Persimmon: an append-only ZNS-first filesystem

Devashish R. Purandare[1], Sam Schmidt[1], and Ethan L. Miller[1,2]
[1]*University of California, Santa Cruz*
[2]*Pure Storage*
Santa Cruz, CA, USA
dpuranda@ucsc.edu, sadschmi@ucsc.edu, elm@ucsc.edu

***Abstract*—While NAND flash has become the centerpiece of modern data center storage, legacy interfaces impact its performance and lifetime. Emulating in-place updates on SSDs results in frequent garbage collection, causing slowdowns, wear, and write amplification. Changing how we utilize modern SSDs to unlock their full potential is necessary. Even though Zoned Namespace SSDs provide an efficient append-only interface, filesystems on such drives still depend upon in-place updates and fixed metadata addresses, making their use with zoned storage complex and inefficient.**

We present Persimmon, a fork of the f2fs filesystem built with append-only metadata structures and tuned for zoned namespaces. Persimmon updates f2fs with new in-memory structures for better management in a zoned context, improved checkpoint logic, and append-only metadata management. Persimmon reduces tail latency, background garbage collection, and write amplification. Persimmon adapts its layout to zoned device constraints to unlock greater utilization and better drive cleanup.

***Index Terms*—Filesystems, ZNS, SSDs, RocksDB, F2FS**

I. Introduction

SSDs have become the centerpiece of modern data storage and need to grow in capacity as data centers move to flash-based primary storage. However, traditional scaling techniques such as die-shrink and increasing flash density come at the cost of durability and performance [1]. With the lifetime of modern flash down to a few hundred program-erase cycles, wear is further exacerbated by the overhead of garbage collection introduced by interfaces not designed for flash. Standard disk-based storage interfaces, which assume random writes and in-place updates, must be emulated by the controller as NAND flash does not support them. Even with append-only log-structured systems, the overhead of log-on-log structures can adversely impact write-amplification and tail latency [2].

To address such issues, Zoned Namespace SSDs (ZNS) were introduced to help align device logs with system logs, exposing the device geometry in equal-sized append-only regions. However, the complexity of adapting current systems to an append-only interface has hurt the adoption of these SSDs. Even log-structured append-only systems such as f2fs, btrfs, and RocksDB still depend on known locations and in-place updates for various metadata operations.

These systems typically employ auxiliary drives that support in-place updates while maintaining the logs in append-only regions. A device-mapper solution like `dm-zoned` can address this requirement but has limitations, such as not supporting zones where the size and capacity differ. These random-write drives still need to perform garbage collection, which can cause slowdowns, and complex multi-namespace or multi-device setups can impact the performance and stability of such systems. Further, the limited compatibility updates can mean odd characteristics in practical use: for instance, f2fs has as low as 36% of the total size as usable (depending on the configuration) with ZNS [3], and btrfs uses less than half the capacity of each zone.

The required volume of such random-write space scales with system size, making adoption difficult. In the case of f2fs, the metadata structures and the checkpoint region grow with the size of the disk, requiring gigabytes of random-write space on larger drives. We observed that above 1 TB, the size of these structures goes beyond 4 GB, which is the amount of random-write space on the 8 TB Western Digital Ultrastar DC ZN540 SSD, necessitating additional drives to support total capacity of the drive. Such cross-device mappings can lead to performance and stability issues. Hence, to unlock the full potential of zoned SSDs in reducing wear and tail latency, we need an append-only filesystem that does not depend on any fixed addresses (which need in-place updates) or random updates. Such a filesystem should partition data by lifetime, performing garbage collection as required while maintaining POSIX compatibility to avoid application rewrites.

We present Persimmon, a ZNS-first filesystem that uses append-only structures and does not depend on in-place updates for any of its metadata operations. We demonstrate reduced write amplification and garbage collection overhead over f2fs while maintaining the same flash-optimized performance. Persimmon adds new in-memory tracking structures to efficiently allocate and free up zones and implement easy-to-garbage-collect checkpoints with rolling logs. Persimmon takes design choices from various log-structure filesystems and puts them together, making ZNS-specific decisions to offer a filesystem designed for zoned storage. We demonstrate the improvements Persimmon offers over f2fs, btrfs, and zoned-f2fs and discuss the lessons learned and the best strategies for zoned filesystems.

II. Background

Host-device communication efforts predate SSDs, with designs such as the host-managed mode of Shingled Magnetic

This work was supported by NSF grants CNS-1841545, CNS-2106259. D. Purandare and S. Schmidt contributed equally to this work.

Recording (SMR) drives. Throughout the years, several approaches have been proposed and implemented, starting with NVMe Streams [4], Open-channel SSDs [5], Zoned Namespaces (ZNS) [6] and Flexible Device Placement (FDP) [7]. In the current NVMe specification, ZNS and FDP are the chief interfaces, the main difference between them being the append-only nature of ZNS, and the backwards compatible hint interface offered by FDP. ZNS offers several advantages over FDP such as reduced complexity, reduced over-provisioning and DRAM requirement, guaranteed reduction in write amplification, and immutability. With the goal of reducing cost of SSDs and improving the lifetime, we focus on ZNS for our work. With the append-only nature of ZNS being a drawback for applications that perform in-place updates, we need a way to *virtualize* these patterns.

A. Background: ZNS

Host-device coordination efforts predate SSDs, with designs like the host-managed mode of Shingled Magnetic Recording (SMR) drives. Throughout the years, several approaches have been proposed and implemented, starting with NVMe Streams [4], Open-channel SSDs [5], Zoned Namespaces (ZNS) [6] and Flexible Device Placement (FDP) [7]. In the current NVMe specification, ZNS and FDP are the chief interfaces, the main difference between them being the append-only nature of ZNS and the backward-compatible hint interface offered by FDP. ZNS offers several advantages over FDP, such as reduced complexity, reduced over-provisioning and DRAM requirement, guaranteed reduction in write amplification, and immutability. For Persimmon, we limit our focus to ZNS due to the availability of kernel API and devices, and we plan to explore FDP in the future.

ZNS works by partitioning the drive into equal-sized append-only logs known as zones. The host can pick zones and use the stored write pointer or the zone append command to append; random writes are rejected. When a zone fills up, it transitions to a read-only state, which needs to be explicitly reclaimed by the host for garbage collection. The host can use zones to group related data, treating it as an erase unit. Reclaiming a zone is *free* if all the data in a zone has a similar lifetime: it needs a single `Zone Reset` command. Thus, one of our aims is to collect data with similar lifetimes in the same zone, reducing the cost of relocation of valid data.

Previous work [6], [8], [9] describes the benefits of zoned storage over conventional SSDs. We focus on features that we use to help us design the Persimmon filesystem: the *on-device write pointers* offer us known locations of the last write, eliminating the need to have known Logical Block Addresses (LBAs) for metadata and checkpoints. The *zones* expose device geometry and allow us to group writes with similar lifetimes on the same zone, and the *host-managed garbage collection* allows us to perform garbage collection on the segments with the highest amount of invalid data.

We focus on a system that only uses *sequential* zones, ensuring append-only access to these logs. A sequential-only device is simpler as it does not need to maintain a complex FTL or a mapping, eliminating the need for large DRAM buffers or complex controllers and reducing the device's cost and wear. While the ZNS protocol supports *conventional* zones, i.e., zones that support in-place updates, these require the drive controller to implement a device-level Flash-Translation Layer (FTL) to emulate in-place updates, negating the benefits of an append-only device. Unlike other filesystems, Persimmon works without a random-write area, making it well-suited for devices with only sequential zones.

B. Approaches: Copy-on-Write vs Log-Structured

To achieve our goal of a ZNS-first filesystem, we first looked at desirable properties in a filesystem from a ZNS perspective. An append-only log-structured filesystem is well suited for such an interface [9]. We looked at two append-only update design families: log-structured merge trees and a copy-on-write design. To explore these, we chose f2fs and btrfs as they are flash-optimized, append-only, open source, and POSIX compliant. btrfs is a Copy-on-Write filesystem, organized as a B-Tree, while f2fs uses up to six separate append-only logs.

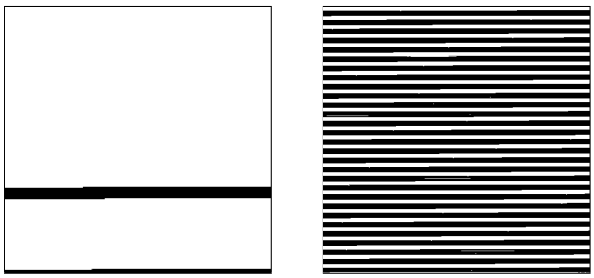
We then ran a simple sequential insert workload to measure the performance and utilization of each of these approaches. We expected btrfs, with its Copy-on-Write nature, to generate large amounts of data invalidated with each RocksDB compaction. To test out this theory, we set up a 32 GB emulated SSD with btrfs and f2fs. We then wrote 100 million records of 10 Bs, each with compression writing 6 GB to each filesystem. The same workload yielded the following utilization results:

TABLE I: Comparison of btrfs and f2fs on a 32 GB SSD.

	Zones		Capacity		
	Used	Full	Used	Invalid	Total Writes
f2fs	10	5	7 GB	1 GB	28 GB
btrfs	25	12	18 GB	12 GB	23 GB

As seen in Table I, f2fs performs more writes than btrfs, mainly due to the relocation of invalid data on garbage collection. On the other hand, btrfs has valid data scattered across 25 zones and cannot garbage collect them due to the limited availability of free zones. With its 6 logs of varying temperatures, f2fs can better identify temporary files and garbage collect them together to free up zones. btrfs, on the other hand, has a single log with all the writes, making it harder to collect garbage.

Further, btrfs' B-Tree structure with Copy-on-Write generates new data on every write, writing it to statically sized pre-allocated chunks, causing a large number of partially filled chunks that require garbage collection to reclaim. Unlike a traditional SSD, these empty regions cannot be written to on an append-only device, causing write and space amplification. For instance, in Fig. 1, we analyze a full block group from this experiment to a full block group on a traditional SSD and see that 60% of it does not contain valid data, resulting in poor utilization. Further, the balancing process is expensive in terms of writing and hurts the performance and the lifetime of the



(a) btrfs (Conventional) (b) btrfs (ZNS)

Fig. 1: We compare "full" block groups on btrfs on a conventional SSD with ZNS; white: valid data, black: unused space. On ZNS btrfs sees half-filled extents and poor utilization

drive. In addition, as we see in Fig. 8, f2fs performs better than btrfs as its multi-log and multi-append point design enables greater parallelism and simpler grouping on a zoned device. The performance and layout benefits inspired Persimmon to inherit f2fs' data segment design.

However, f2fs metadata still expects a traditional in-place update-friendly interface and fixed locations. f2fs optimizes itself for SSDs by maintaining six logs for appending writes. Three logs are used for node data in the 'main area' while the other three keep filesystem data. Node data includes any block of information required to address other blocks, such as inodes or indirect blocks. Filesystem data makes up files or directories. These groups maintain separate logs for hot, warm, and cold data. Separating this data can extend the lifetime of a device by reducing write amplification.

f2fs supports ZNS for the data region [10], but not for checkpoint and metadata region which require fixed addresses and an in-place updates. This can be achieved either by:

- Dedicated *conventional* zones with in-place updates.
- Multi-device setup
- In-place updates virtualized by a device-mapper
- `dm-zoned` mapped random-write region on append-only devices

These options are not particularly desirable as they maintain all the limitations of conventional SSDs (although at a smaller scale) and add complexity to the architecture, and impact performance. Further they have their own limitations, `dm-zoned` for instance, requires zone capacity to match zone size, which is not the case on the drives we tested.

III. Persimmon: A Zone-First Approach

To design an append-only filesystem optimized for ZNS, we started with a fork of f2fs with an unchanged data section. In Persimmon, we eliminate the random writes and in-place updates seen in f2fs associated with metadata and checkpoints. Further, f2fs does not overprovision efficiently on zoned storage, causing large amounts of reserved space. We introduce a new logic for overprovisioning and new in-memory structures to aid zone allocation and cleanup, reducing garbage collection overhead.

A. Append-Only Metadata Handling

f2fs metadata requires fixed Logical Block Addresses (LBAs) and hence in-place updates for maintaining the following structures:

1) The *Segment Information Table (SIT)* stores the number of valid blocks in the log and a bitmap for their validity.
2) The *Node Address Table (NAT)* provides a mapping of node ids and associated LBAs in f2fs.
3) The *Segment Summary Area (SSA)* stores owner information about blocks.

While these structures are necessary, the in-place updates can be virtualized. Persimmon maps the SIT, NAT, and SSA block addresses to physical addresses on the drive *emulating* an in-place update interface. These are stored in dedicated zones as they see frequent updates and can be garbage collected with a low data movement cost. Persimmon's page cache operation modifies the `address_space_operations` function table to make these changes while minimizing the change required in the f2fs codebase. When grabbing pages, Persimmon can switch out the `address_space_operations` object without affecting other filesystem operations. However, simply remapping the metadata is insufficient as it can add performance overhead to operations in Persimmon. To address the slowdown in access and cleanup, we introduce new in-memory structures that offer speedups for zone and metadata operations.

B. New In-Memory Data Structures

We introduce three additional in-memory data structures, as seen in Fig. 2 to manage the metadata mapping, optimize lookups, and to aid in zone allocation and garbage collection:

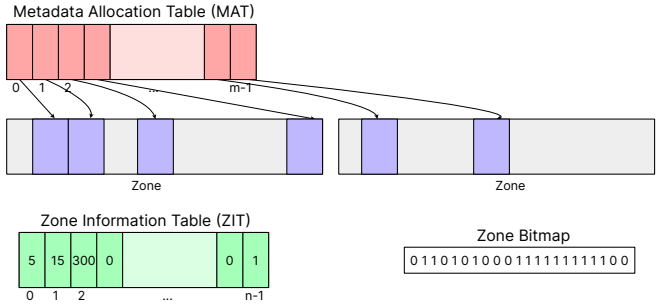


Fig. 2: The three structures that reside in memory: the Metadata Allocation Table (MAT), the Zone Information Table (ZIT), and the Zone Bitmap. Here m is the number of pages in metadata and n is the number of zones.

1) The Metadata Allocation Table (MAT): is an array that associates metadata block indices with block addresses. The f2fs metadata exists in a contiguous address range, with the last SIT address adjacent to the first NAT address. We offset indices by the first SIT address to reduce memory overhead before queries. The MAT size is the difference between the largest and the smallest block addresses.

2) The Zone Information Table (ZIT): is an array that maps a zone index to a count of the number of invalid blocks within that zone. The ZIT allows for speeding up garbage collection. Calculating the number of invalid blocks for a given zone requires iterating through each zone. This lookup table is updated if the address for a block does not match the one under consideration, incrementing the count.

3) The Zone Bitmap (ZB): maintains zone utilization data in memory to speed up new zone allocation. Persimmon reads the associated entry from the ZB instead of issuing a zone management command for the zones, avoiding the penalty of a zone management command to the device.

Persisting In-memory Structures

To repopulate these in-memory structures on `mount` and for crash recovery, we must persist them to the drive. We split the metadata into page-sized chunks, which are appended to dedicated metadata zones. In particular, we divide the MAT into chunks, and its in-memory handle maintains the addresses of each chunk. The MAT chunks provide direct addressing to blocks. Alongside the MAT chunks, we update the number of invalid blocks for a previous zone based on block updates. The full Zone Bitmap is stored at the end of each chunk because its size is relatively small compared to the other structures. In systems with many zones, we may also need to split up the zone bitmap.

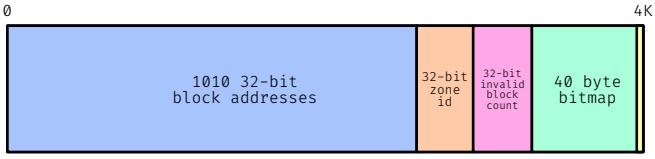


Fig. 3: The page-aligned chunk design.

A chunk comprises 1010 32-bit MAT entries (the changes since the last chunk), a 32-bit zone id, a 32-bit invalid count, and a 40-byte bitmap as seen in Fig. 3. These data structures get appended to the checkpoints because of how infrequently the system generates checkpoints and the small checkpoint size (no more than a few blocks). We store the chunks adjacent to the Persimmon metadata and update the f2fs metadata and checkpoint structures to track them. To avoid small frequent writes, we batch metadata updates in a `bio struct`, then write all dirty chunk pages in the same batch.

The NAT, SIT, and SSA are at least a few megabytes, with the SSA approaching tens of gigabytes at f2fs's maximum supported capacity. These structures are frequently updated, with an updated NAT with every data change, increasing the on-drive footprint size for the metadata and, consequently, the MAT. However, with dedicated zones for frequently updated data, Persimmon can reclaim space relatively cheaply.

Allocating New Zones

When a metadata zone fills up, Persimmon chooses another with the help of the Zone Bitmap to continue accepting writes. Persimmon iterates through zones numerically, consulting the ZB until it finds the first free zone. We chose this approach over defining a queue of zones, assuming there will be few zones to search and NVMe commands can report empty zones. We maintain f2fs' maximum supported size: $16\,\mathrm{TB}$. However, we run our tests on smaller drives, so we define the bitmap in our tests as a fixed size of $40\,\mathrm{B}$ since this is enough to address 320 zones, which would be sufficient for metadata. We allow users to expand ZB size while formatting the drive with `makefs` tools.

C. Append-only Checkpoints

Much like metadata, f2fs checkpoints depend on a known LBA and in-place updates. Since ZNS SSDs do not support in-place updates, we redesigned the checkpoints to be append-only with a rolling buffer of zones. For resetting zones, we adopted a policy similar to btrfs [11], where Persimmon resets the previous zone after writing the most recent data to another zone. Persimmon's checkpoints include additional fields for the newly introduced structures.

However, checkpoints still depend on a known LBA for checkpoint headers. We fixed this issue by using the tail LBA of a zone, maintained in the *on-device write pointer*. Persimmon uses an updated checkpoint design to utilize footers rather than headers, maintaining the changing header address in the footer. Since the footer is a fixed size directly preceding the last-written address maintained by the write pointer, this simplifies keeping track of checkpoints. Persimmon's approach differs from btrfs; checkpoint writes do not spill over into the other zone, maintaining their separation and eliminating the need for data movement on garbage collection.

Updates to Drive Layout

To accommodate the sequentially written checkpoints, the size of the space allocated to the checkpoint is now two full zones, which allows our rolling buffer design. The size allocated for non-checkpoint metadata is increased by 20% over f2fs to provide additional space for persisting in-memory structures discussed in Section III-B. Beyond that, the layout of Persimmon remains similar to f2fs. We include the additional fields in the superblock to allow our metadata mapping to be recovered on a remount. Persimmon uses two checkpoint zones and allocates metadata and data zones as needed.

Metadata Garbage Collection

Persimmon uses the ZIT to inform its decision when choosing a zone for garbage collection to minimize the number of writes incurred from migrating valid data. We trigger garbage collection whenever the number of free zones is equal to 2, allowing for space to write to another zone where necessary. Persimmon allows tuning this threshold based on the performance and capacity requirements of the system. Once a zone with the most invalid blocks is selected, the validity of blocks is determined by querying for the Persimmon queries the associated mapped logical block address to get the valid blocks. The blocks are copied to a new open zone on a match.

D. Limitations

Since Persimmon is a fork of f2fs, we inherit some of the limitations already present in f2fs, namely, a maximum filesystem size of 16TB due to 32-bit logical block addresses. Additionally, Persimmon only works on zoned SSDs. We add a small amount of write and memory overhead with the new metadata structures; however, even with these additions, we reduce the overall amount of writes with better garbage collection and simplified device structure. Further, because Persimmon works on zone granularity, it allocates more on-device real-estate toward checkpoints and metadata relative to the size of these tables, especially on systems with larger zones. We plan to explore techniques to reduce this overhead.

IV. Evaluation

The goal of Persimmon was to create a filesystem with append-only structures that can utilize the zoned interface to its full potential, reduce garbage collection overhead, improve tail latency and space utilization, and reduce write amplification. We evaluate each of these aspects in this section and measure the overhead. For evaluation, we used a 64 Core AMD EPYC server with $256\,\mathrm{GB}$ of DRAM. We used dedicated Western Digital Ultrastar DC ZN540 SSD zoned drives for each filesystem to ensure strict performance isolation. We use conventional zones on the same drive in case of f2fs to ensure that cross-device coordination does not impact its performance. To avoid the impact of on-device operations, we reset the filesystems and drives between each run to ensure no on-device garbage collection occurs unrelated to the benchmark. We capture statistics at the device and filesystem levels to measure writes. We use Yahoo Cloud Serving Benchmark [12] with RocksDB and RocksDB's built-in tool `db_bench` as sample workloads.

A. Improvement in Tail Latency

We test the impact on latency with YCSB's [12] 6 workloads: namely: A. Update-heavy workload: 50% reads, 50% updates, B. Read-heavy: 95% reads, 5% updates, C. Read-only, D. Read-Insert: 5% records are inserted, and latest records are read, E. Short ranges: 5% records are inserted and scanned over short ranges, F. Read-Modify-Write: 50% Reads and 50% Read-Modify-Writes.

For read latency as seen in Fig. 4 Persimmon performs uniformly better, minimizing latency for every workload, especially tail latencies (E and F). We get this advantage due to the faster lookup operations enabled by the in-memory MAT and minimizing tail latency spikes usually caused by background operations. For the Read-Insert workload, the persistence of recent inserts in the metadata structures hurts Persimmon at the tail. Zoned f2fs, on the other hand, see millisecond-to-second spikes in tail latencies, especially during read-heavy workloads, as the device tries to free up space and garbage collection. As the filesystem fills up, garbage collection increasingly impacts performance, and Persimmon cuts down on garbage collection in the filesystem.

B. Improving Garbage Collection Efficiency

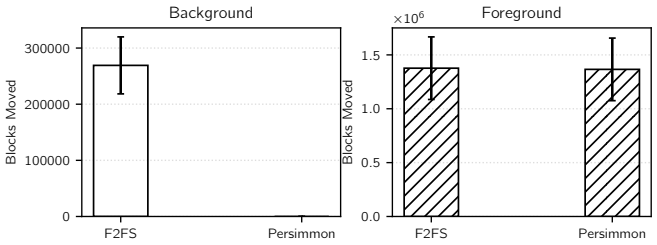


Fig. 5: Persimmon performs virtually no garbage collection in the background as opposed to f2fs.

We ran a multi-threaded benchmark to test garbage collection efficiency by filling the drive with $55\,\mathrm{GB}$ sequential inserts followed by random updates, deletes, and overwrites, and finally by random reads. This is the benchmark we used to measure throughput and write amplification. Effectively, these actions yielded more than $200\,\mathrm{GB}$ of writes on the $128\,\mathrm{GB}$ drives, ensuring in-filesystem garbage collection.

We measured the number of pages moved by the device in the two garbage collection modes: **1. Background:** f2fs and Persimmon's age-sorted background mode, which lazily garbage collects and **2. Foreground:** When space runs out, the filesystem aggressively frees up zones.

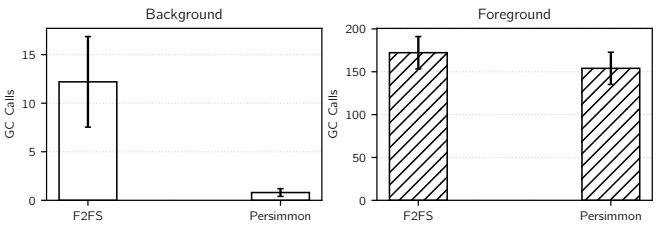


Fig. 6: Persimmon triggers less calls to garbage collection, improving performance.

As seen in Fig. 5, during the benchmark, Persimmon does not need to move large amounts of pages to perform background garbage collection since Persimmon can reset metadata and checkpoint zones without data movement as the data has a short lifetime. Persimmon's greater availability of free space further helps it wait longer to collect garbage, waiting for more data to be deleted over f2fs, resulting in lower data movement. f2fs needs to perform more background garbage collection to ensure free space. On the other hand, both systems perform similar amounts of foreground garbage collection, as this is workload-dependent and in the data region.

For garbage collection calls, as seen in Fig. 6, we look at the statistics maintained by the filesystem. Persimmon manages to garbage collect more efficiently, needing just one call for regular workloads instead of f2fs's twelve, and reduced number of foreground calls. As each call negatively impacts throughput, write amplification, and latency, we see modest improvements due to this reduction in overhead. Persimmon

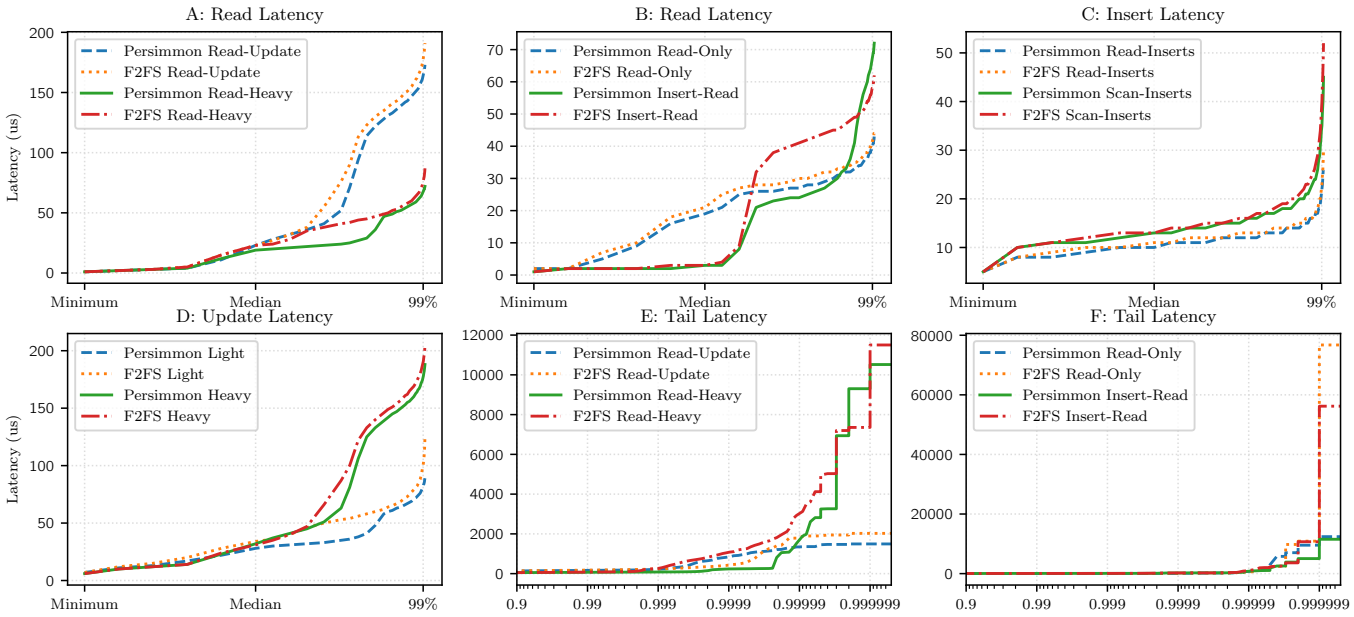


Fig. 4: For Read-Latency, Persimmon performs similar to f2fs offering better tail latency.

moves less data in the background while maintaining more free space.

C. Write and Space Amplification

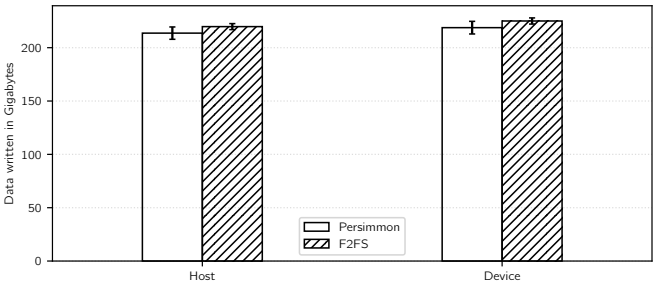


Fig. 7: Persimmon reduces writes both to the device and the filesystem

Persimmon reduces the total writes from the benchmark by about 4 GB, plus the writes saved in avoiding the garbage collection, which the metadata drive for f2fs will have to perform. As seen in Fig. 7, we measure both the writes reported by the filesystem and those reported by SMART data on the device. In our tests, on an average Persimmon writes 215 GB to the filesystem, resulting in 220 GB on the device, where f2fs 218 GB of writes on an average rising up to 224 GB on the SSD. While it is not a large difference, all of these writes are due to the extra garbage collection of metadata and f2fs needs to perform.

Further, Persimmon offers a larger usable space, especially at smaller size, over f2fs as seen in Table II, with metadata overhead shrinking with the filesystem size.

TABLE II: Usable space with 10% over-provisioning and 2 GB zones

	f2fs (GB)		Persimmon (GB)	
Size	Usable	Reserved	Usable	Reserved
64 GB	33	31	47	17
128 GB	93	33	107	21
256 GB	215	41	229	27

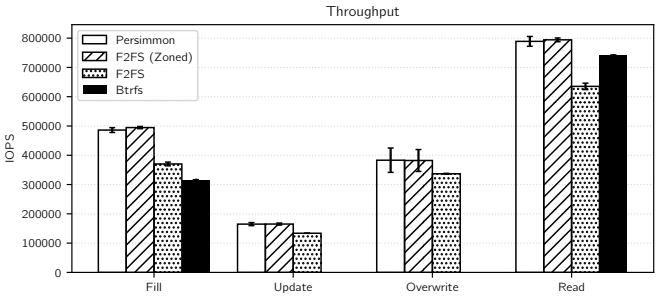


Fig. 8: Persimmon sees a slight slowdown in sequential insert, but is otherwise similar to f2fs.

D. Throughput

While we do not expect significant improvements in the flash-optimized f2fs, as seen in Fig. 8, we see similar performance to f2fs in Persimmon, with a slight lower insert in the read-random workload, where since the workload does not perform writes, both these systems typically perform some background garbage collection. Performance is better than non-zoned f2fs and btrfs, however, we were unable to test btrfs's update or overwrite performance due to stability issues

in zoned mode.

E. Overhead

1) Mounting and Unmounting: As Persimmon introduces new in-memory structures we expect to observe some overhead in terms of mount times and memory usage. However, our approach is faster at mounting than f2fs, as the single-device single-namespace layout is faster to read. The time to mount grows linearly with size as seen in Table III. For unmount, Persimmon takes a small performance hit, due to the need to persist extra structures, but the overhead is insignificant.

TABLE III: Mount and unmount times (Seconds)

	f2fs		Persimmon	
Size	mount (s)	unmount (s)	mount (s)	unmount (s)
256 GB	0.2185	0.016	0.149	0.02
512 GB	0.327	0.018	0.2545	0.022
2 TB	–	–	0.8784	0.026

2) Memory Overhead: With the addition of new in-memory structures, we observe a modest overhead in memory utilization, particularly under heavy workloads. However, this overhead is about 20 MB per each 100 GB added to the filesystem size as seen Fig. 9. Since this could be an issue at larger SSD sizes, we provide an option to limit this overhead with an `ioctl`. Lowering the memory use may impact performance.

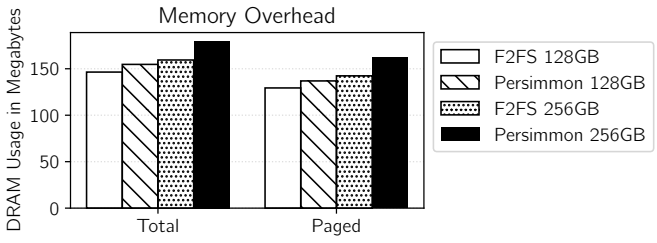


Fig. 9: We observe a modest increase in memory use depending on the workload, however the overhead is fairly small.

V. Related Work

For zoned storage, existing filesystems are broadly in two categories: a. special-purpose (non-POSIX) filesystems like

A. Special-purpose ZNS filesystems

ZenFS [6] and ZoneFS [13], explicitly designed for the zoned interface, and b. general-purpose (POSIX) filesystems like f2fs and btrfs, which have adapted to the ZNS implementation. ZoneFS [13] exposes each zone as a single file. These files support writing data but do not support any changes to the filesystem layout, serving as a block-layer projection of the underlying device.

ZenFS [6] is a plugin for RocksDB that exposes each zone as a writable set of extents that can be appended. However, ZenFS does not support the POSIX interface and systems that require in-place updates.

B. POSIX filesystems on ZNS

btrfs [11] is a Copy-on-Write filesystem with experimental support for Zoned Namespaces. It achieves this by enabling a rolling buffer of zones for the superblock, the only fixed structure, an approach that inspired our checkpoint design. We picked the multi-log approach of f2fs over btrfs for the reasons we discussed in Section II-B.

f2fs [10] is well-suited for a zoned interface but has several limitations in its current version, namely requiring random-write areas for metadata and checkpoints, and excessive over-provisioning which is not designed for a ZNS interface. We use f2fs as the baseline for the log-structured interface it provides, along with temperature-specific logs. However, we updated the metadata management to not depend on in-place updates, improving performance and reducing write-amplification

Other filesystems like EXT4 or ZFS could be adapted to the zoned interface. However, they will require significant re-engineering effort and will not benefit from the flash-optimized decisions in f2fs. Systems such as RocksDB can work with ZNS but do not present a POSIX file interface or allow in-place updates, requiring rewrites of applications for a new storage interface.

VI. Discussion

Persimmon presents an efficient, performant ZNS-optimized filesystem, but its performance gains are modest, mainly focused on garbage collection. It still cannot match the performance of application-specific solutions like ZenFS [6]. During this work, we explored the ZNS design space and would like to present the following observations:

1) When compatibility is not enough: Adopting new hardware interfaces like ZNS requires changes in fundamental abstractions that applications rely on. Rewriting applications is expensive, and it is easier to modify the filesystem to accommodate the benefits offered by these new interfaces. However, changing existing filesystems without breaking backward compatibility is a complex task that results in compromises, as we see with f2fs, requiring a random-write area for metadata and excess over-provisioning to accommodate the new interface as seen in Table II.

2) Complexity inherent to kernel filesystems: While future versions may address these issues, updates depend on Linux kernel releases, which may take years before data centers adopt them. Upgrading the filesystem structure requires reformatting metadata and causes compatibility issues. These issues hurt the adoption of modern storage protocols like ZNS and FDP since rewriting applications is expensive, and filesystems do not support the new standards effectively. However, the interface has much potential beyond the benefits we demonstrate with Persimmon. For instance, these filesystems' hint and grouping mechanisms are rudimentary and cannot generate workload or architecture-based hints.

3) Hint Generation:: An essential property of ZNS that we exploit is the ability to group data with related lifetimes, allowing us low-cost zone resets. The f2fs temperature separation could be tuned further for file lifetimes rather than workloads,

grouping files with related lifetimes. We can improve over such naive hints with approaches that use active learning systems and tune hint generation according to topology and workload. Persimmon uses a `fcntl()` system-call-based hint mechanism for deciding which files end up on which log. Fundamentally, filesystems are application-unaware and will need inputs from the application or the user to generate effective hints.

4) The need for user space approaches:: Since filesystems are in the kernel, updates to filesystem logic are complex. Changes require updating the kernel module, migrating the old structures to the new logic, unmounts, reformats, and remounts. Errors can cause kernel panics and crashes, further complicating the adoption of novel filesystems. An easier way could be to use a simple filesystem, like ZoneFS, which maps the block layer to a file interface. POSIX compatibility and placement logic could then be implemented as a shim layer.

5) Kernel bypass: Another way could be to use frameworks like xNVMe [14] or SPDK [15] to bypass the kernel, implementing all logic in userspace. Userspace approaches can unlock performance benefits at the cost of additional configuration complexity.

6) Other NVMe Protocols: NVMe FDP has been proposed to address the issues with the append-only nature of ZNS, allowing grouping based on hints in a random-write friendly interface. With key-value SSDs, computational storage, and memory-semantic SSDs, interfaces to SSDs are getting more diverse, and the adoption of heterogeneous hardware becomes even more challenging. A filesystem cannot reasonably support different types of devices and tune each application according to the workload. We must rethink our approach to these SSD protocols and filesystems to build data centers with diverse storage types.

VII. Conclusion

With NAND-flash-based SSDs becoming ubiquitous, we need to scale them without severely impacting the lifetime or performance of the drives. ZNS is an effort to enable that. However, it has not seen its full potential explored due to the complexity of modern filesystems, manufacturer-specific extensions, and the need to preserve backward compatibility.

We present Persimmon, the first general-purpose filesystem tuned for append-only ZNS SSDs. Persimmon starts with the f2fs design and improves metadata management, garbage collection, device utilization, and tail latency. Importantly, Persimmon requires no auxiliary devices and can be used solely in the append-only region of a ZNS SSD. We demonstrate that such a filesystem is achievable and can match the performance of flash-optimized systems while reducing the garbage collection overhead.

Acknowledgements:

We are grateful for the help and support from Shel Finkelstein, Daniel Bittman, Darrell D. E. Long, and Center for Research in Storage and Systems (CRSS) members for their input and feedback. We are grateful to Matias Bjørling and Western Digital for the ZN540 drives and feedback on this work. This work was supported by NSF grants CNS-1841545, CNS-2106259, and funding from CRSS industry members. We also thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory." in *FAST*, vol. 7, no. 3.2, 2012, pp. 10–2.

[2] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*. Broomfield, CO: USENIX Association, Oct. 2014.

[3] D. Seo, P.-X. Chen, H. Li, M. Bjorling, and N. Dutt, "Is garbage collection overhead gone? case study of F2FS on ZNS SSDs," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2023.

[4] J. Bhimani, J. Yang, Z. Yang, N. Mi, N. H. V. K. Giri, R. Pandurangan, C. Choi, and V. Balakrishnan, "Enhancing SSDs with multi-stream: What? why? how?" in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, Dec. 2017, pp. 1–2, iSSN: 2374-9628.

[5] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The linux Open-Channel SSD subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 359–374.

[6] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, "ZNS: Avoiding the block interface tax for flash-based SSDs," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 689–703.

[7] C. Sabol and R. Stenfort, "Hyperscale innovation: Flexible data placement mode (FDP)," in *NVMe Flexible Data Placement*. NVMe Express Organization, 2022.

[8] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and W. Lloyd, "Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete," in *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, Jun. 2021, pp. 144–151.

[9] D. Purandare, P. Wilcox, H. Litz, and S. Finkelstein, "Append is near: Log-based data management on ZNS SSDs," in *Conference on Innovative Data Systems Research 2022 (CIDR '22)*, Jan. 2022.

[10] C. Lee, D. Sim, J. Hwang, and S. Cho, "{F2FS}: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 273–286.

[11] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[13] D. L. Moal and T. Yao, "Zonefs: Mapping POSIX file system interface to raw zoned block device accesses," in *Vault 2020*. Santa Clara, CA: USENIX Association, Feb. 2020.

[14] S. A. Lund, P. Bonnet, K. B. Jensen, and J. Gonzalez, "I/O interface independence with xNVMe," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, 2022, pp. 108–119.

[15] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.