HZFS: Breaking the Dilemma of High Performance and Order-Preservation for ZNS-based Flash Storage

Zili Ye*, Nan Zhang*, Yanqi Pan*, Wen Xia*†‡
*Harbin Institute of Technology, Shenzhen, China
†Pengcheng Laboratory, Shenzhen, China

***Abstract*—To comply with the Zoned Namespace (ZNS) sequential write constraint, current I/O stacks preserve the request order through blocking, resulting in performance degradation.**

To address this problem, we propose the Large-Zone Parallelism (LZP) I/O scheduler, which eliminates blocking-based order preservation through a centralized order control mechanism. Building on this scheduler, we develop HZFS, which leverages an orderless submission method and a multi-stream co-update mechanism to enhance intra- and inter-zone parallelism. Evaluations show HZFS outperforms state-of-the-art POSIX-compliant ZNS file systems by 13.6×-27.6× in write bandwidth, and reduces the 99th-percentile latency by 53%-63%.

I. Introduction

Zoned Namespace (ZNS) [1] flash storage is an emerging flash storage designed to mitigate the write-amplification problem [2] of conventional flash storage. Its defining feature is the partitioning of physical storage into independent logical zones, granting the host direct control over programming, reset, and management operations. Compared to conventional flash, ZNS SSDs eliminate device-internal garbage collection [3], [4] and avoid the log-on-log problem [5], thus offering more stable latency and a longer device lifetime. Specifically, large-zone ZNS SSDs use coarse-grained logical partitioning to simplify logical-to-physical (L2P) mapping; more critically, large zones span multiple dies and internal channels, delivering significantly higher intra-zone parallelism [6]. With simpler management methods and higher zone bandwidth, large-zone ZNS SSDs have been applied in database and cloud service scenarios [6]–[8].

However, to comply with the sequential write constraint of the ZNS interface (referred to as the ZNS constraint) and to maintain crash consistency, existing ZNS I/O stacks heavily serialize update streams. This design severely underutilizes the parallelism potential of large-zone ZNS SSDs, leading to significant performance degradation. The inefficiency manifests in two dimensions: intra-zone and inter-zone parallelism.

(1) Intra-zone parallelism remains underutilized due to the coarse-grained request ordering employed by current I/O stacks to respect ZNS constraints. In ZNS storage, data must be written strictly to the zone's write pointer, which advances after each write. Any attempt to write before or after the pointer results in an I/O error (except in conventional zones or zone random write areas). Nevertheless, the ZNS specification [1] does not guarantee that incoming requests will be processed by the SSD controller in their transmission order. Consequently, the multi-queue and out-of-order scheduling mechanisms in the Linux block layer, which are designed to improve I/O parallelism, can violate the ZNS sequential write constraint. To preserve the order of requests, existing I/O schedulers [9] and file systems [10], [11] fall back to a blocking strategy that serializes request streams, limiting each zone to serving only one outstanding write request at a time (i.e., I/O depth = 1). This coarse-grained ordering wastes the parallel I/O capabilities of ZNS SSDs and significantly degrades application performance. Our measurements show that such serialization reduces the bandwidth of a single zone by 66%–95%.

(2) Inter-zone parallelism is underutilized due to coarse-grained data-metadata synchronization. To reduce garbage collection overhead, existing file systems [10], [12]–[14] typically store data and metadata in different zones, which provides the potential for parallel I/O between them. However, current ZNS file systems strictly synchronize cached data in "data-metadata" order, which means that metadata update requests can only be submitted after the corresponding data updates are completed. This restriction prevents ZNS file systems from effectively exploiting inter-zone concurrency in buffered I/O and sync workloads, resulting in a 1.7×–17.8× performance gap compared to the ideal case.

The root cause is that current I/O stacks, including file systems and I/O schedulers, typically build on multi-queue and out-of-order scheduling mechanisms, which can violate the ZNS constraint. To accommodate ZNS SSDs, the Linux block layer restricts the request ordering through a blocking approach, leading to dramatic performance degradation.

To address this issue, we propose LZP, an I/O scheduler for **L**arge-**Z**one **P**arallelism, which replaces blocking-based order preservation with centralized order control. Building on LZP I/O scheduler, we design HZFS, a **H**igh-performance **Z**NS **F**ile **S**ystem that effectively exploits hardware parallelism. To improve the utilization of intra-zone parallelism, HZFS introduces an orderless submission method, which allows the file system to submit multiple requests to a single zone while letting the scheduler assign addresses and enforce ordering. To overcome the limitation of inter-zone parallelism, HZFS employs a data–metadata co-packaging technique, enabling data

‡Corresponding author. Email: xiawen@hit.edu.cn.

and metadata update requests to be submitted simultaneously. The evaluation results show that, compared to existing ZNS file systems, HZFS improves throughput by up to 27.6× and reduces average latency by 63%. In summary, this paper makes the following contributions:

- Root cause analysis. We identify the fundamental cause of the ZNS I/O stack's performance degradation as its over-reliance on blocking to respect ZNS write constraint. Further, our analysis reveals that strict request ordering does not inherently require such coarse-grained blocking mechanisms.
- Design of HZFS. We propose LZP I/O scheduler that allows multiple write requests to be submitted to ZNS SSDs in a non-blocking and ordered manner. Building on the LZP I/O scheduler, we design HZFS, a file system that achieves high performance while adhering to the ZNS constraint. HZFS further introduces a set of techniques to improve both intra-zone and inter-zone parallelism.
- Implementation and evaluation. We implement HZFS and conduct extensive experiments. Evaluations show that HZFS substantially improves bandwidth utilization on ZNS flash storage, achieving performance far beyond state-of-the-art ZNS file systems.

II. Background

ZNS write constraint. The physical space of ZNS flash storage is divided into multiple logical zones that support random reads but only sequential writes. Each zone maintains a write pointer(WP), which advances after data is written. Any attempt to write before or beyond the WP results in an I/O error. Once a zone is full, it must be reset before new data can be written. This constraint matches the "erase-before-program" characteristic of flash media, enabling a much simpler design by eliminating per-zone address mapping tables and garbage collection. In particular, large-zone ZNS SSDs further simplify zone management (e.g., mapping tables and allocation tasks) through coarse-grained logical partitioning and hardware-level striping. Moreover, a larger zone can span more dies and internal channels, thereby offering higher intra-zone performance [6], [15]. At present, large-zone SSDs have been widely deployed in data centers and cloud service scenarios [7], [8].

ZNS file systems. Building on append-only logging, Log-Structured File Systems (LFS) naturally comply with ZNS SSDs' sequential write constraint, making them a prevalent fit for ZNS-enabled storage. To better utilize ZNS SSDs' performance, existing ZNS file systems [10]–[14], [16]–[18] often adopt multi-zone parallel I/O strategies. For example, XFS [11] stripes file data across multiple zones to improve bandwidth utilization. F2FS [10] maintains six major log areas (HOT/WARM/COLD nodes and data), and allocates one active zone to each of them. Z-LFS [13] distributes multiple active zones across log streams to further utilize inter-zone parallelism. It also applies a conflict-aware zone allocation mechanism to mitigate the zone interference problem [19], [20]. ZenFS is a user-space file system for ZNS-based storage, typically serving as the storage backend for RocksDB. It [21] spreads the SSTables across multiple independent zone groups for high-level hardware parallelism. However, ZenFS is implemented under the RocksDB FileSystem wrapper API and does not support POSIX file system operations, limiting its broader applicability. In this work, we mainly focus on POSIX-compliant file systems that support ZNS SSDs and enable multiple applications to access storage with common file semantics.

III. Observation and Motivation

A. Observation

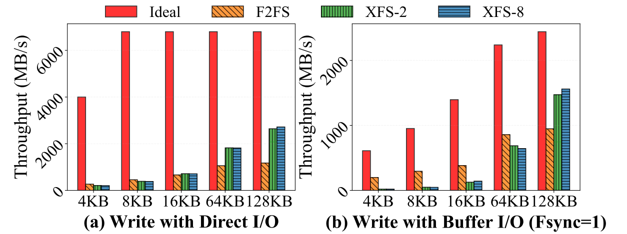

Fig. 1. **Motivation test**. *XFS-n: using n active zones.*

While multi-zone parallel I/O strategies are extensively adopted in existing ZNS file systems, we find that state-of-the-art systems like XFS [11] and F2FS [10] exhibit significant performance gaps compared to the storage device's maximum throughput (see Figure 1). We first characterize the performance upper bounds of intra- and inter-zone parallelism on our large-zone SSD using FIO [22]: Ideal intra-zone bandwidth is measured via FIO with direct I/O, sequential writes, queue depth=8, and single-threaded access to one zone. Ideal inter-zone bandwidth uses 8 parallel threads, each writing to a separate active zone. Both scenarios achieve near-maximum bandwidth with only modest parallelism (ideal bars in Figure 1(a)-(b)). We then perform direct I/O experiments on F2FS and XFS file systems that allocate a dedicated zone for each data log stream (i.e., file data writes). Our results demonstrate that their throughput is up to 23.7× lower than the intra-zone ideal. We further conduct `write()` + `fsync()` experiments, which encompass both metadata and data log streams (multiple log streams), but performance remains suboptimal—indicating inadequate exploitation of inter-zone parallelism. Specifically, for XFS, increasing the number of active zones from 2 to 8 yields negligible performance improvement over XFS-2, highlighting inherent limitations in inter-zone parallelism.

B. Analysis of the I/O Path of ZNS File Systems

To explore the root cause of the performance bottleneck, we revisit the I/O path of existing ZNS stacks and identify the I/O stack's coarse-grained ordering mechanism as the primary driver of ZNS SSD performance underutilization with two key manifestations.

Intra-zone parallelism is sacrificed to enforce ZNS constraint. We observe that file systems alone are insufficient to guarantee the ZNS constraint. As shown in Figure 2, due to the multi-queue and out-of-order architecture of the block layer, the I/O stack cannot ensure that write requests reach

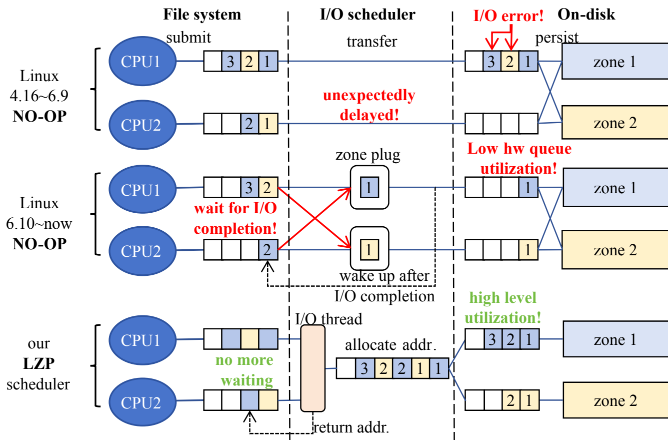

Fig. 2. **The I/O path of different ZNS I/O stacks.** *The numbers in the block denote the write request target address on the target zone, which needs to align with the zone WP. The color of the blocks indicates the target zones. NO-OP: dispatching requests to devices without any scheduling mechanism, also referred to as "none".*

the device in the same order as they were submitted from the file system [23]. For example, consider two threads (running on CPU 1 and CPU 2, respectively) intending to write both zone 1 and zone 2. The file system appends data at the zone's WP and submits requests in order. Figure 2 illustrates a scenario where, due to variable CPU scheduling delays, requests from CPU 1 may arrive at the SSD before those from CPU 2, violating the ZNS sequential write constraint even though they were issued in order. As a result, CPU 1's last two write requests will attempt to write beyond the current WP and incur I/O errors. To prevent these errors, the Linux ZNS I/O stack delegates WP maintenance to the file system (or application) while tasking the block layer with ensuring that the I/O submission order strictly aligns with the pointer allocation order. Starting from Linux 6.10, a Zone Write Plugging mechanism [24] was introduced to support multi-queue submission of requests targeting different zones while adhering to ZNS constraints. However, these approaches still apply the Transfer-and-Flush mechanism [23] for the intra-zone order guarantee: write requests to the same zone must await the prior request's completion to proceed. This coarse-grained ordering caps each zone's I/O depth at 1, severely limiting intra-zone parallelism.

Inter-zone parallelism is sacrificed to preserve consistency ordering. During `fsync()`, ZNS file systems [10], [12], [13] typically apply the soft update mechanism to ensure atomic metadata update, which necessitates a synchronous flush command between the data and node phases. Such a "Transfer-and-Flush" mechanism effectively serializes I/O requests across different zones. Consequently, even when data and metadata are stored in separate zones, the I/O stack remains stalled by these mandatory synchronization barriers, leading to substantial latency and underutilized inter-zone bandwidth.

C. Motivation

Attempts. Several efforts have been made to reduce the overhead of enforcing strict write-order control:

(1) **Order-preserved I/O stack**. Early efforts, such as Barrier-enabled I/O stacks [23], [25], introduced barrier request types to maintain write orders. However, these are primarily designed for single-queue storage devices (e.g., eMMC, SATA) and suffer from significant performance degradation in multi-queue environments due to the overhead of global queue draining. More recently, OPIMQ [26] extended order preservation to multi-queue block devices via a modified on-device Flash Translation Layer (OPFTL). However, while OPIMQ relies on device-side address remapping and delayed mapping updates to hide out-of-order execution, ZNS exposes physical sequential write requirements to the host and removes the on-device page-mapped FTL, making OPIMQ either functionally invalid on native ZNS SSDs. Another ZNS-oriented order-preserving work, oZNS, used a lightweight indirect layer to record write offsets within the SSD, maintaining zone WP alignment even under out-of-order writes. However, it also requires modification of the SSD firmware.

(2) **Zone Append (ZA) command**. Alternatively, the ZA command [1], [27]–[29] allows concurrent write submissions to a single zone by delegating order enforcement to the SSD hardware controller. However, under standard ZA mode, the host cannot determine the final Logical Block Address(LBA) until the SSD returns the allocated address post-completion, forcing synchronous updates to file indexing structures(e.g., data pointers in inodes). Compounding this issue, recent studies [14], [30] have shown that existing ZA hardware implementations exhibit suboptimal performance. Furthermore, for ZNS SSDs lacking native ZA support, the Linux block layer emulates it via Zone Write command, which still relies on blocking-based ordering to preserve request sequence. Collectively, these limitations render ZA ineffective at unlocking intra-zone parallelism.

(3) **Logical-to-physical zone remapping** [6], [13], [18] maps multiple physical zones to a single logical zone, enabling logical intra-zone parallelism via fine-grained data striping across these physical zones. However, requests to each physical zone still require the prior request to be completed before proceeding, thus failing to fully leverage intra-zone parallelism. Moreover, large-zone SSDs usually expose a limited number of active zones, making this approach impractical.

Key Insight. Building on this finding, we propose an optimized ZNS I/O stack centered on centralized order control—letting the I/O scheduler exclusively dictate the final write order without any SSD firmware modification. To materialize this, we design the LZP scheduler around two core principles: (1) write requests are submitted to the I/O scheduler in an unordered manner, with their final write addresses determined exclusively by the scheduler; and (2) each zone is constrained to accept write requests only from a single hardware queue. The second principle is practical because SSD controllers process requests in each hardware queue in strict FIFO order—a straightforward implementation aligned with mainstream SSD design practices.

However, deploying a ZNS file system on top of this scheduler raises two challenges:

- **Challenge 1**—Fast Notification of the Actual Write Address: Similar to Zone Append, the orderless submission method forces file systems to await I/O completion for the actual write address, limiting the subsequent file index update.
- **Challenge 2**—Exploiting Inter-Zone Parallelism under Data–Metadata Synchronization: The inherent two-phase data–metadata synchronization in ZNS file systems serializes operations across zones.

IV. The Design

A. Overview and Design Principles

Driven by the insights in §III-C, we design the LZP scheduler. Building on this foundation, we develop HZFS, incorporating an orderless submission method and a multi-stream co-update mechanism.

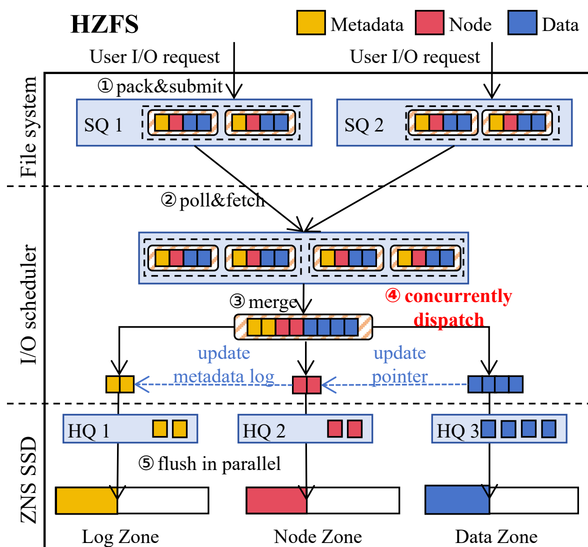

Fig. 3. **The Architecture and workflow of HZFS.** *SQ: Software submission queue. HQ: hardware submission queue.*

Disk layout overview. HZFS adopts a disk layout analogous to Z-LFS [13] (a recent F2FS variant). It consists of two core areas: the Metadata Area (metadata zone for global metadata like superblock/NAT/SIT/SSA, and log zone for incremental metadata updates) and the Main Area (node zone for node blocks, data zone for file data blocks). Note that HZFS does not rely on conventional SSDs for metadata storage. On `fsync()`, modified metadata entries are merged into a log block and appended to the log zone.

Multi-stream overview. In HZFS, each synchronization involves updates to three types of data: file data, nodes, and global metadata, which belong to different write streams (also known as multi-head logging). The global metadata includes updates to entries such as NAT/SIT associated with the synchronization request, which are merged into a single log block. Unlike existing ZNS file systems, HZFS merges updates from different streams into a single request.

Workflow overview. We adopt a per-CPU Software Queue (SQ) model with centralized fetching, avoiding lock contention through non-blocking notification. Each zone maps to a dedicated Hardware Queue (HQ), preserving FIFO semantics without requiring global locks. As shown in Figure 3, after a user thread issues an I/O request, HZFS (①) packages the data, nodes, and metadata log into a single request, then submits it to a per-CPU SQ and notifies the scheduler. Then, the scheduler (②) concurrently fetches all requests from the SQs. Next, the scheduler (③) merges the three types of data, assigns physical addresses to each, updates the pointers in the nodes, and records the global metadata updates into the log. Finally, the scheduler (④) dispatches the requests concurrently to per-zone HQ, where the SSD controller (⑤) flushes them in parallel to the flash medium.

Design Principles. (1) Non-blocking order preservation: The scheduler assigns addresses to file system requests (submitted without preassigned addresses) and submits them directly to HQs, eliminating waiting for prior completion to unlock intra-zone parallelism. (2) Stream merging for inter-zone parallelism: The file system merges multi-stream requests before submission, enabling concurrent dispatch across zones.

B. Large-Zone Parallelism I/O scheduler

In §III-B, we identified that the root cause of the poor intra-zone parallelism lies in the mutual unawareness among order-altering points in the existing I/O stack, forcing each point to reorder write requests through blocking. To address this issue, we design the LZP I/O scheduler. Compared with the Linux scheduler, the LZP scheduler redesigns two key components: (i) request scheduling and (ii) queue mapping.

Request scheduling. After a user I/O thread issues an update request, the file system submits the dirty cache pages to the thread's SQ and notifies the I/O scheduler. Once submitted, the file system can continue processing the next update request without waiting. Upon receiving the notification, the scheduler fetches all pending write requests from the SQ of the notifying CPU and tries to merge them.

During request merging, the scheduler separates data pages belonging to different write streams and dispatches them to the corresponding HQs. Finally, the controller persistently writes all queued requests in each HQ to the flash medium in order.

Queue mapping. To reduce contention among multiple user I/O threads on submission queues, HZFS maintains an SQ for each CPU. All requests from the SQs are fetched by the scheduler into a temporary queue for merging and then dispatched to the target zones in submission order. Further, we restrict each zone to receive requests from a single HQ. The LZP scheduler ensures that all write requests to the zone are submitted to the corresponding HQ in order. When the current active zone becomes full, the file system notifies the scheduler to switch to a new zone. The scheduler then opens an empty zone and remaps the HQ from the full zone to it.

C. Orderless Submission Method

Unlike Zone Append–based file systems [11], [16], the LZP scheduler dictates the final write order in the host-side, rather than the SSD controller. Once the LZP scheduler dispatches write requests to the HQ, it immediately updates

and returns the new write pointer wp, eliminating the need for the file system to wait for request completion. Further, the LZP scheduler actively merges multiple write requests from the same thread, and allocates contiguous address space for them. For example, suppose a user needs to flush n data pages $\{p_1, p_2, \ldots, p_n\}$ and synchronize them. HZFS merges these pages into a single request, submits it to the CPU's SQ, and notifies the I/O scheduler. The scheduler then fetches the request and allocates space for each page sequentially from the target zone's wp. For each data page p_i, HZFS computes its final write location as $LBA_i = wp + (p_i - p_1) \times page_size$.

After allocation, the scheduler immediately returns the updated wp to HZFS, allowing subsequent operations to proceed. HZFS can compute the final LBA of each page through simple computation, without waiting for I/O completion. Based on the orderless submission method, HZFS can continuously submit write requests to the zone, thereby effectively exploiting intra-zone parallelism.

D. Multi-stream Co-update Mechanism

Current ZNS file systems submit update requests during synchronization in a sequential order of "data →node →global metadata", where each subsequent phase must wait for the previous phase to complete. Furthermore, when synchronizing global metadata, different metadata types also wait for each other. This serialization of different write streams undermines inter-zone parallelism. To address this, we design new request and metadata structures that break the synchronization order across different data types and within global metadata. We then develop a new process for handling merged requests.

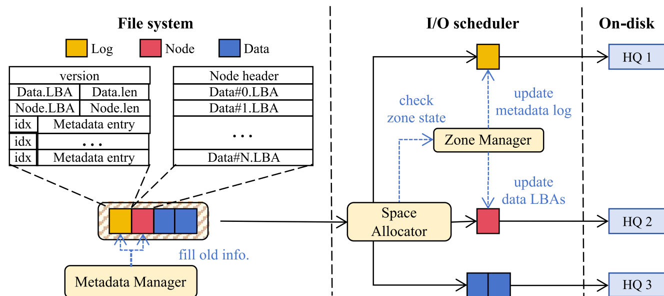

Fig. 4. **The process of Multi-stream Co-update.** *Node header consists of node information, such as node ID.*

Co-update request and metadata log structure. HZFS generates a log block for each operation to record the updated metadata. As shown in Figure 4, the log block contains all global metadata modifications required by the sync operation, including the write ranges of data and nodes, as well as updated global metadata entries. Importantly, due to the orderless submission method, HZFS does not know the exact data write locations before submitting the request. Therefore, metadata related to allocation information in the log block is not updated at submission time. The I/O scheduler will complete these metadata entries after assigning the space.

Co-update request processing. During a `fsync()` operation, HZFS adds the updated file data pages as well as the old node pages to a co-update request. HZFS then generates a log block for the operation and also adds it to the request. The old metadata entries are copied to the log block and assigned a globally incremented version number. After the request is submitted, the scheduler separates the data of different streams and allocates space in the corresponding zones. During the dispatch to HQs, the scheduler first dispatches the data pages, then updates the LBAs in the node pages and dispatches them, and finally updates the metadata entries based on the new data write addresses and zone status (WP, free space, etc). There is no waiting between the dispatches of different data types, enabling a high level of multi-stream parallelism.

E. Implementation

We implemented the LZP scheduler using SPDK to directly issue I/O requests to hardware queues (HQs). On top of LZP, we implemented HZFS as a user-space file system that bypasses the Linux block layer.

Crash Consistency and Recovery. HZFS employs log entries to record the LBA and length of data and node batches. A `fsync()` operation is considered complete only after the corresponding log is successfully appended to the log zone. Drawing inspiration from OPRW [31], HZFS leverages the WP for recovery. Data/node block is deemed valid only if:(1)its corresponding log block exists, (2)its address recorded in the log is less than the current WP of its respective zone. In cases where log entries contradict the WP position, HZFS appends a new invalidation log during recovery to mark the inconsistent region as invalid, ensuring it is subsequently reclaimed by the garbage collection process.

Garbage Collection. Both the main area and the metadata area of HZFS require garbage collection. For the main data area (Data/Node), HZFS adopts the foreground and background reclamation mechanisms from F2FS. For the metadata area (Global Metadata), once the log zone is full, HZFS merges the updated logs with the old metadata entries in the Metadata Zone and writes them into a new metadata zone, while discarding the obsolete or invalid block.

V. Evaluation

A. Experimental Setup

Testbed. Our experimental platform is equipped with a 64-core Intel Xeon Gold 6530 CPU and 128 GB of main memory. We use a Dapustor [32] ZNS SSD with 413 zones (17.72 GB per zone), providing 12.91 TB of total capacity and supporting up to 15 active zones. For ZNS file systems requiring in-place metadata updates, we use a conventional SSD (128 GB). The system runs Linux kernel 6.15.7, with *f2fs-tools 1.15.0* and *xfsprogs 6.15.0* for mkfs. We use FIO [22] for microbench, each write benchmark spans at least 50 zones (886 GB).

Comparison. We compare HZFS with F2FS and XFS (with 8 active zones). We also implement 3 user-space variants of HZFS: (i) HZFS-native (HZFS-N), an unoptimized implementation of HZFS that follows existing file systems by preserving write order through blocking. (ii) HZFS-intra-only, which waits for the completion of the previous write stream before issuing new requests, and (iii) HZFS-inter-only, which

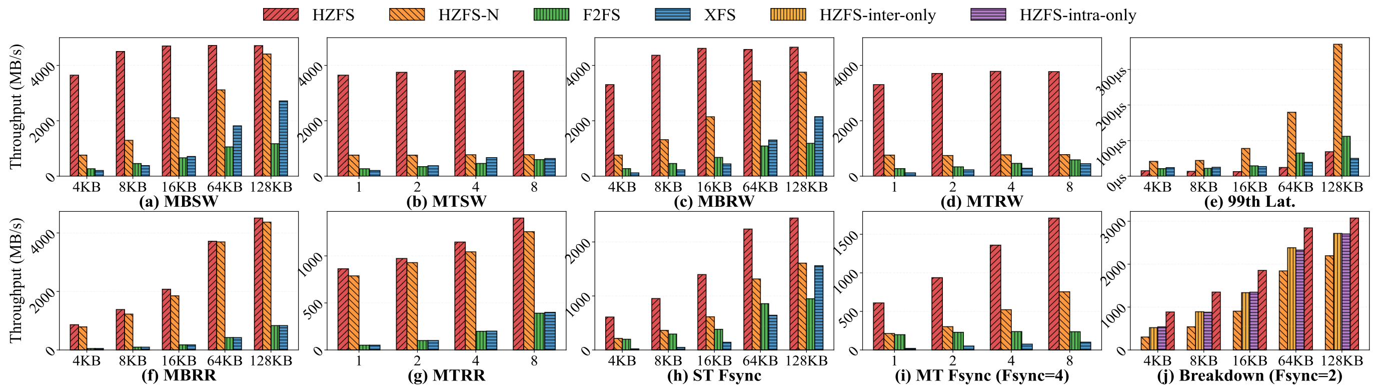

Fig. 5. **Performance overview of HZFS.** *(a) MBSW (Multi-Blocksize Sequential Write); (b) MTSW (Multi-Thread Sequential Write); (c) MBRW (Multi-Blocksize Random Write); (d) MTRW (Multi-Thread Random Write); (e) 99th Percentile Latency of 4KB Sequential Write; (f) MBRR (Multi-Blocksize Random Read); (g) MTRR (Multi-Thread Random Read); (h) Single-Thread Fsync; (i) Multi-Thread Fsync; (j) Breakdown Analysis. Unless otherwise specified, all experiments are conducted with a block size of 4KB and a single thread.*

assigns physical block addresses in the file system and waits for the target zone writes to complete. We omit the logical-to-physical remapping file systems (such as Z-LFS [13]) from comparison, as they need a large number of active zones, which the large-zone SSD cannot provide.

B. Write & Read Performance

In this section, we conduct read and write throughput tests on different file systems. In these experiments, user threads will issue requests directly to the ZNS SSD via Linux direct I/O (`ODIRECT`). Figure 5 (a)-(d) show that HZFS improves the performance by up to 13.6×, 4.8×, 17.8× in MBSW; 12.1×, 4.95×, 27.6× in MTSW; 12.1×, 4.3×, 17.8× in MBRW; 12.1×, 5.0×, 27.6× in MTRW compared with HZFS-N, F2FS, XFS. As expected, HZFS outperforms Linux file systems in sequential and random write workloads. Although some file systems attempt to speed up writes by exploiting multiple I/O zones (XFS), they still suffer from poor intra-zone parallelism. Thanks to the LZP I/O scheduler, HZFS can submit writes to a zone without blocking, and thus achieves extremely high performance even with small block sizes and low-thread workloads. Figure 5 (e) shows the 99th percentile latency. HZFS achieves the lowest tail latency in the 4KB-64KB test. As block size increases, the tail latency of HZFS and F2FS is higher than that of XFS. This is because both HZFS and F2FS merge writes in main memory even under `O_DIRECT`, whereas XFS submits requests directly. We also conducted read performance tests across four file systems, as shown in Figure 5 (f) and (g). HZFS achieves the highest throughput when reading directly from the ZNS SSD. The result is expected because HZFS is implemented in user space and bypasses the overhead of Linux page cache management.

C. Buffered-Write & Fsync Performance

We evaluate the throughput of four file systems under *buffered write + fsync workloads*. Figures 5 (h) and (i) show that HZFS consistently delivers the highest performance among all file systems. This improvement is mainly attributed to two factors: (i) HZFS eliminates the strict sequential flush constraint between data and metadata phases, and (ii) the LZP scheduler achieves much higher intra-zone parallelism through a non-blocking ordering mechanism. Moreover, the throughput of all Linux file systems shows limited improvement even as the number of threads increases, further confirming that serialized data–metadata synchronization restricts scalability.

D. Breakdown Analysis

Figure 5 (j) illustrates the performance improvement contributed by each optimization technique in HZFS under different block sizes. HZFS-N represents the baseline performance of the user-space file system without any optimizations applied. HZFS-inter-only shows the performance when inter-zone parallelism is enabled, allowing metadata and data to be written concurrently during `fsync()`. HZFS-intra-only represents the performance when write I/Os are allowed to execute concurrently. HZFS reports the performance with all optimizations enabled. As shown in Figure 5 (j), both the inter-only and intra-only variants achieve nearly identical performance gains, as they each reduce blocking overhead from different aspects, and both eliminate exactly one blocking point per `fsync()` cycle. With all optimizations applied, HZFS achieves 2.93×, 2.5×, 2.05×, 1.54×, and 1.4× speedups over HZFS-N for different block sizes. The speedup decreases with larger block sizes because the relative impact of blocking overhead becomes smaller for larger block sizes.

VI. Conclusion

Existing ZNS file systems suffer from a 66–95% performance degradation due to the mismatch between the ZNS sequential write constraint and the multi-queue, out-of-order I/O stack. We propose the LZP I/O scheduler, which allows non-blocking, in-order write submissions. We present HZFS, a high-performance ZNS file system exploiting both intra- and inter-zone parallelism, which achieves up to 27.6× bandwidth improvement over F2FS.

VII. Acknowledgment

This research was partly supported by the Major Key Project of PCL under Grant PCL2024A05, the National Natural Science Foundation of China under Grant 62472127.

REFERENCES

[1] NVM Express, Inc., "NVM Express Zoned Namespace Command Set Specification," https://nvmexpress.org/standards/zoned-namespaces/.

[2] N. Agrawal and V. Prabhakaran, "Design tradeoffs for ssd performance," in *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008, pp. 57–70.

[3] M. Bjorling and A. Aghayev, "ZNS: Avoiding the block interface tax for flash-based ssds," in *Proceedings of the 2021 USENIX Annual Technical Conference*, 2021, pp. 689–703.

[4] T. Stavrinos and D. S. Berger, "Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete," in *HotOS '21: Workshop on Hot Topics in Operating Systems*. ACM, 2021, pp. 144–151.

[5] J. Yang and N. Plasson, "Don't stack your log on my log," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW*. USENIX Association, 2014.

[6] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, "ezns: An elastic zoned namespace for commodity ZNS ssds," in *17th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2023, pp. 461–477.

[7] Western Digital Corporation, "Zenfs: A file system plugin for rocksdb optimized for zns ssds," https://github.com/westerndigitalcorporation/zenfs/, 2025, accessed: 2025-09-14.

[8] Y. Zhou and E. Xu, "CSAL: the next-gen local disks for the cloud," in *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024*, 2024, pp. 608–623.

[9] Zoned Storage Initiative, "Linux scheduling for zoned storage," https://zonedstorage.io/docs/linux/sched, 2025.

[10] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 273–286.

[11] LWN.net and Logan, Darrick J., "Rfc: Xfs: support for zoned devices," https://lwn.net/Articles/1001751/, 2024.

[12] D. R. Purandare, S. Schmidt, and E. L. Miller, "Persimmon: an append-only zns-first filesystem," in *41st IEEE International Conference on Computer Design*, 2023, pp. 308–315.

[13] I. Hwang and S. Lee, "Z-LFS: A zoned namespace-tailored log-structured file system for commodity small-zone ZNS ssds," in *Proceedings of the 2025 USENIX Annual Technical Conference*, 2025, pp. 547–562.

[14] J. Zhang and H. Hu, "Schinfs: A file system integrating functions of the block I/O scheduler for ZNS ssds," in *42nd IEEE International Conference on Computer Design*. IEEE, 2024, pp. 324–331.

[15] H. Bae and J. Kim, "What you can't forget: exploiting parallelism for zoned namespaces," in *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems*. ACM, 2022, pp. 79–85.

[16] N. Aota, "[patch 00/17] zns support for btrfs," https://lwn.net/Articles/865988/, 2021.

[17] W. Qi and Z. Tan, "Blzfs: Crash consistent log-structured file system based on byte-loggable zone for zns ssd," in *ICCD*, 2023, pp. 206–213.

[18] J.-Y. Hwang and S. Kim, "ZMS: Zone abstraction for mobile flash storage," in *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2024, pp. 173–189.

[19] H. Shin, M. Oh, G. Choi, and J. Choi, "Exploring performance characteristics of ZNS SSDs: Observation and implication," in *2020 9th Non-Volatile Memory Systems and Applications Symposium*. IEEE, 2020.

[20] H. Bae, J. Kim, M. Kwon, and M. Jung, "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*. ACM, 2022, pp. 79–85.

[21] M. Oh and S. Yoo, "Zenfs+: Nurturing performance and isolation to zenfs," *IEEE Access*, vol. 11, pp. 26 344–26 357, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3257354

[22] Fio Development Community, "Fio documentation," https://fio.readthedocs.io/en/latest/fio_doc.html, 2024, accessed: 2025-09-15.

[23] Y. Won and J. Jung, "Barrier-enabled IO stack for flash storage," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, 2018, pp. 211–226.

[24] D. Le Moal, "Zone write plugging ([patch v3 00/30])," https://lwn.net/Articles/966987/, 2024.

[25] Y. Zhang and H. Zhang, "Lazybarrier: Reconstructing android IO stack for barrier-enabled flash storage," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2024, pp. 601–615.

[26] J. Kim and J. Oh, "OPIMQ: Order preserving IO stack for Multi-Queue block device," in *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, 2025, pp. 425–439.

[27] B. D. Team, "Zoned mode — btrfs documentation," https://btrfs.readthedocs.io/en/latest/Zoned-mode.html, n.d., accessed: 2024-05-14.

[28] J. Lee, D. Kim, and J. W. Lee, "Waltz: Leveraging zone append to tighten the tail latency of lsm tree on zns ssd," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 2884–2896, 2023.

[29] K. Doekemeijer, Z. Ren, N. Tehrany, and A. Trivedi, "Zwal: Rethinkingwrite-ahead logs for zns ssds with zone appends," *ACM SIGOPS Operating Systems Review*, vol. 58, no. 1, pp. 53–60, 2024.

[30] K. Doekemeijer and N. Tehrany, "Performance characterization of nvme flash devices with zoned namespaces (ZNS)," in *IEEE International Conference on Cluster Computing*. IEEE, 2023, pp. 118–131.

[31] E. Lee, I. Son, and J.-S. Kim, "An efficient order-preserving recovery for f2fs with zns ssd," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023, p. 116–122.

[32] D. Corporation, "Dapustor j5100 series pcie 4.0 nvme ssd," https://en.dapustor.com/product/12.html, 2025.