an SST in L_i is selected for compaction, and it is sort-merged with those in the next level (L_{i+1}) whose key ranges overlap with the key range of the selected SST in L_i. By the compaction, new merged SSTs are created in L_{i+1}. Generally, there is a maximum threshold on SST size (e.g., 4 MB). Therefore, if the capacity of an SST is beyond the threshold while writing key-value items in the file, the SST is closed and a new SST is created. Finally, the old SSTs in L_i and L_{i+1} are deleted.

Space Amplification. We need to discuss whether the current LSM-tree compaction is suitable for ZNS. Generally, the compaction costs become larger for larger SSTs, as the SSTs chosen for compaction are read, reordered and rewritten entirely. In addition, the memory space for memtables

HotStorage '22, June 27–28, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539741

also becomes larger. Therefore, we assume that the zone size (a multiple of flash erase block size) is large enough to store multiple SST files at a zone. During SST compaction, each zone can be filled sequentially with new created SSTs. However, the zones with deleted SSTs can be partially invalidated and be fragmented. Since a zone can be reused after all its SSTs are deleted, the invalid space of partially-invalid zones cannot be utilized, and thus space amplification becomes significant. To reclaim the invalid space of a zone, all valid SSTs in the zone must be moved to other zones by host-managed *garbage collection* (GC), which increases the write amplification of ZNS storage and degrades I/O performance.

Partially-invalid zones result from the mismatch between the write order and the deletion order of SSTs in a zone. There are two reasons for the mismatch. First, SSTs of different levels can be mixed in one zone. Since a compaction process invalidates the SSTs only in the involving two adjacent levels, the SSTs of other levels will remain after the compaction. This problem can be solved simply by allocating dedicated zones for each level, as proposed in GearDB [20].

Second, the current LSM-tree compaction algorithm uses an *upper-level driven compaction*, where a target SST is first selected from the upper level, and then its key-range-overlapping SSTs are selected from the lower level. Assuming that each zone has the SSTs of only a single level, the L_i-to-L_{i+1} compaction can invalidate the SSTs of L_i sequentially within each zone by selecting SSTs in the order of write. This is already supported by the current LevelDB implementation, whose compaction algorithm chooses the SSTs of the upper level in the order of key range. However, the deletion order of SSTs in L_{i+1} cannot be controllable. For example, some SSTs of L_{i+1} will be skipped during compaction if no SST of L_i overlaps with them in key range (**long-lived SSTs**). In addition, after an SST of L_{i+1} is created by the j-th L_i-to-L_{i+1} compaction, it can be immediately deleted by the $(j+1)$-th compaction, if the SST is selected as an key-range-overlapping SST at the compaction (**short-lived SSTs**).

We measured actual space consumption by long-lived and short-lived SSTs. We ran LevelDB using the YCSB [8] benchmark (zipfian constant = 0.99), and the zone size was 64MB. The workload wrote 16.5GB data in total, so 264 zones were used if no space amplification occurred. However, the number of zones actually allocated were 348. In our profiling, 81 zones (31% of allocated zones) were additionally allocated due to holes caused by the short-lived SSTs and 3 zones were occupied by long-lived SSTs. Consequently, 31.8% more space was used by short-lived and long-lived SSTs.

Lifetime-Leveling Compaction. To resolve the space and write amplification problems of LSM-tree on ZNS storage, we propose a lifetime-leveling compaction (*LL-compaction*) algorithm tailored for ZNS SSD, which makes the SSTs in each zone have similar lifetimes. First, even when there are no corresponding key-range-overlapping SSTs in the upper level for an SST in the lower level, the SST is also chosen for compaction in the LL-compaction if the compaction pointer (CP) must pass the key range of the SST. The CP of L_i points to the start key location of an SST in L_i to be selected at the next L_i-to-L_{i+1} compaction, and it moves forward after each compaction in the round robin policy. Therefore, the long-lived SSTs can be eliminated. Second, the LL-compaction stores short-lived SSTs at special separate zones, called `T-Zone`, to prevent them from being mixed with normal SSTs within a zone. In particular, the LL-compaction minimizes the size of each short-lived SST by making its start key range is equal to the next CP. Then, the read and rewrite costs for short-lived SSTs can be minimized. As a result, the LL-compaction writes and invalidates SSTs sequentially in each zone and thus eliminates the necessity of zone GCs.

The LL-compaction may increase or decrease the LSM-tree compaction cost compared to the normal compaction. The technique to avoid long-lived SSTs can increase the compaction cost, whereas the techniques to minimize the size of a short-lived SST can decrease the cost. In our experiments, we observed that most of the SSTs in each level have the corresponding key-range-overlapping SSTs in the adjacent levels. Therefore, the amount of long-lived SSTs is generally small. On the contrary, the amount of duplicate compactions for short-lived SSTs is significant in the normal compaction. Consequently, the LL-compaction has more benefit than cost.

To evaluate the proposed compaction algorithm, we implemented LL-compaction at LevelDB and experimented the compaction performance at a real ZNS SSD. The experiments showed that the LL-compaction can mitigate space amplification by about 30% and can show 1.7x better performance by avoiding zone GCs.

2 RELATED WORK

Host-Managed Data Placement. An SSD is composed of many flash chips, each of which consists of many flash blocks. Since the GC cost of SSD is determined by the lifetime similarity of data blocks in each flash block, the lifetime-aware data placement is important to reduce the GC cost. However, first-generation SSDs followed the traditional storage interface for compatibility although their internal structures were quite different with those of hard disk drives. The host cannot control the data placement within SSD, and the SSD cannot receive any hints from the host, which can be used to determine the optimal data placement.

To resolve this problem, second-generation SSDs provide an extended interface to allow the host to control data placement directly or indirectly. For example, the open-channel SSD [2, 4] exposes the hardware geometry of SSD such as flash channels/chips/blocks to the host. The multi-streamed

SSD [12] receives the stream ID along with a write request, and reduces write amplification by separating different-lifetime streams into different flash blocks. For example, SSTs in different levels of LSM-tree can be separated to independent streams. ZNS [15, 19] provides a high-level abstraction called zone, which can be written only sequentially and cannot be overwritten before reset. Although the constraint facilitates low-cost SSDs such as DRAM-less and GC-less SSDs, the host must undertake the host-level GC. To reduce the host-level GC overhead, the recently-proposed new ZNS interface, called ZNS+ [11], supports internal copy and sparse sequential write operations.

KV-Store for ZNS. SMRDB [17] is an SMR-friendly KV-store, which enlarges the SST to the size equivalent to an SMR band (zone) size (e.g., 80 MB) to prevent overwriting a band. SMRDB brings severe compaction latency due to the large data volume involved in each compaction. GearDB [20] is a GC-free KV-store designed for host-managed SMR drives. To eliminate GC, which migrates live SSTs from partially invalidated zones, GearDB proposed a gear compaction algorithm. By descending compaction level by level if the newly generated data overlaps the compaction window of the next level, zones can be reused without GC. However, the gear compaction invokes latency and space spikes by long compactions. ZenFS [3, 7] is a storage backend for RocksDB, which allows control of data placement through zones. ZenFS assumes that the zone size is smaller than the SST size, whereas we target large zone systems. Therefore, ZenFS does not consider the space amplification by partially invalid zones and does not provide GC for reclaiming invalid space.

3 ZNS-AWARE LSM-TREE COMPACTION

3.1 Motivation

Traditional LSM-tree Compaction. The traditional upper-level compaction algorithm is as follows:

(1) Determine the level L_i for compaction based on score.
(2) Select an SST T_j^i pointed by CP from L_i and initialize the set of SSTs to be merged (M) and the compaction window (W) as follows: $M = \{T_j^i\}$, $W = [\kappa_s(T_j^i), \kappa_e(T_j^i)]$ ($\kappa_s(\cdot)$ and $\kappa_e(\cdot)$ represent the start key location and the end key location of an SST or a compaction window, respectively.)
(3) For each SST T_k^{i+1} in L_{i+1} where $\kappa_s(T_k^{i+1}) \le \kappa_e(W)$ and $\kappa_s(W) \le \kappa_e(T_k^{i+1})$, insert T_k^{i+1} to M, and expand W as follows:
   - $\kappa_s(W) = \kappa_s(T_k^{i+1})$ if $\kappa_s(T_k^{i+1}) < \kappa_s(W)$
   - $\kappa_e(W) = \kappa_e(T_k^{i+1})$ if $\kappa_e(W) < \kappa_e(T_k^{i+1})$
(4) Insert any SST in L_i into M if its key range is fully included in W.
(5) Create new SSTs in L_{i+1} by merging all the SSTs in M.

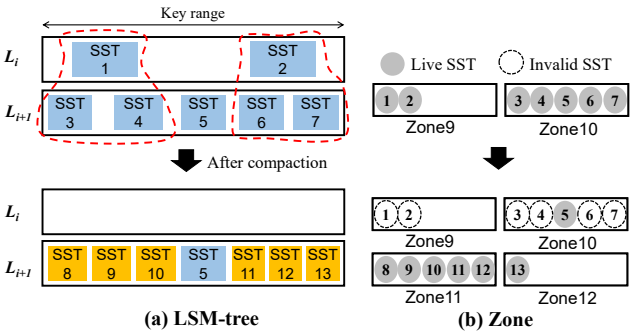


Figure 1: Example of long-lived SST.

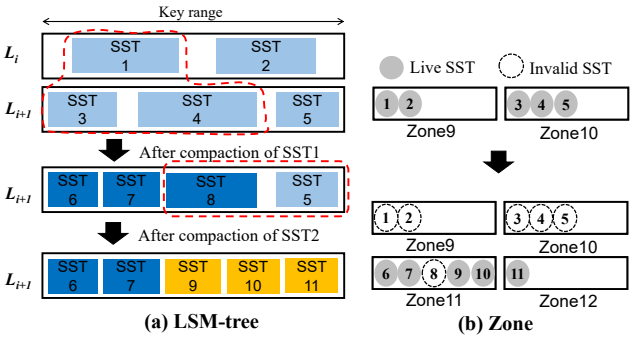


Figure 2: Example of short-lived SST.

(6) Delete all the merged SSTs.
(7) Change the CP of L_i (CP_i) to the start key location of the SST in L_i which is in the next order of compaction.

Long-lived SST. Fig.1 shows an example of LSM-tree compaction and the data change of zones by the compaction. Before the compaction, SST1 and SST2 in the upper level (L_i) are located at Zone9 while the SSTs of the lower level (L_{i+1}) are located at Zone10. The L_i-to-L_{i+1} compaction first selects SST1 from L_i, and then selects SST3 and SST4 from L_{i+1}, which overlap with SST1 in key range. By the compaction, the new merged SSTs (SST8, SST9, SST10) are created in L_{i+1}, and they are sequentially written at Zone11. The old SSTs of L_{i+1}, SST3 and SST4, are invalidated in Zone10. By the next compaction for the same level, the new merged SSTs (SST11, SST12, SST13) are written at Zone11 and Zone12, and the old SSTs of L_{i+1} in Zone10 (SST6 and SST7) are invalidated. SST5 is not involved in the compactions because there is no SST in L_i whose key range overlaps with the key range of SST5. As a result, although Zone10 has a large invalid space, it cannot be reset for reuse owing to only one valid SST, SST5, which will live until CP_i or CP_{i+1} passes its key range again. Such long-lived SSTs increase space amplification.

Short-lived SST. Fig.2 shows another example of LSM-tree compaction. The L_i-to-L_{i+1} compaction first selects key-range-overlapping SSTs (SST1, SST3, SST4) and creates the new merged SSTs (SST6, SST7, SST8), which are written at

Zone11. By the next compaction for the same level, SST2, SST8, and SST5 are merged into SST9, SST10, and SST11. SST8 is deleted by the second compaction just after it is created by the first compaction. This is because the compaction windows of the first compaction and the second compaction overlap. The lifetime of SST8 is short and makes a hole within Zone11, which cannot be reused until all the SSTs of L_{i+1} are deleted by the next compactions. Such short-lived SSTs increase not only space amplification but also write traffic because their key-value items are written twice.

Space Amplification and Write Amplification. The space amplification of the LSM-tree compaction can be mitigated by zone GC, which migrates valid SSTs in a partially invalid zone to other zones to reclaim the invalid space of the zone. For the example in Fig. 1, if SST5 is moved from Zone10 to Zone12, Zone10 can be reused and the space amplification can be mitigated. However, there is a trade-off between space amplification and write amplification. The SST migration by GC increases the write amplification, and thus the overall performance will be degraded. We need a ZNS-aware compaction algorithm which can improve zone utilization without GC.

3.2 Lifetime-Leveling Compaction

Our LL-compaction algorithm is based on the following principles: (1) It allocates dedicated zones for each level since different levels have different lifetime SSTs.

(2) To avoid long-lived SSTs, each compaction must involve all the lower-level SSTs located between the current CP and the next CP. For example, in Fig. 1, SST5 needs to be merged along with SST1, SST3, and SST4 during the first compaction, because CP_i is changed to κ_s(SST2) after the compaction bypassing the key range of SST5.

(3) Short-lived SSTs are separated from normal SSTs and the size of a short-lived SST is minimized. In Fig. 3, the first compaction driven by SST1 creates four new SSTs in L_{i+1}. The last two SSTs, SST10 and SST11, are divided by the next CP_i. Even though the capacity of SST10 is lower than the capacity threshold of an SST, the SST is closed before the next CP_i, and another SST is created starting from the key location of the CP. Only the latter will be involved in the next compaction driven by SST2. Without such an SST split, the key range of SST10 will be included in the next compaction window. Since SST11 is a short-lived SST, it is written at T-Zone instead of Zone10. Then, the short-lived SST can be separated from the normal SSTs in Zone10. T-Zone will have only short-lived SSTs, and thus it can be easily reclaimed.

(4) While creating SSTs in L_{i+1}, the key range of a created SST cannot include CP_{i+1} in the middle. If this is not guaranteed, CP_{i+1} can be changed to point to the start location of an SST because a compaction cannot be performed starting

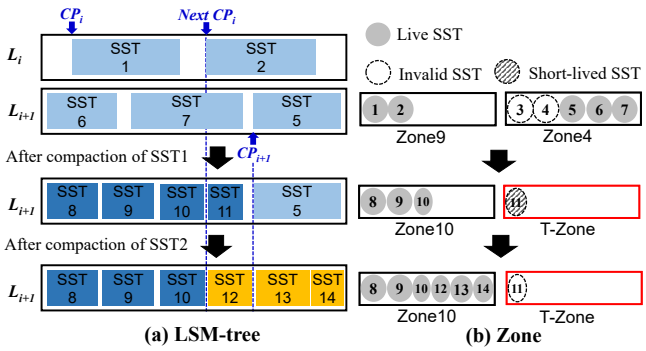


Figure 3: SST Split

from a middle key location of an SST. If CP_{i+1} is changed by the L_i-to-L_{i+1} compaction, the sequentiality of SST selection in the L_{i+1}-to-L_{i+2} compaction can be broken. In Fig. 3, CP_{i+1} points to SST5. The second compaction driven by SST2 creates three SSTs in L_{i+1}. The first SST (SST12) is closed before CP_{i+1}, and the key range of the following SST (SST13) starts from CP_{i+1}. Then, we don't need to change CP_{i+1}.

Based on the principles, we modified the original compaction algorithm in §3.1 by adding sub-steps in the steps of (3) and (5) as follows:

- (3)-1 **Compaction window expansion for principle (2)**: for each SST T_k^{i+1} in L_{i+1} where $\kappa_e(W) < \kappa_s(T_k^{i+1})$ and T_k^{i+1} has no key-range-overlapping SSTs in L_i, insert T_k^{i+1} to M and expand the compaction window W s.t. $\kappa_e(W) = \kappa_e(T_k^{i+1})$
- (5)-1 **SST Split for principle (3)**: if the key range of a created SST should pass the next CP_i, close the SST before the next CP_i and write following key-value items at another new SST file, and write only the latter at T-Zone
- (5)-2 **SST Split for principle (4)**: if the key range of a new created SST should pass CP_{i+1}, divide the new SST in two by CP_{i+1}

Instead of allocating a zone for T-Zone, we can use a memory space since the SSTs in T-Zone will be immediately deleted. Instead, the old SSTs with the key-value pairs of a short-lived SST must be deleted after the short-lived SST is merged into other normal SSTs.

4 EXPERIMENTS

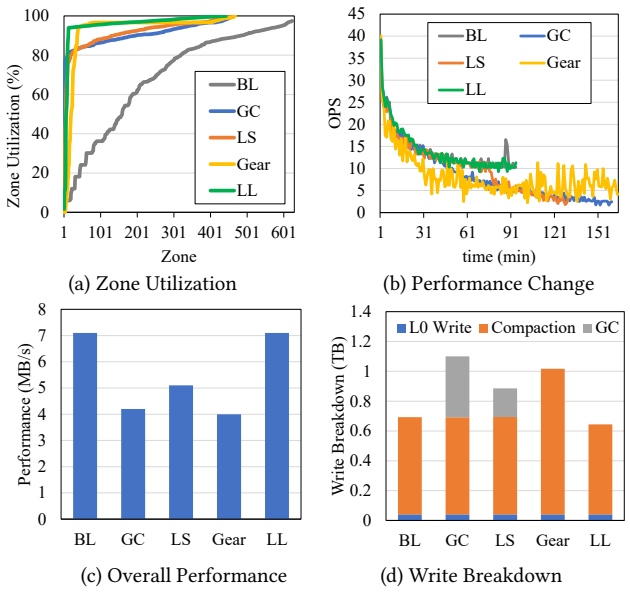
For evaluation, we used LevelDB 1.19. We implemented the host-managed GC, which is triggered when the number of free zones becomes one and is halted when more than two free zones are obtained. The GC algorithm uses the greedy policy, which chooses the zone with the lowest copy cost as a victim. We also implemented the level separation technique, which allocates dedicate zones for each level. We implemented our own ZNS storage system using Cosmos+

openSSD [13]. The zone size is 64 MB, and the SST size is 4 MB. The ZNS storage is accessed via our user-level LSM-tree management layer, which is composed of the libzbd [6] for accessing ZNS storage interface, a zone management layer for zone GC and level-to-zone mapping, and an 1.5 GB of memory cache for storing recently-accessed SSTs. The host computer system was equipped with 4 GHz quad-core Intel i7-4790k CPU and 16 GB DDR4 memory.

We first evaluated the space amplifications under five different compaction techniques, BL, GC, LS, Gear, and LL. BL and GC mean the GC-disabled original LevelDB and the GC-enabled version, respectively. LS uses the level separation technique in addition to GC. Gear and LL use GC-less compaction algorithms; the first is from GearDB [20] and the second is our LL-compaction. LL uses the original compaction algorithm at the L_0-to-L_1 compaction since the SSTs in L_0 have overlapping key ranges. The workload is the fill-random benchmark of db_bench. The key size is 16 B, and the value size is 512 B. The total 27 GB of data were written by the workload. The storage space was limited to 29 GB to invoke GCs at the experiments of GC, LS, and LL. In the experiment of Gear, which required more zone space owing to the space spike problem of the gear compaction, 31 GB was allocated.

Fig. 4(a) shows the distribution of zone utilization. The x-axis represents the zone ID sorted by utilization. BL consumed 624 zones and about 200 zones had a utilization lower than 60% owing to partially-invalid zones. The space amplification can be mitigated by zone GC, which migrates valid SSTs in a zone to other zone to reclaim the invalid space of the zone. By enabling GC, the number of allocated zones was reduced to 463, and most zones had a utilization higher than 80%. From the result, we can know that the zone GC is essential to reduce space amplification. LS improved utilization slightly compared to GC. The utilization at LL was higher than 90% at most zones (30% increase over BL).

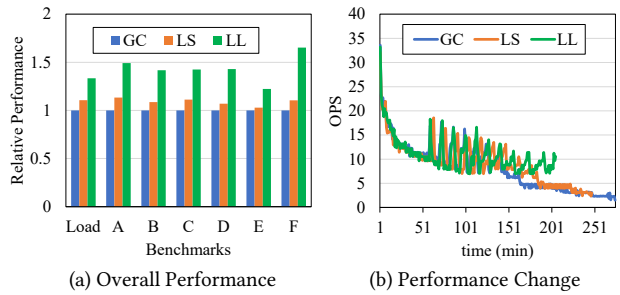
There is a trade-off between space amplification and write amplification. Fig. 4(b) shows the write performance change. As the number of written key-value items increased, the write performance degraded owing to LSM-tree compaction overhead. When GC was enabled, the performance degraded more significantly. Gear showed a similarly slow performance as GC. In addition, it showed a significant fluctuation on performance, because the gear compaction process proceeds recursively level by level. Fig. 4(c) compares the overall performances of different algorithms. LL achieved about 1.7x better performance than GC, and the improved performance was similar to that of BL, which has no GC overhead. According to GearDB [20], the gear compaction showed a better performance than the GC-enabled compaction at an SMR device. However, Gear and GC showed similar performances at our ZNS SSD, which has no seek time overhead unlike SMR drives. Fig. 4(d) shows the breakdown of the total write



(a) Zone Utilization (b) Performance Change

(c) Overall Performance (d) Write Breakdown

Figure 4: Fill-random workload



(a) Overall Performance (b) Performance Change

Figure 5: YCSB workload

traffic on storage. LL reduced the GC cost and the compaction cost compared to GC and Gear, respectively. Although Gear eliminates the GC cost, it shows a significant compaction overhead owing to its recursive compaction algorithm.

Fig. 5(a) shows the performance comparison result for the YCSB [8] workload (zipfian constant = 0.99, key = 16 B, value = 512 B). The total 75 GB of data were written by the workload. The storage space was limited to 78 GB to invoke GCs. Gear, which required too much space for its gear compaction, was not examined for the YCSB workload.

LL showed 22-65% performance improvements compared to GC. Fig. 5(b) shows the performance change while executing the load scenario of YCSB. Since the workload has an access locality, there is a performance fluctuation while running compactions. This is because the number of SSTs involved in a compaction varies.

5 CONCLUSION & DISCUSSION

Reducing space and write amplifications is a primary object in designing LSM-tree key-value stores. We revealed the current LSM-tree compaction can suffer from space and write amplifications at ZNS SSD owing to its lifetime-unaware algorithm. Our lifetime-leveling compaction can reduce space amplification without invoking zone garbage collections.

Under the LL-compaction, many small SST files can be created owing to its SST split policy. According to our evaluation, the number of files was increased by 9% at LL-compaction. However, the additional indexing overhead by the increased number of SST files was negligible. When executing *Get* operations, the metadata search overhead accounted for only 0.6% of the total execution time, and most of the overhead resulted from SSD access time. Therefore, the execution time increase was only 0.005%.

When an SST in the upper level has no corresponding key-range-overlapping SSTs in the lower level, the original compaction utilizes *trivial move*, which changes only the index information without rewriting key-value items. However, the LL-compaction cannot use *trivial move* because each level has its dedicated zones. GearDB [20], where each zone can serve only SSTs from one level, also has the same problem. However, the level separation has a higher benefit than the cost, as shown in experiments. In addition, we will be able to take advantage of *simple copy* [11] of ZNS, through which the host can order the SSD to copy data internally.

The LL-compaction cannot use a priority-driven SST selection algorithm. For example, RocksDB [9] considers the age, the overlapping key range, and the number of deleted items when selecting a victim SST to improve read performance or reduce compaction cost and space amplification [18]. However, the priority-driven policy will invalidate the SSTs in a zone randomly, which incurs a significant GC overhead at ZNS-based KV stores. It is our future work to analyze the impact of several priority-driven compression algorithms on the cost of garbage collection. The seek compaction, employed by LevelDB, also breaks the sequential selection order by prioritizing to move the SSTs in read-intensive key regions to lower levels to reduce the seek overhead. However, the seek overhead is insignificant at SSD devices. In addition, some SSTs can be cached in memory, and bloom filter can be used instead of the seek compaction.

ACKNOWLEDGMENTS

We thank our shepherd Youngjin Kwon and the anonymous reviewers for their valuable feedback. This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2095125) and Samsung Electronics.

REFERENCES

[1] Ahmed Amer et al. 2011. Data Management and Layout for Shingled Magnetic Recording. *IEEE Transactions on Magnetics* 47, 10 (2011), 3691–3697.

[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 *2021 USENIX Annual Technical Conference (ATC'21)*. 689–703.

[4] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*. 359–374.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (jun 2008), 26 pages.

[6] Western Digital Co. 2021. libzbd user library. https://zonedstorage.io/projects/libzbd/.

[7] Western Digital Co. 2021. ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs. https://github.com/westerndigitalcorporation/zenfs.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[9] Facebook. 2013. RocksDB. http://rocksdb.org/.

[10] Google. 2012. LevelDB. http://code.google.com/p/LevelDB.

[11] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Joo-Young Hwang. 2021. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[12] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*.

[13] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage* 16, 3, Article 15 (2020).

[14] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40.

[15] NVM Express. 2022. Zoned Namespace Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Zoned-Namespace-Command-Set-Specification-1.1b-2022.01.05-Ratified.pdf.

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

[17] Rekha Pitchumani, James Hughes, and Ethan L Miller. 2015. SMRDB: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*. 1–11.

[18] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2216–2229.

[19] Western Digital Co. 2022. Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io.

[20] Ting Yao et al. 2019. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. 159–171.