WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD

Jongsung Lee
Seoul National University and
Samsung Electronics
Korea
leitia@snu.ac.kr

Donguk Kim
Seoul National University and
Samsung Electronics
Korea
dongukim12@snu.ac.kr

Jae W. Lee
Seoul National University
Korea
jaewlee@snu.ac.kr

ABSTRACT

We propose WALTZ, an LSM tree-based key-value store on the emerging Zoned Namespace (ZNS) SSD. The key contribution of WALTZ is to leverage the *zone append* command, which is a recent addition to ZNS SSD specifications, to provide tight tail latency. The long tail latency problem caused by the merging process of multiple parallel writes, called *batch-group writes*, is effectively addressed by the internal synchronization mechanism of ZNS SSD. To provide fast failover when the active zone becomes full for a write-ahead log (WAL) file during parallel append, WALTZ introduces a mechanism for WAL zone replacement and reservation. Finally, lazy metadata management allows a put query to be processed fast without requiring any other synchronizations to enable lock-free execution of individual append commands. For evaluation we use both microbenchmarks (db_bench) with varying read/write ratios and key skewnesses, and realistic social-graph workloads (MixGraph from Facebook). Our evaluation demonstrates geomean reduction of tail latency by 2.19× and 2.45× for db_bench and MixGraph, respectively, with a maximum reduction of 3.02× and 4.73×. As a side effect of eliminating the overhead of batch-group writes, WALTZ also improves the query throughput (QPS) by up to 11.7%.

PVLDB Reference Format:
Jongsung Lee, Donguk Kim, and Jae W. Lee. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. PVLDB, 16(11): 2884-2896, 2023.
doi:10.14778/3611479.3611495

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SNU-ARC/WALTZ.git.

1 INTRODUCTION

The key-value store is a widely-used storage engine for data management. LSM-tree is the de-facto standard data structure that manages key-value pairs. It has been used as a basis for popular key-value stores, such as RocksDB [21], LevelDB [22], BigTable [10], Cassandra [1], and HBase [2]. LSM-tree stores key-value pairs that are received via put requests to an in-memory structure called *MemTable* for fast write performance. When MemTable reaches a specific size, it is sorted, merged, and stored in storage as a Sorted String Table (SST) file (*flush*). SST files are arranged in a multi-level structure to provide good read performance. When the space limit for each level is reached, a victim SST file is selected and moved to the lower level by merging with the SST files of the lower level (*compaction*). Both flush and compaction operations are performed in the background. To overcome the slow storage speed, SST files are written sequentially in bulk, which is favored by storage devices.

MemTable constructed with volatile memory cannot retain data in the event of a power failure. To address this consistency issue, a *write-ahead log (WAL)* is maintained in the persistent storage. The record of a put (write) request is first stored in WAL and then buffered in memory. WAL enables the key-value store to recover its data even after a failure. However, with this structure, a stream of put requests generates frequent small writes to the storage, hence reducing write throughput to introduce a scalability bottleneck. In a multi-threaded environment, multiple workers can be blocked by simultaneous WAL record writes to storage media, which causes write latency spikes due to lock contentions.

A popular solution to this problem is merging the small writes and writing them sequentially in bulk [1, 21, 22]. In particular, RocksDB introduces a process called *batch-group writes* [21], which dynamically selects one leader thread among multiple workers with pending put requests, collects all the remaining records, and make the leader write them at once on behalf of the other workers. However, this technique does not improve (or even exacerbates) the tail latency problem. In the process of collecting and writing these records, all records in the batch share the writing time to potentially increase the waiting time of an individual write, especially when the record size increases.

Recently, zoned namespace (ZNS) SSD [7] has been introduced. The ZNS SSD divides the storage space into *zones*, where only sequential writes are allowed within a zone. This greatly simplifies the flash translation layer (FTL), which is required to satisfy the erase-before-write constraint of NAND flash media. A zoned structure is a proven concept in the context of SMR HDDs [23] and already in production use [26]. As the zoned structure alleviates the management overhead of NAND flash media, it reduces the garbage collection overhead in the SSD.

Researchers have recently identified the potential suitability of ZNS SSD as a backing store for the LSM tree generating a stream of sequential writes. For example, ZenFS [7] proposes to use the ZNS SSD as a storage media of RocksDB. ZenFS introduces a lightweight file structure for ZNS SSD. All the files created by the background jobs are stored in a list of *extents*, each of which maintains the zone index, start location, and length. In addition, ZenFS supports a

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611495

recovery mechanism for the internal metadata, such as the status of zones and files, by persisting them into a specific area called meta zone. However, ZenFS provides only the plugin of the zoned interface, so the aforementioned batch-group write problem remains. Existing solutions to address this problem on non-ZNS devices, such as lock-free allocation of LBAs for parallel writes in SpanDB [12], are difficult to apply due to the constraints from the ZNS interface. This would require additional serialization to ensure the consistency of the write sequence.

To provide tight tail latency on ZNS SSD-backed LSM trees, we identify opportunities for leveraging the *zone append* command. This command is a recent addition to the ZNS SSD specifications [19]. The conventional NVMe interface does not guarantee write ordering among the requests in the submission queue, and they may not be processed in the pre-specified LBA order. To serialize the writes, it is difficult to increase the queue depth greater than one, which has negative impact on write throughput. The append command provides an efficient mechanism to increase the queue depth while maintaining the write order. Unlike the conventional write command, the append command specifies the data and a designated zone (but not a specific LBA), stores it in the zone, and returns the start location of the appended data to the host through a field called *Assigned LBA (ALBA)* [5, 6]. Since the exact LBA location is determined by the device, even if we increase the queue depth, the synchronization is done efficiently within the ZNS SSD, lifting the restrictions for the queue depth on the host side. This can greatly increase the write throughput, alleviate the scalability issue, and hence improve the tail latency.

To capitalize on these opportunities, we propose WALTZ, a novel key-value store that improves write-ahead log using the zone append for LSM trees on the ZNS SSD. We first augment RocksDB on ZenFS [7] with Intel SPDK [14], a lightweight user-level NVMe driver, which serves as the baseline. We evaluate WALTZ using both db_bench microbenchmarks [20, 21] with varying read/write ratios and key skewnesses, and Facebook MixGraph benchmarks [9], which represent realistic production workloads. WALTZ reduces the tail latency by up to 3.02× for db_bench and by up to 4.73× for MixGraph. As a side effect of eliminating the overhead of batch-group writes on the host, WALTZ also improves the query throughput (QPS) by 5.9% on average and 11.7% at maximum.

Our contributions are summarized as follows:

- We identify the synchronization overhead of batched-group writes as the primary cause of loose tail latency in a key-value store as the number of worker threads increases.
- We are the first to propose leveraging the newly introduced append command in the ZNS SSD specifications for the WAL record to reduce the tail latency.
- We also introduce techniques for zone replacement, reservation, and lazy metadata management to process parallel appends efficiently.
- We prototype WALTZ on RocksDB using real ZNS SSD devices as a backing store and evaluate it with both db_bench microbenchmarks and Facebook MixGraph benchmarks. Evaluation demonstrates the effectiveness of WALTZ for greatly reducing the tail latency, by up to 4.73×, while slightly improving the query throughput.

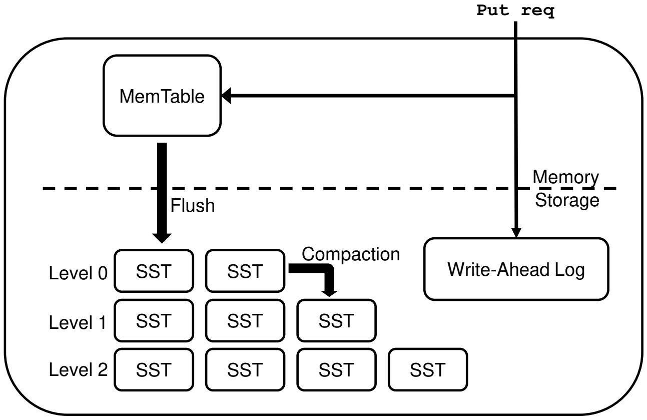


Figure 1: LSM-tree architecture

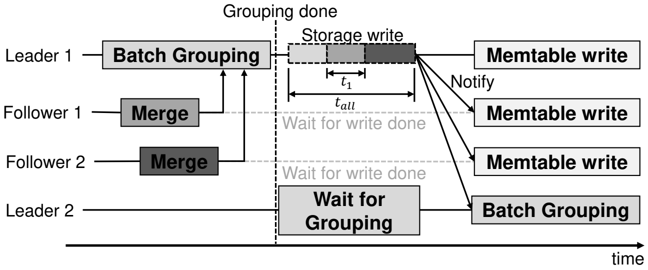


Figure 2: Batch-group write process of RocksDB

2 BACKGROUND

2.1 Log-Structured Merge (LSM) Tree

Overall Structure. The LSM tree is at the core of a key-value store and widely used in practice. Figure 1 illustrates the overall structure of the LSM tree. It consists of a MemTable, an in-memory data structure, and sorted string table (SST) files stored in persistent storage. The MemTable buffers incoming requests until it reaches its designated size, at which point it is marked as immutable and materialized into an SST file stored in persistent storage. SST files are managed in a multi-level structure, with all SST files initially stored at Level 0 (highest) and subsequently propagated to lower levels. SST files in all levels except Level 0 are managed in a sorted and disjoint manner so that the keys from the SST files in the same level do not overlap to optimize the read path.

Background Operations. To provide fast write speed, the *mutable* MemTable temporarily stores all key-value pairs received through put queries. If the size of this MemTable exceeds a certain threshold, it is marked as *immutable*, and later flushed into a sorted-string table (SST) file in persistent storage. At a flush operation, duplicate keys are removed from the MemTable, and the remaining key-value pairs are sorted using the specified comparator. These sorted key-value pairs are concatenated with SST file metadata such as Bloom filter and index block and then stored to Level 0. When the size of each level reaches a certain threshold, a compaction operation is triggered. It selects a victim SST file at the corresponding level

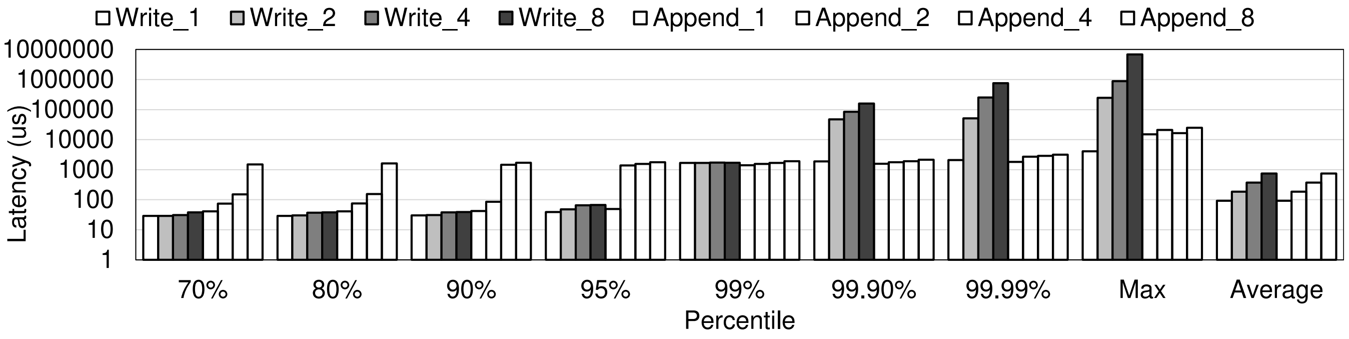


Figure 3: Write vs. Append

and then picks the files from the next lower level that contain overlapping keys to the victim file. The reason why compaction picks the files with the overlapping keys is to maintain the aforementioned property of managing the keys at each level in a sorted and disjoint manner. After reading all the input files, the compaction operation sorts them and materializes them at the next level. In this process, the files selected as input are deleted. Both flush and compaction operations trigger bulk sequential writes at the granularity of files. In particular, compaction merges and rewrites files instead of overwriting the existing file so that the storage receives only sequential write requests, thereby maximizing the performance of the background operations.

Write-Ahead Log and Batch-Group Write. Since put queries are stored at MemTable upon arrival, which is an in-memory data structure, all written records would be lost if there is a sudden power failure. To recover the data from such failures, LSM tree also utilizes a write-ahead log (WAL) in persistent storage. As illustrated in the Figure 1, all writes are stored in the WAL and then buffered in MemTable, which degrades put latency due to the slow speed of storage write. To alleviate this problem, RocksDB, one of the most popular LSM tree-based key-value stores, introduces the *batch-group write* mechanism. RocksDB organizes put query internally in the form of WriteBatch and then writes it to the WAL. When the batches are received simultaneously from multiple threads, RocksDB attempts to merge them into a *batch-group* during the batch-group write process. Figure 2 illustrates the example flow of a batch-group write process. When a write occurs, all writer threads compete to become the leader thread (leader competition stage). The winner thread becomes the leader and collects all the batches from the other follower threads. Then the leader makes the merged record persist on behalf of the other followers and notifies them of the availability of the record at MemTable.

2.2 Zoned Namespace (ZNS) SSD

Overview. The conventional SSD provides a block interface with 512-byte sector-sized in-place updates. However, since NAND flash memory does not allow overwrite, the flash translation layer (FTL) [24] should manage the internal mapping to emulate the operation of in-place updates. FTL maintains logical-to-physical address mappings and performs a background job called *garbage collection*, which collects live data and erases invalidated data generated by out-of-place updates. These background jobs become the major source of unpredictable performance to users [11, 30, 31] and extreme tail latencies [17, 25]. To eliminate this overhead of the background jobs in FTL, efforts have been made to overcome the limitations of the block interface, such as open channel SSD (OC-SSD) [8] and multi-stream SSD [30]. Zoned namespace (ZNS) SSDs [16] are a most recent addition to the list. ZNS is included as part of the NVMe standard specifications [19], which aims to improve user experience by bridging the semantic gap between the block interface and NAND device-specific restrictions.

Characteristics of Zone. The entire storage space in the ZNS SSD is managed as a collection of logical units called *zones*. Like the conventional namespace (CNS) SSD, sector-sized random reads are allowed, but writes must always be performed sequentially within each zone. The size of the zone is typically from 96MB [3, 18, 27] to 1GB [7, 13, 40], which is vendor specific. Each zone is in one of four states at any given time: EMPTY, OPEN, CLOSE, or FULL. Writing is allowed only for those in the OPEN state. When a user tries to read, the zone's status is not affected. When the zone is explicitly opened for writing or writing is performed without an OPEN process, the zone enters the OPEN state. The device performs internal bookkeeping operations such as preparing write buffers for OPEN zones and setting write pointers. Due to resource limitations of the ZNS device, the number of zones that can be kept OPEN is limited (specified by the `open_zone_limit` parameter). Due to these limitations, it is impossible to use all zones in parallel, and the zones that have performed a sufficient number of writes must be explicitly closed to free internal resources for other zones.

The write pointer of each zone can be retrieved explicitly through a *zone report* command. However, different parallel contexts often manage their own copies of this pointer separately for higher performance by controlling the metadata at the user level and sending a *zone write* command with the cached write pointer. However, in the multi-threaded case, the cached write pointer can easily become invalid by a write from another thread. If the thread sends a write to an address that does not match the (real) write pointer within the ZNS device, it immediately fails. Therefore, for the thread to run the application on the ZNS SSD, all writes to the same zone must be strictly serialized [6]. This severely limits the parallelism of writes across multiple threads when accessing the same zone.

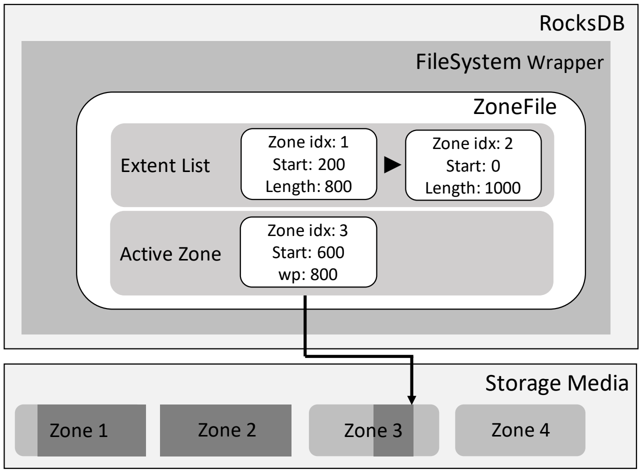


Figure 4: ZenFS file system structure

Zone Append. The zone append command [5, 6] has been recently introduced into the NVMe ZNS protocol [19]. The key idea is that, for the data whose physical location is not important, the user can delegate the task of synchronizing the write pointer to the ZNS SSD. The ZNS device is now responsible for synchronization of the write pointer for the requested zone internally, and it should return the actual location of the write called *Assigned LBA (ALBA)* as a response to the append command. Since the responsibility of synchronization is now moved from the worker to the ZNS SSD, all the worker threads can send the append command without any mutex or the like, to protect the write pointer and just send the zone indicator, buffer address, and size of the record.

Furthermore, the append command has advantages over the conventional write command in terms of fairness and quality-of-service. Figure 3 shows the latency analysis with varying number of threads writing to the same zone from 1 through 8. In the case of the write command, LBA must be pre-determined similarly to the block I/O interface. Since multiple requests cannot be sent to the single zone simultaneously, we utilize a synchronization mechanism. In this experiment, the average latency is similar for both commands, but their tail behaviors differ very much. The write command maintains low latency compared to the append command until around 95th percentile, but latency increased rapidly from a 99th percentile, and max latency increased by geomean 85× compared to the append command. This phenomenon is attributed to the unfairness of the lock mechanism. In contrast, the fairness among append commands is guaranteed inside the device, which leads to much more favorable tail behaviors with zone appends.

2.3 LSM Tree with ZNS SSD

ZenFS [7] is a plugin developed by Western Digital so that RocksDB can utilize the ZNS interface. ZenFS uses POSIX API-based `pread` and `pwrite` to perform I/O on the ZNS SSD and `ioctl` to perform management such as open, close, and reset of zones. As illustrated in Figure 4, ZenFS is implemented under the FileSystem wrapper APIs of RocksDB. To support the ZNS interface, ZenFS implements its own FileSystem class suitable for ZNS SSD instead of using the basic FileSystem of RocksDB. The ZenFS file system manages information such as metadata in the file system unit and each file's name, size, and data location. ZenFS utilizes direct I/O and uses the mq-deadline scheduler to prevent reordering at the kernel level.

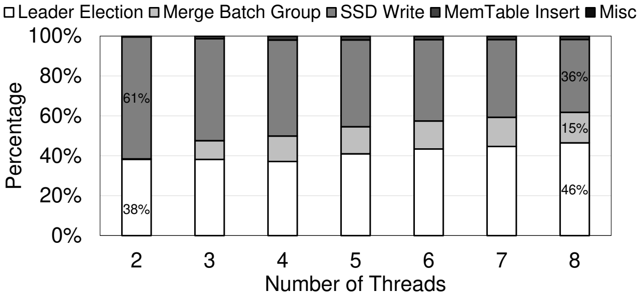


Figure 5: Scalability bottlenecks of batch-group writes

Figure 4 shows how ZenFS FileSystem places files into the zone structure and manages this information. Data continuously stored in a specific zone is expressed in an *extent unit*, which stores zone information, starting point, and length. For files stored across multiple zones, the extent information is managed as a list in an order within the file so that, when a read request at a specific point in the file arrives, the data at the exact location is delivered. In the example of Figure 4, the file read at offset 1200 will correspond to offset 400 of Zone 2 because the length of the first extent is only 800, and the remaining offset is calculated to 400.

ZenFS reduces the write amplification factor (WAF) by utilizing the features of RocksDB [21] when allocating zones. RocksDB provides a write lifetime hint (WLTH) based on the observation that the lifetimes of SST files in the same level are similar. ZenFS records the WLTH value of the data initially written at the time of zone allocation as the lifetime of the zone, and makes best effort to ensure that data with the same WLTH can be stored in the same zone. However, considering the space utilization and the limitations in open/active zones, ZenFS first allocates the zone from the opened zones whose lifetime is similar to the requested write and then searches for an empty zone only if it fails.

2.4 Overhead of Batch-Group Write

The batch-group write makes trade-offs between the overhead of synchronization and that of storage resource management. However, the synchronization overhead becomes more significant [12] as the storage resource management overhead is reduced with fast storage media such as NVMe. In addition, synchronization overhead further increases as the number of threads increases [38]. Finally, in the case of tail latency, all the put queries of the grouped batch should wait for the storage write phase, which includes the writing time of other queries. Figure 2 also illustrates a case when the tail latency gets worse. The write latency of Follower 1 is only t_1, but the leader groups three queries in this case so that all the group members, including Leader and Follower 1 and 2, should wait for t_{all} time to be done. If the record size is different across the threads, the unfairness of the put query time becomes even more severe.

We first analyze the performance overhead of batch-group writes on ZNS SSD. We vary the number of worker threads from 2 to 8 and measure the write latency breakdown by five steps. In this experiment, we set the key-value pair size for all threads to 4KB. *Leader election* is the process in which one of the worker threads is elected as a leader using the lock-free compare_exchange method. The elected leader is now responsible for all records to be written to the WAL file, and the *merge batch group* phase is gathering all the records from the follower threads and merging them into the batch-group. We poll the completion queue right after sending the ZNS write command and measure the *SSD write* time between the insertion at the submission queue and fetch from the completion queue. *MemTable insert* is the time elapsed from the MemTable insertion, and *Misc* is the remainder. As illustrated in Figure 5, the portion of leader election and merge batch group process increases as the number of worker threads increases. In the case of two worker threads, which would have the lowest degree of contention among the worker threads, about 38% of time is consumed at the leader election phase, which occurred from the batch-group write process. Also, the synchronization overhead becomes more severe when the number of worker threads increases, around 61% in the case of eight worker threads. This observation shows that the current batch-group write process limits the scalability of write performance.

3 WALTZ: DESIGN AND IMPLEMENTATION

3.1 Overview

WALTZ is an LSM-tree based key-value store using ZNS SSD as primary storage and utilizing append commands [5, 6] for writing the WAL. It builds on RocksDB as the baseline but bypasses its batch grouping process, which adversely affects tail latency due to the long leader election phase and write time. To protect the zone write pointer of the WAL file, we use the append command instead of a mutex or other synchronization mechanism.

Figure 6 shows the overall structure of WALTZ. It includes Replacement Checker and Zone Manager and two queues to communicate with each other. Replacement Checker determines when to replace the current active zone for the WAL file and notifies the worker thread handling the corresponding put query if the active zone should be replaced (i.e., close the current active zone and open a new one). To manage the replaced zone quickly, Replacement Checker uses a dummy append, which is much faster than a zone report command to find the exact write pointer of the current zone. Zone Manager is responsible for reserving new empty zones to supply if the worker thread requests a new zone for the WAL file much more efficiently than the baseline zone allocation mechanism of ZenFS. Also, Zone Manager takes over the zone finish and close operation for the zone being closed. These operations take a significant amount of time in the baseline implementation to account for a major fraction of the write tail latency of the worker thread.

3.2 Design of WALTZ

Write Path. The write path of WALTZ is simpler and lighter than the original RocksDB. RocksDB performs leader election by invoking `JoinBatchGroup()`, and the leader collects records from the followers and makes them persist by invoking `Enter/ExitAsBatchGroupLeader()`. However, WALTZ bypasses this part. Instead, all worker threads request to append their records to the WAL file. Once this is done, the worker thread immediately starts to update MemTable, without any concerns about the status of the other worker threads.

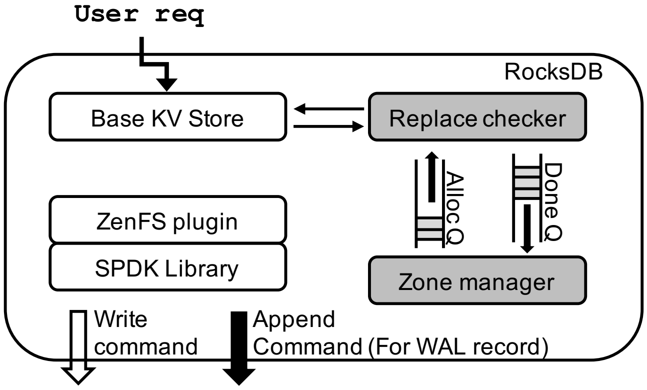


Figure 6: System overview of WALTZ

Replacement Checker. If a zone is FULL, a failure response will be raised to an append command. In this case, a retry is triggered, whose latency would increase the tail latency. Instead, we propose to prevent a retry from occurring. Immediately after performing an append, WALTZ calculates the remaining space based on the returned ALBA and proactively initiates a zone replacement if the remaining space in the active zone goes below a threshold (say, less than 1% of the zone space). If this threshold is set too high, the zone space is wasted so that the WAF value rises, but the possibility of increasing tail latency will be reduced.

We integrate a low-cost protection mechanism into WALTZ to prevent a situation where multiple worker threads attempt to perform zone replacement simultaneously. Compared to the baseline RocksDB in which the existing batch-group write always goes through a costly leader election phase, the protection overhead of WALTZ is negligible as this occurs only when the remaining space in the active zone is below the threshold. While zone replacement is in progress with the replacement thread, the other worker threads can continue to append their records to the remaining space of the current zone, hence minimizing blocking time. If the zone replacement task takes a long time to not complete until the current zone space is exhausted, a failure response by zone FULL is raised. In this case, the append is retried after the replacement task is finished.

Once a new zone is allocated, the extent information of the replaced zone must be stored in the extent list of ZoneFile. However, since other threads may have appended their records while the replacement thread was performing zone replacement, ALBA received by the replacement thread may not precisely match the last position of the data written in the actual extent. To address this, after assigning a new zone, the replacement thread throws dummy data to the replaced zone through the append command and then checks the status code and ALBA field of the completion entry. If the zone full status code is returned, it means that the last sector of the zone holds valid data of the current WAL file, so we use the zone's last LBA to calculate the extent's size. If the append command succeeds, since ALBA returned by the dummy append is the location where dummy data is stored, we can determine that the area up to the returned ALBA is valid data of the WAL file. The extent size is calculated based on the ALBA.

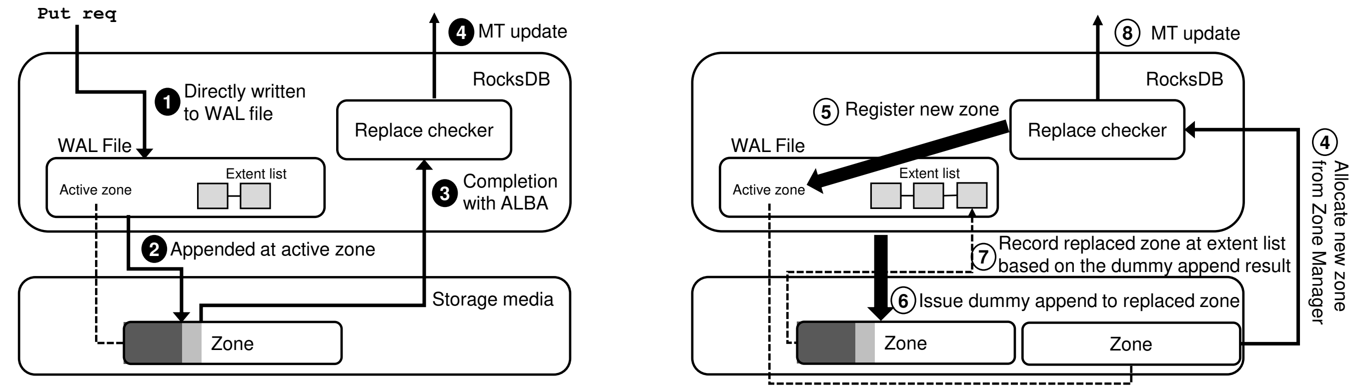
Dummy Append. The reason why a dummy append is preferred over a zone report command to retrieve the up-to-date zone write pointer is due to its lower overhead. The average latency of the zone report command of a single zone was measured as 6687μs, which is too long for the replacement thread to resolve while processing put queries by other threads. However, the zone append command showed a much faster speed, with an average of about 93μs in a single-thread environment. Since this process operates separately after the replacement thread registers the new zone as the active zone, the other worker threads now perform the WAL record append to the new active zone. Therefore, the replacement thread is almost always the only one accessing the replaced zone, making the dummy append faster. However, even assuming that other worker threads were still appending to the remaining space of the replaced zone, the worst performance of an append command (e.g., multi-threaded, tail-latency case) is still much faster than the zone report command (see Figure 3). Besides, since the background thread invalidates the remaining space with a zone finish command, the dummy append command only invalidates one sector in advance and does not incur additional overhead.

Zone Allocation with Zone Manager. The zone replacement process may also increase the write tail latency of the thread in charge of replacement (replacement thread). Let us first review issues with the existing work and then present our approach. In the case of ZenFS [7], when performing zone allocation, it grabs the I/O zone mutex and iterates through all zones while performing allocation considering the remaining space and lifetime of the zone. Zones whose remaining space below a designated threshold are finished while in the allocation phase. Then ZenFS attempts to allocate the zone if the WLTH of the first file stored in this zone is longer than the WLTH of the file requested for zone allocation. Therefore, zone allocation may be unexpectedly delayed when many zones need management, such as zone finish, or if the other background threads for compaction and flush operations try to allocate zone and hold the I/O zone mutex for creating new SST files. Besides, due to the implementation of full iteration over I/O zones, as the number of zones increases, the overhead of the iteration of the zone allocation loop increases, limiting scalability. And this problem is even more severe in the small-sized zone ZNS case, which provides features to enable more fine-grained control of a single device. Even on NAND Flash media with the same capacity, it exposes small-sized zones so that the number of zones is much more than the large-sized zone ZNS SSD, so loop iteration overhead increases several times, and tail latency is adversely affected.

To minimize this overhead, we add *Zone Manager* running on a background thread to reserve two zones to respond immediately to the zone replacement case of the WAL file. The main reason for reserving two zones is that during the MemTable switch process, a situation occurs where two WAL files are temporarily available and simultaneously written. One primary role of Zone Manager is reserving the WAL zone mentioned above. In addition, Zone Manager is also responsible for finishing and closing the replaced zone. Zones replaced from the WAL file are either OPEN or FULL state, depending on the presence or absence of remaining capacity. When entering the FULL state, the internal resources of the ZNS SSD have already been released, and the `open_zone_limit` is not occupied. However, if there is remaining space and the zone is in the OPEN state, `open_zone_limit` is occupied. Therefore, the ZNS SSD cannot be fully utilized if there is no management of resource starvation. Therefore, we perform not only WAL zone allocation but also status checks for replaced zone and process the finish and close zone in the background. Based on this, the replacement thread receives a zone through Zone Manager while processing a put query. After registering it into the active zone of ZoneFile, the replacement thread passes the replaced zone into done_queue. It can respond quickly without concerns about the post-processing of the replaced zone, which greatly reduces the tail latency. Zone Manager performs the remaining process for the replaced zone, which monitors the done_queue and handles it in the background.

Lazy ZoneFile Metadata Management. There are several file metadata that are managed in ZoneFile of ZenFS, such as active zone, extent list, file size, write pointer, capacity, etc. Since the active zone is where other worker threads try to append the WAL records, active zone replacement should be executed immediately. Also, the order of extents stored in extent list is important because it determines the position of data stored in a ZoneFile, which makes lazy update harder. Other metadata elements, such as file size, write pointer of active zone, and capacity, are not critical in the write path. Under the write case of ZenFS, synchronization is guaranteed for each ZoneFile, so another protection for these elements is not required. However, in our parallel append architecture, ZoneFile can concurrently be appended by worker threads which request the put queries, so that additional protection is necessary. We apply lazy updates replacing the existing method of managing non-critical ZoneFile metadata at every append, to avoid the synchronization and support fast response to a put query. Instead, based on the ALBA returned from a dummy append, WALTZ calculates the file size, remaining capacity of zone, and the latest write pointer, and updates the ZoneFile metadata at once during zone replacement.

Recovery. The vanilla RocksDB assigns a sequence number to each key-value pair or write batch according to the database configuration in the write phase. We identify the overhead of the batch-group write process and propose a parallel append architecture to accelerate WAL write in concurrent put queries. However, even with this structure, the minimum unit of WAL append is still a write batch so it does not affect RocksDB's sequence number structure. ZenFS provides an interface to permanently sync files through the `Fsync()` function, which is invoked at every WAL record write. When `Fsync()` is called, ZenFS writes the update information to ZenMetaLog for future recovery. Thus, WAL records are primarily recovered by ZenFS, and RocksDB can recover write batches or key-value pairs from individual records. However, in WALTZ, because of the lazy metadata update architecture, the WAL file records metadata only when the active zone is replaced instead of updating the metadata every time WAL append is performed. Therefore, at the point of failure recovery, only the allocation of the last active zone and the write pointer information of the starting point of active zone remain, and the total size of the WAL data appended to the zone is not recorded. To address this, during a recovery phase, we use the dummy append approach to figure out the current write pointer of the active zone, and examine the returned write pointer as a last location of valid records stored in this WAL file. The reason


Put req
❶ Directly written to WAL file
❷ Appended at active zone
❸ Completion with ALBA
❹ MT update
RocksDB
Replace checker
WAL File
Active zone
Extent list
Storage media
Zone
⑧ MT update
RocksDB
⑤ Register new zone
Replace checker
WAL File
Active zone
Extent list
④ Allocate new zone from Zone Manager
⑦ Record replaced zone at extent list based on the dummy append result
⑥ Issue dummy append to replaced zone
Zone
Zone

(a) Example walk-through of **Put** (normal case)   (b) Example walk-through of **Put** (zone replace)

Figure 7: Example walk-through of Put request

for this is that when a ZoneFile is allocated an active zone in ZenFS, it provides separate protection to prevent another ZoneFile from taking the same zone. Therefore, all data from the starting point of the active zone is guaranteed to be originated from the recovered WAL file. As for impact on recovery time, our design should restore the write pointers of up to two WAL files if power loss occurs at the time of the MemTable switch. This is the worst-case scenario. Therefore, the recovery overhead added for WALTZ is up to two dummy append operations. The average latency of the append command is measured to be about 93μs. Since host write is disabled at the time of recovery, there is no resource contention. Therefore, the recovery time increase of WALTZ is about 200μs at most, which is negligible.

Implementation. We implement WALTZ on RocksDB v6.25.3 with the ZenFS v1.0.2 plugin for ZNS support. ZenFS uses the libzbd [15] library to manage the ZNS SSD, but this library does not support the append command. We port Intel Storage Performance Development Kit (SPDK) [14] v22.01.2 to the ZenFS to support the append command. WALTZ takes 1600 lines of code (LOC) in total—some 700 LOC to attach Intel SPDK to ZenFS and some 900 LOC for the rest of the implementation.

3.3 Example Walk-through of Put Query

Figure 7(a) shows the example walk-through of a put query. When a put request arrives, it is first written to the WAL file ❶. We find the active zone registered in the WAL file and throw an append command to the zone ❷. After the append is performed, we retrieve the ALBA from the completion entry and deliver it to the Replacement Checker ❸. The Replacement Checker calculates the remaining space of the active zone based on ALBA, and if enough space remains, it goes directly to the MemTable insert phase ❹.

Figure 7(b) illustrates the zone replacement case. The replacement procedure is triggered if the Replacement Checker detects a failure of the append command or the remaining space calculated from ALBA is less than the designated threshold. First, we retrieve a new zone from Zone Manager ④, and the assigned zone is registered as a new active zone for the WAL file ⑤. And then, we throw the dummy append ⑥ to check the valid area of the replaced zone. If the dummy append fails, the extent size is calculated based on the end LBA of the replaced zone, and if it is successful, the extent size is calculated based on the ALBA returned from the dummy append ⑦. Finally, the worker thread can proceed to the MemTable insert phase ⑧.

Table 1: System configurations.

CPU	Intel(R) Core(TM) i9-9900K CPU 8 cores @ 3.6GHz
Memory	Samsung DDR4 64GB
Storage	PM1731a Samsung NVMe ZNS SSD 4TB
OS	Ubuntu 18.04.6 LTS, Linux 5.16.11
RocksDB	Version 6.25.3
SPDK	Version 22.01.2 LTS

Table 2: Percentile latency of microbenchmarks, 20M requests on 256GB database.

Uniform	Write only		Mixed 7:3		Mixed 3:7	
	ZenFS	WALTZ	ZenFS	WALTZ	ZenFS	WALTZ
50%	220.5	152.84	145.77	128.52	98.93	69.7
75%	312.71	239.15	203.16	169.54	157.99	120
Average	2547.84	2494.74	2408.66	2317.09	2237.42	2043.56

4 EVALUATION

We implement WALTZ upon the integrated system of Intel SPDK and ZenFS. Therefore, to fair comparison, we set the SPDK integrated ZenFS as our baseline.

System Configurations. We evaluate several experiments on a single-node workstation with a PM1731a Samsung NVMe ZNS SSD [18], which contains 40304 zones of 96MB size and a total size of 4TB. Detailed hardware and software configurations are described in Table 1. We use ZenFS, an open-source ZNS-based RocksDB plugin, as a baseline. The db_bench tool [20, 21] provided by RocksDB is used for our experiments. We configure the sync option as enabled for strict consistency of WAL. Also, we set the compression option as disabled and the number of background jobs as 4 for flush and 4 for compaction. The number of worker threads, which simultaneously issue the queries, was set to 4. For other options, the default option of RocksDB is used.

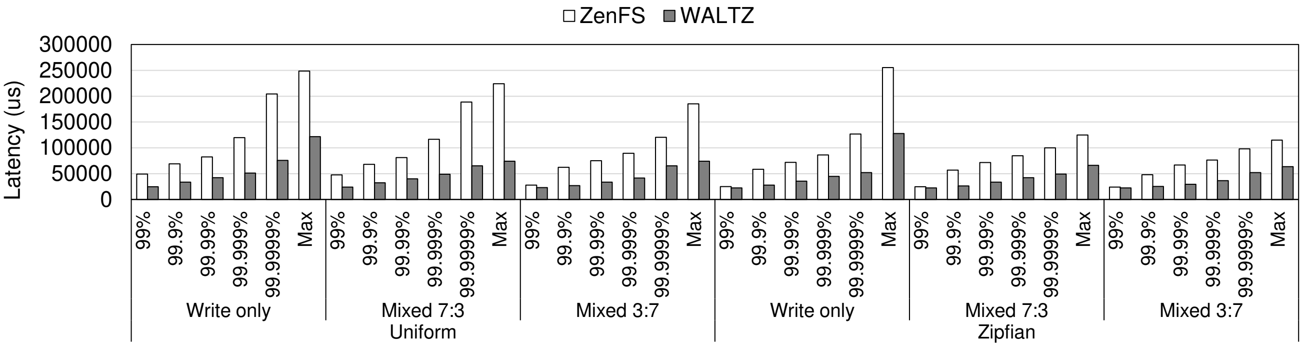


Figure 8: Percentile write latency of microbenchmarks, 20M requests on 256GB database. Lower is better.

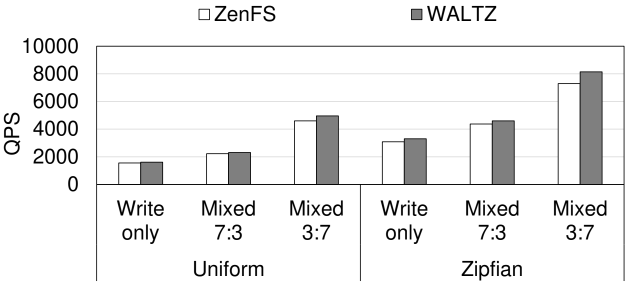


Figure 9: Throughput of microbenchmarks, 20M requests on 256GB database.

Workloads. We run the microbenchmarks and MixGraph [9] benchmark for our evaluation using the db_bench tool. In our experiments, we set the KV pair size as 4KB [42] and sequentially filled the keys in the range of 64M to construct a total DB of 256GB for the precondition. For the microbenchmarks, we conduct experiments while varying the write ratio to 100%, 70%, and 30% on the db_bench. Also, two types of key skewness are used: uniform random and Zipfian distribution with a constant of 0.99. Finally, we experimented with MixGraph, a synthetic workload that mimics Facebook's social graph workload with query composition and key access patterns. The options of four workloads are introduced in the paper [9]: All Random, All Dist, Prefix Random, and Prefix Dist (AllRand, AllDist, PreRand, PreDist in short).

4.1 Microbenchmarks

Latency. Table 2 and Figure 8 show the percentile write latency of the microbenchmark results. The overall latency distribution shows a similar pattern. It means that although the decrease in background jobs (flush, compaction), which is the effect of increasing key skewness, actually affects the improvement of latency, both the baseline and the WALTZ are equally affected. This shows that WALTZ efficiently reduces synchronization overhead regardless of key skewness.

As the ratio of write operations increases, the frequency of the flush and compaction also increases. In ZenFS, the increasing ratio of write operations causes conflicts between the put queries, such as delaying the composition of batch-group writes. Also, the post-processing of zones replaced from the active zone of the SST files written by flush and compaction is exposed to the zone allocation, which worsens the tail latency. WALTZ effectively hides such zone allocation overhead to prevent deterioration in tail latency. Still, as shown in Figure 3, synchronization overhead increases when multi-threads simultaneously request the append command at a single zone. Due to the characteristic of synchronization overhead, WALTZ is affected by the conflicts between commands, so the tail latency increases as the proportion of writes increases. However, the degree of deterioration is relatively less than that of ZenFS, and WALTZ achieves the max latency improvement by geomean 2.19× and up to 3.02×.

Throughput. As shown in Figure 9, QPS is also improved up to 7.8% in a uniform random pattern and up to 11.7% in a Zipfian pattern. The more reads are mixed, the higher the QPS because of the fast speed of read compared to write, and the improvement over baseline tended to be higher. As this result is an overall QPS analysis in a multi-thread environment, the sync overhead of about 35.6%, shown in the individual write latency breakdown at Figure 5, did not fully appear in the total improvement. Mainly because, as can be seen in Figure 3, the append command in the multi-thread environment also showed an increase in average latency due to internal synchronization overhead, which is similar to the average latency of the write command used for comparison. Considering that the write command is the result of including the synchronization overhead from the host, there was a limitation in the performance of the device to fully improve the sync overhead. Nevertheless, latency spikes were improved as batch-group writes were eliminated, and reads, which were limited by relatively slow writes and background jobs, were able to achieve high performance. As a result of the experiment, the average latency of read was also improved by up to 7.46% even though we only the modified the WAL write path.

4.2 MixGraph Benchmarks

Latency. Figure 10 plots the percentile latency of four MixGraph benchamrks. The baseline shows that tail latency varies greatly depending on the workloads, but WALTZ performs put queries with consistent tail latency. In ZenFS, it can be seen that the tail latency performance deteriorated considerably in PreRand and PreDist, which was caused by the concentration of keys in the hot

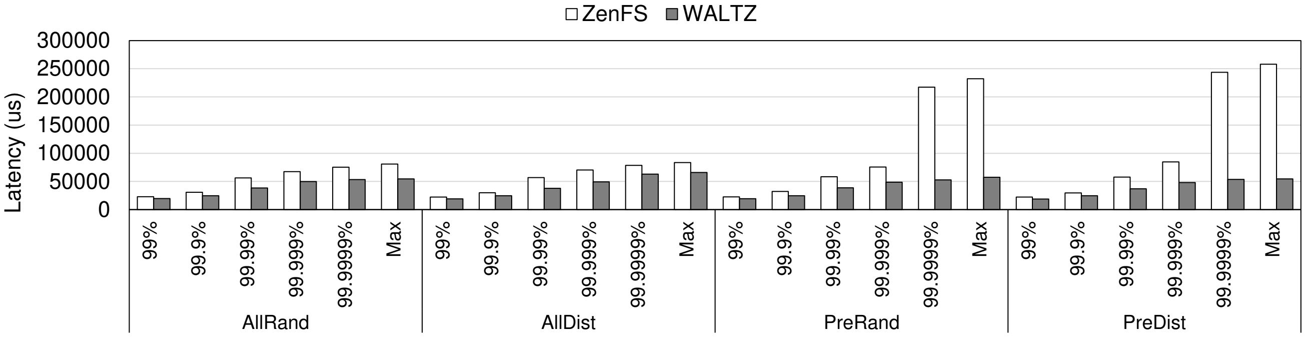


Figure 10: Percentile write latency of four MixGraph benchmarks with 20M requests on 256GB database. Lower is better.

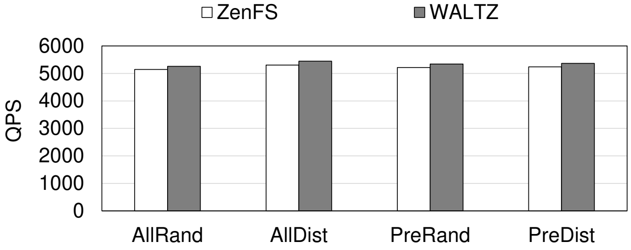


Figure 11: Throughput of four workloads in MixGraph benchmark, 20M requests on 256GB database.

Table 3: Average Latency of MixGraph Benchmarks

Average Latency	AllRand	AllDist	PreRand	PreDist
ZenFS	1304.55	1158.41	1218.45	1129.41
WALTZ	1195.98	1041.66	1109.30	1012.15
Improvement	8.32%	10.08%	8.96%	10.38%

region. Prefix workload has the property of maintaining a higher locality, which reduces the overhead caused by compaction. As the compaction overhead is reduced, the frequency at which multiple put queries are batched increases, and tail latency rather soars. As we remove the synchronization overhead, it limits the tail latency, and the tail latency improvement is achieved up to 4.73× and geomean 2.45×. As shown in Table 3, the average latency of WALTZ is also improved by up to 10.38% by reducing the extreme tail cases and several optimizations such as zone reservation.

Throughput. Figure 11 shows the experimental results for throughput. AllDist shows the highest QPS value among the four experiments for both ZenFS and WALTZ, and the performance improvement of WALTZ is also the highest with 2.70%. The rest of the experiments also show slightly improved performance compared to the baseline, and there is an improvement in 2.42% in geomean. The reason why the improvement seems relatively smaller than the microbenchmark is because of the scan operation. Although the portion of scan query is only 3%, the average latency of scan queries is measured at 14-15 ms, which is about 100 times higher than the latency of a single get query. Overall DB stress is increased due to these heavy-weight commands, and as a result, QPS improvement is relatively small compared to the microbenchmark, but all cases show an improvement compared to the baseline.

4.3 Sensitivity Study

Number of threads. We evaluate the throughput and latency of the MixGraph PreDist workload by increasing the number of threads from 2 to 8 to measure the effect of the number of threads. As shown in Figure 12(a), QPS for both WALTZ and ZenFS improves as the number of threads increases, and the difference widens up to 5.35%. In Figure 12(b), we observe an increasing trend in latency as the number of threads increases for both ZenFS and WALTZ. ZenFS exhibits latency increase due to the synchronization overhead associated with the batch-group write, while WALTZ, which has effectively mitigated synchronization overhead, faces latency increase due to the serialized processing of append commands by the ZNS SSD. Despite the increment in average latency for both systems, WALTZ consistently exhibits lower average latency than ZenFS, achieving a reduction of up to 14.41%. Regardless of the number of threads, the maximum latency of WALTZ shows a geomean 4.45× improvement compared to ZenFS and by up to 5.10×.

Value size. To quantify the influence of key-value pair size, we evaluated the performance of the MixGraph PreDist workload while changing the key-value pair size to 0.5KB, 1KB, and 2KB. To test the same amount as the 4KB experiment, in which 20M queries were processed for a 256GB database, the key range and query count were doubled by dividing the key-value pair size in half. As shown in Figure 12(d), the decrement of key-value pair size makes the QPS scales due to the overall decrease in traffic. Also, the average latency decreases due to the smaller record size, hence reducing the total amount of traffic within RocksDB. However, despite the decreased record size, WALTZ still outperforms ZenFS due to the existence of the batch-group write process, which can cause lock overhead.

4.4 Write Amplification Factor (WAF) Analysis

We calculate the WAF of the entire database by dividing the amount written to the device by the amount written by the host. Device writes can be broadly categorized into flush, compaction, and WAL writes. As WALTZ does not modify the LSM tree architecture, the amount of device writes generated by flush and compaction is comparable. The difference in WAF between ZenFS and WALTZ

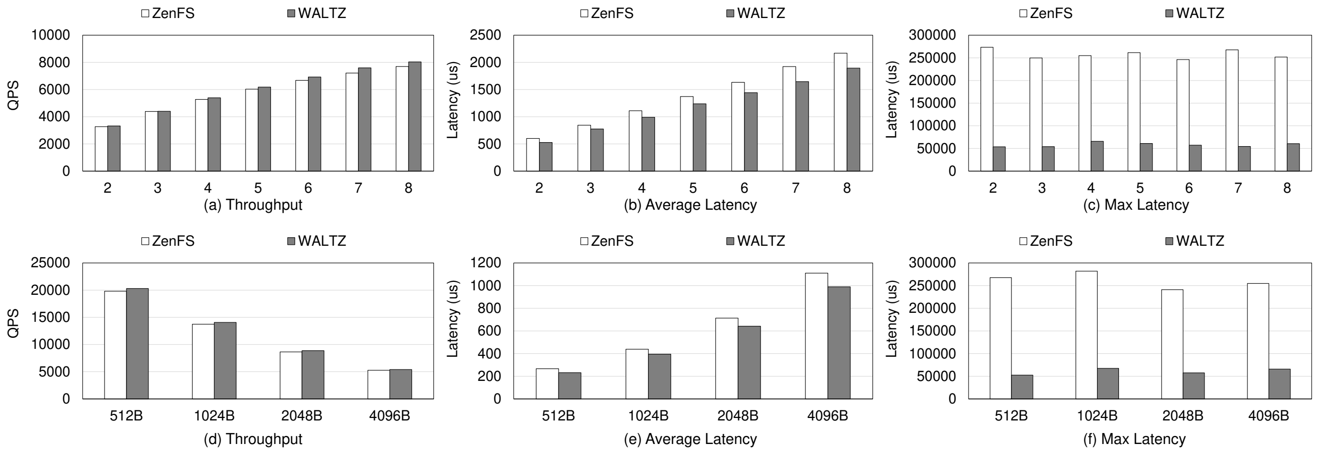


Figure 12: Sensitivity study while varying (a-c) thread counts and (d-f) value sizes for MixGraph PreDist workload

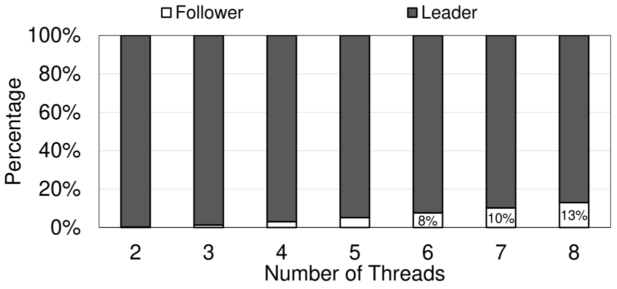


Figure 13: Ratio of group leader and follower in PreDist workload on the baseline ZenFS

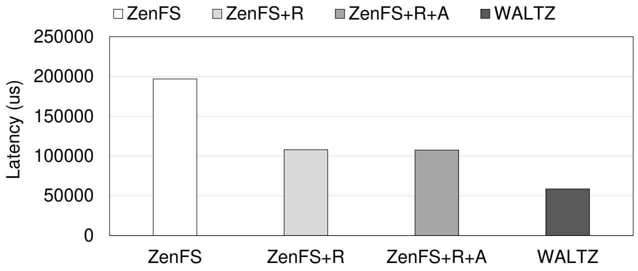


Figure 14: Max latency of uniform random write-only workload, 5M requests on 64GB database. Lower is better.

originates from the batch-group writes used by ZenFS. In batch-group writes, the leader thread merges the records of followers and writes them to the device at once. This can lead to lower WAF than if each thread wrote its own records independently. Figure 13 shows the ratio of leaders and followers in ZenFS as the number of threads is changed for the MixGraph PreDist workload to analyze the WAF difference caused by batch writes in ZenFS and WALTZ. WALTZ treats all writes as leaders. As the number of threads increases, there is a higher chance of overlapping write commands, so the write portion of follower threads increases, reaching up to 13% for 8 threads. This means that ZenFS can reduce WAL writes by up to 13%. However, the amount of compaction writes is 81.58 GB for WALTZ and 81.4 GB for ZenFS, while the amount of WAL writes is 10.7 GB for WALTZ and 9.3 GB for ZenFS, which indicates a smaller proportion. As a result, although WALTZ's WAF is slightly increased by about 1.7% compared to ZenFS, this does not have a significant impact compared to the serious tail latency issues and bandwidth improvement.

4.5 Ablation Study

We propose three methods: a zone reservation feature that performs zone allocation and replacement of WAL files in the background, an append feature that uses the append command instead of the write command to a WAL file, and parallel processing that removes batch-group writes. We incrementally enable these features and evaluate how each method affects the tail latency. Two more cases are evaluated with baseline and WALTZ. *ZenFS+R* is the ZenFS, in which Zone Manager is implemented and working for zone allocation and replacement. In the *ZenFS+R+A*, we change the recording function of the WAL file from the write command to the append command. Still, batch-group write is enabled in this case. We configure the 64GB DB and test 20GB of writes in a uniform pattern for evaluation, and the results are illustrated in Figure 14. ZenFS shows the worst tail latency with 196807us, and we can find that the reservation and replacement improve the tail latency with 107867us, which is almost two times improved than ZenFS. It also means that the current design, which iterates through all zones for zone allocation and zone post-process of replacement, worsens the tail latency.

The *ZenFS+R+A* shows the tail latency as 107396us, which is almost the similar result of *ZenFS+R*. The only difference between the two is the usage of the write command and append command. The batch-group write is enabled for both, so the write and append commands always work in single-threaded mode. Based on this, it can be seen that it is challenging to improve tail latency simply by changing the write command to the append command. Finally, WALTZ with all features applied showed a tail latency of 58712us.

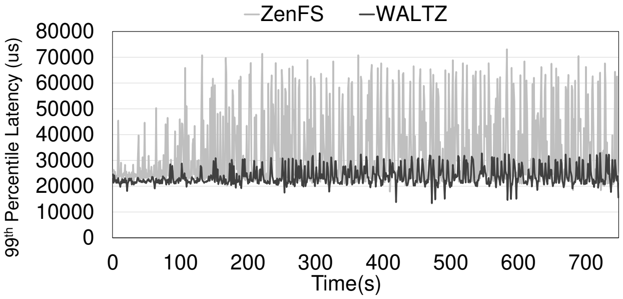


Figure 15: 99% latency over time in uniform, write only workload, 5M requests on 64GB database

This result is about 1.84× improved from *ZenFS+R+A*, showing how much the tail latency can be improved by skipping the batch-group write and exposing only the write time of a single record. Through these observations, although the zone append command has several advantages, its advantages cannot be maximized by simply replacing the write command with the append command. We improved both tail latency and QPS through Zone Manager and Replacement Checker and various optimizations in the WALTZ design.

4.6 99th Percentile Latency over Time

Figure 15 illustrates the 99th percentile latency of ZenFS and WALTZ by dividing the entire experiment time into 1-second units and measuring 99% tail latency within each unit. A 64GB database is configured for the measurement, and the latency of all put queries is measured while giving 5M writes in a uniform pattern. In the case of ZenFS, there are relatively few latency spikes from the 0s to the 100s, which is the first part of the experiment. The main reason is that a random write is started right after the DB is filled, so the background jobs, such as flush and compaction, not yet heavily occurs. This aspect is similar to WALTZ and shows a flat tail latency without any latency spike in the first 100s section. However, from the later section, ZenFS generates a 99th percentile latency of about 70 ms, and the frequency of the latency spikes gradually increases over time. As put queries are continuously applied to the DB, compaction frequently occurs across all levels, and the cascaded compaction caused by upper-level compaction repeatedly occurs. Hence, the increase of fragmented zones generated from the compaction operations makes the tail latency of the put query even worse in ZenFS because it allocates the zone during the put query processing. In the case of WALTZ, tail latency is no longer maintained flat as the background operation increases, but the degree of fluctuation is still much lower than that of the ZenFS. The reason is that, in the WALTZ structure, Zone Manager reserves the zone for the WAL file in the background so that the overhead of zone allocation is not revealed during the put query of the worker thread.

4.7 Effect of Batch-Group Writes

We further analyze how the batch-group write affects tail latency in the RocksDB. Figure 16 is a graph plotted as a cumulative distribution function of write latency classified by batch count. For this, we use the evaluation result of the baseline ZenFS while running the PreDist benchmark of MixGraph. When the batch count is 1, the write latency mainly spans the low latency section. It means that most writes end quickly if batch-group is not performed. However, as the number of batch-group increases, it can be seen that the proportion of write latency is biased towards the high latency section. This trend can also be seen in Table 4. We calculate the average write latency of each batch count, and the increment tendency is discovered while the number of grouped batches increases. In Figure 17, 99%, 99.9%, 99.99%, and max latency are plotted for in-depth

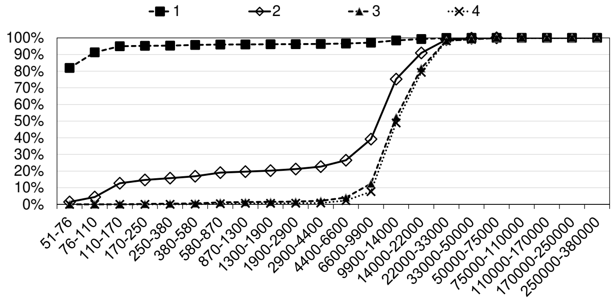


Figure 16: Cumulative distribution function of write latency over batch counts in PreDist workload of ZenFS

Table 4: Average write latency over batch counts

	Batch 1	Batch 2	Batch 3	Batch 4
Average write latency	625.51us	10834.81us	15785.25us	16388.01us

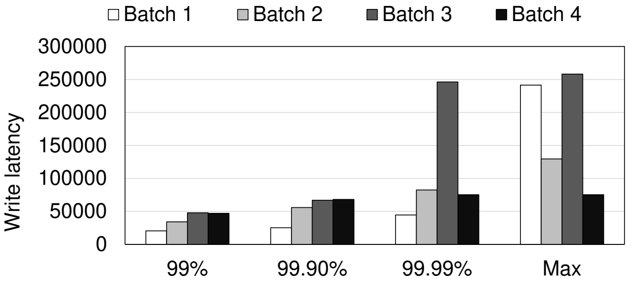


Figure 17: Tail write latency over batch counts in PreDist workload of ZenFS

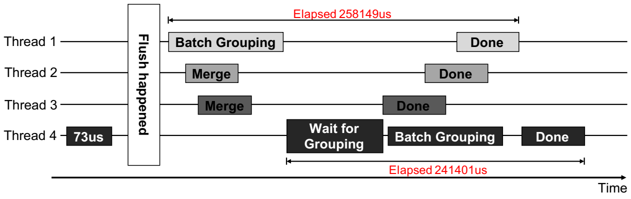


Figure 18: Timeline of tail latency case of batch 1 and 3 in PreDist workload of ZenFS

analysis of tail latency, respectively. In this figure, the latency of batch 3 soared sharply and that the maximum value for each batch count occurred in batch 1 and 3 cases.

The reason why the batch 1 case shows extreme tail latency can be seen again in Figure 2. Leader 2, which came in immediately after leader 1's batch grouping, waits for the entire storage write time before starting batch grouping, so it is fully affected by the latency spike of the preceding put queries. For this reason, the batch 4 case is a case in which all four threads participate in the group, corresponding to the case where there is no preceding batch-group in progress. The latency of the batch 4 case is only affected by the increase in storage writing time and synchronization overhead, without any interference between batch-group.

The detailed timeline for the max case of batch 1 and batch 3 can be found in Figure 18. Firstly, the put queries of Thread 4 is done within 73us, and then, the flush operation is triggered in the background. Immediately after the flush occurred, the put query of Thread 1 became the leader, and the put queries of Threads 2 and 3 are merged. The second put query in Thread 4 is waiting for the leader done without participating in the batch-group. Due to the effect of the background job, Thread 1's group write ended with a delay, and Thread 1 terminated the group write with a write latency of 258149us. After that, the write of thread 4 proceeded with storage write as a group containing only a single batch and ended with a latency of 241401us. This case is also a kind of overhead related to the grouping of the batch-group write, and WALTZ improves extreme tail latency by removing this overhead.

5 RELATED WORK

Zoned Namespace SSD with LSM Tree. Several studies [7, 28, 32, 36, 39, 40, 43] have been introduced to utilize the ZNS SSD. ZoneFS [32] proposed a file system for ZNS SSDs and merged it into the mainline of the Linux kernel. ZenFS [7] implemented the plugin of the RocksDB for ZNS SSD and provided it as open-source. ZoneFS and ZenFS are focused on analyzing the characteristics of ZNS SSDs and implementing a base library to utilize them in applications. They do not pay attention to the latency aspect of put queries, which is the main target of WALTZ.

Write Amplification in LSM Tree over ZNS SSDs. In addition, Lu et al. [37], CAZA [33], LL-compaction [29], and ZNSKV [41] focused on the write amplification factor (WAF) of ZNS SSD and proposed various methods to store the data with a similar lifetime at the same zone. Lu et al. [37] proposed a level-based zone assignment based on the observation that upper-level files have shorter lifetimes. This mechanism is similar to ZenFS, which uses RocksDB's write lifetime hint (WLTH) but provides a little more fine-grained control than WLTH, which ties everything after level 3 to WLTH_EXTREME. Based on the fact that files are simultaneously deleted at the time of compaction, CAZA [33] proposed allocating zones based on key range overlapping between adjacent levels instead of a level-based allocation. LL-compaction [29] proposed a method of splitting the SST based on the checkpoint of the upper level at the time of compaction, minimizing the short-lived SST of the lower level. Based on the SST splitting, LL-compaction attempted to reduce write amplification incurred by compaction. ZNSKV [41] also observed that the write amplification problem by the key range overlapping at the adjacent level is more critical in the ZNS SSD environment and introduced the adaptive weight calculation to reduce GC overhead. These attempts suggested a way to efficiently use ZNS SSD by reducing WAF but did not suggest a way to reduce the tail latency.

Conventional Namespace SSD with LSM Tree. There have been studies trying to accelerate key-value stores by utilizing the fast storage media, such as NVMe. KVell [34] uses the in-memory B+ tree to manage the index structure of KV pairs, to fully utilize the fast random read/write speed characteristic of NVMe SSD. SpanDB [12] and p2KVS [38] tried to solve the problem of batch-group write in various ways. SpanDB [12] proposes a parallel mechanism that allocates LBA first by lock-free fetch-and-add command and then store the WAL record in parallel by using the allocated LBAs. p2KVS [38] is configured to be applicable to various key-value databases by portablely applying the request batching mechanism. Through the batching technique of write and read requests, it is possible to achieve outstanding performance in workloads with many small KV items. Unfortunately, all of these methods are difficult to apply directly to ZNS SSD.

Improving Tail Latency of LSM Tree. There is a work focused on analyzing tail latency to improve user experience in key-value stores. SILK [4] focuses on the fact that the user requests are affected by compaction, causing latency spikes, and aims to improve tail latency by prioritizing compaction likely to affect tail latency. CruiseDB [35] applies a method to enhance tail latency by limiting user requests entering the memory buffer of the LSM tree. These existing methods attempt to improve the tail latency by controlling the write rate or compaction priority. However, they are different from WALTZ because they do not address the effect of batch-group write and targeting conventional namespace (CNS) SSDs.

6 CONCLUSION

We proposed WALTZ, a key-value store on ZNS SSD that improves the tail latency of LSM-tree using zone append commands. In this study, we first analyzed how the batch-group write affects the tail latency and eliminated the synchronization overhead of batch-group writes by leveraging the append command of ZNS SSD. The latency impact was minimized by utilizing zone reservation and lazy metadata management techniques in handling zone-full cases that occur during the parallel append process. In the experiment, we achieved maximum 3.02× tail latency improvement and geomean 2.19× improvement compared to ZenFS for db_bench microbenchmarks and up to 4.73× tail latency improvement for Facebook MixGraph benchmarks. We also demonstrate that QPS is improved by up to 11.7% due to write path optimization.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. We also thank Meongchul Song and Sangyeun Cho from Samsung Electronics for providing PM1731a ZNS SSDs for our evaluation. This work was supported in part by an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (2021-0-00853, Developing Software Platform for PIM Programming) funded by the Korea Government (MSIT). Jae W. Lee is the corresponding author.

REFERENCES

[1] Apache. [online]. Apache Cassandra. https://cassandra.apache.org. [Accessed 25-07-2023].

[2] Apache. [online]. Apache HBase. https://hbase.apache.org. [Accessed 25-07-2023].

[3] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. 2022. What you can't forget: exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 79–85.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[5] Matias Bjørling. 2019. From Open-Channel SSDs to Zoned Namespaces. In *Linux Storage and File systems Conference (Vault'19)*.

[6] Matias Bjørling. 2020. Zone Append: A New Way of Writing to Zoned Storage. In *Linux Storage and File systems Conference (Vault'20)*.

[7] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[8] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 359–374. https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[11] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives.. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, Vol. 11. USENIX Association, 77–90.

[12] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[13] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. 2020. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/choi

[14] Intel Corporation. [online]. Storage Performance Development Kit (SPDK). https://spdk.io/. [Accessed 25-07-2023].

[15] Western Digital Corporation. [online]. libzbd. https://github.com/westerndigitalcorporation/libzbd. [Accessed 25-07-2023].

[16] Western Digital Corporation. [online]. Zoned Storage. https://zonedstorage.io/. [Accessed 25-07-2023].

[17] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[18] Samsung Electronics. [online]. Samsung Introduces Its First ZNS SSD With Maximized User Capacity and Enhanced Lifespan. https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan. [Accessed 25-07-2023].

[19] NVM express workgroup. [online]. NVMe Zoned Namespaces (ZNS) Command Set Specification. https://nvmexpress.org/specification/nvme-zoned-namespaces-zns-command-set-specification/. [Accessed 25-07-2023].

[20] Facebook. [online]. Benchmarking tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. [Accessed 25-07-2023].

[21] Facebook. [online]. A persistent key-value store for fast storage environments. http://rocksdb.org. [Accessed 25-07-2023].

[22] Sanjay Ghemawat and Jeff Dean. [online]. LevelDB: A Fast Persistent Key-Value Store. https://github.com/google/leveldb. [Accessed 25-07-2023].

[23] Simon Greaves, Yasushi Kanai, and Hiroaki Muraoka. 2009. Shingled Recording for 2–3 Tbit/in^2. *IEEE Transactions on Magnetics* 45, 10 (2009), 3823–3829.

[24] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1508244.1508271

[25] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 263–276. https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao

[26] HGST. 2017. Libzbc Version 5.4. 1. (2017).

[27] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. 2022. Accelerating RocksDB for small-zone ZNS SSDs by parallel I/O mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*. 15–21.

[28] Peiquan Jin, Xiangyu Zhuang, Yongping Luo, and Mingchen Lu. 2021. Exploring index structures for zoned namespaces SSDs. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 5919–5922.

[29] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-leveling LSM-tree compaction for ZNS SSD. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 100–105.

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

[31] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 183–189. https://www.usenix.org/conference/fast15/technical-sessions/presentation/kim_jaeho

[32] Damien Le Moal and Ting Yao. 2020. zonefs: Mapping POSIX File System Interface to Raw Zoned Block Device Accesses. (2020).

[33] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 93–99.

[34] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[35] Junkai Liang and Yunpeng Chai. 2021. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1032–1043.

[36] Renping Liu, Zhenhua Tan, Yan Shen, Linbo Long, and Duo Liu. 2022. Fair-ZNS: Enhancing Fairness in ZNS SSDs through Self-balancing I/O Scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).

[37] Mingchen Lu, Chen Tang, and Peiquan Jin. 2022. Revisiting LSM-Tree-Based Key-Value Stores for ZNS SSDs. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 6772–6774.

[38] Ziyi Lu, Qiang Cao, Hong Jiang, Shucheng Wang, and Yuanyuan Dong. 2022. P2KVS: A Portable 2-Dimensional Parallelizing Framework to Improve Scalability of Key-Value Stores on SSDs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys 22)*. Association for Computing Machinery, New York, NY, USA, 575–591. https://doi.org/10.1145/3492321.3519567

[39] Yanqi Lv, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, Liming Fang, Yuanjin Lin, and Kuankuan Guo. 2022. ZonedStore: A Concurrent ZNS-Aware Cache System for Cloud Data Storage. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1322–1325.

[40] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. 2020. Exploring performance characteristics of ZNS SSDs: Observation and implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–5.

[41] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. 2022. ZNSKV: Reducing Data Migration in LSMT-Based KV Stores on ZNS SSDs. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 411–414.

[42] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[43] Yiwen Zhang, Ting Yao, Jiguang Wan, and Changsheng Xie. 2022. Building GC-free key-value store on HM-SMR drives with ZoneFS. *ACM Transactions on Storage (TOS)* 18, 3 (2022), 1–23.