Received 26 February 2023, accepted 12 March 2023, date of publication 15 March 2023, date of current version 20 March 2023.

*Digital Object Identifier 10.1109/ACCESS.2023.3257354*

RESEARCH ARTICLE

# ZenFS+: Nurturing Performance and Isolation to ZenFS

**MYOUNGHOON OH[1], SEEHWAN YOO[2], (Senior Member, IEEE), JONGMOO CHOI[1], (Member, IEEE), JEONGSU PARK[3], AND CHANG-EUN CHOI[3]**

[1]Department of Software, Dankook University, Yongin-si, Gyeonggi-do 16890, South Korea
[2]Department of Mobile Systems Engineering, Dankook University, Yongin-si, Gyeonggi-do 16890, South Korea
[3]SK Hynix Inc., Bubal-eup, Icheon-si, Gyeonggi-do 17336, South Korea

Corresponding authors: Jongmoo Choi (choijm@dankook.ac.kr) and Seehwan Yoo (seehwan.yoo@dankook.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) under Grant 2022R1A2C1006050, in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korean Government (MSIT) under Grant 2021-0-01475, and in part by the Development of Novel Key-Value DB for Unstructured Bigdata (SW StarLab) under Grant 2022-0-01022.

**ABSTRACT** This paper proposes ZenFS+, a new storage backend of RocksDB for small-zone ZNS SSD. RocksDB has complicated internal operations such as flush and compaction. Flush and compaction run in separate threads, and it is well known that they closely interact around the sstables. Due to the concurrent storage I/O between flush and compaction, ZenFS presents unsatisfactory performance for small-zone ZNS SSD. We believe that ZenFS+ presents how ZNS SSD can support the performance and isolation of modern key-value stores. Leveraging the ZNS SSD, ZenFS+ intelligently identifies the independent zones group (IZG), which reveals the device's internal parallelism. With the IZG information, ZenFS+ effectively isolates the flush workloads of RocksDB from compaction. Besides, ZenFS+ spreads the sstables to multiple IZGs so that the storage write can leverage the hardware parallelism. ZenFS+ presents up to 4.8x higher storage throughput for write-intensive microbenchmark and stabilizing 99.9P tail latency by 1/51 from existing ZenFS. Further, ZenFS+ implements proactive garbage collection, and presents sustainable performance over longer lifespan of the device.

**INDEX TERMS** Data storage systems, flash memories, parallel architectures, storage management, system software.

## I. INTRODUCTION

A key-value store is one of the essential components in modern big data systems. With the help of powerful storage engines, Google, Facebook, Twitter, and Amazon could handle a large volume of unstructured data in a timely manner. In a recent report [1], Amazon's cloud key-value store handles 68.2 million requests per second with a high-availability configuration. That shows a requirement for decent big data systems, handling queries with significant I/O throughput and small latency.

RocksDB is an open-source key-value store developed by Facebook [2], [3]. It is widely used in research and industrial projects such as social graph analysis [2], distributed file systems [2], structured/unstructured DB [4], etc. RocksDB is popularly used in large-scale big data systems because it presents improved write performance with concurrent threads. Write performance is critical because large volumes of data keep incoming in such systems, and KVS needs to handle intensive 'PUT' requests within a small time budget. Also, the recency in a database is essential to keep the freshness of input data.

RocksDB manages multiple threads that conduct concurrent I/O operations. The user thread, flush, and compaction threads make complicated interactions affecting overall performance. Flush and compaction concurrently run in separate threads, and they can interfere with each other, collapsing the performance of RocksDB. In a recent study [5], the authors observed the latency spikes in `Put()` operations due to the delayed flush of in-memory data.

This paper presents a new approach to improve the latency and controlled performance of RocksDB using zoned

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

namespace (ZNS) SSD. ZNS SSD divides the address space of an SSD device into sequentially-writeable zones [6]. ZNS SSD allows the software to control the address space of the storage device directly, minimizing the overhead from device-level garbage collection and performance deterioration [7]. ZNS SSD claimed several benefits over traditional SSD, including improved storage access latency thanks to the unique firmware architecture that omits device-level garbage collection and that supports per-zone controlled performance.

Recently, ZenFS has been proposed in [9], a storage backend for RocksDB. ZenFS focused on the benefits of ZNS SSD thanks to the simplified garbage collection and improved performance deterioration throughout the device's lifespan. However, the original ZenFS disregards the small-zone ZNS SSD, which allows more flexible and fine-grained software control.

The primary goal of ZenFS+ is to improve ZenFS to support high throughput and controlled latency of RocksDB for big data systems with the help of ZNS SSD. ZenFS+ achieves controlled performance for I/O threads of RocksDB, leveraging the ZNS SSD. With the independent zone group (IZG) information, ZenFS+ effectively isolates the flush performance from compaction. ZenFS+ intelligently manages the internal parallelism of the SSD device by supporting sstables striping, which improves the overall storage performance. ZenFS+ devises the proactive garbage collection (GC) to minimize the run-time overhead and to quick zone reclaiming, not to delay the flush operation.

There are two challenges in our ZenFS+: First, ZNS SSD has some hardware design parameters about the zone size, and the parameters significantly affect the storage performance. Some ZNS SSD devices have a large zone size (e.g., 1GB), and a zone has high internal parallelism. With a large-zone ZNS SSD device, writing to a zone utilizes the entire channels, presenting high bandwidth. However, the large-zone ZNS SSD devices are weak at performance isolation. For example, flush to using the entire channels could interfere with reading or writing from compaction or user requests.

In contrast, other ZNS SSD devices have a smaller zone size (e.g., 72 MB), and a zone has limited internal parallelism. With a small-zone ZNS SSD device, reading or writing to a zone utilizes limited internal channels, presenting low bandwidth [8]. However, it is better to control bandwidth between flush and compaction because we can flexibly choose the flush zones to avoid interference from the compaction zone. Therefore, we need to balance and leverage the performance and isolation of ZNS SSD by intelligently making design choices. This paper advocates the small-zone ZNS SSD for better isolation among complicated flush and compaction threads. This paper advocates the small-zone ZNS SSD for better isolation among complicated flush and compaction threads. ZenFS+ improves the previous ZenFS design so that software concurrency can intelligently leverage underlying hardware parallelism. Thereby, we can balance and leverage the performance and isolation of ZNS SSD by making intelligent design choices.

Second, garbage collection has not been implemented in the state-of-the-art ZNS SSD software for RocksDB, ZenFS [9]. In the current implementation of ZenFS, zone reclaiming is triggered only when we use up all the zones. ZenFS+ spreads the sstable over multiple zones for high performance. With the current ZenFS zone reclaiming, this incurs a considerable run-time latency because once the zone is all used up, the ZenFS should find multiple zones, repeatedly running its reclaiming algorithm. To mitigate the zone reclaiming overhead, ZenFS+ needs a proactive yet lightweight garbage collection.

According to our experimental results, ZenFS+ provides guaranteed bandwidth to flush, isolated from compaction jobs. Also, ZenFS+ improves the write performance of RocksDB. In our microbenchmark results, ZenFS+ achieves up to 4.8x higher flush bandwidth, 2.6x higher `put()` bandwidth, and reduces 99.9P tail latency by 1/51 compared with original ZenFS. Namely, ZenFS+ improves the user-perceived application performance as well as complicated internal operations, achieving the promise of ZNS SSD: stable and predictable storage performance.

The contribution of this paper can be summarized as follows.

- This paper proposes ZenFS+, a new approach to achieving performance isolation using small-zone ZNS SSD.
- ZenFS+ improves the performance of RocksDB by exploiting the internal parallelism via striping data across multiple zones.
- ZenFS+ implements proactive GC for minimizing the run-time overhead of zone reclaiming.
- The paper presents the viability of the performance evaluation on a real ZNS SSD device.

## II. BACKGROUND

In this section, we first explain the structure of an LSM tree-based key-value store (KVS) and its internal operations, including flush and compaction. Then, we discuss issues of traditional SSDs and how ZNS SSDs address these issues.

### A. ROCKSDB AND LSM TREE-BASED

#### 1) KEY-VALUE STORE

KVS is an application that stores arbitrary typed key and value pairs. There are diverse KVS implementations, but KVS-based on log-structured merge tree (LSM-tree) is popularly used for big data systems because it is write optimized and manages data in recency order. KVS places more recent data in higher-level storage in the hierarchy so that the user can quickly obtain it. RocksDB is one of LSM tree-based KVS implementations [3]. To enhance the storage performance of LevelDB [10], RocksDB introduces concurrent flush and compaction operations.

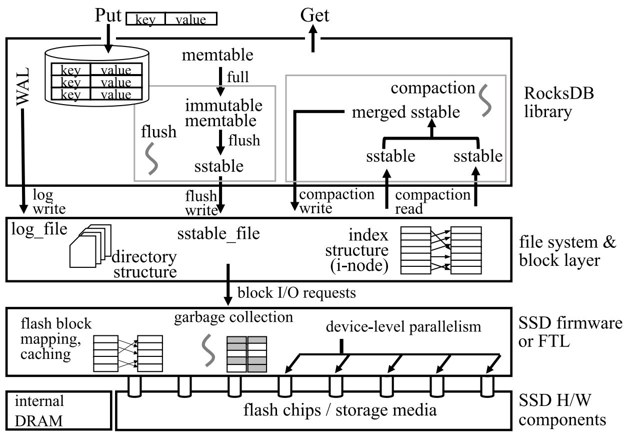


**FIGURE 1.** The RocksDB storage stack.

Figure 1 illustrates the overall storage stack of RocksDB. In RocksDB, flush and compaction run in separate threads from user's `get()` and `put()` operations. RocksDB takes advantage of storage hierarchy, placing data to memory, cache, and storage, according to the recency order.

First, when a user puts data into KVS, it is stored in the memory. RocksDB uses a memtable, in-memory data structure using skiplist. The size of the memtable increases as the user writes data, and when the size hits a threshold, it is changed into an immutable (read-only) state.

Second, RocksDB flushes immutable memtables to the storage. RocksDB uses an in-storage structure called sstable. In practical settings, a sstable is stored in a file. For efficiency, sstables are also hierarchically structured. To search for a key, RocksDB searches through multiple sstables from the top level to the bottom level. In the current implementation, RocksDB uses the Bloom filter to quickly checking the existence of data in a sstable.

Third, RocksDB manages sstables in recency order, allowing multiple entries for the same key in sstables. That is, new and valid entry is always located at the higher level, and obsolete entries are found in the lower-level hierarchy. RocksDB introduces compaction of sstables to avoid wasting the storage capacity and performance from redundant and obsolete entries. When the number of sstables reaches a threshold, compaction is triggered. The compaction is a merge sort that combines sstables and removes unnecessary entries. The compaction is repeatedly executed over multiple levels, reading and writing a volume of sstables.

Along with the flush and compaction operations, RocksDB uses WAL (write-ahead-log) for the recovery of in-memory data. Although WAL uses small storage bandwidth, it is known that WAL has a considerable impact on latency.

### B. ZNS SSD AND ZENFS

As a next-generation SSD, ZNS SSD draws attention for predictable latency and stable performance [11]. ZNS SSD is a kind of open-channel SSDs that allows more direct control of the flash device from users [6]. ZNS SSD divides the entire flash address space into multiple zones, and the user can directly write data on a specifically named zone. ZNS SSD does not support random writing on a zone; thus, all writes on the zone must be sequential.

Because ZNS SSD removes the complicated FTL abstraction from the device, the hardware operation becomes more predictable, which promises predictable storage performance. Besides, ZNS SSD reduces the cost of garbage collection from the device firmware to software. Thus, the hardware performance never deteriorates over the lifespan.

Recent ZNS studies [7], [8], [12] focus on various aspects of hardware and software co-design. Traditional SSDs have internal DRAM and over-provisioning (OP) flash capacity for garbage collection, coined as *block interface tax* in [7], which can be minimized by Zone-based flash management. The study proposed ZenFS, a zoned storage backend for RocksDB, and presented performance with RocksDB. In ZNS+ study [12], the garbage collection overhead from the file system (F2FS) can be efficiently mitigated by introducing an internal block copy operation on ZNS SSD.

Another potential benefit of ZNS SSDs is performance isolation [8]. The authors presented two kinds of ZNS SSDs: typical large-zone ZNS SSD and small-zone ZNS SSD. They also discussed the performance implication of ZNS SSD according to the internal structure. The authors advocate the small-zone ZNS SSD because it can provide isolated performance for concurrent workload leveraging multiple small zones. They also proposed a technique for detecting an inter-zone interference and an I/O scheduling for avoiding the interference among zones.

Now let us discuss ZenFS in more detail [7], [9]. It is a RocksDB storage backend for ZNS SSDs using hardware from a commercial vendor [13]. In the original design of ZenFS, it has a journaling zone and a data zone. Files for sstables are stored in the data zone, whereas metadata for recovery, superblock, and mapping information for WAL and data files are stored in the journal zone. Sstables from different levels are allocated into different zones [14].

ZenFS does not define any operation related to garbage collection. Instead, ZenFS has a data zone selection algorithm that best efforts for storing sstables and WAL. The algorithm tries to choose the zone so that the zone's data has a similar lifetime. In the original design, a user can heuristically choose $6 \sim 12$ active zones for running RocksDB on ZenFS.

## III. A MOTIVATION: BROKEN PROMISE OF ZNS SSD

ZNS SSD claims benefits over traditional SSD. The claimed benefits includes 1) ZNS SSD presents stable latency and bandwidth because ZNS SSD removes the semantic gap of using FTL, and 2) ZNS SSD minimizes block interface tax from garbage collection and over-provisioning. This section presents some experiments with ZNS SSD on RocksDB, illustrating the motivating cases of ZenFS+. Our question is how considerable the performance interference between

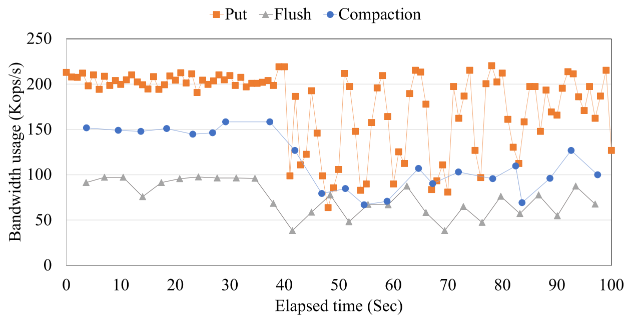


**FIGURE 2.** **Throughput fluctuation of put, flush, and compaction operations of ZenFS.**

flush and compaction is in ZNS SSD, even on a single KVS application.

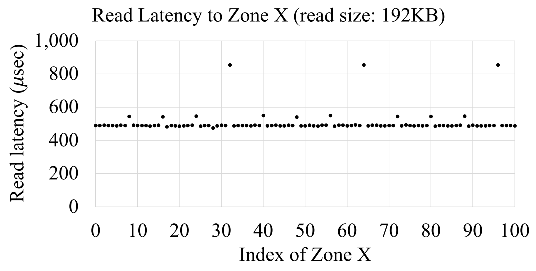
We use a real small-zone ZNS SSD device, such that the size of a zone is 72MB, the number of zones is 29,172, and the total size of the ZNS SSD is 2TB. The host PC is equipped with an Intel i7 CPU (8 cores) and 16GB DRAM. ZNS SSD is connected through PCIe 3.0 × 4 (max 3.94 GB/s). The baseline system configuration is an Ubuntu 20.04 distribution, Linux kernel 5.15, RocksDB v6.22, and ZenFS v2.1. To run with RocksDB, we use ZenFS from a previous study [7] and fill random key-value pairs for about 20 GB using dbbench_fillrandom.

Figure 2 shows the measured throughput for `put()`, flush, and compaction I/O operation in (Kops/sec). In the figure, the throughput seems stable for the first 40 seconds. Recall that `put()` operation inserts key-value pairs in the memtable, and no direct I/O operations are involved. Flush writes memtable to ZNS SSD, making a new sstable. Compaction reads sstables, and writes the merged sstable to ZNS SSD. Flush and compaction introduce writing and reading/writing I/O jobs, respectively.

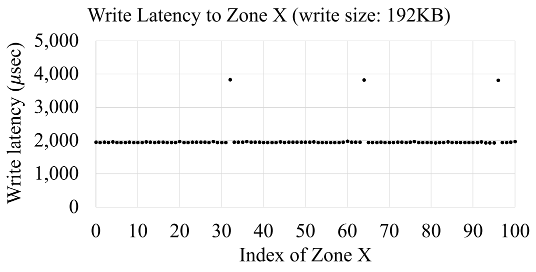
After the first 40 seconds, the throughput in the figure significantly fluctuates, and the throughput decreases for flush and compaction. Specifically, the average throughput of compaction, flush, and put operations decreases from 151 to 94.7 kops/s, from 91.3 to 63.2 kops/s, and from 204.3 to 158 kops/s, respectively. The fluctuation of `put()` throughput is dramatic. The throughput of `put()` changes from 204.3 kops/s to 63∼213 kops/s, of which standard deviation changes from 6.6 to 46.1.

The result is quite surprising because the ZNS SSD promises stable performance and low latency for long-running and write-friendly workloads. However, the results shows the significant fluctuation in `put()` latency, that might be due to the write stall for $L_0$ flush, as pointed in [5]. The question to us is why flush has been delayed because ZNS SSD could write sstables on different zones, and the next level write or compaction should not affect the $L_0$ flush, and the flush latency should be small.

We further investigated the reason for such degradation and fluctuation at a low-level ZNS SSD design. Using the raw ZNS interface, we concurrently access data blocks from



(a) Read latency to Zone X.



(b) Write latency to Zone X.

**FIGURE 3.** **Read and Write latency to Zone X.**

different zones and observe read and write latency from different threads. Figure 3a shows the read latency from two threads: one reads from zone 0, and the other thread reads data from zone X on the X-axis. The measured latency for reading zone X is presented in the graph. Similarly, Figure 3b shows the measured latency for writing zone X while another thread is reading from zone 0. The read and write block size is 192KB.

Interestingly, the read and write latency show an interesting pattern such that some zones are interrelated and interfere with each other, and some other zones do not. In Figure 3a, the read latency for zone X has three bands: 1) the read latency for zone 32, 64, and 96 is much larger than the latency of reading the other zones. For those zones, the read latency is as large as 850 microseconds. 2) The latency for zones 8, 16, 24, and some more are slightly larger than the latency of reading the rest of the zones. The latency in the spikes is as large as 550 microseconds. 3) The read latency is as small as 490 microseconds for the rest of the zones. Similar to the read, peak write latency is observed in Figure 3b. In the graph, the latency is as large as 4 milliseconds. In the figure, we can distinguish a group of zones. Some zones affect the performance of another zone, and some zones do not. If a zone is independent from the other zones, the zones do not affect the performance each other, keeping low latency. In the figure, the low-latency zones are independent from zone 0. However, zone 8, 16, 24 affect the performance of zone 0, meaning that the zones are in the same group.

Based on the observation, we can define IZGs (independent zone group) of an ZNS SSD device. An IZG is a set of zones that are independently operable from the rest of the zones. For example, zone 0, 8, 16, 24, 32, . . . are affecting each other, but independently operable with the rest of the

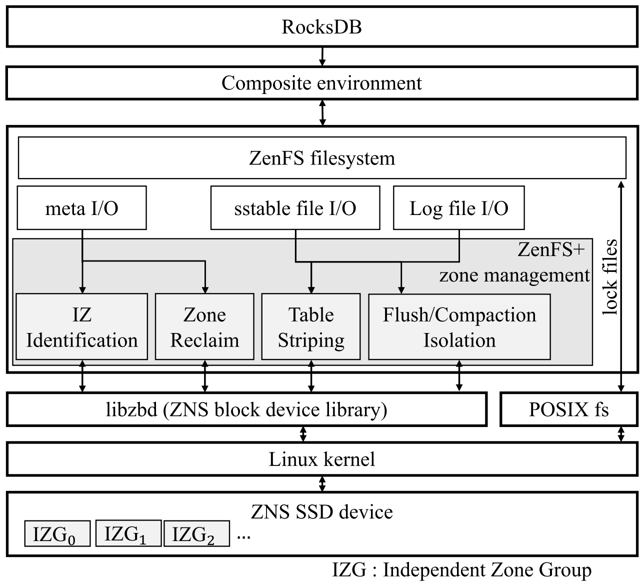


**FIGURE 4.** Overall structure of ZenFS+.

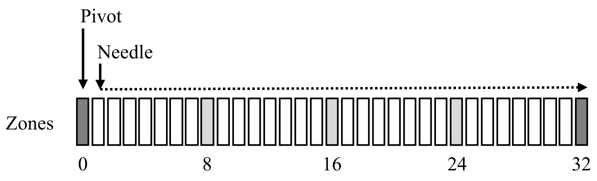


**FIGURE 5.** Independent Zones Group (IZG) identification using pivot and needle zones.

zones. Thus, the zones are grouped as an IZG. Note that there could be multiple IZGs in the same ZNS SSD.

A reason could be the internal hardware design of ZNS SSD that share parallelisms such as channels and ways. That is, some zones share the internal channels. That causes much worse performance in terms of bandwidth and latency and results in the fluctuating performance of the flush and compaction in Figure 2.

Also, we observe a possibility of performance enhancement for concurrent workloads. If we carefully select the multiple zones that do not affect the current zone, the aggregated throughput can be improved over utilizing a single zone. In summary, the zones in the same resources (e.g. channels and ways) affect each other; however, the zones in different resources could be concurrently utilized, without any performance penalty, in terms of latency and bandwidth.

The current ZenFS manages zone reclaiming with lifetime hint [7]. ZenFS tries to allocate sstables so that data in the same zone would have a similar lifetime and minimize the copy overhead for the live data at zone cleaning time. In the current implementation, ZenFS limits the number of active zones to 6∼12. That would work for large-zone ZNS SSD that has GB-sized zones. However, the zone reclaiming cost is considerable in large-zone ZNS SSDs [8]. For small-zone ZNS SSD, zone utilization increases relatively quickly than large-zone ZNS SSD. Thus, zone reclaiming cost is relatively small but is more frequently required.

## IV. DESIGN

As we observed in the previous section, small-zone ZNS SSD needs more consideration for better performance and isolation. Therefore, this section presents the design of ZenFS+.

The overall structure of ZenFS+ is in Figure 4. Extending the RocksDB interface of ZenFS [9], ZenFS+ introduces intelligent zone management functions. In the figure, ZenFS+ has IZ identification for identifying independently operable zones groups or Independent Zone Group (IZGs). Based on the IZG information, ZenFS+ isolates the performance of flush and compaction operations so that they can concurrently run without harming the performance of another. Also, ZenFS+ improves the performance of flush and compaction by sstable striping. Leveraging the small-zone ZNS SSD, we can achieve performance and isolation of ZNS SSD. Further, ZenFS+ allows a quick zone reclaiming using proactive garbage collection. The following subsections will elaborate on the design points in detail.

### A. IZG IDENTIFICATION

We devise an apparatus for identifying IZGs from a ZNS SSD device that shares the internal parallelism as illustrated in Figure 5. We prepare two threads: Pivot and Needle threads. Pivot points to a zone (pivot zone), and the needle moves around the zones (needle zones) from the pivot to the last zone. We measure the read latency from the needle zone thread while the pivot thread is concurrently reading the pivot zone. If the read latency from the needle is similar to baseline latency, that implies the pivot and the needle are in different IZG and can independently operate. However, if the latency is larger than a threshold, the needle zone is added to the IZG of the pivot zone.

Once the IZG for the pivot is constructed, the zones are independently operable with the rest of the zones. Similarly, IZG identification is done for the next pivot zone and constructs the next IZG. The overall process repeats until IZG sets make the entire partition for all the zones.

Leveraging the IZG information, ZenFS+ can exploit the internal parallelism as well as performance isolation between flush and compaction.

### B. FLUSH/COMPACTION ISOLATION

ZenFS+ isolates the performance between flush and compaction. With the help of IZG identification, all the zones are partitioned into IZGs. The SSD used in our system has a total of 32 different IZGs ($IZG_0 \sim IZG_{31}$), and each IZG has about 910 zones.

A flush thread writes a sstable to the storage, and a compaction thread reads sstables from storage, merges them in the memory, and writes the merged sstable to the storage. That is, flush makes write I/O, compaction makes read and write I/O. Note that the read and write speeds are different and write latency is about four times longer than read in Figure 3.

With our design, we use different IZG zones for flush and compaction. Reading sstable from ZenFS+ is presented in Algorithm 1. In the pseudo-code, each IZG has a state, idle or busy. As shown in the algorithm, compaction read should operate on the designated zone where the sstable is stored. Once the target zone is selected, the designated IZG is set to busy. Once the sstable is read, the IZG is set to idle.

**Algorithm 1** Read sstable$_i$

```
Data: $IZG_i = \{idle, busy\}$
Data: $sstable_i := sstable\_index$
$len \leftarrow sizeof(sstable_i)$;
while $len > 0$ do
    $zone \leftarrow FindZone(sstable_i)$;
    $target \leftarrow FindIZG(zone)$;
    $mutex\_lock(target)$;
    $IZG_{target} \leftarrow busy$;
    Read the sstable$_i$;
    $IZG_{target} \leftarrow idle$;
    $mutex\_unlock(target)$;
    $len \leftarrow len - sizeof(sstable_i)$;
end
```

Writing sstable from ZenFS+ is presented in Algorithm 2. Each IZG again has a state, idle or busy. On flush write, ZenFS+ first finds one of the idle IZGs. Then, it sets the selected IZG to busy and writes data on the zone in the selected IZG. Once the write is completed, the IZG is set back to idle. Note that the flush and the compaction threads can concurrently operate. Thus, `FindFreeZone()` should be a synchronized function. Further, compaction read should wait for the target IZG if the IZG is busy, and used by flush thread.

**Algorithm 2** Write sstable$_i$

```
Data: $IZG_i = \{idle, busy\}$
Data: $sstable_i := sstable\_index$
$len \leftarrow sizeof(sstable_i)$;
while $len > 0$ do
    $target \leftarrow FindFreeIZG()$;
    $zone \leftarrow FindFreeZone(target)$;
    $mutex\_lock(target)$;
    $IZG_{target} \leftarrow busy$;
    Write the sstable$_i$;
    $IZG_{target} \leftarrow idle$;
    $mutex\_unlock(target)$;
    $len \leftarrow len - sizeof(sstable_i)$;
end
```

If the compaction thread is working on a zone, the IZG should be busy, and the IZG would not be selected for the flush. Because the flush operates on idle IZG, it can efficiently avoid interference with compaction reading. If the flush thread writes on a zone first, and the compaction wants to operate on that IZG, the flush thread always preempts the compaction and delays the compaction job.

On compaction write, ZenFS+ can also flexibly choose the zones from free IZGs. It also sets the selected IZG to be busy during the writing and IZG to be idle after the writing. For flush and compaction write, the flush and the compaction threads compete for the IZG. However, the number of IZG is as large as 32; the concurrency seems enough to isolate the flush and compaction writes, considering the case when the RocksDB have two threads (the flush and the compaction threads) compete for IZG.

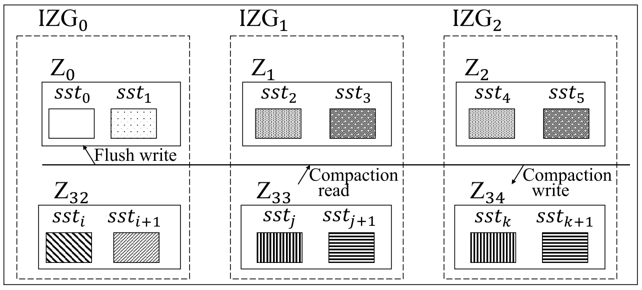


**FIGURE 6.** ZenFS+ with isolated flush from compaction.

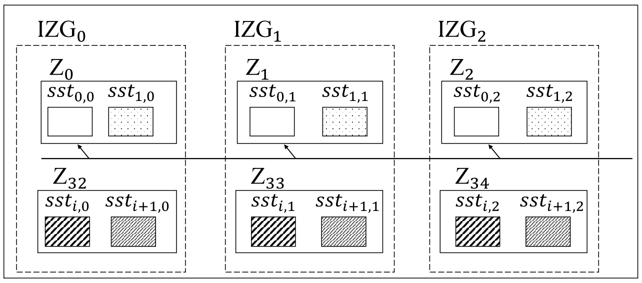


**FIGURE 7.** ZenFS+ sstable striping: concurrent write to $sst_{0,0}$, $sst_{0,1}$ and $sst_{0,2}$.

Figure 6 shows the conceptual illustration of flush write from compaction read and write. In the figure, IZG$_0$, IZG$_1$, and IZG$_2$ can be concurrently accessed without any interference. Thus, flush writing on zone 0 ($Z_0$) is isolated from compaction on zone 33's read and zone 34's write. In summary, ZenFS+ tries to isolate the performance between flush and compaction, which is our primary design goal.

## C. SSTABLES STRIPING

Besides the performance isolation, ZenFS+ improves the overall performance by sstable striping. ZenFS+ can concurrently utilize multiple zones for faster reading and writing. To increase the aggregated I/O bandwidth, free IZGs can also be used for reading and writing if available.

Striping sstables over multiple zones is illustrated in Figure 7. In the figure, sstable$_0$ is spread over three zones, zone$_0$, zone$_1$ and zone$_2$. sst$_{0,0}$, sst$_{0,1}$, and sst$_{0,2}$ represents striped sstable$_0$ in each zones, respectively. Then, ZenFS+ can concurrently flush sstable$_0$ to three zones, increasing the bandwidth utilization by three times.

Striping of sstable makes ZenFS+ operations complicated. If we spread a sstable over all IZGs when we flush the sstable, then flush performance would get the best bandwidth; however, compaction would starve until the flush is completed. Also, once a compaction read has to utilize all the IZGs, the flush cannot timely get free IZG because all the IZGs are busy. If we use only one IZG for flush write, the flush write will under-utilize the device bandwidth, and RocksDB will suffer from low flush performance. To balance between the wide striping for maximum bandwidth and limited striping for guaranteed performance for flush and compaction, we define the *maximum striping width* as a control parameter of ZenFS+ so that the flush can get guaranteed bandwidth, avoiding write stall. We set the default maximum striping width as 16.

### D. PROACTIVE GARBAGE COLLECTION

Zone garbage collection (GC) is a job that reclaims used zones to regenerate newly available zones. It consists of three steps: 1) selecting candidate zones, 2) copying valid data from selected zones, and 3) sending the *zone_reset* command to ZNS SSD. ZenFS+ requires more consideration on GC than the existing ZenFS since it is based on small-zone ZNS SSD. In this case, zone GC may takes less time due to small size, but regenerating less free space, meaning that zone GC need to be triggered more frequently.

There are several issues related to zone GC such as triggering time, zone selection policy, hot-cold separation policy, and metadata management after copying. We follow most existing schemes of ZenFS except the triggering time. In ZenFS, zone GC is triggered when all the zones are used up. It requires considerable run-time, having a high potential to delay user requests. In contrast, in ZenFS+, zone GC is triggered periodically (e.g. every 10 seconds) or when the available space is below a threshold (e.g. 10% utilization).

We implement two versions of zone GC, called as minor GC and full GC, respectively. Note that ZenFS+ (also ZenFS) allocates different levels into different zones to achieve the hot-cold separation benefit since the lifetime of sstables in level 0, $L_0$, is short [15]. Hence, a zone allocated for $L_0$ has a high probability of containing all invalid data. Hence, we search zones in $L_0$ first and then find a zone that has all invalid data. Then, we just send the *zone_reset* command without copying. This is called as the minor GC. For the full GC, ZenFS+ scans all zones ranging from $L_0$ to $L_{max}$ and selects candidate zones using the greedy policy selecting lower zones first.

As already mentioned, ZenFS+ proactively triggers zone GC every 10 seconds. In this case, ZenFS+ uses the minor GC. ZenFS+ also runs zone GC when the ZNS SSD's available space is below a given threshold (*fullGC_trigger*). In this case, it invokes the full zone GC. Although our proactive GC runs frequently, the design makes it easy and light weight because 1) the zone size is small and so does the zone reset cost, and 2) minor GC avoids data copy, which is the major source of the overhead in GC.

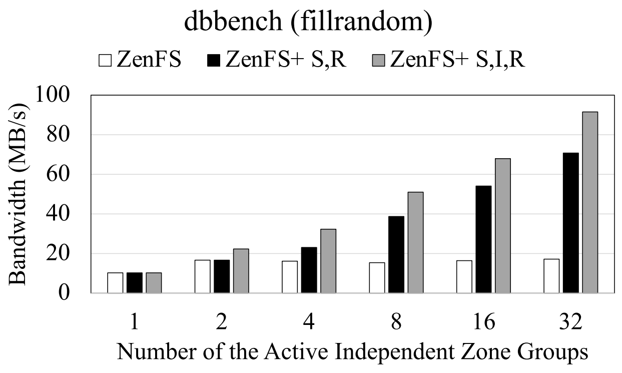


**FIGURE 8.** ZenFS+ write performance with fillrandom workload.

One additional issue in ZenFS+ is that it stripes sstables over multiple zones for the high write throughput. Thus, zone GC is conducted at the unit of multiple zones, like the superblock based FTL in traditional SSDs [16]. The merit of this coarse-grained GC is reducing the overall GC time by utilizing IZGs in parallel.

## V. EVALUATION

### A. PERFORMANCE AND ISOLATION OF ZENFS+ UNDER FILL RANDOM WORKLOAD

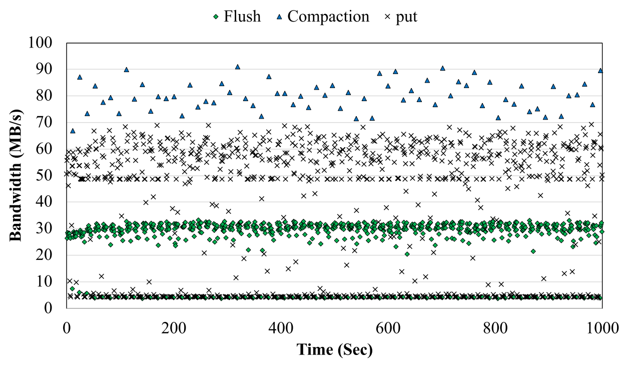
We evaluate ZenFS+ with different configurations on RocksDB benchmarks. Our hardware and software configuration for experiments are the same one in Section III. We comparatively present the result of ZenFS+ with the open source ZenFS [9]. Firstly, we present the write performance with a fillrandom workload.

To measure the bandwidth, we write about 20GB of random key-value pairs. The default key size is 20 bytes, and the value size is 800 bytes. Memtable size is 64MB.

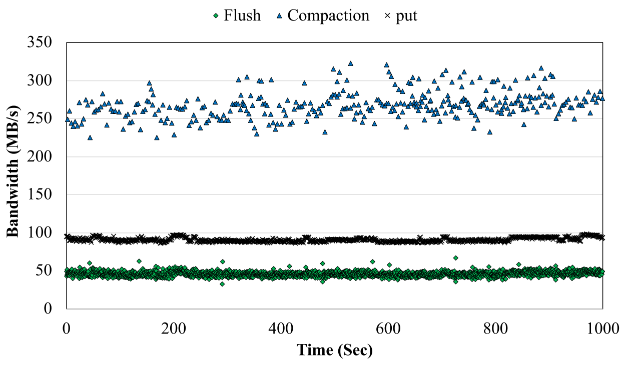
Figure 8 shows the measured bandwidth during the fill random. In the figure, ZenFS+ S, R represents the ZenFS+ with sstable striping and zone reclaiming. ZenFS+ S,I,R represents the ZenFS+ with sstable striping, flush/compaction isolation, and zone reclaiming.

To present the performance enhancement from sstable striping, we change the number of active IZGs. In the graph, X-axis is the number of active IZGs. Original ZenFS uses 6∼12 active zones, and the performance in the range is almost the same. A possible reason for the low bandwidth of ZenFS is that we are using a small-zone ZNS SSD device. For the small-zone ZNS SSD, the limited internal parallelism in a zone makes it difficult to scale performance. ZenFS+ S, R shows the performance scalability, overcoming a zone's limited internal parallelism. Utilizing multiple zones, ZenFS+ achieves higher aggregated bandwidth. Besides, flush/compaction isolation further improves performance because ZenFS+ isolates the IZG to guarantee flush performance, avoiding the write stalls.

In terms of utilized bandwidth, ZenFS and ZenFS+ show different characteristics for fillrandom workload. We measured the utilized bandwidth for `put()`, flush, and



(a) Bandwidth usage with ZenFS



(b) Bandwidth usage with ZenFS+

**FIGURE 9.** Fill random utilized bandwidth during the benchmark execution.

compaction during the execution. For `put()`, we calculated the amount of data by accounting for the completed transactions multiplied by the number of bytes per transaction. We plotted raw flush and compaction and put bandwidth from the RocksDB log to present the time-series variation of device utilization. The measured bandwidth with ZenFS (9a) and ZenFS+ (9b) are presented in Figure 9.

In Figure 9a, flush shows the device utilization around 30MB/s, compaction ranges from 70∼90 MB/s. The `put()` has two ranges: a high band around 50∼68 MB/s and a low band around 5∼7 MB/s. With ZenFS, flush presents stable bandwidth utilization; however, it would delay the `put()` operation quite frequently, and `put()` operation cannot enough bandwidth, resulting in low `put()` bandwidth.

With the ZenFS+ in Figure 9b, `put()` and flush can get guaranteed bandwidth, which shows strong isolation of flush from compaction. It shows much more stable flush bandwidth usage than ZenFS and higher bandwidth usage. A possible reason is that ZenFS+ segregates flush IZGs from compaction IZG, and both operations can exploit the internal parallelism of ZNS SSD hardware. In addition, compaction bandwidth ranges from 230 ∼ 300 MB/s, which is about three times higher device utilization compared with ZenFS. The fluctuation in the compaction might be due to the fact that the priority of the compaction thread is lower than that of flush.

Additionally, observe that `put()` performance is stable and predictable with ZenFS+. It has a very narrow single band, keeping the bandwidth as high as 91MB/s, whereas the original ZenFS's average `put()` bandwidth is about 35MB/s. The result supports that ZenFS+ improves the perceived performance for the user 2.6 times higher.

**TABLE 1.** Tail latency of ZenFS+ for put() requests.

| unit (usec) | ZenFS | ZenFS+S,R | ZenFS+S,I,R |
|---|---|---|---|
| 99P | 20.825 | 7.322 | 7.690 |
| 99.9P | 1,166.235 | 22.094 | 22.736 |
| 99.99P | 1,191.488 | 285.695 | 175.499 |
| 99.999P | 2,121.915 | 461.809 | 258.489 |

Overall tail latency for 1,000 seconds fillrandom workload is given as following Table 1. ZenFS+S shows the tail latency of ZenFS+ with sstable striping, and ZenFS+S,I,R shows the tail latency with sstable striping and flush/compaction isolation. The 99th percentile tail latency of ZenFS and ZenFS+ are 20.8 usec and 7.3 usec, respectively. ZenFS+ reduces tail latency by a factor of 3, which is a good result. The gap increases when we magnify the 99.9 and 99.99 percentile tail latency. The tail latency of ZenFS is over 1 millisecond, and ZenFS+ with sstable striping still remains in 22.736 ms, 175.499 ms, respectively for 99.9P and 99.99P tail latency. A reason is that the sstable writing is spread over multiple IZGs; thus, write stall due to slow flush could be further relaxed. ZenFS+ with flush/compaction isolation further reduces the tail latency. For 99.999P tail latency, ZenFS+S,I,R shows 1/8 latency values thanks to the flush/compaction isolation. Note that the maximum latency improvement is observed in 99.9P tail latency, in which case ZenFS+ reduces the tail latency by 1/51.

To plot the latency trends in time-series, we separately gather the tail latency statistics for every 100 ms `put()` requests (requests bucket). Then, we measure the tail latency for each `put()` requests bucket. Then, we plot 99P, 99.9P, and 99.99P tail latency values. Thus, the graph in Figure 10 shows the time series trend of tail latency. Note that the Y-axis has different scales due to the large gap in latency values between the configurations.

In the graph, ZenFS+ presents much-improved latency along with the sstable striping and flush/compaction isolation. With sstable striping, the 99P tail latency decreases from 1.5 ms to 20 microseconds. The 99.99P tail latency shows much dramatic change of flush/compaction isolation, compared with ZenFS+ stable striping.

### B. ZENFS+ UNDER DIVERSE WORKLOADS

More than the fillrandom workload, we run the RocksDB benchmark for the different workloads, including overwrite, updaterandom, readrandom, and readwhilewriting. We measured the overall performance in terms of bandwidth.

Figure 11 shows the results from dbbench benchmarks. ZenFS+ shows comparably better performance than previous ZenFS. For Fillrandom and overwrite, the bandwidth has increased by 4.7∼4.8 times, and for updateRandom, ReadRandom, and ReadwhileWriting, the ZenFS+ presents improved bandwidth by 2.5∼2.8 times higher than ZenFS.

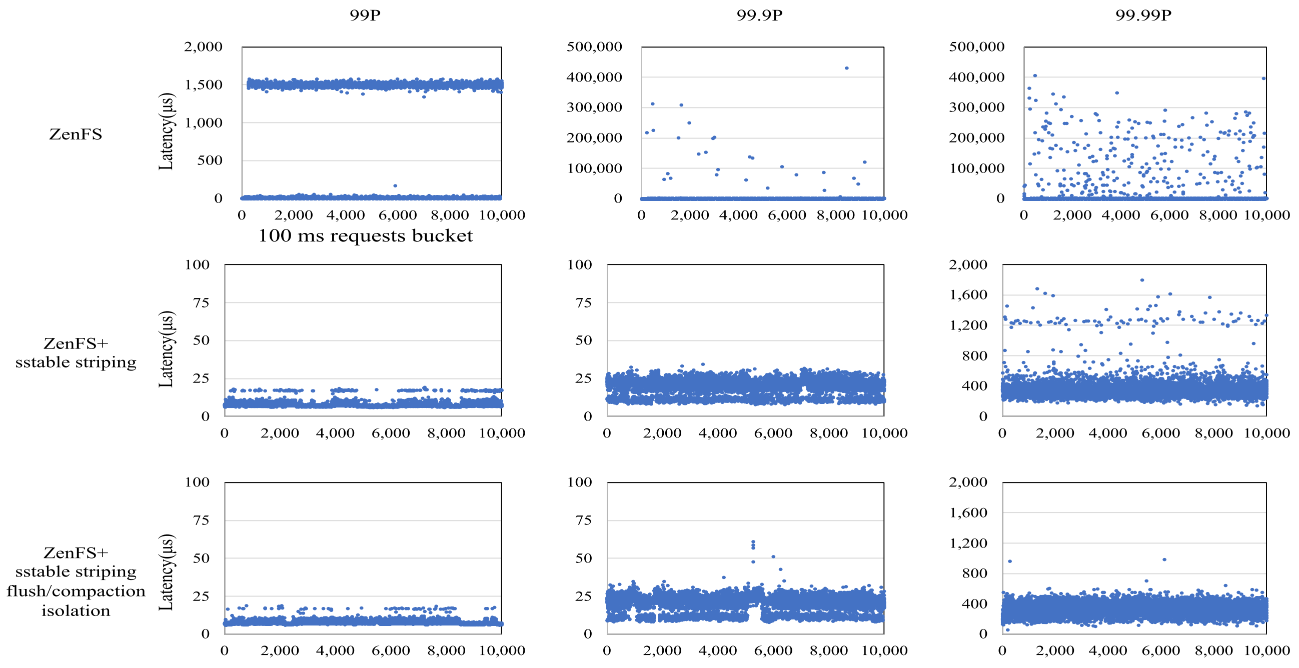


**FIGURE 10.** ZenFS+ Tail latency per 100ms *put*() requests bucket.

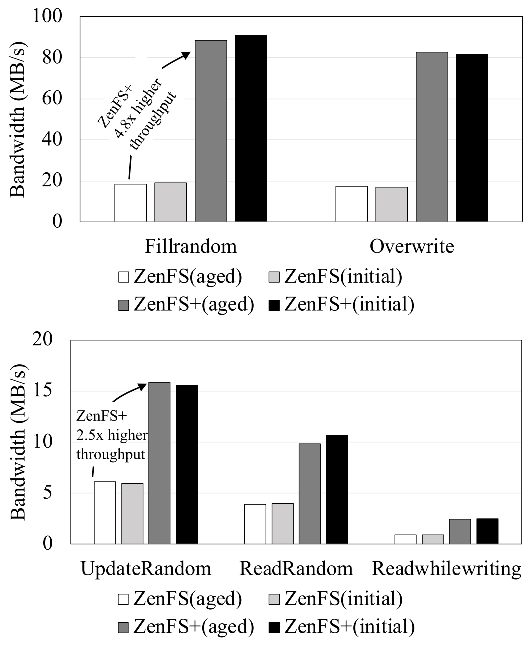


**FIGURE 11.** More dbbench results with ZenFS+.

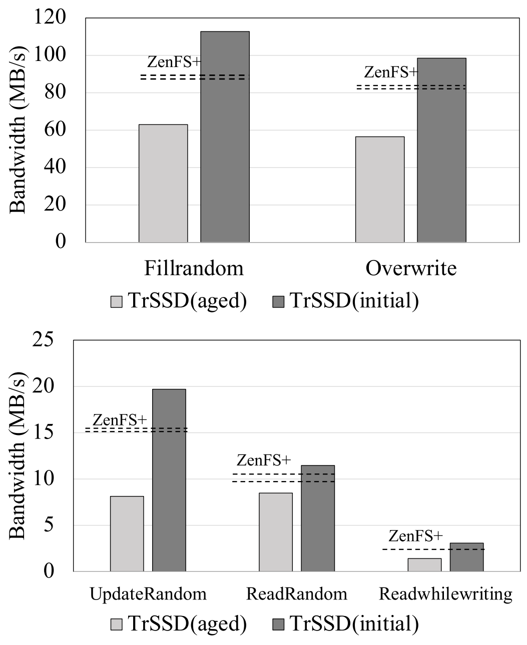


**FIGURE 12.** dbbench results from traditional SSD.

To make the ZNS SSDs aged, we write 10TB on the device, resetting random zones. After writing the 10 TB of data, we reset 20% of the zones for further writing. The aged ZNS SSD performance in the graph shows almost negligible differences (-2∼4%) for most cases. The largest outlier is ReadRandom, of which overhead is about 8%.

The result contrasts with the result from traditional SSD. We ran the same experiments with the traditional SSD. The traditional SSD used for this experiment has the same internal architecture as the small-zone ZNS SSD except only for the feature that supports the ZNS interface. The traditional SSD uses the traditional block interface. The traditional SSD is aged with writing 10TB of data on the 2TB SSD. The device has about 15% of over-provisioning capacity and runs with full FTL that runs garbage collection.

With the traditional SSD, considerable performance deterioration along with the lifespan of SSD is observed, as shown in Figure 12. Due to the garbage collection and the block

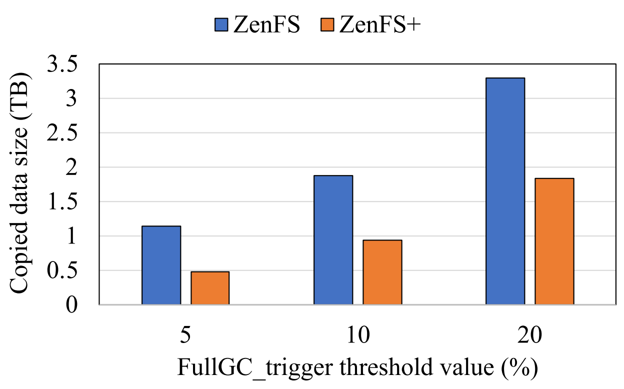


**FIGURE 13.** **Copied bytes from GC during 5 TB put transaction.**

interface tax, the performance drop ranges from 44∼58%. The smallest overhead was 25% when we ran ReadRandom.

ZenFS+ shows a very narrow band of performance change over time. Due to the limited IZGs in flush width, ZenFS+ performance is slightly lower than the SSD in the initial state. However, the performance after aging contrasts with the clear advantage of ZNS SSD over traditional SSD in the lifespan.

## C. PROACTIVE GC AND WRITE AMPLIFICATION RESULT

ZenFS+ implements proactive GC with quick minor GC, which affects the performance in a more prolonged execution of RocksDB. To present the viability of proactive GC, we measured the amount of copied data for a longer execution run.

First, we fill the 1.5TB ZNS SSD storage with random data using randomfill workload. Second, we run the overwrite workload for 3.5 TB. In the given workload, RocksDB will continuously put data to the memtable, flush them to the sstables, and eventually trigger compaction. The compaction would eventually result in storage consumption, and the working set size is larger than the physical storage size; thus, compaction should trigger garbage collection in the end.

ZenFS has no garbage collection. Thus, if the write size is larger than the storage, RocksDB stops at the point. We slightly modified the ZenFS to trigger the full GC at a specific threshold (*fullGC_trigger*) like ZenFS+. ZenFS+ conducts minor GC every 10 seconds and full GC when the device utilization hits the threshold.

Figure 13 shows the amount of copied data from GC during the execution. In the graph, full GC copies 1.1 ∼ 3.3 TB of data due to GC, whereas ZenFS+ reduces the amount of write to 0.5 ∼ 1.8 TB, which shows our proactive GC is effective and improves write amplification for the long-running workloads, also.

## D. YCSB ON ZENFS+

To present the viability of ZenFS+ in more realistic cases, we run the YCSB on ZenFS+. YCSB is a popular benchmark suite for cloud data center transaction processing. With the 20GBs of data, we ran YCSB A∼F workloads and measured the execution time for the workloads. A summary for workload A∼F is given in the following Table 2.

**TABLE 2.** **YCSB workload.**

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| r:w:u ratio | 1:0:1 | 95:0:5 | 1:0:0 | 95:5:0 | 95:5:0 | 1:1:1 |
| descrip-tion | Update-heavy | Read-mostly | Read-only | Read-latest | Short range query | Read-modify-write |
| Req. dist. | Zipfian | Zipfian | Zipfian | Latest | Zipfian | Zipfian |

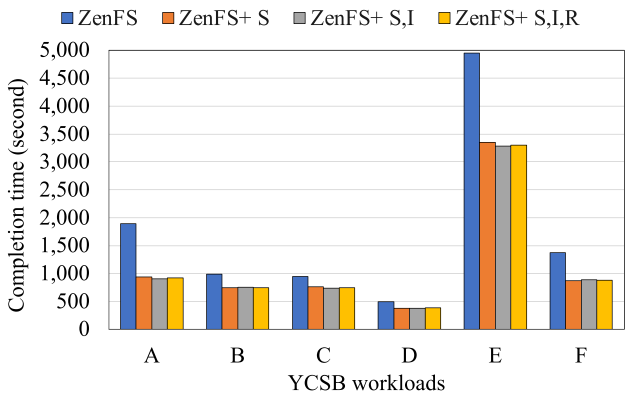


**FIGURE 14.** **ZenFS+ performance under YCSB workloads.**

Figure 14 shows the completion time of each workload under different configurations. Note that the Y-axis is the runtime in seconds, and the lower values present the better results. In the graph, ZenFS represents the result from the original ZenFS, ZenFS+ S represents the ZenFS with sstables striping, ZenFS+ S, I adds the flush/compaction isolation, ZenFS+ S, I, R adds the zone reclaim to the previous configurations, respectively.

In the graph, workloads A, E, and F show meaningful execution time enhancement. Workloads A and F have write-heavy and update-heavy operations, and the sstable write performance is going to be important. Workload E is a range scan workload, and the workload reads 100k data for the scanned entry. If we stripe data over multiple zones, the reads could also get help from striped reading.

In the graph, the primary performance gain comes from sstable striping. Also, flush/compaction isolation helps in overall performance. The performance gain in terms of throughput for A, E, and F are 105%, 50%, and 57%, respectively. The result supports that ZenFS+ design effectively achieves performance enhancements.

For read-heavy workloads (B, C, D), the execution time has slightly increased in ZenFS+. That might also be due to sstable striping. Contrary to the write, when a user reads sstable, the reading zones are designated based upon the location of the data file. Thus, the reading threads could compete for the IZG, presenting some overhead of reading from the same IZG. Note that YCSB workload queries randomly distributed key-value pairs. The distribution of accessing keys are Zipfian, yet the spacial locality of

**TABLE 3.** Reduced execution time and performance gain (ratio over ZenFS).

| workload | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| exec.time (s) ZenFS+ | 921.9 | 749.4 | 750.8 | 383.7 | 3301.1 | 877.1 |
| exec.time (s) ZenFS | 1891.3 | 990.3 | 948.8 | 496.3 | 4950.5 | 1374.9 |
| reduced time(s) | 969.4 | 240.9 | 198 | 112.6 | 1,649.4 | 497.8 |
| perf. gain | 105% | 32% | 26% | 29% | 50% | 57% |

accessing key is not considered. The locality of accessing keys makes random access in different sstables, rather than reading keys from nearby sstables; thus, it is known that read performance in production could be better in practice [2].

Thus, the performance gain is less than expected. The reduced performance for B, C, and D are 32%, 26%, and 29%, respectively. A reason could be the asymmetric read-write performance of flash I/O, and the writing has a more significant impact than the reading.

The reduced execution time and more detailed results compared with ZenFS is presented in Table 3. In summary, the ZenFS+ results from YCSB macro benchmark shows 50% $\sim$ 100% enhancements in write-intensive workload, and about 20% $\sim$ 30% enhancements in read-intensive workloads.

### E. ZIPPYDB ON ZENFS+

Although YCSB has long been a standard benchmark for NoSQL databases [17], a recent study [2] reported that the real-world workload on production systems is slightly different from synthetic YCSB workloads. We further run ZenFS+ with ZippyDB workload, where ZippyDB is a storage engine for Paxos-based distributed KVS, and RocksDB stores metadata of ObjStorage file or a data block with its address information. The exact parameter we used is the same in the paper [2].

ZippyDB workload has much more reading requests than writing requests, and the key-value size varies according to the modeled pattern. ZippyDB creates randomly generated 50 million KV entries before execution, and the loaded RocksDB size is about 4GB. Then, according to the workload patterns, put, get, and seek operations are conducted on the database. For specific parameters, the key size is 48 bytes, and `get()`, `put()`, and `seek()` operations are 83%, 14%, and 3% of total queries, respectively.

We measured the throughput for the main workload execution, mixed reading, writing, and seeking workloads. ZenFS achieves 17.6 MB/s, and ZenFS+ achieves 20.5 MB/s in terms of throughput. The result from our experiment shows ZenFS+ improves 18% throughput under production-close workloads.

## VI. DISCUSSION

ZNS SSDs, like any other SSDs, have several hardware design parameters. The number of zones, the size of zones, the number of channels and ways, the number of pages in blocks, etc. One concern around the ZenFS+ is whether the design fits large-zone ZNS SSD or not. For example, a large-zone ZNS SSD device would easily get the write performance without sstables striping. ZenFS+ intelligently identifies the IZG information; thus, ZenFS+ configures a single IZG for large-zone ZNS SSD device. In the case, it inherently limits the sstable string width to 1, and compatibly work with ZenFS settings.

Currently, ZenFS+ assumes small-zone ZNS SSD as an alternative to large-zone ZNS SSD because small-zone ZNS SSD has a substantial benefit with regard to performance isolation under concurrent workloads. Also notice that large-zone ZNS SSD is inherently hard to achieve performance isolation because it does not allow to control the internal parallelism from software side. Namely, it gives much freedom in software design, considering the zones allocation and fine-grained garbage collection.

ZenFS+ focuses on RocksDB and KVS applications. However, the proposed primitives, such as IZG-based striped writing and minor GC, are generally applicable to small-zone ZNS SSD and can be used in different applications. Large-zone and small-zone ZNS SSD devices would have different applications that fits its usages. Large-scale machine learning system's model check pointing would like to leverage large-zone ZNS SSD, maximizing the device bandwidth. On the other hands, multi-tenant applications' performance provisioning would require small-zone ZNS SSD, guaranteeing isolated performance between each tenant. ZenFS+ provides a new perspective of leveraging ZNS SSD, not only in terms of long-running GC-related benefits, but also in terms of performance and isolation, extending the ZenFS study.

Yet, some information from the hardware could improve the current ZenFS+. For example, IZG identification is a software guessing method for obtaining information on internal parallelism. For some ZNS SSD devices, the internal mapping between the channels and zones could be different. The software can flexibly leverage internal parallelism if any public interface gives such information.

## VII. RELATED WORK

There are lots of studies on RocksDB. First of all, Siying et al. presented extensive experience in developing RocksDB in practical systems [3]. It presents how RocksDB has been evolving and optimizing various aspects of performance, including write/space amplification, data format and compression, backward compatibility, backup and data corruption handling, and scalability under multi-RocksDB instances.

YCSB has been a standard benchmark in NoSQL databases. Workload statistics from real-world applications are the precious basement for further optimization. Zhichao et al. have reported how a real-world application's workload differs from the synthetic model in YCSB [2]. The authors revealed the locality pattern in searching keys, the distribution of keys and value sizes, and time-varying search patterns with the three representative workloads.

Also, Yoshinori et al. have reported how RocksDB supports large-scale social graph applications [4].

To optimize the search time, MatrixKV [18], DiffKV [19] separates keys from values. By separating the keys from values, we can efficiently utilize the I/O bandwidth for searching keys. Write with a guard, PebblesDB [20] divides the key space into fragments and proposes a fragmented LSM tree-based KVS. Bourbon [21] uses learned index for quickly pinpointing the index location in the sstable, TridentKV [22] also uses adaptive index for read-optimization. Remix [23] proposes re-indexing the tree for range query, S3 [24] uses CPU-optimized data structure [25] and semi-ordered skiplist, respectively.

Some recent studies observed that complicated flush and compaction would interfere with each other resulting in long tail latency [5], [26]. In a study, the authors proposed a scheme that prioritizes the flush thread over compaction threads [5]. In the scheme, the prioritized flush can preempt compaction so that the flush and write operation cannot be delayed by compaction. In a recent study [26], the authors proposed a KVS flush scheme on a new operation mode (Predictable Latency Mode or PLM) of SSD. With the PLM mode, the device's internal channel can be flexibly configured so that other operations cannot block the flush channel.

Optimizing KVS performance, some studies leverages the storage hierarchy and advanced memory technologies. Studies focus on the different performance characteristics of DRAM, NVM, and relatively slow SSDs. SpanDB [27], [28] allows the user to place low-level sstables in slower SSDs, whereas putting top-level sstable and WAL on the faster NVMe SSD. With high-performance hardware on a higher-level storage hierarchy, SpanDB adopts parallel WAL writes via SPDK, improving throughput and latency. MatrixKV [18] and TriangleKV [29] are approaches to leverage NVMe with SSD. The authors proposed a matrix container that packs level-0 sstable within NVM. Also, the authors focused on write stalls from $L_0 - L_1$ compaction and proposed column compaction that conducts $L_0 - L_1$ compaction at fine-grained key ranges. Pacman [30] intelligently leverages persistent memory in garbage collection and compaction operation of log-structured KVS. The authors minimize the indexing and copying overhead in compaction and garbage collection. Placing index and staged buffer [31] and WAL [32] on persistent memory could accelerate LSM-style KVS performance.

Minwoo et al. recently presented a new RocksDB acceleration scheme leveraging parallel I/O access of ZNS SSD [33]. The authors pointed out that the level of SST could affect the parallelism in ZNS SSD utilization. The proposed striping idea takes the advantage of small-zone SSD, similarly to our paper. On the other hand, our work considers leveraging the hardware parallelism aligning with the software threading such as flush / compaction. Thus, we improve the application's tail latency as well as throughput. Also, minor GC of ZenFS+ largely reduces the unpredictable overhead

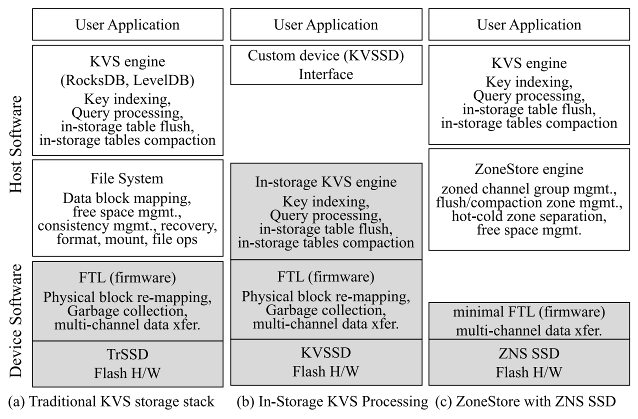

Host Software
Device Software
User Application
KVS engine (RocksDB, LevelDB) Key indexing, Query processing, in-storage table flush, in-storage tables compaction
File System Data block mapping, free space mgmt., consistency mgmt., recovery, format, mount, file ops
FTL (firmware) Physical block re-mapping, Garbage collection, multi-channel data xfer.
TrSSD Flash H/W
Custom device (KVSSD) Interface
In-storage KVS engine Key indexing, Query processing, in-storage table flush, in-storage tables compaction
KVSSD Flash H/W
KVS engine Key indexing, Query processing, in-storage table flush, in-storage tables compaction
ZoneStore engine zoned channel group mgmt., flush/compaction zone mgmt., hot-cold zone separation, free space mgmt.
minimal FTL (firmware) multi-channel data xfer.
ZNS SSD Flash H/W
(a) Traditional KVS storage stack
(b) In-Storage KVS Processing
(c) ZoneStore with ZNS SSD

**FIGURE 15. Various key-value store storage stack with different SSD architecture.**

from GC, which is essential component for long-term execution.

Figure 15 presents the software layers for KVS and the internal structure of traditional SSD devices. The software layer consists of device firmware and host software. Device firmware includes a flash translation layer (FTL) that maps logical blocks to physical blocks, conducts garbage collection (GC), and so on. Host software includes a file system inside the OS kernel and KVS engine in the application.

There have been concerted efforts toward efficient, stable, and intelligent next-generation SSDs. One approach tries to move the host software into an SSD device, known as ISP (In-Storage Processing), as depicted in Figure 15(b). A typical example of this approach is KVSSD (Key-Value SSD) [34], [35], [36], [37], which implements a key-value store within SSDs. In the figure, the user application directly sends `put()` and `get()` commands to the device, and device firmware handles all the application-specific operations internally. KVSSD have already been standardized [38] although there are several application-specific SSD studies such as in-flash acceleration [39], in-storage ransomware filter [40] and in-storage file system indexing [41].

The other approach tries to move FTL functionalities out of the device and let the host software drives the hardware, as presented in Figure 15(c). In this approach, hardware exposes the internal structure of SSDs so that the host software makes the best use of underlying hardware. In the figure, the device has minimal FTL with device software, and host software drives the hardware and runs the KVS application. Typical examples of this category include OCSSD (Open-Channel SSD) [42], [43], [44], [45], SDF (Software Defined Flash) [46] and Multi-streamed SSD [47].

In the context of ZNS SSD, several optimizations in the storage stack have been published in the literature. For example, an RDB application running on an LSM-style based storage engine makes duplicate logging functions at different layers. Kecheng et al. presented an LSM-tree-based RDB system, removing the redundant logging function from the storage stack [48]. Also, some KVS studies optimize the software overhead of unnecessary file system interfaces for KVS applications, introducing minimal filesystem adaptation layers such as TopFS [27] and ZenFS [7]. Hee-rock et al. observed the zone's lifetime prediction inaccurately in ZenFS [14]. Instead, the authors proposed compaction-aware zone allocation that makes the lifetime of a zone more accurately predictable. Jung and Shin observed that compaction of KVS makes partially invalid zones in ZNS SSD [15]. To minimize the garbage collection overhead, the authors selectively put sstables that have similar lifetimes into the same zone, relaxing the write and space amplification from garbage collection. Shai et al. presented an approach to using ZNS SSD as a swap device [49]. Because ZNS SSD has a little cost of firmware-level garbage collection, the swap performance has significantly increased.

## VIII. CONCLUSION

This paper proposes a new approach to KVS, using ZNS SSD. Extending ZenFS, the state-of-the-art ZNS-supporting software, we present ZenFS+. ZenFS+ presents how ZNS SSD can support the performance and isolation of modern KVS. ZenFS+ intelligently distinguishes independent zones group and isolates the performance of flush job from compaction. Thus, flush utilizes much stable bandwidth and latency for write intensive workloads. In our experiments, ZenFS+ reduces 99.9P tail latency by 1/51. Also, ZenFS+ leverages the IZG information to exploit the device's internal parallelism. With the striped sstable, ZenFS+ achieves up to 4.8x higher flush throughput and 2.6x higher application throughput for microbenchmark, and about twice throughput in macro benchmark, compared with the current ZenFS implementation. We further implement a proactive GC, a missing part in the current ZenFS, making it more sustainable in real-world systems.

## ACKNOWLEDGMENT

The authors appreciate all reviewers and editors for their precious comments and works on this article.

## REFERENCES

[1] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, S. Sosothikul, D. Terry, and A. Vig, "Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service," in *Proc. USENIX ATC*, 2022, pp. 1037–1048.

[2] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in *Proc. USENIX FAST*, 2020, pp. 209–223.

[3] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "RocksDB: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, pp. 1–32, Oct. 2021, doi: 10.1145/3483840.

[4] Y. Matsunobu, S. Dong, and H. Lee, "MyRocks: LSM-tree database storage engine serving Facebook's social graph," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, Aug. 2020, doi: 10.14778/3415478.3415546.

[5] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in log-structured merge key-value stores," in *Proc. USENIX ATC*, 2019, pp. 753–766.

[6] M. Bjørling, "From open-channel SSDs to zoned namespaces," in *Proc. USENIX VAULT*, 2019.

[7] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis, "ZNS: Avoiding the block interface tax for flash-based SSDs," in *Proc. USENIX ATC*, 2021, pp. 689–703.

[8] H. Bae, J. Kim, M. Kwon, and M. Jung, "What you can't forget: Exploiting parallelism for zoned namespaces," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, Jun. 2022, pp. 79–85.

[9] *ZENFS: Rocksdb storage backend for ZNS SSDS and SMR HDDS*. Accessed: Mar. 14, 2023. [Online]. Available: https://github.com/westerndigitalcorporation/zenfs

[10] S. Ghemawat and J. Dean. *A Fast Key-Value Storage Library Written at Google*. Accessed: Mar. 14, 2023. [Online]. Available: https://github.com/google/leveldb

[11] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and W. Lloyd, "Don't be a blockhead: Zoned namespaces make work on conventional SSDs obsolete," in *Proc. Workshop Hot Topics Operating Syst.*, Jun. 2021, pp. 144–151.

[12] K. Han, H. Gwak, D. Shin, and J. Hwang, "ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *Proc. USENIX OSDI*, 2021, pp. 147–162.

[13] *Ultrastar DC ZN540 Data Sheet*, Western Digit., USA.

[14] H.-R. Lee, C.-G. Lee, S. Lee, and Y. Kim, "Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, Jun. 2022, pp. 93–99.

[15] J. Jung and D. Shin, "Lifetime-leveling LSM-tree compaction for ZNS SSD," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, Jun. 2022, pp. 100–105.

[16] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proc. 6th ACM IEEE Int. Conf. Embedded Softw.*, Oct. 2006, pp. 161–170.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 2010, pp. 143–154.

[18] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. USENIX ATC*, 2020, pp. 17–31.

[19] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui, "Differentiated key-value storage management for balanced I/O performance," in *Proc. USENIX ATC*, 2021, pp. 673–687.

[20] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 497–514.

[21] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. C. Arpaci-Dusseau, and H. R. Arpaci-Dusseau, "From WiscKey to bourbon: A learned index for log-structured merge trees," in *Proc. USENIX OSDI*, 2020, pp. 155–171.

[22] K. Lu, N. Zhao, J. Wan, C. Fei, W. Zhao, and T. Deng, "TridentKV: A read-optimized LSM-tree based KV store via adaptive indexing and space-efficient partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1953–1966, Aug. 2022, doi: 10.1109/TPDS.2021.3118599.

[23] W. Zhong, C. Chen, X. Wu, and S. Jiang, "REMIX: Efficient range query for LSM-trees," in *Proc. USENIX FAST*, 2021, pp. 51–64.

[24] J. Zhang, S. Wu, Z. Tan, G. Chen, Z. Cheng, W. Cao, Y. Gao, and X. Feng, "S3: A scalable in-memory skip-list index for key-value store," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2183–2194, Aug. 2019.

[25] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: Fast architecture sensitive tree search on modern CPUs and GPUs," in *Proc. ACM SIGMOD*, New York, NY, USA, 2010, pp. 339–350, doi: 10.1145/1807167.1807206.

[26] M. Kwon, S. Lee, H. Choi, J. Hwang, and M. Jung, "Vigil-KV: Hardware-software co-design to integrate strong latency determinism into log-structured merge key-value stores," in *Proc. USENIX ATC*, 2022, pp. 755–772.

[27] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage," in *Proc. USENIX FAST*, 2021, pp. 17–32.

[28] C. Li, H. Chen, C. Ruan, X. Ma, and Y. Xu, "Leveraging NVMe SSDs for building a fast, cost-effective, LSM-tree-based KV store," *ACM Trans. Storage*, vol. 17, no. 4, pp. 1–29, Oct. 2021, doi: 10.1145/3480963.

[29] C. Ding, T. Yao, H. Jiang, Q. Cui, L. Tang, Y. Zhang, J. Wan, and Z. Tan, "TriangleKV: Reducing write stalls and write amplification in LSM-tree based KV stores with triangle container in NVM," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4339–4352, Dec. 2022.

[30] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for log-structured key-value store on persistent memory," in *Proc. USENIX ATC*, 2022, pp. 773–788.

[31] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. R. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. USENIX FAST*, 2019, pp. 191–205.

[32] W. Kim, C. Park, D. Kim, H. Park, Y. R. Choi, A. Sussman, and B. Nam, "ListDB: Union of write-ahead logs and persistent SkipLists for incremental checkpointing on persistent memory," in *Proc. USENIX OSDI*, 2022, pp. 161–177.

[33] M. Im, K. Kang, and H. Yeom, "Accelerating RocksDB for small-zone ZNS SSDs by parallel I/O mechanism," in *Proc. 23rd Int. Middleware Conf. Ind. Track*, Nov. 2022, pp. 15–21.

[34] R. Pitchumani and Y.-S. Kee, "Hybrid data reliability for emerging key-value storage devices," in *Proc. USENIX FAST*, 2020, pp. 309–322.

[35] Y. Kang, R. Pitchumani, P. Mishra, Y.-S. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proc. 12th ACM Int. Conf. Syst. Storage*, May 2019, pp. 144–154.

[36] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 373–384.

[37] J. Im, J. Bae, C. Chung, and S. Lee, "PinK: High-speed in-storage key-value store with bounded tails," in *Proc. USENIX ATC*, 2020, pp. 173–187.

[38] *Key Value Storage API Specification*. Accessed: Mar. 14, 2023. [Online]. Available: https://www.snia.org/tech_activities/standards/curr_standards/kvsapi

[39] M. Chun, J. Lee, S. Lee, M. Kim, and J. Kim, "PiF: In-flash acceleration for data-intensive applications," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, Jun. 2022, pp. 106–112.

[40] S. Lee, Y. Kim, D. Lee, I. Choi, and J. Kim, "Alohomora: Protecting files from ransomware attacks using fine-grained I/O whitelisting," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, Jun. 2022, pp. 113–118.

[41] J. Koo, J. Im, J. Song, J. Park, E. Lee, B. S. Kim, and S. Lee, "Modernizing file system through in-storage indexing," in *Proc. USENIX OSDI*, A. D. Brown and J. R. Lorch, Eds., 2021, pp. 75–92.

[42] M. Bjørling, J. González, and P. Bonnet, "LightNVM: The Linux open-channel SSD subsystem," in *Proc. USENIX FAST*, 2017, pp. 359–373.

[43] J. Zhang, Y. Lu, J. Shu, and X. Qin, "FlashKV: Accelerating KV performance with open-channel SSDs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–19, Oct. 2017.

[44] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014, pp. 1–14.

[45] I. L. Picoli, N. Hedam, P. Bonnet, and P. Tözün, "Open-channel SSD (what is it good for)," in *Proc. CIDR*, 2020.

[46] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined flash for web-scale internet storage systems," in *Proc. ACM ASPLOS*, 2014, pp. 471–484.

[47] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, "Fully automatic stream management for multi-streamed SSDs using program contexts," in *Proc. USENIX FAST*, 2019, pp. 295–308.

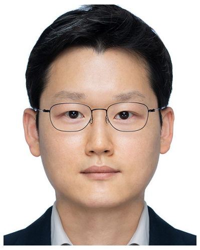
[48] K. Huang, Z. Shen, Z. Jia, Z. Shao, and F. Chen, "Removing double-logging with passive data persistence in LSM-tree based relational databases," in *Proc. USENIX FAST*, 2022, pp. 101–116.

[49] S. Bergman, N. Cassel, M. Bjørling, and M. Silberstein, "ZNSwap: Unblock your swap," in *Proc. USENIX ATC*, 2022, pp. 1–18.

**MYOUNGHOON OH** received the B.S. and M.S. degrees in computer science and engineering from Dankook University, South Korea, in 2015 and 2016, respectively, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include operating systems, file systems, flash memory, and embedded systems.

**SEEHWAN YOO** (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Korea University, Seoul, Republic of Korea, in 2002, 2004, and 2013, respectively. From 2013 to 2014, he was with the Software Platform Laboratory, LG Electronics, Seoul. He has been with Dankook University, South Korea, since 2014, where he is currently an Associate Professor with the Department of Mobile Systems Engineering. He has published academic research papers in journals and conferences in the area of cloud computing and operating systems. Also, he has been actively participating in the Open-Source Community, including LinuxCon, and Linux Security Summit. He was a recipient of the Best Paper Award and Best Presented Paper Award by KIISE in 2015, 2018, and 2021.

**JONGMOO CHOI** (Member, IEEE) received the B.S. degree in oceanography and the M.S. and Ph.D. degrees in computer engineering from Seoul National University, South Korea, in 1993, 1995, and 2001, respectively. He is currently a Professor with the Department of Software, Dankook University, South Korea. Previously, he was a Senior Engineer with Ubiquix Company, Seoul, South Korea, from 2001 to 2003. He held a visiting faculty position with the University of California at Santa Cruz, from 2005 to 2006, and Carnegie Mellon University, from 2014 to 2015. His research interests include operating systems, file systems, key-value store, flash memory, and non-volatile memory.



**JEONGSU PARK** received the B.S. degree in electrical engineering, the M.S. degree in computer engineering, and the Ph.D. degree from Seoul National University, South Korea, in 2008, 2010, and 2014, respectively. He was the Manager of SK Telecom, from 2014 to 2017. He joined SK Hynix, in 2017, where he currently leads ZNS FW development.

**CHANG-EUN CHOI** received the B.S. degree in computer science from Chung-Ang University, South Korea, in 1997. He is currently a corporate VP and an in charge of the Firmware Group, NAND Solution Division, SK Hynix. His journey to the current position started from Samsung, in 2005, by developing the world first eMMC firmware, and had been directly contributing to the top NAND M/S and technical leadership of Samsung across the market, until 2017. He also has been technically engaging with many NAND flash customers and ecosystem partners worldwide. His team in SK Hynix is currently focusing on the firmware development for UFS, client SSD, and enterprise SSD.