

ZWAN: Leveraging Zone Random Write Area (ZRWA) for Avoiding WAL Tax in LSM-tree

Minwook Kim
Seoul National Univeristy
Seoul, Republic of Korea
ace@snu.ac.kr

Kyu-Jin Cho
Seoul National University
Seoul, Republic of Korea
bori19960@snu.ac.kr

Jin-Soo Kim
Seoul National University
Seoul, Republic of Korea
jinsoo.kim@snu.ac.kr

Abstract

Zoned Namespace (ZNS) SSDs eliminate device-internal management overhead but impose a strict sequential write constraint. While this append-only discipline suits LSM-tree key-value stores, concurrent Write-Ahead Log (WAL) writes must be serialized behind a single zone write pointer (WP), causing synchronization overhead. RocksDB's *group commit* mitigates this bottleneck and reduces the write amplification of sub-block records, but inflates tail latency.

We propose ZWAN, an LSM-tree key-value store that resolves this tradeoff via the Zone Random Write Area (ZRWA) feature in the ZNS Command Set. ZWAN packs multiple WAL records into a single ZRWA block via in-place updates, eliminating sub-block write amplification without batching delay. Additionally, ZWAN employs a dedicated I/O engine built on ownership-based allocation and WP-aware prioritization to fully utilize the performance benefits of ZRWA. Compared to the state-of-the-art LSM-tree on ZNS SSDs, ZWAN achieves up to 3.4× higher throughput, 3.0× lower P50 latency, and 9.8× lower P99 latency while simultaneously reducing write amplification by 98.6%.

CCS Concepts

• **Information systems** → **Key-value stores**; *Storage management*; • **Hardware** → *External storage*.

ACM Reference Format:

Minwook Kim, Kyu-Jin Cho, and Jin-Soo Kim. 2026. ZWAN: Leveraging Zone Random Write Area (ZRWA) for Avoiding WAL Tax in LSM-tree. In *18th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '26)*, September 28–29, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3837053.3837352>



This work is licensed under a Creative Commons Attribution 4.0 International License.

HotStorage '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2902-7/26/09

<https://doi.org/10.1145/3837053.3837352>

1 Introduction

Zoned Namespace (ZNS) SSDs [3] have emerged as a storage solution that eliminates the device-internal overhead inherent in block-interface SSDs and are being increasingly adopted across diverse application domains [9, 11, 13, 14, 18]. ZNS SSDs partition their address space into fixed-size *zones*, each spanning a contiguous range of logical block addresses (LBAs), and impose a strict sequential write constraint: writes must be appended in LBA order and the entire zone must be explicitly reset before any block can be rewritten. Consequently, ZNS SSDs are a natural fit for I/O-intensive workloads with predominantly sequential access patterns.

The Log-Structured Merge (LSM) tree [15] is a write-optimized data structure widely adopted in persistent Key-Value (KV) stores [5, 7, 12] whose bulk I/O pattern inherently aligns with the sequential write constraint of ZNS SSDs. An LSM-tree buffers KV pairs in *MemTables* and flushes them as sorted *SSTables* when full, converting small random writes into large sequential ones. To protect KV entries in volatile *MemTables* from power failures, every committed Put is first appended to a persistent Write-Ahead Log (WAL) before being applied to the *MemTable*. However, this durability guarantee comes at the cost of frequent fine-grained WAL record commits to the storage device. This WAL persistence not only increases write amplification by inflating sub-block writes into full-block I/O, but also bottlenecks aggregate throughput under concurrent writes by turning the WAL path into a contention point [13]. We collectively refer to these overheads of WAL persistence as the *WAL tax*.

In this paper, we propose ZWAN, an LSM-tree-based key-value store that avoids the costly WAL tax by leveraging the Zone Random Write Area (ZRWA) [1], a small overwrite-capable window adjacent to the write pointer (WP). ZWAN commits each WAL record individually for low tail latency, while incrementally packing multiple records into a single block via ZRWA overwrites, without inflating the write amplification factor (WAF). However, leveraging the ZRWA window for WAL records poses two key challenges. First, unlike normal zones, the ZRWA window can accept multiple in-flight writes concurrently, and their completions may be returned out of submission order. ZWAN keeps block allocation off the critical write path so that each writer owns a

block exclusively without scanning the window, minimizing contention. Second, the WP of ZRWA window advances only over consecutive fully packed blocks starting from the WP. Therefore, a single straggling block can stall the entire window advancement and degrade throughput. ZWAN biases allocation toward blocks near the WP to accelerate window advancement, ensuring a steady supply of fresh blocks.

Our evaluation shows that, compared to a state-of-the-art LSM-tree implementation on ZNS SSDs, ZWAN improves throughput by up to 3.4 $\times$, reduces P50 and P99 latency by up to 3.0 $\times$ and 9.8 $\times$, respectively, while reducing WAL-level WAF by 98.6%. The source code of ZWAN is publicly available at <https://github.com/snu-csl/zwan>.

2 Background and Motivation

2.1 Extended Features of the ZNS SSD

ZNS SSDs [3] divide the address space into fixed-size zones. Each zone only permits sequential writes, and the host and device synchronize the writable position through a WP. Due to these constraints and the potential reordering in the request queue, the host must wait for each write to complete before submitting the next one [11]. Beyond conventional *Zone Writes*, modern ZNS SSDs provide *Zone Append* and *ZRWA* to relax host-side ordering.

Zone Append [2] lets the host write to a zone without specifying the WP: the device places the data at a writable location and returns the assigned LBA, freeing the host from tracking the WP and serializing writes. Multiple appends can thus be issued concurrently within a zone, reducing tail latency caused by WP serialization [13].

ZRWA [1] provides a small overwrite-capable window adjacent to the WP that acts as a non-volatile cache. Within it, the sequential constraint is lifted, so the host can issue multiple random writes concurrently and overwrite the same LBA—both forbidden in a normal zone [6, 11, 18].

Writes to the window do not immediately advance the WP; instead, it advances in multiples of the ZRWA flush granularity (ZRWAFG) via *Implicit* or *Explicit Flush*. Implicit Flush, triggered when the host writes beyond the ZRWA boundary, can race non-deterministically with concurrent in-window writes and fail them [11]; Explicit Flush is a dedicated command that advances the WP in ZRWAFG units.

2.2 WAL Tax in RocksDB

In RocksDB, variable-sized records (a fixed-size header + KV payload) are persisted to the WAL for durability, incurring the *WAL tax* in two forms. Since the smallest unit of a write to an SSD is a single block (4 KB here), each sub-block WAL record is written as a full block, inflating WAF. In addition, concurrent writers must serialize on the WP, adding synchronization overhead to the latency-critical I/O path [13].

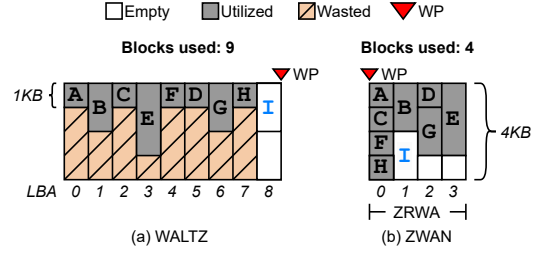


Figure 1: Comparison of WAL write layouts between WALTZ and ZWAN.

To mitigate this, RocksDB employs *group commit* [16]: a dynamically elected *leader* concatenates *followers' Write-Batches* into a single WAL chunk and issues one I/O for the group, then wakes the followers to insert their KV entries into the MemTable in parallel. Packing multiple records into one block reduces both the number of WAL writes and WAF, but every writer's latency now includes the leader-election, batching, and I/O time of the entire group, so tail latency grows with batch size. The same tradeoff persists on ZenFS [3], a filesystem plugin for the ZNS interface.

WALTZ [13], a state-of-the-art LSM-tree KV store on ZNS, instead commits each WAL record individually via *Zone Append*, offloading host-side WP serialization to the device. This *per-record commit* achieves throughput comparable to group commit's at lower tail latency, but reintroduces severe WAF from sub-block records: in §4.2, WALTZ incurs up to 2.5 $\times$ higher WAL-level WAF than RocksDB on ZenFS.

3 Design

3.1 Sub-Block WAL Packing via ZRWA

ZWAN exploits ZRWA's in-place overwrite capability to accumulate WAL records at sub-block granularity within each block. Fig. 1 illustrates the WAL layouts of WALTZ and ZWAN on a 4 KB-block ZNS SSD when a new 2 KB entry I arrives, after eight prior entries (A–H) of 1–3 KB have been written. As shown in Fig. 1(a), since *Zone Append* does not support in-place overwrites, WALTZ must write each entry as a full 4 KB block, wasting 25–75% of each block as padding. Therefore, the 8 records A–H occupy 8 blocks and record I is placed at LBA 8. In contrast, ZWAN can pack each new entry into the remaining space of a block within the ZRWA window by overwriting the partially filled block. For example, in Fig. 1(b), up to four 1 KB WAL records can be packed into a single block because ZRWA Write allows in-place update.

To manage WAL record allocation within each block, ZWAN maintains a buffer page per ZRWA block to stage the block's in-progress contents. Once a buffer page is assigned, the new WAL record is written at the next free offset within the page and the updated page triggers a device write that overwrites the previous version of its block in place. Under concurrent writers, ZWAN limits each page to one in-flight

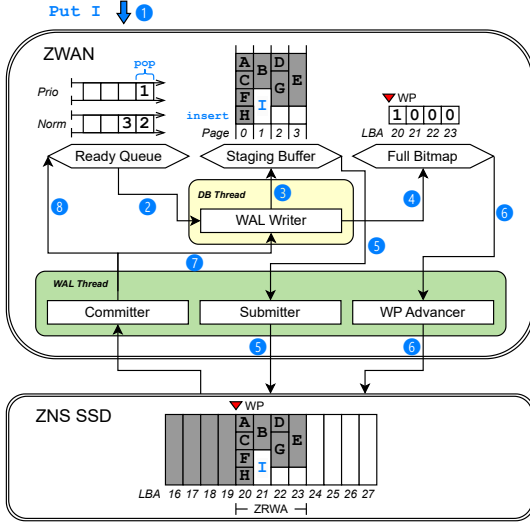


Figure 2: Control path of ZWAN WAL I/O engine.

write to prevent a stale version from overwriting a newer one (more details in §3.2). Although the per-page queue depth is limited to one, the zone-level queue depth remains high because each ZRWA window covers multiple buffer pages.

3.2 Buffer Allocation and Recycling

The I/O engine of ZWAN pursues two objectives to fully exploit the ZRWA window: (1) minimizing contention among concurrent writers on the WAL path, and (2) accelerating the consumption of blocks near the WP. ZWAN achieves these objectives through two complementary mechanisms: *ownership-based buffer page allocation*, which grants each writer exclusive access to a buffer page upon acquisition, and a *WP-aware priority queue*, which biases allocation toward low-LBA buffer pages. Fig. 2 illustrates the control path of ZWAN’s WAL I/O engine using the same example as in Fig. 1.

The engine comprises three core data structures and two types of workers. The state of each ZRWA block is maintained in a *Staging Buffer*, which also serves to stage in-progress WAL records before they are flushed to the device. The *Ready Queue* stores the indices of buffer pages ready to accept a new record. The *Full Bitmap* tracks which buffer pages have been marked full. On the worker side, a *WAL Writer* is assigned to each *DB Thread* to issue WAL entries into a buffer page, while a dedicated *WAL Thread* runs an event loop that handles WAL I/O submission, completion handling, and WP advancement.

When a *Put* request arrives (1), the *WAL Writer* pops page index 1 from the *Ready Queue* (2), transferring ownership of the corresponding page. Concurrent writers therefore never share a buffer page or scan for free pages linearly. Then, the *Writer* appends the WAL record to the next free offset within its assigned buffer page and sets the page’s *dirty* flag (3). In the figure, *Put I* inserts its record into page 1 at the

offset following the previous record B. If the remaining space is insufficient for the incoming record, the *Writer* marks the page as full in the *Full Bitmap* (4) and retries from (2) to acquire another page. For example, if page 2 had been acquired instead, its remaining space (under 2 KB) could not accommodate I, so ZWAN would treat the page as full.

The *WAL Thread* executes three roles within a single event loop. The *Submitter* scans the *Staging Buffers* and issues a ZRWA write for every page whose *dirty* flag is set (5). The *WP Advancer* scans the *Full Bitmap* for full pages starting from the current *WP* and issues an explicit ZRWA flush at the device’s flush granularity (6). ZWAN advances the *WP* only over a consecutive sequence of fully packed pages starting from the current *WP*, as prematurely flushing a partially packed page would waste its unfilled space and inflate WAF. The *Committer* processes WAL write completions from the device by signaling the waiting *DB Thread* (7) and returning the page to the queue (8). Upon explicit ZRWA flush completion, it updates the in-memory *WP* state and enqueues the newly exposed LBAs into the *Ready Queue* (8).

Since the *WP* only advances over consecutive full pages from its current position, a page that fills slowly near the *WP* can stall the window and starve later writers. To bias acquisition toward these urgent pages, ZWAN introduces a *Priority Ready Queue* alongside the *Normal Ready Queue*. The *Committer* classifies each returned page by its LBA distance from the current *WP* and pushes low-LBA pages to the *Priority Queue* rather than the *Normal Queue* (8). The *WAL Writer* drains the *Priority Queue* first and falls back to the *Normal Queue* only when the former is empty (2), accelerating the saturation of *WP*-adjacent pages. In the figure, the priority window is shown as two blocks for illustration; in practice, we empirically set it to four blocks from the *WP*.

3.3 Crash Recovery

ZWAN persists WAL records in a non-deterministic on-device order, as does WALTZ via Zone Append. Since WALTZ’s recovery is already designed for this property, ZWAN largely inherits its procedure. Recovery retrieves the start offset of the WAL region from the ZenFS metadata journal, scans the region, and applies each record to the *MemTable* according to its embedded sequence number. The rebuilt *MemTable* is then flushed to L0 as in stock RocksDB. ZWAN differs from WALTZ in how it determines the end of the WAL region and parses the packed records.

In WALTZ, the *WP* exactly marks the scan end. In ZWAN, valid blocks may exist beyond the *WP*, so recovery extends the scan to the end of the ZRWA window and identifies written blocks by the per-block checksum. Each valid block is then parsed sequentially from its start, with the length field in each record header locating the next.

4 Evaluation

Table 1: System configurations.

CPU	Intel® Xeon® Silver 4114 CPU 40 cores
Memory	352 GB DDR4 DRAM
Storage	Samsung PM1731a NVMe ZNS SSD 4 TB
OS	Ubuntu 20.04, Linux 5.15.0
RocksDB	6.25.3
SPDK	22.01.2 LTS

System configurations. We evaluate ZWAN on the testbed in Table 1, built around a Samsung PM1731a NVMe ZNS SSD [17] used in prior studies [9, 13, 14]. It exposes 40,704 zones of 96 MiB, up to 384 open, each with a 64 KB ZRWA window and 32 KB ZRWAFG. As baselines, we use ZenFS [3] (RocksDB on the ZenFS plugin) and WALTZ [13] at their latest published revisions. We implement ZWAN on WALTZ by replacing its Zone Append-based WAL with our ZRWA Write design, leaving the rest of the stack unchanged; all three share the same SPDK-based I/O backend [10].

Following WALTZ’s evaluation setup, we enable synchronous WAL writes to ensure strict consistency and disable compression. RocksDB uses four flush and four compaction threads with direct I/O on the read and flush/compaction paths; remaining options follow RocksDB defaults.

Workloads. We employ two classes of workloads.

(1) Device-level write characterization (§4.1): We first measure raw performance of the ZNS SSD with three interfaces: Zone Write, Zone Append, and ZRWA Write. Each run lasts 60 seconds and writes to one active zone at a time, moving to the next zone when the previous one is fully filled. To obtain saturated-throughput results, we use Zone Append with a queue depth of 128 (the device’s MQES) and issue ZRWA Writes across the entire ZRWA window. For latency measurements, we run experiments with a queue depth of 1. (2) Application-level benchmarks (§4.2, §4.3): We use db_bench [8] for microbenchmarks and MixGraph [4]. The microbenchmarks prefill 20 M KV pairs and issue 6 M ops per configuration, sweeping value size (100 B, 500 B, 1 KB, 2 KB, 4 KB), write:read ratio (10:0, 7:3, 3:7), and key distribution (uniform, Zipfian skew 0.99). MixGraph prefills 64 M pairs and issues 20 M ops under its four standard variants (AllRand, AllDist, PreRand, and PreDist) with default parameters from [4], including its Pareto-distributed value sizes. Throughout all experiments, we use four worker threads.

4.1 Performance Benefit of ZRWA

Fig. 3 reports the write performance of the PM1731a across the three interfaces. For ZRWA Write, we distinguish variants by the number of writes issued to each 4 KB block, which indicates how many times a single block is overwritten. ZRWA-1 writes each block once (no overwrite), ZRWA-N issues N writes before advancing and ZRWA-inf repeatedly overwrites one block in place. Once the series of completed

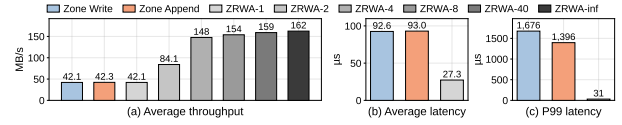


Figure 3: Performance comparison of Zone Write, Zone Append, and ZRWA Write.

blocks from the WP reaches ZRWAFG, the host issues an Explicit Flush that advances the WP by the same amount.

Fig. 3(a) shows that Zone Write, Zone Append, and ZRWA-1 achieve nearly identical throughput. However, ZRWA Write throughput increases with the number of overwrites per block, reaching up to 3.8× that of Zone Append. This implies that PM1731a buffers ZRWA Writes in device DRAM, rather than persisting them directly to NAND. Note that, even though the data resides in DRAM, ZRWA Writes provide the same durability as Zone Writes, including persistence across power failures and atomic block-write guarantees [1].

Fig. 3(b) and (c) report the average and P99 tail latencies, respectively. For ZRWA Write, we show only ZRWA-1, as all ZRWA variants exhibit nearly identical latency. We observe that ZRWA Write reduces average latency by 71% and P99 latency by 97.8% compared to Zone Append.

4.2 Microbenchmarks

Overall performance. Fig. 4 presents the throughput and P50 latency of ZenFS, WALTZ, and ZWAN across representative microbenchmark configurations. ZWAN achieves the highest throughput in all 30 configurations, outperforming ZenFS by 34% on average (up to 1.9×) and WALTZ by 64% (up to 3.4×). The P50 latency follows the same trend: ZWAN reduces it by up to 67% over ZenFS and up to 62% over WALTZ.

Notably, WALTZ underperforms ZenFS by up to 54% in write-intensive, small-value workloads. Its per-record commit inflates WAL-level WAF to 35 for 100 B KV pairs, 2.5× higher than ZenFS, whose group commit consolidates multiple records. In contrast, ZWAN avoids this tradeoff. ZWAN’s packing keeps WAL-level WAF to only 1.15 at 100 B KV pairs, which is 30× and 12× lower than WALTZ and ZenFS. Even at 4 KB, where all three write one block per record and the WAF advantage diminishes, ZWAN still outperforms, because ZRWA-1’s lowest per-write latency (§4.1) directly raises throughput on every Put’s synchronous WAL path.

Tail latency. Fig. 5 shows the P99, P99.9, and P99.99 tail latencies for the write-only workload under uniform random; other workloads behave similarly. For 100 B KV pairs, ZWAN achieves a P99 latency of 191 μs, which is 4.4× lower than ZenFS (841 μs) and 9.8× lower than WALTZ (1,872 μs). This is consistent with ZRWA Write’s 97.8% P99 reduction over Zone Append (§4.1). WALTZ exhibits the highest P99 latency because its per-record commit policy generates substantially higher I/O command intensity than ZenFS’s group commit.

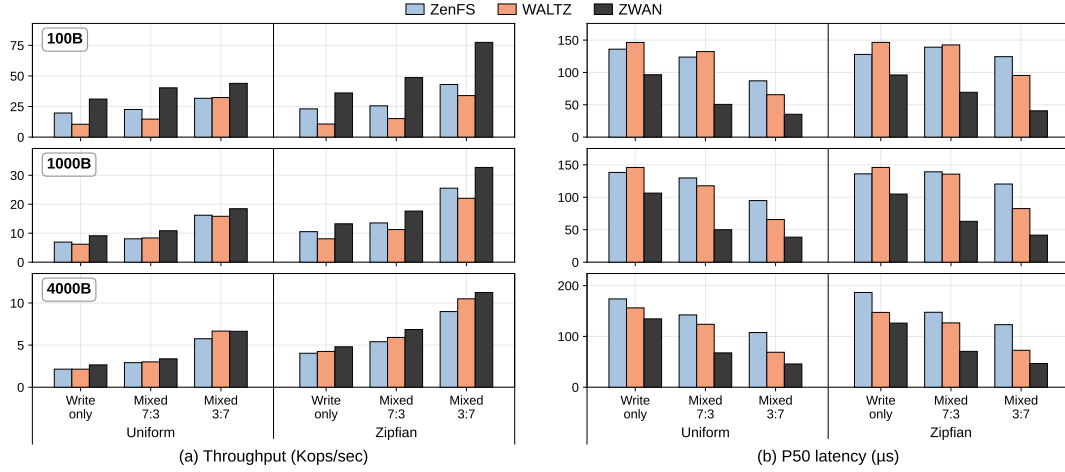


Figure 4: (a) Throughput and (b) P50 latency of the microbenchmarks. Each row fixes the value size, from 100 B (top) to 4 KB (bottom); intermediate sizes (500 B, 2 KB) exhibit consistent trends and are omitted for brevity.

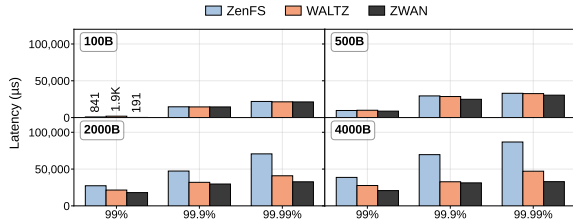


Figure 5: Tail latency of the uniform write-only workload. Each panel fixes the value size from 100 B to 4 KB.

However, all three systems converge to 14 ms and 21 ms at P99.9 and P99.99, respectively. This convergence arises because ZWAN sustains $3.4\times$ higher IOPS than WALTZ at this value size, yielding a higher I/O command rate that, at high percentiles, offsets ZRWA Write’s latency advantage.

For 4 KB KV pairs, ZenFS’s P99 latency reaches 38.7 ms, $1.9\times$ higher than ZWAN’s 20.8 ms, because larger group-commit batches inflate tail latency. WALTZ and ZWAN avoid this batching delay through per-record commits, while ZWAN benefits from the lower latency of ZRWA Write. Unlike at 100 B, the throughput – and thus the command rate – gap narrows at 4 KB, so ZRWA’s latency advantage stands out instead of being absorbed by higher command intensity.

4.3 MixGraph Benchmarks

Overall performance. Fig. 6 presents the throughput, P50, and P99 latency of ZenFS, WALTZ, and ZWAN under the four MixGraph workloads [4]. We additionally evaluate *ZenFS-NM*, a variant of vanilla ZenFS with per-record metadata synchronization disabled. A key optimization in WALTZ is lazy metadata management, which defers metadata updates until zone replacement instead of issuing an fsync on every WAL append [13]. ZenFS-NM therefore isolates the effect of metadata synchronization from WALTZ’s other mechanisms.

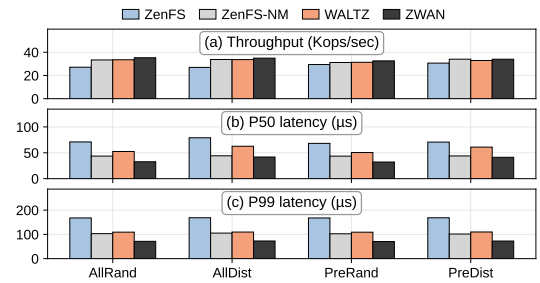


Figure 6: Performance comparison of MixGraph. Table 2: PreDist workload WAF breakdown.

On average, ZWAN achieves 20% higher throughput than ZenFS and roughly halves its P50 and P99 latency. Compared to WALTZ, ZWAN achieves 4% higher throughput and reduces P50 and P99 latency by about 33%. The throughput gain is smaller than in §4.2 because MixGraph workloads have an average write ratio of 14%, reducing the contribution of ZWAN’s WAL-path optimization to aggregate throughput. Still, ZRWA’s latency advantage persists: since every Put traverses the WAL synchronously, ZWAN has the lowest P50 and P99 latency across all four MixGraph workloads.

Among baselines, ZenFS-NM and WALTZ show similar throughput, indicating that the gap between vanilla ZenFS and WALTZ stems from eliminating per-record metadata synchronization. ZenFS-NM also shows lower P50 than WALTZ, consistent with WALTZ’s observation that Zone Write has lower latency than Zone Append at typical percentiles [13]. **WAF Analysis.** Table 2 decomposes the host-issued WAF of the PreDist benchmark into WAL, metadata, and SST. Each

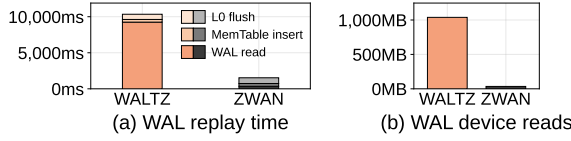


Figure 7: Average recovery costs.

component reports the distinct 4 KB LBAs written, normalized by total user payload. ZenFS incurs a WAF of 197.99, over half from per-record metadata synchronization on every WAL write. WALTZ eliminates this traffic through lazy metadata management, cutting total WAF by nearly half.

Notably, ZenFS and WALTZ now show similar WAL-level WAF, unlike the write-only microbenchmark where ZenFS’s group commit pushed it well below WALTZ’s (§4.2). Mix-Graph’s low Put ratio leaves few concurrent WAL records to batch, negating ZenFS’s group commit benefit. As a result, both ZenFS and WALTZ typically consume one 4 KB block per WAL record; with WAL records averaging only 42.8 B, about 99% of each block is wasted. Although the WAL is only transient metadata, WAL writes alone consume over 20× the device bandwidth of SSTable writes in both baselines.

ZWAN reduces WAL-level WAF to 1.35 and total WAF to 5.83, a 17× reduction over WALTZ. In ZWAN, WAL writes amount to only 30% of SSTable writes. This comes from packing 71 WAL records per 4 KB block on average via ZRWA’s overwrite capability. ZWAN issues one device write per WAL record, as WALTZ does, but its in-place updates are absorbed by the device’s DRAM-backed ZRWA window and effectively coalesced, so only the final packed block reaches NAND.

4.4 Recovery Costs

To evaluate how ZWAN’s WAL packing affects crash recovery, we run `db_bench fillrandom` with 100 B values and terminate the process after a target operation count sampled uniformly between 1 M and 8 M. Fig. 7 reports the results across 100 trials. As shown in Fig. 7(a), ZWAN completes WAL replay 6.7× faster than WALTZ, cutting mean replay time from 10.3 s to 1.54 s. Fig. 7(b) explains this difference. Both systems replay the same logical volume of roughly 30 MB of records, but WALTZ reads 30× more WAL bytes from the device than ZWAN, 1,038 MB versus 35 MB per recovery on average. This is because WALTZ does not pack WAL records, and the resulting read gap roughly matches the WAL-level WAF gap reported in §4.2. The WAL read phase accounts for 89.4% of WALTZ recovery time, compared with only 22.6% for ZWAN. Thus, ZWAN reduces not only WAF but also recovery-time read amplification.

We verify recovery correctness in two steps: (1) the largest recovered sequence number covers every write acknowledged before the crash, and (2) a checksummed full scan validates all SST keys, including every record replayed from the WAL. All trials pass both checks with zero loss.

5 Discussion

ZNS-Specific Benefits of ZWAN. The idea of packing is not ZNS-specific. A conventional SSD’s write buffer can absorb the same overwrites and provide a similar reduction in WAL-level WAF. However, the device-internal benefit is particularly pronounced on ZNS SSDs. On a conventional SSD, the host cannot control buffer eviction, so WAL and SSTable data inevitably co-locate in the same garbage collection (GC) units. Because WAL records expire much earlier than SSTable data, this inflates device-internal WAF during GC. ZNS SSDs instead isolate zones architecturally, letting applications separate WAL and SSTable data. ZRWA extends this separation into the device’s DRAM-backed buffer by assigning each open zone its own region. The host can implicitly retain data in this buffer by refraining from advancing the ZRWA window. ZWAN leverages this control to retain partially filled WAL blocks, so each block is expected to be flushed to NAND only after it has been fully packed.

Sensitivity to ZRWA parameters. Devices may expose ZRWA parameters different from the PM1731a’s 64 KB window (W) and 32 KB flush granularity (F). We evaluate all combinations of $W \in \{32, 40, 48, 56, 64\}$ KB and $F \in \{32, 64\}$ KB via host-side emulation, confining ZWAN to the first W bytes of the window and issuing Explicit Flushes in units of F . Under the same workload as in §4.2, performance remains nearly unchanged across configurations: throughput within 2.2%, P50 within 0.5%, and P99 within 6.1%. Even the minimal window ($W=F=32$ KB) sustains full throughput, suggesting ZWAN’s benefit is insensitive to the ZRWA parameters.

6 Conclusion

ZWAN leverages the in-place overwrite capability of ZRWA to eliminate the WAL tax in LSM-tree key-value stores on ZNS SSDs. By packing multiple WAL records into a single ZRWA block, ZWAN effectively eliminates sub-block WAF without incurring the batching delay of group commit. ZWAN further exploits the performance benefits of ZRWA under concurrent writers through ownership-based page allocation and WP-aware prioritization. Across diverse workloads, ZWAN exhibits superior performance and lower WAF than existing ZNS-aware LSM-tree designs.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Research Foundation of Korea (NRF) grant (No. RS-2023-00222663), and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-II211363), funded by the Korean government (MSIT). This work was also supported in part by a research grant from Samsung Electronics.

References

- [1] 2025. NVM Express® Zoned Namespace Command Set Specification. <https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-Revision-1.4-2025.08.01-Ratified.pdf>.
- [2] Matias Björling. 2020. Zone Append: A New Way of Writing to Zoned Storage. In *USENIX Vault '20*. Santa Clara, CA.
- [3] Matias Björling, 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] 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.
- [5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
- [6] Minkyu Choi, Joonseong Hwang, Minjin Park, and Seokin Hong. 2026. Braid-ZNS: Leveraging Zone Random Write Area for Efficient In-Storage Compression on ZNS SSDs. In *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*.
- [7] 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.
- [8] Facebook. [n. d.]. Performance Benchmarks. <https://github.com/facebook/rocksdb/wiki/Benchmarking-tools>.
- [9] Inhwil Hwang, Sangjin Lee, Sunggon Kim, Hyeonsang Eom, and Yongseok Son. 2025. Z-LFS: A Zoned Namespace-tailored Log-structured File System for Commodity Small-zone ZNS SSDs. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 547–562.
- [10] Intel. [n. d.]. Storage Performance Development Kit. <https://spdk.io>.
- [11] Minwook Kim, Seongyeop Jeong, and Jin-Soo Kim. 2025. ZRAID: Leveraging Zone Random Write Area (ZRWA) for Alleviating Partial Parity Tax in ZNS RAID. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1151–1165.
- [12] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.
- [13] 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.
- [14] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 461–477.
- [15] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta informatica* 33, 4 (1996), 351–385.
- [16] RocksDB Wiki. 2020. WAL Performance. <https://github.com/facebook/rocksdb/wiki/WAL-Performance>.
- [17] Samsung. [n. d.]. Samsung Introduces Its First ZNS SSD With Maximized User Capacity and Enhanced Lifespan. <https://semiconductor.samsung.com/news-events/news/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan/>.
- [18] Shushu Yi, Shaocong Sun, Li Peng, Yingbo Sun, Ming-Chang Yang, Zhichao Cao, Qiao Li, Myoungsoo Jung, Ke Zhou, and Jie Zhang. 2024. Biza: Design of self-governing block-interface zns afa for endurance and performance. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 313–329.