ZoneKV: A Space-Efficient Key-Value Store for ZNS SSDs

Mingchen Lu[†], Peiquan Jin[†*], Xiaoliang Wang[†], Yongping Luo[†], Kuankuan Guo[‡]
[†]*University of Science and Technology of China, Hefei, China*
[‡]*ByteDance Inc., Beijing, China*
{lmc123, wxl147, ypluo}@mail.ustc.edu.cn, *jpq@ustc.edu.cn, guokuankuan@bytedance.com

***Abstract*—In this paper, we propose a new space-efficient key-value store called ZoneKV for ZNS (Zoned Namespace) SSDs. We observe that existing work on adapting RocksDB to ZNS SSDs will cause fragmentation of zones and severe space amplification. Thus, we propose a lifetime-based zone storage model and a level-specific zone allocation algorithm to store SSTables with a similar lifetime in the same zone. We evaluate ZoneKV on a real ZNS SSD. The results show that ZoneKV can reduce up to 60% space amplification and maintain higher throughputs than two competitors, RocksDB and ZenFS.**

***Index Terms*—ZNS SSD, key-value store, space efficiency**

I. Introduction

ZNS SSDs (Zoned Namespace SSDs) [1], [2], is a new kind of SSDs adopting the new NVMe storage interface. ZNS SSDs divide the logical block address (LBA) into multiple zones of the same size. Each zone can only be written sequentially and used again after resetting. Because of the mandatory sequential writing of zones, ZNS SSDs can perform more coarse-grained address translation (e.g., mapping a zone to flash blocks) without sacrificing performance. This coarse-grained mapping allows for smaller DRAM inside ZNS SSDs, resulting in significant cost savings [3]. In addition, ZNS SSDs leave functions like garbage collection and data placement to the host side [1], [4]–[6], which enables users to devise efficient techniques to utilize the advantages of ZNS SSDs, e.g., new indexes [7] and buffer managers [8].

ZNS SSDs have a natural fit with the Log-Structured Merge tree (LSM-tree) [9], [10]. LSM-tree converts random writes into sequential writes, greatly improving write performance. It has been employed in many key-value stores, such as BigTable, LevelDB, and RocksDB. An LSM-tree-based key-value store contains a memory component, which is composed of MemTable and Immutable MemTable. MemTable stores the latest key-value in an append-only mode. When it reaches a storage threshold, the Memtable will be converted to an Immutable MemTable that is read-only. LSM-tree-based key-value stores also consist of multiple levels in the disk or SSD, and each level is composed of several Sorted String Tables (SSTables) [11], [12]. Each level has a size restriction, and the size limit grows exponentially. When a level exceeds its size limit, a compaction operation will be triggered to push the data down by merging SSTables in L_i to L_{i+1}.

Recently, ZenFS [1] was proposed to make RocksDB adaptive to ZNS SSDs. To the best of our knowledge, so far, ZenFS is the only LSM-tree-based key-value store supporting ZNS SSDs. ZenFS is a storage back-end module developed by Western Digital. It can allocate zones for newly produced SSTables and reclaim free space from zones with invalid data. ZenFS adopts an empirical method to put SSTables into zones, generating many invalid data inside zones because the SSTables in a zone may have a different lifetime. For example, when an SSTable in L_2 is selected to perform a compaction and all SSTables in L_3 are supposed to be stored in $Zone_1$, the newly generated SSTables by the compaction may be written to a different zone, e.g., $Zone_2$. As a result, ZenFS will produce zones containing SSTables with a different lifetime. Thus, the invalid data in zones will not be reclaimed in time, increasing the space amplification of LSM-tree.

To address the above problem, in this paper, we propose a new space-efficient key-value store called ZoneKV for ZNS SSDs. ZoneKV adopts a lifetime-based zone storage model and a level-specific zone allocation algorithm to make the SSTables in each zone have a similar lifetime, yielding lower space amplification, better space efficiency, and higher time performance than ZenFS. Briefly, we make the following contributions in this paper.

- To solve the space amplification problem of LSM-tree on ZNS SSDs, ZoneKV proposes a lifetime-based zone storage model to maintain SSTables with a similar lifetime in one zone to reduce space amplification and improve space efficiency.
- ZoneKV proposes a level-specific zone allocation algorithm to organize the levels of the LSM-tree in ZNS SSD. Especially, ZoneKV puts L_0 and L_1 in one zone because the SSTables in these two levels have a similar lifetime. In addition, ZoneKV horizontally partitions all the SSTables in $L_i (i \geq 2)$ into slices, and different slices are stored in different zones so that the SSTables of each zone have the same lifetime.
- We compare ZoneKV with RocksDB [12] and the state-of-the-art system ZenFS [1] on a real ZNS SSD. Compared with RocksDB, ZoneKV can mitigate space amplification by about 50%-60%, and this improvement is about 40%-50% when compared with ZenFS. Also, ZoneKV achieves the highest throughput.

II. Related Work

In conventional SSDs, the device is transparent to the host. Thus, the host cannot perform application-aware data placement or other operations to reduce write amplification and space amplification [13]. Additionally, garbage collection (GC) on the device side makes the performance of the SSD unpredictable [14]. ZNS SSDs can be regarded as a refined version of the open-channel SSDs. ZNS is compliant with the ZAC/ZBC specification [1], [3]. ZNS SSDs support a complete storage stack, including the underlying block driver to the upper layer file system. Furthermore, in a ZNS SSD, the device side does not perform GC, meaning that the host has to perform host-level GC. To minimize the GC overhead in the host level, ZNS+ [15], a new ZNS interface, was proposed to support in-storage zone compaction. Other approaches focusing on the compaction on ZNS SSDs include CAZA [16] and LL-Compaction [17]. However, both of them will make the SSTables after compaction scattering among different zones. Conceptually, the SSTables produced by the same compaction operation are supposed to have the same lifetime [5]. Putting them among different zones is not friendly for space efficiency.

So far, to the best of our knowledge, ZenFS [1] is the only key-value store that adapts RocksDB to ZNS SSDs. However, ZenFS does not fully consider the lifetime of SSTables and does not have complete GC logic. Thus, SSTables with different lifetimes may be stored in the same zone, leading to serious space amplification. In this paper, we mainly focus on improving ZenFS and regard ZenFS as the main competitor. Unlike ZenFS, we propose to store SSTables with a similar lifetime within the same zone to reduce space amplification of the LSM-tree.

III. Design of ZoneKV

In this section, we detail the idea and designs of ZoneKV. First, we analyze the limitations of existing approaches. Then, we explain the main idea of ZoneKV. Following the introduction of ZoneKV's architecture, we describe the key techniques in ZoneKV, including lifetime-based zone storage and level-specific zone allocation.

A. Limitations of Existing Approaches

The basic approach of porting RocksDB to ZNS SSDs is to allow RocksDB to store SSTables into specific zones, which can be realized by the file-operating interfaces [1]. Although such a simple implementation can support read/write operations on ZNS SSDs, it is not space efficient. This is because SSTables are randomly scattered among zones. As the SSTables in each zone may have different life cycles, e.g., some SSTables are frequently updated but others are never updated, we can infer that there will be many invalid data within each zone because ZNS SSDs have to invalidate old data and write the new data at the end of the influenced zone. These invalid data will not be reused until a zone-reset operation is executed. However, as zone-reset operations are costly (many data movements have to be performed), ZNS SSDs will not reset a zone until the space usage of the zone reaches its limitation. To this end, if all SSTables are randomly stored in zones, more and more invalid data will be generated in each zone, which leads to high space amplification and space waste. ZenFS [1] proposed an improved approach for storing SSTables in zones. It maintains the largest lifetime for each active zone; If the lifetime of a newly generated SSTable is smaller than the largest lifetime in an active zone, the SSTable will be put in the zone. ZenFS can reduce the space amplification compared to the raw RocksDB, but the SSTables with different lifetimes may still be distributed in the same zone. Note that SSTables with different lifetimes are with different updating frequencies. Hence, it is possible that some SSTables within a zone have been updated but others are not, which wastes zone capacities and is not space efficient.

B. Idea of ZoneKV

In this paper, we define the lifetime of an SSTable as the interval between its creation and invalidated time. Suppose that an SSTable was created at t_1 and was compacted or updated at t_2; its lifetime is defined as $|t_2 - t_1|$. As we do not want the SSTables with different lifetime scattering among different zones, which will produce more invalid data and increase the space amplification, we propose to store all SSTables with similar lifetimes into the same zone.

However, it is not trivial to get the exact lifetime of an SSTable. For instance, when we need to write an SSTable into a level in the LSM-tree, we only know the creation time of the SSTable. Until the SSTable is updated or compacted, we will not know its invalidated time. We observe that the SSTables in higher levels (e.g., L_0) have smaller lifetimes because they are more likely to be compacted caused by data flushing from the memory [18]. Thus, we do not maintain the lifetime information of each SSTable explicitly. Instead, we use the level number as an indicator of a lifetime. Moreover, as RocksDB adopts tiering compaction for L_0 [12], which will invalidate all the SSTables in L_0 and merge them with SSTables in L_1 and produce new SSTables that are finally written to L_1. Thus, we can infer that all SSTables in L_0 have the same lifetime. On the other hand, as the key ranges of the SSTables in L_0 are overlapped, it is much more likely that all the SSTables in L_1 will be read and merged with L_0 when performing tiering compaction for L_0 [19]. Hence, almost all the SSTables in L_1 will be invalidated by a tiering compaction triggered in L_0, meaning that all SSTables in L_1 have a similar lifetime with the SSTables in L_0.

In addition, for lower levels like L_2 and L_3, RocksDB uses leveling compaction [12] that selects one SSTable in $L_i (i \geq 2)$ in a round-robin way and merges the SSTable with all the overlapped SSTables in L_{i+1}. To this end, we can see that the SSTables with small keys will be first selected as victims for compaction, meaning that they have a similar lifetime. Thus, we make a horizontal partition for the SSTables in $L_i (i \geq 2)$ and put each partition in different zones. With such a mechanism, we can ensure that all SSTables in each zone have a similar lifetime, which is expected to offer high space efficiency.

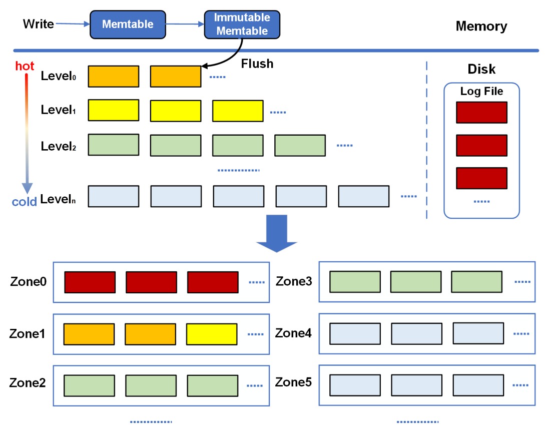


Fig. 1. The architecture and data layout of ZoneKV.

To sum up, the ideas of ZoneKV can be summarized as follows:

(1) Lifetime-Based Zone Storage. ZoneKV proposes to maintain SSTables with a similar lifetime in one zone to reduce space amplification and improve space efficiency. ZoneKV does not maintain the lifetime information for each SSTable explicitly but uses the level number of each SSTable to infer the lifetime implicitly. Such a design can avoid memory usage and maintain cost of the lifetime information.

(2) Level-Specific Zone Allocation. First, ZoneKV proposes to put L_0 and L_1 in one zone because the SSTables in these two levels have a similar lifetime. Second, ZoneKV horizontally partitions all the SSTables in $L_i (i \geq 2)$ into slices, and each slice is stored in one zone.

C. Architecture of ZoneKV

Figure 1 shows the architecture and data layout of ZoneKV. Like RocksDB, ZoneKV also includes a memory component and a persistent component (which is a ZNS SSD). The memory component acts the same as RocksDB, and the unique designs in ZoneKV focus on the persistent component, which consists of two aspects. First, when SSTables are written to a specific level, either by a flushing operation or a compaction operation, ZoneKV writes SSTables to specific zones (the algorithm of selecting suitable zones will be discussed in Section III-E) according to the lifetime information of SSTables. Second, each level in the LSM-tree is stored in different zones to make each zone contain SSTables with a similar lifetime.

As shown in Fig. 1, ZoneKV puts the log files in a separate zone because logs are written in an append-only way and never updated; thus their lifetime can be regarded as infinity.

The implementation of ZoneKV is based on RocksDB. Like ZenFS [1], ZoneKV modifies the interfaces of the *FileSystemWrapper* class in RocksDB and interacts directly with zones through *libzbd* [1]. The traditional disk I/O stack requires a series of I/O subsystems such as kernel file system, block layer, I/O scheduling layer, and block device driver layer to reach the disk. These long chains invariably slow down the data storage efficiency, indirectly reducing disk throughput and increasing request latency. ZoneKV is optimized for ZNS SSDs by performing end-to-end data placement directly on ZNS SSDs and bypassing the huge I/O stack.

D. Lifetime-Based Zone Storage

As mentioned earlier, ZoneKV proposes to maintain SSTables with a similar lifetime in one zone to reduce space amplification. Specifically, we put the SSTables in L_0 and L_1 in one zone, and the SSTables in $L_i (i \geq 2)$ in multiple zones. For each $L_i (i \geq 2)$, the SSTables are grouped according to their key ranges, forming a set of SSTable groups. Then, we put each SSTable group in one zone. Such partitioning has two reasons. First, the zone capacity (e.g., 1 GB in our experiment) is not large enough to store all SSTables at a low level. Second, the SSTables at the same level have different lifetimes. Particularly, in RocksDB, the SSTables with small keys will be selected early to be merged with other SSTables. The lifetime-based zone storage can store SSTables with a similar lifetime in the same zone or in the same group of zones.

Algorithm 1: Search for suitable zones

```
Function Find_zone(SSTable)
    lifetime_SSTable ← Get_lifetime(SSTable);
    zone ← Get_first_active_zone();
    while zone do
        if lifetime_SSTable = zone.lifetime_zone and
          Is_not_full(zone) then
            return zone;
        zone ← Get_next_active_zone();
    end
    if Get_num_empty_zone() > 0 then
        if Get_num_active_zone() <
          MAX_NUM_ACTIVE_ZONES then
            zone ← Assign_new_zone();
            return zone;
        else if Get_num_active_zone() =
          MAX_NUM_ACTIVE_ZONES then
            Close_active_zone();
            zone ← Assign_new_zone();
            return zone;
    else if Get_num_empty_zone() = 0 then
        zone ← Get_first_zone();
        while zone do
            if lifetime_SSTable < zone.lifetime_zone and
              Is_not_full(zone) then
                return zone;
            zone ← Get_next_zone();
        end
        zone ← Get_first_zone();
        while zone do
            if Is_not_full(zone) then
                return zone;
            zone ← Get_next_zone();
        end
    return NULL;
```

E. Level-Specific Zone Allocation

ZoneKV adopts the level-specific zone allocation scheme to realize lifetime-based zone storage. To avoid the memory cost and maintain the overhead of storing lifetime explicitly in ZoneKV, we use a level-related value to represent the lifetime of an SSTable. More specifically, ZoneKV marks the lifetime of log files as -1, which means that log files have an infinite lifetime. Next, we set the lifetime of SSTables in L_0 and L_1 to

1, meaning that the lifetime of these SSTables is only longer than that of log files. In addition, the lifetime of the SSTables in $L_i (i \geq 2)$ is set to i. The larger number means the lifetime of SSTables is longer, indicating that the SSTables are less likely to be updated or compacted.

ZoneKV assigns appropriate zones to the newly generated SSTables for storage based on the level-specific lifetime values. The steps to find suitable zones for an SSTable are described in Algorithm 1. The main goal of the algorithm is to find the zone with the same lifetime as the current newly generated SSTable from all active zones. If two or more zones are found, we select the first-found zone as the candidate. If no suitable zone is found from all active zones, a new zone is assigned directly to the SSTable, and the lifetime of this zone is set to the lifetime value of the SSTable. If no suitable zone is found, and no empty zone exists in the ZNS SSD, we look for a zone with a lifetime value larger than the SSTable. If no such zone is found, then any zone with enough space can be allocated to the current SSTable.

IV. Performance Evaluation

A. Experimental Setup

All experiments were conducted on Linux (5.15.0) configured with an Intel Xeon Gold 6348 CPU 2.60 GHz and 256 GB DDR4. Our experiments use the micro-benchmark (`db_bench`) provided by RocksDB (7.8.0). The keys in our experiment are all 16 bytes, and the values are all 1024 bytes. For RocksDB, the size of an SSTable is 64 MB, and the threshold of both L_0 and L_1 is 256 MB. We used ZN540, a ZNS SSD provided by Western Digital, in all experiments. The characteristics of ZN540 are shown in Table I.

In addition, we compare ZoneKV with RocksDB and ZenFS. Here, RocksDB refers to the RocksDB ported directly to ZNS SSDs without any modification, which is unaware of the ZNS interface. ZenFS [1] is the only key-value store that optimizes RocksDB for ZNS SSDs. We take ZenFS in the experiments as state-of-the-art work.

B. Write Performance

In this experiment, we use the *fillseq* and *fillrandom* workloads in `db_bench`. The *fillseq* workload contains sequential writes, while *fillrandom* contains random writes. Before performing *fillrandom*, we first use *fillseq* to sequentially write 50 GB key-value data to warm the ZNS SSD.

After warming the ZNS SSD, we randomly write 50 GB, 100 GB, 200 GB, and 400 GB key-value data using the *fillrandom* workload, respectively. Figure 2 shows the experimental results for the random-write performance.

Figure 2(a) indicates that the throughput of ZoneKV is significantly higher than that of RocksDB under four random-write loads with different database sizes, and all are slightly higher than that of ZenFS. When writing 50 GB data randomly, ZoneKV achieves 1.25x and 1.04x higher write performance than RocksDB and ZenFS, respectively. Because the SSTables with different lifetimes may be distributed in one zone, RocksDB and ZenFS need to copy valid data when resetting

TABLE I
Characteristics of ZN540.

Type	Parameter
Number of Zones	3,624
Zone Size	2,048 MB
Zone Capacity	1,077 MB
SSD Size	7.1 TB
SSD Capacity	4 TB

the zone, which occupies part of the write bandwidth. As a result, the write performance of RocksDB and ZenFS degrades with the running. The performance of RocksDB is worse than ZenFS because the data in each zone is more cluttered.

Figure 2(b) shows that the number of zones used by ZoneKV, which is 47.35%, 28.28%, 53.87%, and 46.09% less than that of RocksDB, and 34.25%, 17.24%, 52.27%, and 46.26% less than that of ZenFS, when the random-write data is 50 GB, 100 GB, 200 GB, and 400 GB, respectively. Most of the used zones are full zones, and only a few (about ten) are not full.

Figure 2(c) shows that under the above four write loads, the space amplification of ZoneKV is 48.59%, 28.42%, 54.43%, and 46.47% less than that of RocksDB, and 34.40%, 17.33%, 52.93%, and 46.63% less than that of ZenFS. The number of zones used by ZoneKV is much smaller than that of RocksDB and ZenFS, and the space amplification of ZoneKV is also lower, indicating that ZoneKV is more space-efficient than RocksDB and ZenFS.

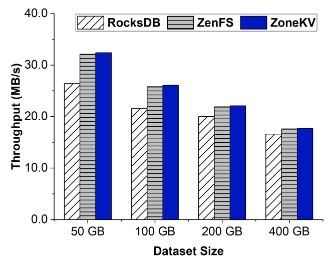
Figure 2(d) shows the standard deviation (*StdDev*) of ZoneKV and its competitors. ZoneKV has a smaller *StdDev* than RocksDB and ZenFS, which indicates that ZoneKV has a more stable performance (a smaller *StdDev* represents a more stable throughput). This is mainly because only a few data migrations are caused by ZoneKV during zones resetting.

Additionally, as Fig.2(b) shows, when the data load grows from 50 GB to 100 GB, then to 200 GB, and finally, to 400 GB, the number of zones used by ZenFS skyrockets even outgrows RocksDB. In this process, the number of zones used by ZoneKV increases slowly, which shows that ZoneKV has better scalability. As the capacity of the ZNS SSD is limited, when the data written to the key-value store increases from 400 GB to a specific value, ZenFS and RocksDB are already out of space, while ZoneKV can still run with high performance. We also demonstrate this in our experiments. When we try writing 2 TB key-value data to the ZNS SSD, both ZenFS and RocksDB fail, but ZoneKV can still work.

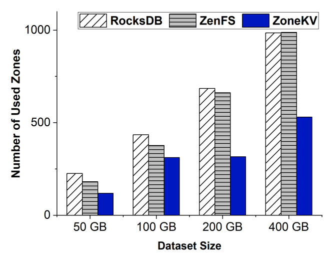
C. Update Performance

In this experiment, we use the *overwrite* workload in `db_bench` to test the random-update performance of ZoneKV. Like the random-write experiment, we warm the ZNS SSD by randomly writing 50 GB data before overwriting the data. Again, the data size of overwriting is set to 50 GB.

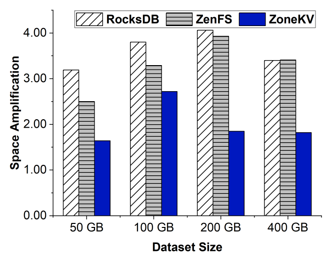
As shown in Fig. 3, ZoneKV achieves the highest throughput. Meanwhile, it uses the least number of zones and gets



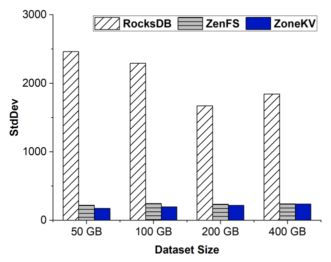
(a) Throughput.



(b) Used zones.

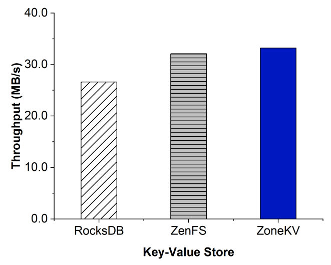


(c) Space amplification.

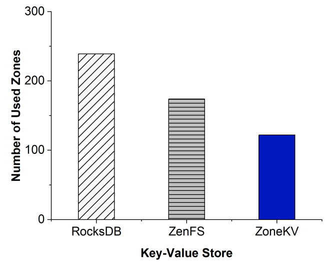


(d) StdDev.

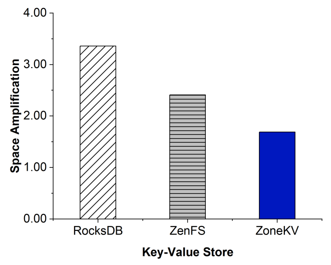
Fig. 2. Write performance.



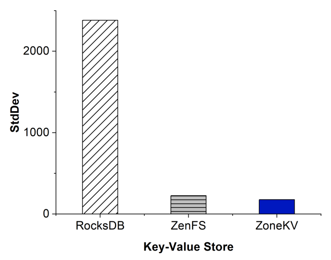
(a) Throughput.



(b) Used zones.

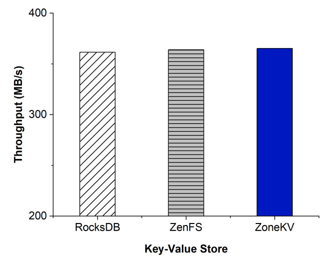


(c) Space amplification.

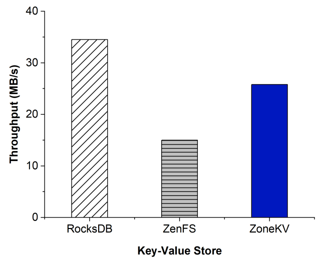


(d) StdDev.

Fig. 3. Update performance.



(a) Sequential read.



(b) Random read.

Fig. 4. Read performance.

the lowest space amplification. In addition, the overwrite performance of ZoneKV is also the most stable. This is because the updates in LSM-tree are done by appending to the latest data regardless of the old data. Therefore, random-update operations are equivalent to random-write operations; thus, the update performance is similar to the random-write performance in the experiment.

D. Read Performance

In this experiment, we use the *readseq* and *readrandom* workloads in `db_bench`. The *readseq* workload contains sequential reads, while *readrandom* contains random reads. We first write 50 GB data sequentially with *fillseq*, then write 50 GB data randomly with *fillrandom*. Next, we update 50 GB data with overwriting, then read 5 GB data sequentially and read 5 GB data randomly. All the above operations are executed without interruption.

As shown in Fig. 4, all three key-value stores offer similar sequential-read performance, while RocksDB performs a little better than the other two. ZoneKV has the second-highest random-read performance, and ZenFS has a deficient random-read performance. This is because ZenFS has many compaction operations before performing random reads, which will lower the system's throughput. In contrast, RocksDB's compaction completes quickly, which will not highly impact random reads. In addition, ZoneKV causes fewer compactions than ZenFS, which helps maintain high random-read performance.

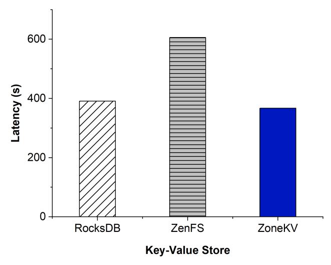
E. Mixed-Read/Write Performance

In this experiment, we use the *readrandomwriterandom* workload in `db_bench` to test the mixed-read/write performance. The *readrandomwriterandom* workload allows us to specify the read-write ratio in the workload. In this experiment, we use a 1:1 read-write ratio, i.e., reading 5 GB data while writing 5 GB data simultaneously. To reflect the performance of ZoneKV in terms of other metrics, we report the latency of each system in this experiment.

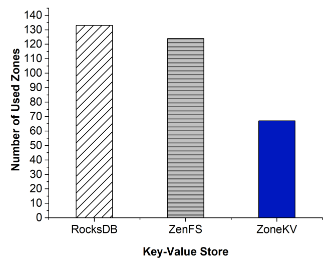
Figure 5(a) shows that ZoneKV achieves the lowest latency. From Fig. 5(b) and Fig. 5(c), we can see that ZoneKV costs the least number of zones and has the lowest space amplification. From Fig. 5(d), we can see that ZoneKV has the highest throughput almost all the time, while ZenFS has a very low throughput. To sum up, ZoneKV is not only space efficient but also with high performance in read/write-mixed workloads.

F. Multi-Thread Performance

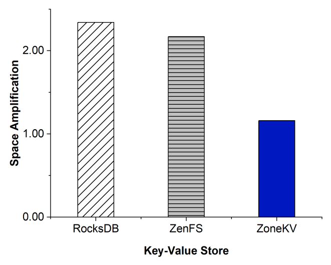
In this experiment, we test the performance of ZoneKV in a multi-thread environment. We conduct experiments with 2, 4, 8, and 16 threads, and the random-write workload is used by default. Before the experiment, we write 50 GB



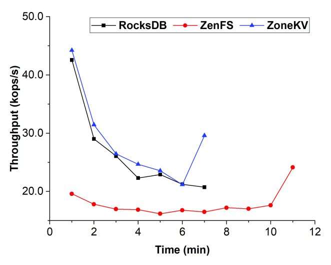
(a) Latency.



(b) Used zones.

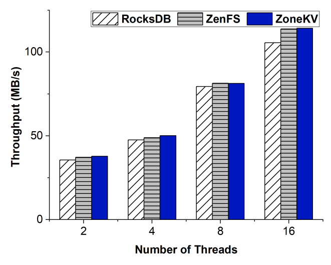


(c) Space amplification.

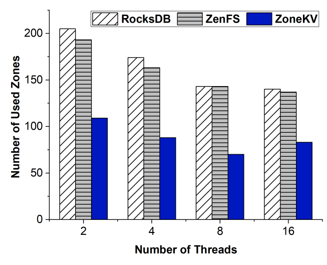


(d) Operations per second.

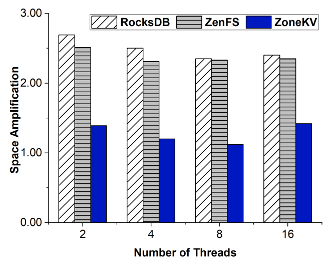
Fig. 5. Mixed-read/write performance.



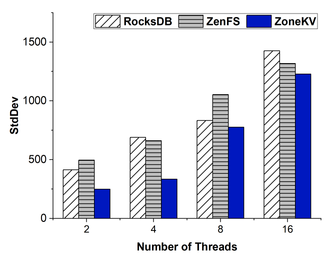
(a) Throughput.



(b) Used zones.



(c) Space amplification.



(d) StdDev.

Fig. 6. Multi-thread performance.

data sequentially and then write 50 GB data separately using multiple threads.

Figure 6 shows that ZoneKV can scale with the threads. It can always maintain the highest throughput in all settings. Moreover, ZoneKV also consumes the least number of zones and achieves the lowest space amplification. Also, the *StdDev* of ZoneKV is smaller than that of RocksDB and ZenFS. In general, the performance of ZoneKV in multi-thread environments is consistent with that in single-thread running, which means that ZoneKV can maintain stable and scalable performance in multi-thread settings.

V. Conclusions

In this paper, we proposed a new key-value store for ZNS SSDs, which was called ZoneKV, to mitigate space amplification and maintain high performance. The novelty of ZoneKV lies in the lifetime-base zone storage and level-specific zone allocation, which can lower the number of used zones and reduce space amplification of the LSM-tree on ZNS SSDs. We experimentally demonstrated that ZoneKV outperformed RocksDB and the state-of-the-art ZenFS under various workloads and settings. Especially, ZoneKV can reduce up to 60% space amplification while keeping higher throughput than its competitors. Also, the experiments under a multi-thread setting showed that ZoneKV could maintain stable and high performance under multi-thread environments.

Acknowledgements. This work was supported by the National Science Foundation of China (No. 62072419) and the ByteDance Inc. Peiquan Jin is the corresponding author.

References

[1] M. Bjørling, A. Aghayev, and et al., “ZNS: avoiding the block interface tax for flash-based SSDs,” in *USENIX ATC*, 2021, pp. 689–703.
[2] H. Shin, M. Oh, and et al., “Exploring performance characteristics of ZNS SSDs: Observation and implication,” in *NVMSA*, 2020, pp. 1–5.
[3] T. Stavrinos, D. S. Berger, and et al., “Don’t be a blockhead: zoned namespaces make work on conventional SSDs obsolete,” in *HotOS*, 2021, pp. 144–151.
[4] G. Choi, K. Lee, and et al., “A new LSM-style garbage collection scheme for ZNS SSDs,” in *HotStorage*, 2020.
[5] H. Wang, Y. Liu, and et al., “Efficient data placement for zoned namespaces (ZNS) ssds,” in *NPC*, 2022, pp. 302–314.
[6] H. Bae, J. Kim, and et al., “What you can’t forget: exploiting parallelism for zoned namespaces,” in *HotStorage*, 2022, pp. 79–85.
[7] P. Jin, X. Zhuang, and et al., “Exploring index structures for zoned namespaces ssds,” in *IEEE BigData*, 2021, pp. 5919–5922.
[8] Y. Lv, P. Jin, and et al., “Zonedstore: A concurrent zns-aware cache system for cloud data storage,” in *ICDCS*, 2022, pp. 1322–1325.
[9] P. E. O’Neil, E. Cheng, and et al., “The log-structured merge-tree (LSM-tree),” *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[10] D. R. Purandare, P. Wilcox, and et al., “Append is near: Log-based data management on ZNS SSDs,” in *CIDR*, 2022.
[11] F. Chang, J. Dean, and et al., “Bigtable: A distributed storage system for structured data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.
[12] S. Dong, A. Kryczka, and et al., “Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience,” in *FAST*, 2021, pp. 33–49.
[13] N. Agrawal, V. Prabhakaran, and et al., “Design tradeoffs for SSD performance,” in *USENIX ATC*, 2008, pp. 57–70.
[14] J. Kim, D. Lee, and et al., “Towards SLO complying SSDs through OPS isolation,” in *FAST*, 2015, pp. 183–189.
[15] K. Han, H. Gwak, and et al., “ZNS+: advanced zoned namespace interface for supporting in-storage zone compaction,” in *OSDI*, 2021, pp. 147–162.
[16] H. Lee, C. Lee, and et al., “Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs,” in *HotStorage*, 2022, pp. 93–99.
[17] J. Jung and D. Shin, “Lifetime-leveling LSM-tree compaction for ZNS SSD,” in *HotStorage*, 2022, pp. 100–105.
[18] T. Yao, J. Wan, and et al., “Geardb: A GC-free key-value store on HM-SMR drives with gear compaction,” in *FAST*, 2019, pp. 159–171.
[19] T. Yao, Y. Zhang, and et al., “Matrixkv: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM,” in *USENIX ATC*, 2020, pp. 17–31.