ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends

Krijn Doekemeijer
Vrije Universiteit Amsterdam
The Netherlands

Zebin Ren
Vrije Universiteit Amsterdam
The Netherlands

Nick Tehrany*
BlueOne Business Software LLC
Beverly Hills, CA, USA

Animesh Trivedi
Vrije Universiteit Amsterdam
The Netherlands

Abstract

KV-stores are extensively used databases that require performance stability. Zoned Namespace (ZNS) is an emerging interface for flash storage devices that provides such stability. Due to their sequential write access patterns, LSM trees, ubiquitous data structures in KV stores, present a natural fit for the append-only ZNS interface. However, LSM-trees achieve limited write throughput on ZNS. This limitation is because the largest portion of LSM-tree writes are small writes for the write-ahead log (WAL) component of LSM-trees, and ZNS has limited performance for small write I/O. The ZNS-specific zone append operation presents a solution, enhancing the throughput of small sequential writes. Still, zone appends are challenging to utilize in WALs. The storage device is allowed to reorder the data of zone appends, which is not supported by WAL recovery. Therefore, we need to change the WAL design to support such reordering.

This paper introduces ZWALs, a new WAL design that uses zone appends to increase LSM-tree write throughput. They are resilient to reordering by adding identifiers to each append along with a novel recovery technique. We implement ZWALs in the state-of-the-art combination of RocksDB and ZenFS and report up to 8.56 times higher throughput on the YCSB benchmark. We open-source all our code at https://github.com/stonet-research/zwal.

CCS Concepts: • **Information systems** → **Storage management**; *Flash memory*; • **Software and its engineering** → *Secondary storage*.

*Work done while the author was at the Vrije Universiteit Amsterdam.

CHEOPS '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0538-0/24/04
https://doi.org/10.1145/3642963.3652203

Keywords: Write-ahead log, Key-Value store, ZNS SSDs

ACM Reference Format:
Krijn Doekemeijer, Zebin Ren, Nick Tehrany, and Animesh Trivedi. 2024. ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends. In *4th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3642963.3652203

1 Introduction

Log-structured merge-tree (LSM-tree) based KV-stores are extensively used databases, with workloads ranging from graph processing to machine learning [5, 7, 12]. KV-stores store application data as KV-pairs with the PUT operation. The average KV-pair size issued by applications is small (e.g., 1 KiB) [5], resulting in many small writes to the LSM-tree. This paper focuses on optimizing LSM-tree write throughput for small writes on ZNS, an emerging storage interface.

We visualize the LSM-tree PUT operation in Fig. 1. Large sequential writes achieve higher throughput than small writes, therefore, LSM-trees buffer KV-pair updates in memory and periodically flush data to storage. The LSM-tree first store KV-pairs inside volatile memory to a size-bounded component known as the memtable. When this memtable is sufficiently large, the LSM-tree *flushes* the memtable to a tree-like structure on storage. To ensure no data is lost on shutdown, the LSM-tree writes PUT operations to an on-storage log known as the *write-ahead log* (WAL). The WAL maintains all KV-pair changes over time. When the KV-store restarts, the LSM-tree recovers its state using a process known as WAL recovery. WAL recovery reads all WAL data sequentially and (re)applies it to the memtable. Data must be applied sequentially, as only the most recent change to a KV-pair is valid. The WAL is crucial for achieving high write throughput because each PUT writes to the WAL.

LSM-trees are typically deployed on fast and highly parallel NVMe flash SSDs. Flash storage performs better with sequential- than with random writes [17], precisely the access pattern of LSM-trees. However, with NVMe the SSD issues internal management operations that compete for storage resources with LSM-trees. This competition results in unstable throughput, which hinders achievable LSM-tree

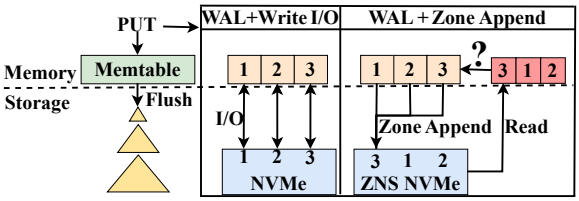


Figure 1. *LSM-tree* PUT *operation with* write *I/O or zone* append *for the WAL.*

throughput [3, 34]. Therefore, researchers and industry have proposed leveraging different interface(s) for flash SSDs [3, 4, 22]. One such interface is the recently standardized Zoned Namespaces (ZNS) interface [3]. ZNS presents the storage as sequential write-only regions known as *zones* and exposes the management operations to applications. ZNS delivers stable throughput by exposing these operations. Consequently, ZNS has led to several LSM-tree designs [21, 27, 31].

While ZNS achieves stable LSM-tree throughput, it leads to significant write throughput challenges for the WAL component of the LSM-tree. ZNS prohibits applications from issuing write I/Os concurrently to the same zone. ZNS prohibits this because (1) write I/Os need to be issued to sequential addresses of the zone (sequential write-only zones), and (2) SSDs are free to reorder I/O requests [30]. Consequently, PUTs to the WAL are serialized, limiting throughput to the WAL as only 1 PUT can be processed concurrently [28].

To address write I/O's limited throughput, ZNS has introduced an operation known as zone appends. Zone appends allow concurrent write operations to a zone, saturating device parallelism and significantly increasing small write throughput [2]. High concurrency makes it a good alternative to use for WALs [2, 28, 32]. Nevertheless, we can not interchange zone appends for write I/O's without modifications. The main challenge is that zone appends are issued to a zone, not an address, and only return their address on completion. This address can be anywhere in a zone, and consequently, the SSD can reorder WAL data. Thus, the WAL needs to be resistant to data reordering. Therefore, current WAL designs on ZNS (such as RocksDB + *ZenFS* [36]) only use write I/O or only allow scaling zone appends by increasing threads [28]. We visualize the reordering challenge as "?" in Fig. 1.

This work proposes ZWALs, a zone append-friendly WAL for ZNS. ZWALs improve write throughput on ZNS and are resilient against data reordering. They achieve this feat by adding 64-bit atomically increasing sequence numbers to each PUT request. The sequence numbers specify the absolute ordering of data and are used to infer the order within the WAL. On recovery, the WAL reads all of its KV-pair changes and then sorts them back into their original order using the sequence number. After sorting, the LSM-tree applies the changes in sequence. Considering that LSM-tree WALs are generally only recovered during database startup and WALs are small (e.g., 32 MiB), we consider trading WAL read for better write throughput acceptable. To reduce the overhead of reordering and to prevent reading the entire WAL, we introduce the notion of WAL barriers. A ZWAL synchronizes all zone appends at a barrier. Barriers ensure that a read to the WAL only needs to read and sort between subsequent barriers, increasing WAL read performance.

We implement ZWALs in *ZenFS*, a state-of-the-art custom file system backend for RocksDB, and report that ZWAL leads to significantly higher write throughput than traditional WALs on commercially available ZNS SSDs, up to 33.02% higher throughput on the YCSB benchmark suite. Similarly, we repeat our experiments on the ConfZNS [33] emulator and report that with high internal parallelism, ZWAL can deliver up to 8.56 times higher write throughput on *YCSB*.

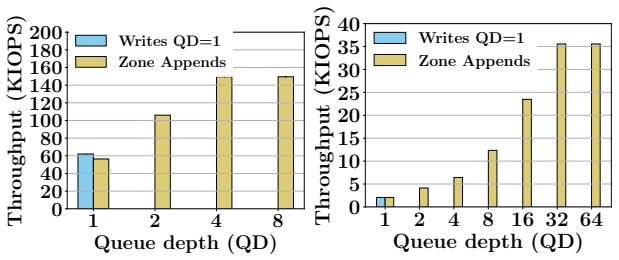
In this paper, we make the following key contributions:

1. We characterize the performance of the zone append operation and explain how we can leverage them for WALs.
2. We design and implement ZWALs—a new WAL design for ZNS zone appends.
3. We evaluate ZWALs on both the micro- and macrolevel.
4. We open-source the code of our ZWAL implementation at https://github.com/stonet-research/zwal.

2 Motivation: Why use zone appends?

Below, we demonstrate a performance characterization of zone appends. The design of ZWAL relies on high write concurrency and throughput for small writes. In this section, we show how zone appends lead to higher write concurrency and throughput than write I/O to motivate their use-case in WALs. In our benchmarking, we use *fio* [19] (v3.32) as a workload generator. We use the *io_uring* storage interface with NVMe passthrough [20] since the Linux block layer does not support zone appends and follow recommended *io_uring* performance optimizations [10]. We modify *fio* to support zone appends for passthrough (~10 LOC). We show the rest of our benchmarking setup in Tab. 1.

We evaluate the concurrency of zone appends by increasing the queue depth (QD)—the maximum number of concurrent zone appends—and measure throughput in I/O operations per second (IOPS). Since ZNS prohibits multiple write I/Os to the same zone, we only evaluate write I/O at QD 1. We issue all requests at a granularity of 8 KiB, which we evaluate as the optimal request size (i.e., lowest request latencies). Fig. 2a shows the throughput of zone appends in IOPS (y-axis, higher is better) with increasing QD (x-axis). Zone appends scale up to a QD of 4, beyond which the device's peak bandwidth is reached according to the device's specification sheet. We observe that write throughput is up to 2.41 times higher for zone appends (at high QD) than for write I/Os. At QD 1, write I/O leads to higher throughput (10.01%). We also investigate the impact of request size



(a) *Commercially available SSD* **(b)** *Emulated ConfZNS SSD*

Figure 2. *ZNS write throughput of zone* appends *and* writes.

on the difference between zone append and `write` throughput. While writes to the WAL are small, state-of-the-art file systems such as *ZenFS* buffer writes, resulting in large periodic writes (e.g., 1 MiB). The results (not shown) reveal that for larger requests (32 KiB and beyond), throughput is similar for both `write` and zone appends at any QD. In short, zone appends have higher write concurrency and throughput than `write` I/Os for small sequential writes (confirming [11]).

We design our work for SSDs with high parallelism within zones (i.e., intra-zone parallelism). The commercial SSD we evaluated scales up to QD 4, we also explore the configuration space for ZNS with higher intra-zone parallelism using the emulator ConfZNS [33]. We tailor the emulated SSD for high intra-zone parallelism (exact configuration in the source code). In all further experiments (unless explicitly stated), we utilize this emulated SSD. Fig. 2b shows the throughput of 8 KiB zone appends for this SSD. On this SSD, zone appends scale up to QD 32. For larger request sizes, we observe similar performance for `write` I/Os and zone appends.

In conclusion, zone appends lead to higher write throughput and concurrency than `write` I/Os for small request sizes with high QD. Therefore, we recommend zone appends for applications that issue frequent small writes, e.g., WALs.

3 Design and Implementation of ZWAL

In this section, we detail the design and implementation of ZWAL. First, we explain ZWAL's design goals and how their design differs from a traditional WAL. Second, we explain how we implement ZWAL within the *ZenFS* file system.

3.1 ZWAL Design

We design ZWALs, a new type of WAL for ZNS SSDs. The design is not limited to LSM-trees and is applicable to other databases, such as SQLite [15]. Further on, it can be used for any storage medium/interface that reorders write requests. Nevertheless, we limit our discussion to LSM-trees on ZNS and use *ZenFS* as a reference model, to explain how we change an existing WAL design for ZNS (more about this in §6). We design ZWALs around three key WAL characteristics: (1) WALs are write-heavy and primarily issue small writes; (2) WALs are read during database recovery only, and; (3) WALs are typically small (i.e., 64 MiB [14]). The key insight is that WAL performance is dominated by small writes to the WAL and reads only happen sporadically. Hence we consider *trading lower read performance for increased write performance* an acceptable trade-off. Our design goal is to increase write performance and match peak zone append performance (see §2). In a WAL, we differentiate between four major operations: writing to the WAL, recovering all data from the WAL, allocating a WAL and deleting the WAL. We explain the design using Fig. 3, representing ZWAL's design.

WAL write. On a WAL write, the data of a PUT request is written to the end of the WAL. In *ZenFS*'s design this eventually results in a `write` I/O to the zone's write pointer. If another PUT operation is issued concurrently, it has to wait for the previous PUT to finish. To increase concurrency, ZWALs issues zone appends to the head of a zone and do not wait for zone appends to finish. As a result, multiple PUT operations can be written to the WAL concurrently. Considering that zone appends can be reordered, the WAL has no guarantee on the order of PUT requests. A ZWAL achieves resilience against reordering by prepending each WAL write with a small header (128-bit) (❶). We refer to the combination of WAL data and header as a *WAL entry*. The header (depicted in yellow) exists out of a 64-bit sequence number and the WAL entry's size. Sequence numbers increase atomically and represent the absolute data ordering. For example, the first write to a WAL has sequence number 0, and the tenth write 9. Each WAL maintains its own sequence number to negate the risk of rollover (i.e., it is unlikely for a WAL to be 2^{64} pages). The size of an entry is used to infer the location of the subsequent appended WAL entry (if any). An alternative to WAL headers is using the address returned by zone appends to determine where data is stored. The disadvantage of this solution is that the return address is ephemeral, requiring another write to storage to store this address.

As discussed in §2, the ideal request size for appends might differ from the page size, and KV-pairs may be significantly smaller than the page size. Therefore, ZWAL's design allows for buffering (similar to *ZenFS*). On writing to the WAL, the WAL first copies data to a buffer (❷). Once the buffer is full or the WAL is synced (e.g., fsync, close), we write the data to the SSD using a zone append (❸). Practitioners should decide on the WAL buffer size; larger buffers generally lead to higher throughput but lower persistence guarantees. Note that regardless of the buffer size, ZNS does not allow zone appends to cross zone boundaries. If a zone append request crosses a boundary, the ZWAL splits the request, one to each zone, each with its own sequence number.

WAL recovery. During WAL recovery, an LSM-tree scans its WAL sequentially and applies the read data to its memtable. An LSM-tree reads a few KiB at a time and assumes the data is in sequential order. However, a ZWAL is stored out of order

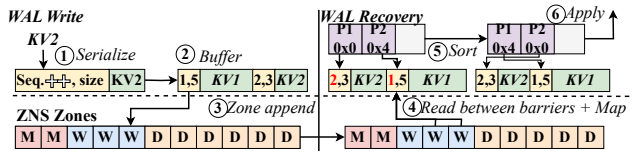


Figure 3. *Overview of a* ZWAL*, depicting the stages of writing WAL entries to ZNS zones, and recovering the WAL.*

Table 1. *Details of the benchmarking environment*

Component	*Configuration details*
CPU	Dual socket Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 2 sockets, 10 cores/socket, 20 virtual cores/socket
DRAM	256 GiB, DDR4
ZNS	Western Digital Ultrastar DC ZN540 1TB, PCIe 3.0
OS	Ubuntu (v22.04), kernel (v6.3.8, built from source)

on the SSD; hence, the ZWAL needs to restore its original order. A ZWAL achieves this by creating mappings from logical addresses (i.e., offsets) to physical addresses. On a read, it first finds the WAL entry corresponding with a logical address. It finds this entry using the information in the WAL headers. Since sequence numbers are monotonically increasing and data is only appended to the WAL, WAL entries with a higher sequence number have a strictly higher logical address. Specifically, the logical address of the WAL entry with sequence number x is equal to the logical address of the WAL entry *x-1* plus its size. For example, a PUT with sequence number 1 is stored at the logical address of PUT 0 plus the size of PUT 0. On a read to the WAL, the ZWAL reads the entry with the corresponding logical address.

Retrieving WAL headers is challenging; each WAL entry has a potentially different size, and hence—apart from the first request—the ZWAL can not infer a priori where WAL headers are stored. That is because, for each entry, we first need to know its size to determine where the subsequent entry is stored. Since we do not know where WAL headers are stored a priori, the ZWAL needs to be scanned sequentially, from first to last, entry by entry. A naive implementation would (re)read the entire WAL for each read request, but this does not scale. An alternative solution is to read the WAL once and keep a mapping to logical addresses in memory, but this requires memory proportional to the WAL size. For ZWALs, we opt for an alternative: we use *barriers*. A ZWAL inserts barriers at a predefined page interval—defined as $P_{barrier}$. At a barrier the ZWAL syncs all zone appends (i.e., wait for all to be completed). Requests are ordered between subsequent barriers, i.e., an append beyond a barrier has a strictly higher physical address than an append before the barrier. On a read, the ZWAL first finds the closest barrier, and then it reads all data up to the subsequent barrier (❹). It finds the closest barrier with: $floor(logical\_address/P_{barrier})$. After reading all data between the two barriers, it creates a mapping from sequence number to WAL entries. Then it sorts the mapping on sequence number (can be limited to metadata updates) (❺). In this design, a ZWAL is only required to read and maintain a mapping of up to $P_{barrier}$ pages at a time, reducing both I/O and memory footprint to a configurable upper limit. As an optimization, this mapping can be cached, as reads are sequential and subsequent reads are likely to appear in the same barrier. In our design, we expect the recovery cost to be slightly higher than that of traditional WALs and to scale proportionally ($O(n \log n)$ for sorting) with the barrier size.

WAL allocation. On WAL allocation, storage resources are allocated for the WAL. On *ZenFS*, this involves assigning a zone to the WAL. This zone is not exclusive to the WAL and can be shared with other LSM-tree components. A ZWAL has more restrictions. A ZWAL requires dedicated zones for WALs ("W" in Fig. 3). This requirement is because of two reasons. Firstly, this is to prevent other LSM-tree components from issuing writes to the same zone as zone appends, as this would require zone appends to wait for writes to finish (nullifying their advantages). Secondly, ZWAL's recovery procedure requires all WAL data to be stored contiguously. After reading the data of a PUT, the subsequent read to the WAL will read the data next to this PUT. It does not check if this data belongs to the WAL. A solution (unverified) is to add the filename to the WAL header. The WAL can then ignore the entry if the filename does not match its name.

WAL deletion. On WAL deletion, its storage resources are released. *ZenFS* does not treat WAL deletion differently from other data and neither do ZWALs. Since a ZWAL has a dedicated set of zones, it is safe to reset (a ZNS-specific operation to mark a zone for deletion) all WAL zones at any point in time. Deletion can be done actively (i.e., immediately) or lazily (i.e., when the zones are needed).

3.2 ZWAL Implementation

We implement ZWALs in RocksDB's *ZenFS* file system since it is the state-of-the-art for ZNS [3, 31, 36]. ZWALs issue zone appends using *io_uring* with NVMe passthrough since the Linux block layer does not support zone appends (§2). To separate data from WAL zones, a dedicated set of zones is reserved for WALs between *ZenFS*'s metadata and data zones; see Fig. 3 ("W" is WAL, "M" is metadata, "D" is data). The exact number of WAL zones ("W") is configurable during formatting. We also implement configurable barriers, configurable in multiples of the page size.

To support ZWALs in RocksDB, we modify one function. On a WAL delete, RocksDB first moves WALs to an archival directory and only physically deletes them at a later point in time. Since we use a limited number of zones for WALs, this will

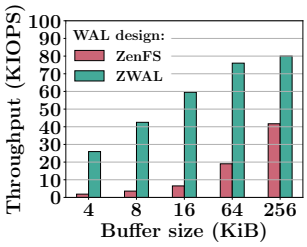


Figure 4. *RocksDB throughput for db_bench with fillrandom and 4 KiB requests;* ZWAL *configured at QD 32.*

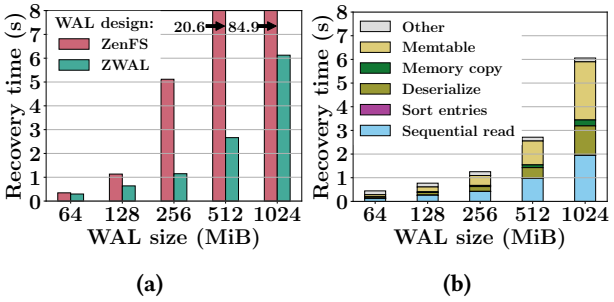


Figure 5. *(a) WAL recovery time within RocksDB; (b)* ZWAL *recovery latency breakdown.*

quickly lead to out-of-space errors—hence, we force RocksDB to delete old WALs immediately. Additionally, RocksDB can only delete a WAL if all column families that use a WAL are flushed. Therefore, it is important to flush all column families regularly for WALs to prevent out-of-space errors (automized by setting the max WAL size). Note that RocksDB provides various WAL filters, we do not expect filters to function differently for ZWALs.

4 ZWAL Evaluation

In this section, we evaluate the performance of ZWAL's implementation in RocksDB + *ZenFS*. First, we compare ZWAL's write throughput to raw zone append's (§2) and conventional *ZenFS*'s write throughput. Next, we analyze ZWAL's recovery time and compare it against conventional *ZenFS*'s recovery time. Then, we measure ZWAL's performance with *YCSB* [9] application workloads. Lastly, we evaluate the impact of the barrier size on WAL throughput and recovery time. Note that unless stated, we use a barrier size equal to the SSD's zone size. We show our benchmarking configuration in Tab. 1.

4.1 WAL Write Performance

We design ZWALs to improve write throughput—in this section, we evaluate if throughput is also improved in practice. *ZenFS* and ZWALs both buffer write requests for the WAL as it can lead to significant throughput improvements (at the cost of lesser persistence). Due to its performance potential, we also investigate the impact of buffer size on throughput, enabling practitioners to make a trade-off between throughput and persistence. We evaluate both WALs with RocksDB's *db_bench* [14] benchmark and the *fillrandom* workload, with 5 GiB of 4 KiB KV-pairs. We configure ZWALs to issue zone appends at a maximum QD of 32 (optimal; see Fig. 2b).

Fig. 4 shows the throughput (y-axis) with increasing buffer sizes (x-axis). The larger the buffer size, the more PUT requests the WAL merges into one I/O request (e.g., with 16 KiB buffers and 4 KiB requests, every 4 requests are merged). ZWALs outperform *ZenFS*'s WALs for all evaluated buffer sizes, from a 1.92 times throughput increase with 256 KiB buffers to 13.94 times with 4 KiB buffers. The peak throughput of *ZenFS*'s WALs (not shown) is achieved with a 1 MiB buffer and is 48.71 KIOPS. We do not show results with buffer sizes larger than 256 KiB because of an implementation bug in ZWALs. From these results, we conclude that ZWALs significantly increase LSM-tree write throughput on ZNS SSDs with high intra-zone parallelism.

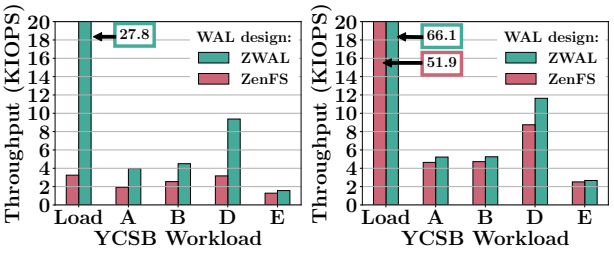
While not shown in a plot, ZWALs do not scale for I/O requests larger than the *zone append size limit* (ZASL). ZASL defines the maximum request size of zone appends; hence, if requests are larger than ZASL, they need to be split into multiple subrequests. If a request is split and each fragment is sent individually, the SSD can reorder them. The current ZWAL implementation does not support unordered fragmentation and falls back to issuing one request at a time, sending all request fragments serially, reducing performance significantly. We can solve this challenge (not evaluated) by supporting unordered fragmentations, e.g., by giving each fragment its own sequence number.

4.2 WAL Recovery Time

We define recovery overhead as an increase in recovery time (e.g., 10% longer recovery). In this section, we do a quantitative analysis of the recovery overhead of ZWAL's recovery procedure compared to *ZenFS*'s WALs. Further on, we determine the relation between WAL size and recovery time, facilitating the reasoning for WAL size configuration.

We first fill a WAL using *db_bench* with the *fillrandom* workload. We use a minimal buffer size of 4 KiB and write all data as 4 KiB KV-pairs to ensure that every I/O request contains exactly one KV-pair (i.e., each PUT leads to its own WAL entry). We evaluate WAL sizes ranging from the default of 64 MiB up to 4 GiB. After filling the WAL, we reload the database and measure WAL recovery time, measured from the moment we open the WAL until we have read all data and applied it to the LSM-tree's memtable.

Fig. 5a shows the WAL recovery time (y-axis, lower is better) with increasing WAL size (on the x-axis). ZWALs take significantly less time to recover than *ZenFS*'s WALs, up to 13.86 times (6.13s compared to 84.9s) for a 1 GiB WAL. ZWALs have a low recovery time because our read implementation is optimized, particularly because we cache mappings of the



(a) *Emulated ConfZNS SSD* **(b)** *Commercially available SSD*

Figure 6. *YCSB throughput for RocksDB.*

region between subsequent barriers. Instead of reading each entry individually (i.e., what *ZenFS* does), we issue larger read requests occasionally. We have confirmed this behavior using a barrier size of 4 KiB, where our 1 GiB recovery time increases to 33.62 seconds (an increase of 2.43 times). ZWAL recovery time increases proportionally with the WAL size.

To further investigate the recovery time of ZWALs (and where its bottlenecks are), we conduct a latency breakdown, shown in Fig. 5b. Notably, the recovery time is dominated by applying entries to the memtable (close to 40%) and I/O (close to 30%)—note that the exact percentages depend on WAL size and hardware used. Sequential I/O is a large portion of the recovery as the WAL reads synchronously—With *fio*, we confirmed read throughput to be about 417 MiB/s for the emulated SSD at a granularity of 64 KiB (we evaluate 64 KiB as the average read size to the WAL). The combined time it takes to sort and deserialize the WAL and copy data from the in-memory buffers, while not negligible, is, in our evaluations, 24.39% of the recovery time at most.

In short, ZWAL's recovery time is up to 13.86 times lower than *ZenFS*'s WAL recovery time. Its recovery time is dominated by applying entries to the memtable, followed by I/O.

4.3 Application Workload YCSB

We evaluate the application performance of ZWALs using the state-of-the-practice *Yahoo Cloud Services Benchmark* (YCSB) [9] workloads A (50% reads, 50% updates), B (95% reads, 5% updates), D (95% reads, 5% insertions) and E (95% scans, 5% insertions). These workloads are selected as they use PUTs, i.e., write to the WAL. Note that an update constitutes a read and a write in sequence. Before each load phase we `reset` all ZNS zones and reformat the file system. All workloads have the same load phase; filling the database with 25 GiB of KV-pairs for the commercially available SSD and 20 GiB for the emulated SSD. In the run phase, we issue 1 million operations for each workload. We set the RocksDB write buffer- and target file- size equal to the zone size and the KV-pair size to 4 KiB and 1 KiB (default *YCSB* size) for the emulated and commercially available device, respectively. The WAL buffer size is 8 KiB, and we issue zone appends at a maximum QD 32 (see Fig. 6).

Fig. 6a shows the *YCSB* throughput for the emulated SSD. The largest throughput increases occur in the load phase (8.56 times)—matching Fig. 2b—since it only issues PUT requests to the WAL. The run phase also shows significant performance increments, but to a smaller degree (e.g., up to 2.96 times for workload D), because these workloads include reads, and the read and write requests compete for the same SSD resources. Repeating the experiment (not shown) with larger buffer sizes shows little throughput benefits for ZWALs, similar to what we observed in §2.

Fig. 6b shows the throughput for the commercially available SSD. Akin to the emulated device, there are significant throughput improvements for the workloads that are PUT-heavy, such as the load phase (27.39%) and workload D (33.02%), and less for update or scan-heavy workloads, such as A (12.51%) and E (6.26%). Throughput differences for this device are less because zone appends scales up to QD 4 (see Fig. 2a). Similar to previous experiments, repeated experiments with larger buffers show no significant performance differences between `write` I/Os and zone appends.

In conclusion, ZWALs lead to significant throughput improvements on commercially available ZNS SSDs (up to 33.02%) for PUT-heavy workloads. However, throughput improvements are less for workloads that include a mix of writes and reads (e.g., 6.26% for scan-heavy workloads).

4.4 Barrier Size Performance

Below, we evaluate the impact of barrier sizes on ZWAL's performance. We repeat all experiments from §4.1 and §4.2 on the emulated SSD and evaluate with barrier sizes ranging from one page to an entire zone.

We observe (not visualized) that larger barrier sizes (16 MiB) lead to significantly higher write throughput. For example, we observe 4.13 times higher write throughput with 16 MiB barriers (28.25 KIOPS) than with 4 KiB barriers (1.99 KIOPS). However, beyond a threshold (>16 MiB), further increasing the barrier size has less impact on the throughput (a 3.91% increase from 16 MiB to a zone-sized barrier). The reason that throughput is increased for larger barriers is that barriers act as synchronization points, forcing the requests that cross a barrier to wait for all previous requests. This effect becomes especially prevalent when the barrier size is smaller than $size_{buffer} * QD_{max}$, since it lowers the effective QD.

We also observe that larger barrier sizes significantly decrease WAL recovery time (again, not visualized). We observe up to 5.20 times shorter recovery times with zone-sized barriers (6.13s) than with 4 KiB barriers (31.87s). Similar to the throughput experiment, we observe no significant effect on total recovery time beyond a (device-specific) threshold (>16 MiB). The decrease in recovery time can be explained due to higher read throughput from storage. Data is read from storage in a granularity of the distance between subsequent barriers. Therefore, with small barriers, ZWALs read smaller chunks of data. This leads to many small reads and

small reads generally take longer to complete than a few large reads. We observe up to 14.51 shorter I/O read time for zone-sized barriers (1.95s) compared to 4 KiB barriers (28.23s).

5 Related Work

The recent standardization of ZNS has given rise to a plethora of active research [1, 3, 23], with numerous proposals evaluating the ZNS integration into the design of KV-stores and zone append integration into applications [1, 16, 28, 32].

There are a couple of WAL designs for ZNS [28, 29, 32]. WALTZ [28] uses zone appends to increase write concurrency and allows multiple concurrent RocksDB client threads to zone append to the same zone concurrently. The biggest difference with ZWALs is that WALTZ issues zone appends synchronously within threads (i.e., waits for completion). Such an approach prevents a complicated recovery but does not allow for concurrent writes within a single thread. Purandare et al. propose issuing small zone appends in WALs instead of buffering [32]. This work, however, does not take into consideration data recovery from reordered zone appends.

Several applications employ zone appends in their design [1, 16, 23, 35]. Most of these applications use the return address of zone appends and, as a result, either require an extra I/O operation for persistence [16] or do not need to be persistent [1]. RAIZN [23] is similar to ZWALs since it uses sequence numbers. However, the recovery procedure is different due to the nature of RAID—only the most recent number for each data entry is valid (i.e., no sorting needed), and the number of valid entries is explicitly maintained elsewhere. This would not scale for WALs as it requires maintaining the entries elsewhere (i.e., an additional write).

Our concept of sequence numbers in ZWALs is based on cross-referencing logs (CRL) [8, 18]. CRLs use per-thread logs (WALs) and use sequence numbers to infer ordering between those logs. We apply this concept to a situation where only one log exists, but the log itself can be reordered.

6 Discussion and Future Work

The current ZWAL implementation is tightly coupled to the *ZenFS* file system. This coupling is because ZWALs have to issue zone appends to storage directly, which requires modifying the file system or issuing I/O to storage directly from an application. The advantage of this approach is its transparency to applications; the disadvantage is the need to modify file systems. We modify *ZenFS* to treat all I/O to ".log" files differently by storing data in separate zones and using zone appends. However, such a file system modification is generally applicable to other ZNS-friendly file systems (i.e., F2FS [26] and Btrfs [13]). F2FS, for example, already uses file extensions to estimate file hotness (i.e., cold, warm, hot) and stores data with different temperatures separately [26]. To support ZWALs in F2FS, we propose adding an extra classification for WAL-like files such as ".log" and allocating a separate region for this data, similar to our *ZenFS* implementation.

This paper focuses on WALs for LSM-trees in a local setting. However, our design extends beyond this configuration. Firstly, ZWALs are beneficial for any application that makes use of WALs. As such, we propose adding ZWALs to SQLite's WAL mode [15]. Secondly, we argue that ZWALs are also valuable for distributed settings. A common database optimization is to store WAL data on a separate storage device [6]. Furthermore, storage disaggregation is ubiquitous and approaching the performance of local storage [24, 25]. Storing WAL data to different storage devices prevents WAL traffic from interfering with other I/O traffic and allows multiple databases to share one device for WAL traffic. The design of ZWAL comes with the additional benefit that the ZNS device does the scheduling for concurrent applications. Currently, ZWALs store WAL data in different logical areas than the rest; as a next step, we will deploy ZWALs to separate (remote) devices and measure their performance characteristics. Lastly, we propose extending our work beyond WALs and optimizing other components of the LSM-tree for ZNS, such as the write path of flush and compaction operations.

7 Conclusion

Zoned Namespace (ZNS) SSDs enable stable throughput for LSM-trees, but are known to lead to write throughput challenges in the LSM-tree's WAL. In this work, we utilize the ZNS-specific zone append operation to address this challenge and showcase that our approach improves write throughput significantly. We believe the contributions of this work showcase the potential of zone appends beyond LSM-trees as our sequence number procedure is generally applicable, encouraging the further utilization of zone appends in applications and data structures. To facilitate further research, we publish the artifacts of this study at https://github.com/stonet-research/zwal.

Acknowledgments

This work is supported by generous hardware donations from Western Digital arranged by Matias Bjørling. This work was partially supported with Netherlands-funded projects NWO OffSense and GFP 6G FNS, and EU-funded projects MCSA-RISE Cloudstars and Horizon Graph-Massivizer. Krijn Doekemeijer is funded by the VU PhD innovation program. We thank the anonymous reviewers and the AtLarge group at the Vrije Universiteit Amsterdam for their feedback.

References

[1] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. 2023. ZNSwap: un-Block your Swap. *ACM Transactions on Storage* 19, 2, 1–25.

[2] Matias Bjørling. 2020. Zone Append: A New Way of Writing to Zoned Storage. https://www.usenix.org/conference/vault20/presentation/

bjorling. *USENIX Vault* (2020).

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

[4] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*. Santa clara, CA, USA, 359–373.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[6] 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)*. 17–32.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[8] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. https://doi.org/10.1145/3534056.3534945

[11] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. 2023. Performance characterization of nvme flash devices with zoned namespaces (zns). In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 118–131.

[12] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-scale Applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[13] Ohad Rodeh et al. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* (2013).

[14] Facebook. Accessed: 2024-01-19. RocksDB. https://github.com/facebook/rocksdb.

[15] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. SQLite: Past, Present, and Future. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3535–3547.

[16] Jin Yong Ha and Heon Young Yeom. 2023. zCeph: Achieving High Performance On Storage System Using Small Zoned ZNS SSD. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 1342–1351.

[17] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. The unwritten contract of solid state drives. In *Proceedings of the twelfth European conference on computer systems*. 127–144.

[18] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 967–979.

[19] Jens Axboe. Accessed: 2024-01-19. Fio. https://github.com/axboe/fio.

[20] Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. 2024. I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 107–121.

[21] 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.

[22] 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)*.

[23] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G Andersen, Gregory R Ganger, George Amvrosiadis, and Matias Bjørling. 2023. RAIZN: Redundant Array of Independent Zoned Namespaces. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 660–673.

[24] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–15.

[25] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash= local flash. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 345–359.

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

[27] 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.

[28] Jongsung Lee, Donguk Kim, and Jae W Lee. 2023. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2884–2896.

[29] Jinhong Li, Qiuping Wang, and Patrick PC Lee. 2022. Efficient LSM-Tree Key-Value Data Management on Hybrid SSD/HDD Zoned Storage. *CoRR* abs/2205.11753 (2022). https://doi.org/10.48550/ARXIV.2205.11753 arXiv:cs.PF/2205.11753

[30] NVMe Consortium. Accessed: 2024-01-19. NVMe® 2.0 Specification. https://nvmexpress.org/nvme-2-0-specifications-and-new-technical-proposals/.

[31] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. 2023. ZenFS+: Nurturing Performance and Isolation to ZenFS. *IEEE Access* 11 (2023), 26344–26357.

[32] Devashish Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. 2022. Append is Near: Log-based Data Management on ZNS SSDs. In *12th Annual Conference on Innovative Data Systems Research (CIDR'22)*.

[33] Inho Song, Myounghoon Oh, Bryan Suk Joon Kim, Seehwan Yoo, Jaedong Lee, and Jongmoo Choi. 2023. ConfZNS: A Novel Emulator for Exploring Design Space of ZNS SSDs. In *Proceedings of the 16th ACM International Conference on Systems and Storage*. 71–82.

[34] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2592798.2592804

[35] Qiuping Wang and Patrick PC Lee. 2023. ZapRAID: Toward High-Performance RAID for ZNS SSDs via Zone Append. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 24–29.

[36] Western Digital. Accessed: 2024-01-29. ZenFS. https://github.com/westerndigitalcorporation/zenfs.