ZNS: Avoiding the Block Interface Tax for Flash-based SSDs

Matias Bjørling, *Western Digital;* Abutalib Aghayev, *The Pennsylvania State University;* Hans Holmberg, Aravind Ramesh, and Damien Le Moal, *Western Digital;* Gregory R. Ganger and George Amvrosiadis, *Carnegie Mellon University*

https://www.usenix.org/conference/atc21/presentation/bjorling

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

ZNS: Avoiding the Block Interface Tax for Flash-based SSDs

Matias Bjørling[⋆], Abutalib Aghayev[◇], Hans Holmberg[⋆], Aravind Ramesh[⋆], Damien Le Moal[⋆],
Gregory R. Ganger[†], George Amvrosiadis[†]
[⋆]*Western Digital* [◇]*The Pennsylvania State University* [†]*Carnegie Mellon University*

Abstract

The Zoned Namespace (ZNS) interface represents a new division of functionality between host software and flash-based SSDs. Current flash-based SSDs maintain the decades-old block interface, which comes at substantial expense in terms of capacity over-provisioning, DRAM for page mapping tables, garbage collection overheads, and host software complexity attempting to mitigate garbage collection. ZNS offers shelter from this ever-rising *block interface tax*.

This paper describes the ZNS interface and explains how it affects both SSD hardware/firmware and host software. By exposing flash erase block boundaries and write-ordering rules, the ZNS interface requires the host software to address these issues while continuing to manage media reliability within the SSD. We describe how storage software can be specialized to the semantics of the ZNS interface, often resulting in significant efficiency benefits. We show the work required to enable support for ZNS SSDs, and show how modified versions of f2fs and RocksDB take advantage of a ZNS SSD to achieve higher throughput and lower tail latency as compared to running on a block-interface SSD with identical physical hardware. For example, we find that the 99.9^{th}-percentile random-read latency for our zone-specialized RocksDB is at least 2–4× lower on a ZNS SSD compared to a block-interface SSD, and the write throughput is 2× higher.

1 Introduction

The *block interface* presents storage devices as one-dimensional arrays of fixed-size logical data blocks that may be read, written, and overwritten in any order. Introduced initially to hide hard drive media characteristics and to simplify host software, the block interface worked well for many generations of storage devices and allowed great innovation on both sides of the storage device interface. There is a significant mismatch, however, between the block interface and current storage device characteristics.

For flash-based SSDs, the performance and operational costs of supporting the block interface are growing prohibitively [22]. These costs are due to a mismatch between the operations allowed and the nature of the underlying flash media. Although individual logical blocks can be written to flash, the medium must be erased at the granularity of larger units called *erase blocks*. The SSD's Flash Translation Layer (FTL) hides this mismatch by using substantial DRAM for dynamic logical-to-physical page mapping structures, and by reserving substantial fractions (*over-provisioning*) of the drive's media capacity to lower the garbage-collection overhead of erase-block data. Despite these efforts, garbage collection often results in throughput limitations [17], write-amplification [3], performance unpredictability [33, 64], and high tail latencies [16].

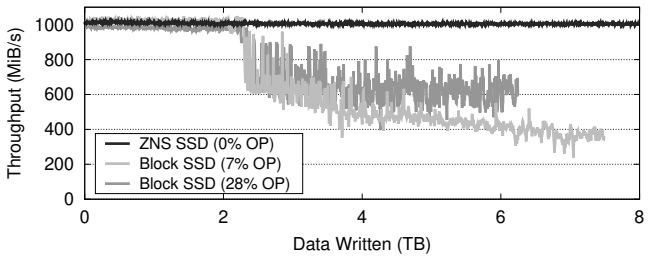

Figure 1: Throughput of a multi-threaded write workload that overwrites usable SSD capacity four times. The SSDs all have 2 TB raw media and share the same hardware platform.

The NVMe Zoned Namespace Command Set specification [8], or ZNS for short, has recently been introduced as a new interface standard for flash-based SSDs. Instead of a single array of logical blocks, a ZNS SSD groups logical blocks into *zones*. A zone's logical blocks can be read in random order but must be written sequentially, and a zone must be erased between rewrites. A ZNS SSD aligns zone and physical media boundaries, shifting the responsibility of data management to the host. This obviates the need for in-device garbage collection and the need for resource and capacity over-provisioning (OP). Further, ZNS hides device-specific reliability characteristics and media-management complexities from the host software.

Figure 1 illustrates some of the costs of the block interface and the corresponding potential benefits of ZNS. The three lines represent throughput for identical SSD hardware exposed as a ZNS SSD (0% OP) and as a block-interface SSD (measured with 7% OP and 28% OP), while concurrently overwriting the drive's usable capacity 4 times. All three SSDs provide high throughput for the first 2TB (which is their raw capacity), but then the block-interface SSD expe-

riences a sharp drop in throughput as in-device garbage collection kicks in. As expected, due to the reduction in garbage collection overhead, 28% OP provides higher steady-state write throughput than 7% OP. The ZNS SSD sustains high throughput by avoiding in-device garbage collection and offers more usable capacity (see line length) by eliminating the need for over-provisioning.

This paper describes the ZNS interface and how it avoids the block interface tax (§2). We describe the responsibilities that ZNS devices jettison, enabling them to reduce performance unpredictability and significantly eliminate costs by reducing the need for in-device resources (§3.1). We also describe an expected consequence of ZNS: the host's responsibility for managing data in the granularity of erase blocks. Shifting FTL responsibilities to the host is less effective than integrating with the data mapping and placement logic of storage software, an approach we advocate (§3.2).

Our description of the consequences and guidelines that accompany ZNS adoption is grounded on a significant and concerted effort to introduce ZNS SSD support in the Linux kernel, the fio benchmark tool, the f2fs file system, and the RocksDB key-value store (§4). We describe the software modifications required in each use case, all of which have been released as open-source to foster the growth of a healthy community around the ZNS interface.

Our evaluation shows the following ZNS interface advantages. First, we demonstrate that a ZNS SSD achieves up to 2.7× higher throughput than block-interface SSDs in a concurrent write workload, and up to 64% lower average random read latency even in the presence of writes (§5.1). Second, we show that RocksDB on f2fs running on ZNS SSD achieves at least 2-4× lower 99.9^{th}-percentile random read latency than RocksDB on file systems running on block-interface SSDs. RocksDB running directly on a ZNS SSD using ZenFS, a zone-aware backend we developed, achieves up to 2× higher throughput than RocksDB on file systems running on block-interface SSDs (§5.2).

The paper consists of five key contributions. We present the first evaluation of a production ZNS SSD in a research paper, directly comparing it to a block-interface SSD using the same hardware platform, and optional multi-stream support. Second, we review the emerging ZNS standard and its relation to prior SSD interfaces. Third, we describe the lessons learned adapting host software layers to utilize ZNS SSDs. Fourth, we describe a set of changes spanning the whole storage stack to enable ZNS support, including changes to the Linux kernel, the f2fs file system, the Linux NVMe driver and Zoned Block Device subsystem, the fio benchmark tool, and the development of associated tooling. Fifth, we introduce ZenFS, a storage backend for RocksDB, to showcase the full performance of ZNS devices. All code changes are open-sourced and merged into the respective official codebases.

2 The Zoned Storage Model

For decades, storage devices have exposed their host capacity as a one-dimensional array of fixed-size data blocks. Through this *block interface*, data organized in a block could be read, written, or overwritten in any order. This interface was designed to closely track the characteristics of the most popular devices at the time: hard disk drives (HDDs). Over time, the semantics provided by this interface became an unwritten contract that applications came to depend on. Thus, when SSDs were introduced, they shipped with complex firmware (FTL) that made it possible to offer the same block interface semantics to applications even though they were not a natural fit for the underlying flash media.

The Zoned Storage model, originally introduced for Shingled Magnetic Recording (SMR) HDDs [19,20,35], was born out of the need to create storage devices free of the costs associated with block interface compatibility. We detail those costs with respect to SSDs (§2.1), and then continue to describe existing improvements to the block interface (§2.2) and the basic characteristics of zoned storage (§2.3).

2.1 The Block Interface Tax

Modern storage devices, such as SSDs and SMR HDDs, rely on recording technologies that are a mismatch for the block interface. This mismatch results in performance and operational costs. On a flash-based SSD, an empty flash page can be programmed on a write, but overwriting it requires an erase operation that can occur only at the granularity of an *erase block* (a set of one or more flash blocks, each comprising multiple pages). For an SSD to expose the block interface, an FTL must manage functionality such as in-place updates using a write-anywhere approach, mapping host logical block addresses (LBAs) to physical device pages, garbage collecting stale data, and ensuring even wear of erase blocks.

The FTL impacts performance and operational costs considerably. To avoid the media limitations for in-place updates, each LBA write is directed to the next available location. Thus, the host relinquishes control of physical data placement to the FTL implementation. Moreover, older, stale versions of the data must be garbage collected, leading to **performance unpredictability** [1,10] for ongoing operations.

The need for garbage collection necessitates the allocation of physical resources on the device. This requires media to be over-provisioned by up to 28% of the total capacity, in order to stage data that are being moved between physical addresses. Additional DRAM is also required to maintain the volatile mappings between logical and physical addresses. Capacity over-provisioning and DRAM are the most expensive components in SSDs [18,57], leading to **higher cost per gigabyte of usable capacity**.

2.2 Existing Tax-Reduction Strategies

Two major approaches for reducing the block interface tax have gained traction: SSDs with stream support (Stream

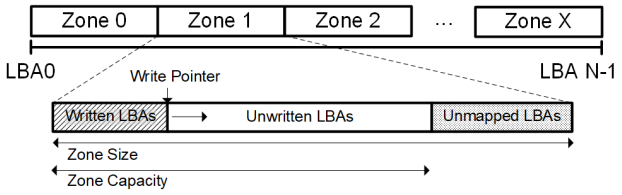

Figure 2: Zones within the LBA address space of a storage device. A write pointer per zone increases on successful writes, and is reset by issuing an explicit reset command.

SSDs) and Open-Channel SSDs (OCSSDs).

Stream SSDs allow the host to mark its write commands with a stream hint. The stream hint is interpreted by the Stream SSD, allowing it to differentiate incoming data onto distinct erase blocks [32] which *improves the overall SSD performance and media lifetime*. Stream SSDs require that the host carefully marks data with similar lifetimes in order to reduce garbage collection. If the host mixes data of different lifetimes into the same stream, Stream SSDs behave similarly to block-interface SSDs. A Stream SSD must carry the resources to manage such an event, so *Stream SSDs do not shed the costs of block-interface SSDs for extra media over-provisioning and DRAM*. We compare the performance of a Streams SSD with a ZNS SSD in §5.3.

Open-Channel SSDs allow host and SSD to collaborate through a set of contiguous LBA chunks [9, 38, 46, 60]. OCSSDs can expose these chunks such that they align with the physical erase block boundaries of the media. This eliminates in-device garbage collection overhead and reduces the cost of media over-provisioning and DRAM. With OCSSDs, the host is responsible for data placement. This includes underlying media reliability management such as wear-leveling, and specific media failure characteristics (depending on the OCSSD type). This has the potential to improve SSD performance and media lifetime over Stream SSDs, but *the host must manage differences across SSD implementations to guarantee durability*, making the interface hard to adopt and requiring continual software upkeep.

The ZNS interface, which we describe next, builds on the lessons of OCSSDs and SMR HDDs. It utilizes, and is compatible with, the zoned storage model defined in the ZAC/ZBC [28, 29] specifications. It adds features to take advantage of the characteristics of flash-based SSDs. ZNS aims to eliminate the mismatch between SSD media and the device interface. It also provides a media-agnostic next-generation storage interface by avoiding to directly manage media-specific characteristics like OCSSD [59].

2.3 Tax-free Storage with Zones

The fundamental building block of the zoned storage model is a *zone*. Each zone represents a region of the logical address space of the SSD that can be read arbitrarily but must be written sequentially, and to enable new writes, must be explicitly reset. The write constraints are enforced by a per-zone state machine and a write pointer.

A per-zone *state machine* determines whether a given zone is writeable using the following states: EMPTY, OPEN, CLOSED, or FULL. Zones begin in the EMPTY state, transition to the OPEN state upon writes, and finally transition to FULL when fully written. The device may further impose an open zone limit on the number of zones that can simultaneously be in the OPEN state, e.g., due to device resource or media limitations. If the limit is reached and the host attempts to write to a new zone, another zone must be transitioned from the OPEN to the CLOSED state, freeing on-device resources such as write buffers. The CLOSED zone is still writeable, but must be transitioned to the OPEN state again before serving additional writes.

A zone's *write pointer* designates the next writeable LBA within a writeable zone and is only valid in the EMPTY and OPEN states. Its value is updated upon each successful write to a zone. Any write commands issued by the host that (1) do not begin at the write pointer, or (2) write to a zone in the FULL state will fail to execute. When a zone is reset, through the reset zone command, the zone is transitioned to the EMPTY state, its write pointer is updated to the zone's first LBA, and the previously written user data is no longer accessible. The zone's state and write pointer eliminate the need for host software to keep track of the last LBA written to a zone simplifying recovery, e.g., after an improper shutdown.

Whilst the write constraints are fundamentally the same across zoned storage specifications, the ZNS interface introduces two concepts to cope with the characteristics of flash-based SSDs.

The *writeable zone capacity* attribute allows a zone to divide its LBAs into writeable and non-writeable, and allows a zone to have a writeable capacity smaller than the zone size. This enables the zone size of ZNS SSDs to align with the power-of-two zone size industry norm introduced with SMR HDDs. Figure 2 shows how zones can be laid out over the logical address space of a ZNS SSD.

The *active zone limit* adds a hard limit on zones that can be in either the OPEN or CLOSED state. Whereas SMR HDDs allow all zones to stay in a writeable state (i.e., CLOSED), the characteristics of flash-based media, such as program disturbs [15], require this quantity to be bounded for ZNS SSDs.

Although the ZNS interface increases the responsibilities of host software, our study shows various use cases can benefit from a set of techniques (§3.2) that ease its adoption.

3 Evolving towards ZNS

This section outlines aspects of ZNS SSDs that affect application performance, both in terms of hardware impact (§3.1) and adapting host applications to the ZNS interface (§3.2).

3.1 Hardware Impact

ZNS SSDs relinquish responsibilities traditionally carried out by the FTL, associated with supporting random writes. The ZNS interface enables the SSD to translate sequential zone writes into distinct erase blocks, thus eliminating the interface-media mismatch. Since random writes are disallowed by the interface and zones must be explicitly reset by the host, the data placement managed by the device occurs at the coarse-grained level of zones. This means that the SSD garbage collection routine responsible for moving valid data between erase blocks (to free up writeable capacity) becomes the responsibility of the host. This implies that write amplification on the device is eliminated, which eliminates the need for capacity over-provisioning, while also improving the overall performance and lifetime of the media [17, 23]. We quantify these benefits in §5.

While ZNS offers significant benefits for the end-user, it introduces the following tradeoffs in the design of the SSD's FTL.

Zone Sizing. There is a direct correlation between a zone's write capacity and the size of the erase block implemented by the SSD. In a block-interface SSD, the erase block size is selected such that data is striped across multiple flash dies, both to gain higher read/write performance, but also to protect against die-level and other media failures through per-stripe parity. It is not uncommon for SSDs to have a stripe that consists of flash blocks from 16-128 dies, which translates into a zone with writeable capacity from hundreds of megabytes to low single-digit gigabytes. Large zones reduce the degrees of freedom for data placement by the host, so we argue for the smallest zone size possible, where die-level protection is still provided, and adequate per-zone read/write performance is achieved at low I/O queue depths. If the end-user is willing to compromise on data reliability, the stripe-wide parity can be removed, and thus smaller writeable sizes can be achieved, but at the expense of host complexity, such as host-side parity calculation and deeper I/O queue depths to get the same performance as ZNS SSDs with larger zones or block-interface SSDs.

Mapping Table. In block-interface SSDs, the FTL maintains a fully-associative mapping table [21] between LBAs and their physical locations. This fine-grained mapping improves garbage collection performance, but the table size often requires 1GB of mappings per 1TB of media capacity. For consumer and enterprise SSDs, mappings are typically stored within device DRAM, whereas embedded SSDs may deploy a caching scheme at the expense of lower performance. Because ZNS SSD zone writes are required to be sequential, we can transition from complex, fully-associative mapping tables to coarse-grained mappings maintained either entirely at the erase block level [34] or in some hybrid fashion [31, 48]. As these mappings account for the largest usage of DRAM on an SSD, this can significantly reduce or even completely eliminate the need for DRAM.

Device Resources. A set of resources is associated with each partially-written erase block (i.e., active zone). This set includes hardware resources, such as XOR engines, memory resources, such as SRAM or DRAM, and power capacitors to persist parity data following a power failure. The data and parity can range from hundreds of kilobytes to megabytes, e.g., due to two-step programming [13]. Due to these requirements and associated costs, ZNS SSDs are expected to have 8-32 active zones. Although the number of active zones can be further increased by adding extra power capacitors, utilizing DRAM for data movement, reduce parity requirements, or deploying a form of write-back cache (e.g., SLC).

3.2 Host Software Adoption

We now discuss three approaches for adapting host software to the ZNS interface. Applications that perform mostly sequential writes are prime candidates for adopting ZNS, such as Log Structure Merge (LSM) tree-based databases. Applications that primarily perform in-place updates are more challenging to support without fundamental modifications to core data structures [47].

Host-side FTL (HFTL). An HFTL acts as a mediator between (1) a ZNS SSD's write semantics and (2) applications performing random writes and in-place updates. The HFTL layer is similar to the responsibilities of the SSD FTL, but the HFTL layer manages only the translation mapping and associated garbage collection. Although it has less responsibility than an SSD FTL, an HFTL must manage its utilization of CPU and DRAM resources because it shares them with host applications. An HFTL makes it easier to integrate host-side information and enhances control of data placement and garbage collection, while also exposing the conventional block interface to applications. Existing work, such as dm-zoned [44], dm-zap [24], pblk [9], and SPDK's FTL [40], shows the feasibility and applicability of an HFTL, but only dm-zap currently supports ZNS SSDs.

File Systems. Higher-level storage interfaces (such as the POSIX file system interface) enable multiple applications to access storage by means of common file semantics. By integrating zones with higher layers of the storage stack, i.e., ensuring a primarily sequential workload, we eliminate the overhead that is otherwise associated with both HFTL and FTL data placement [30], as well as the indirection overhead [64] associated with them. This also allows additional data characteristics known to higher storage stack layers to be used to improve on-device data placement (at least to the degree that the application's actual workload allows this information to permeate to such layers).

The bulk of the file systems developed today primarily perform in-place writes and are generally difficult to adapt to the Zoned Storage model. Some file systems, however, such as f2fs [36], btrfs [53], and zfs [41] exhibit overly-sequential

write patterns and are already adapting so that they can support zones [4, 43]. Although not all of their writes are sequential (such as superblock and some metadata writes), file systems like these can be extended with strict log-structured writes [43], a floating superblock [4], and similar functionality to bridge the gap. These file systems effectively mimic the HFTL logic (with the LBA mapping table managed on-disk through metadata) while also implementing garbage collection to defragment data and free up space for new writes. Although zone support exists for f2fs and btrfs, they support only the zone model that is defined in ZAC/ZBC. As part of this work, we implement the necessary changes to f2fs to showcase the relative ease of supporting ZNS's zone model, and evaluate its performance (§4.1).

End-to-End Data Placement. In an ideal world, zone-write semantics would be aligned with an application's existing data structures. This would allow the highest degree of freedom by enabling the application to manage data placement, while at the same time eliminating indirection overheads from file-system and translation layers. While end-to-end data placement enables a collaboration between the application and ZNS SSD, and has the potential to achieve the best write amplification, throughput, and latency improvements, it is as daunting as interacting with raw block devices.

File semantics are a useful abstraction, and by forgoing them one must not only integrate zone support, but also provide tools for the user to perform inspection, error checking, and backup/restore operations. Like file systems, applications that exhibit sequential write patterns are prime candidates for end-to-end integration. This includes LSM-tree-based stores such as RocksDB, caching-based stores such as CacheLib [7], and object stores such as Ceph SeaStore [55]. To showcase the benefits of end-to-end integration, we introduce ZenFS, a new RocksDB zoned storage backend and compare it to both (1) the XFS file system and (2) the f2fs file system, with and without integrated ZNS support.

4 Implementation

We have added support to four major software projects to evaluate the benefits of ZNS. First, we made modifications to the Linux kernel to support ZNS SSDs. Second, we modified the f2fs file system to evaluate the benefits of zone integration at a higher-level storage stack layer. Third, we modified the fio [6] benchmark to support the newly added ZNS-specific attributes. Fourth, we developed ZenFS [25], a novel storage backend for RocksDB that allows control of data placement through zones, to evaluate the benefits of end-to-end integration for zoned storage. We describe the relatively few changes necessary to support ZNS when building upon the existing ZAC/ZBC support for the first three projects [5, 42, 52] (§4.1) and finally detail the architecture of ZenFS (§4.2).

Table 1 shows the lines modified for each software project. All the modifications have been contributed and accepted into the respective codebases [12, 25, 50, 51].

Project	Lines Added	Lines Removed
Linux Kernel	647	53
f2fs (kernel)	275	37
f2fs (mkfs tool)	189	15
fio	342	58
ZenFS (RocksDB)	3276	2
Total	4729	165

Table 1: Lines modified across projects to add ZNS support.

4.1 General Linux Support

The Linux kernel's Zoned Block Device (ZBD) subsystem is an abstraction layer that provides a single unified zoned storage API on top of various zoned storage device types. It provides both an in-kernel API and an ioctl-based user-space API supporting device enumeration, report of zones, and zone management (e.g., zone reset). Applications such as fio utilize the user-space API to issue I/O requests that align with the write characteristics of the underlying zoned block device, regardless of its low-level interface.

To add ZNS support to the Linux kernel and the ZBD subsystem, we modified the NVMe device driver to enumerate and register ZNS SSDs with the ZBD subsystem. To support the ZNS SSD under evaluation, the ZBD subsystem API was further extended to expose the per zone capacity attribute and the active zones limit.

Zone Capacity. The kernel maintains an in-memory representation of zones (a set of zone descriptor data structures), which is managed solely by the host unless errors occur, in which case the zone descriptors should be refreshed from the specific disk. The zone descriptor data structure is extended with a new zone capacity attribute and versioning, allowing host applications to detect the availability of this new attribute.

Both fio and f2fs are updated to recognize the new data structure. Whereas fio simply had to avoid to issue write I/Os beyond the zone capacity, f2fs required additional changes.

f2fs manages capacity at the granularity of segments, typically 2MiB chunks. For zoned block devices f2fs manages multiple segments as a section, the size of which is aligned to the zone size. f2fs writes sequentially across a section's segments, and partially writeable zones are not supported. To add support for the zone capacity attribute in f2fs, the kernel implementation, and associated f2fs-tools, two extra segment types are added to the three conventional segment types (i.e., free, open, and full): an *unusable* segment type that maps the unwriteable part of a zone, and a *partial* segment type that covers the case where a segment's LBAs cross both the writeable and the unwriteable LBAs of a zone. The partial segment type explicitly allows optimizing for the case when segment chunk size and the zone capacity of a specific zone are unaligned, utilizing all of the writeable capacity of a zone.

To allow backward compatibility for SMR HDDs, which do not expose zone capacity, the zone capacity is initialized

to the zone size attribute.

Limiting Active Zones. Due to the nature of flash-based SSDs, there is a strict limit on the number of zones that can be active simultaneously, i.e., either in OPEN or CLOSED state. The limit is detected upon zoned block device enumeration, and exposed through the kernel and user-space APIs. SMR HDDs do not have such a limit and the attribute is initialized to zero (read: infinity).

No modifications were made to fio to support the limit, so it is the responsibility of the fio user to respect the active-limit constraint, which otherwise results in an I/O error when no more zones can be opened.

For f2fs, the limit is linked to the number of segments that can be open simultaneously. f2fs limits this to a maximum of 6, but can be reduced to align with the available active zone limit. This limit is set at the time the file system is created, and f2fs-tools is modified to check for the zone active limit, and if the zoned block device does not support enough active zones, the number of open segments is configured to align with what is available by the device.

f2fs requires its metadata to be stored on a conventional block device, necessitating a separate device. We do not directly address this in our modifications, as the evaluated ZNS SSD exposes a fraction of its capacity as a conventional block device. If this is not supported by the ZNS SSD, write-in-place functionality can be added similar to btrfs [4], or a small translation layer [24] can expose a limited set of zones on the ZNS SSD through the conventional block interface. Note that the Slack Space Recycling (SSR) feature (i.e., random writes) is disabled for all zoned storage devices, which decreases the overall performance. However, due to overall higher performance achieved by ZNS SSDs, we demonstrate superior performance to block-interface SSDs which have SSR enabled (§5.2).

4.2 RocksDB Zone Support

In this section we show how to adapt the popular key-value database RocksDB to perform end-to-end data placement onto zoned storage devices using the ZenFS storge backend. ZenFS takes advantage of the log-structured merge (LSM) tree [45] data structure of RocksDB that it uses to store and maintain its data, and its associated immutable sequential-only compaction process.

The LSM tree implementation consists of multiple levels, as shown in Figure 3. The first level (L_0) is managed in-memory and flushed to the level below periodically or when full. Intermediate updates between flushes are made durable using a Write-Ahead Log (WAL). The rest of the levels ($L_1, \ldots, L_n$) are maintained on-disk. New or updated key-value data pairs are initially appended to L_0, and upon flush the key-value data pairs are sorted by key, and then written to disk as a Sorted String Table (SST) file.

The size of a level is typically fitted to a multiple of the level above, and each level contains multiple SSTs with each

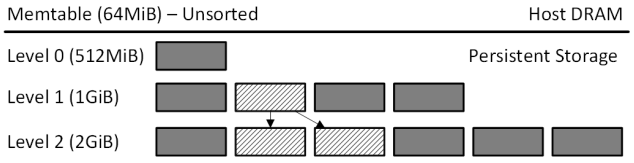

Figure 3: Data organization in RocksDB. Dark grey squares represent SSTs. Light grey with lines squares represent SSTs selected for compaction.

SST containing an ordered set of non-overlapping key-value data pairs. Through an explicit compaction process, an SST's key-value data pairs are merged from one level (L_i) to the next (L_{i+1}). The compaction process reads key-value data pairs from one or more SSTs and merges them with the key-value pairs from one or more SSTs in the next level. The merged result is stored in a new SST file and replaces the merged SSTs in the LSM tree. As a result of this process, SST files are immutable, written sequentially, and created/removed as a single unit. Furthermore, hot/cold data separation is achieved as key-value data pairs are merged into levels below.

RocksDB has support for separate storage backends through its file system wrapper API that is a unified abstraction for RocksDB to access its on-disk data. At its core, the wrapper API identifies data units, such as SST files or Write-Ahead Log (WAL), through a unique identifier (e.g., a file-name) that maps to a byte-addressable linear address space (e.g., a file). Each identifier supports a set of operations (e.g., add, remove, current size, utilization) in addition to random access and sequential-only byte-addressable read and write semantics. These are closely related to file system semantics, where identifier and data is accessible through files, which is RocksDB's main storage backend. By using a file system that manages files and directories, RocksDB avoids managing file extents, buffering, and free space management, but also looses the ability to place data directly into zones, which prevents end-to-end data placement onto zones, and therefore reduces the overall performance.

4.2.1 ZenFS Architecture

The ZenFS storage backend implements a minimal on-disk file system and integrates it using RocksDB's file-wrapper API. It carefully places data into zones while respecting their access constraints, and collaborates with the device-side zone metadata on writes (e.g., write pointer), reducing complexity associated with durability. The main components of ZenFS are depicted in Figure 4 and described below.

Journaling and Data. ZenFS defines two types of zones: journal and data zones. The journal zones are used to recover the state of the file system, and maintains the superblock data structure, and mapping of WAL and data files to zones, whereas the data zones store the file content.

Extents. RocksDB's data files are mapped and written to a

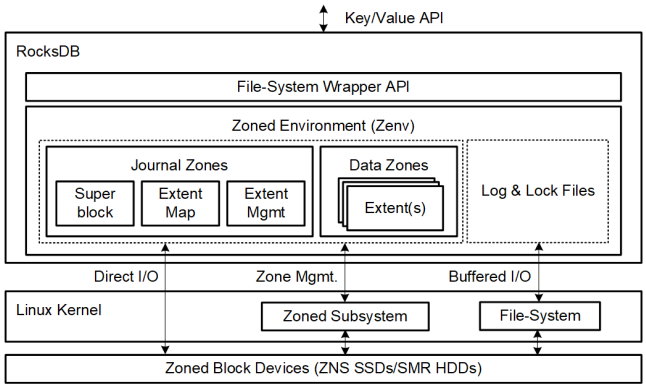

Figure 4: ZenFS Architecture

set of extents. An *extent* is a variable-sized, block-aligned, contiguous region that is written sequentially to a data zone, containing data associated to a specific identifier. Each zone can store multiple extents, but extents do not span zones. Extent allocation and deallocation events are recorded in an in-memory data structure and written to the journal when a file is closed or the data is requested to be persisted by RocksDB through an `fsync` call. The in-memory data structure keeps track of the mapping of extents to zones, and once all files with allocated extents in a zone has been deleted the zone can be reset and reused.

Superblock. The superblock data structure is the initial entry point when initializing and recovering ZenFS state from disk. The superblock contains a unique identifier for the current instance, magic value, and user options. A unique identifier (UUID) in the superblock allows the user to identify the file-system even if the order of block device enumeration on the system changes.

Journal. The responsibility of the journal is to maintain (1) the superblock and (2) the WAL and data file to zone translation mapping through extents.

The journal state is stored on dedicated journal zones, and is located on the first two non-offline zones of a device. At any point in time, one of the zones are selected as the active journal zone, and is the one that persists updates to the journal state. A journal zone has a header that is stored at the beginning of the specific zone. The header stores a sequence number (incremented each time a new journal zone is initialized), the superblock data structure, and a snapshot of the current journal state. After the header are stored, the remaining writable capacity of the zone is used to record updates to the journal.

To recover the journal state from disk, three steps are required: ① the first LBA for each of the two journal zones must be read to determine the sequence number of each, where the journal zone with the highest value is the current active zone; ② the full header of the active zone is read and initializes the initial superblock and journal state; and finally ③ journal updates are applied to the header's journal snapshot.

The amount of updates to apply is determined by the zone's state, and its write pointer. If the zone is in the OPEN (or CLOSED) state, only records up to the current write pointer value are replayed to the journal, whereas if the zone is in the FULL state, all records stored after the header are replayed. Note that if the zone is full, after recovery a new active journal zone is selected and initialized to enable persisting journal updates.

The initial journal state is created and persisted by an external utility that is similar to existing file system tools that generate the initial on-disk state of a file system. It writes the initial sequence number, superblock data structure and an empty snapshot of the journal to the first journal zone. When ZenFS is initialized by RocksDB, the above recovery process is executed, after which it is ready for data accesses from RocksDB.

SST file size	128 MiB	256 MiB	512 MiB
Write Amp.	11.9×	12.0×	12.0×
Runtime (s)	15,430	15,461	14,918
99.99 RW lat (ms)	102	102	105
99.99 RWL lat (ms)	77	73	73

Table 2: RocksDB's write amplification and runtime during the *fillrandom* benchmark, and 99.99-th percentile tail latencies during the *readwhilewriting* benchmark with no rate limits (RW) and with writes rate-limited to 20 MiB/s (RWL), with different SST file sizes.

Writeable Capacity in Data Zones. Ideal allocation, resulting in maximum capacity usage over time, can only be achieved if file sizes are a multiple of the writeable capacity of a zone allowing file data to be completely separated in zones while filling all available capacity. File sizes can be configured in RocksDB, but the option is only a recommendation and sizes will vary depending on the results of compression and compaction processes, so exact sizes are not feasible. ZenFS addresses this by allowing a user-configurable limit for finishing data zones, specifying the percentage of the zone capacity remaining. This allows the user to specify a file size recommendation of, e.g., 95% of device's zone capacity by setting the finish limit to 5%. This allows for the file size to vary within a limit and still achieve file separation by zones. If the file size variation is outside the specified limit, ZenFS will make sure that all available capacity is utilized by using its zone allocation algorithm (described below). Zone capacities are generally larger than the RocksDB recommended file size of 128 MiB and to make sure that increasing the file size does not increase RocksDB write amplification and read tail latencies we measured the impact on different file sizes. Table 2 shows that increasing SST file sizes does not significantly reduce performance.

Data Zone Selection. ZenFS employs a best-effort algorithm to select the best zone to store RocksDB data files. RocksDB separates the WAL and the SST levels by setting a write-lifetime hint for a file prior to writing to it. Upon the first write to a file, a data zone is allocated for storage. ZenFS tries first to find a zone based on the lifetime of the file and the max lifetime of the data stored in the zone. A match is only valid if the lifetime of the file is less than the oldest data stored in the zone to avoid prolonging the life of the data in the zone. If several matches are found, the closest match is used. If no matches are found, an empty zone is allocated. If the file fills up the remaining capacity of the allocated zone, another zone is allocated using the same algorithm. Note that the write life time hint is provided to any RocksDB storage backend, and is therefore also passed to other compatible file systems and can be used together with SSDs with stream support. We compare both approaches to pass hints in §5.3. *By using the ZenFS zone selection algorithm and the user-defined writeable capacity limits, the unused zone space or space amplification is kept at around 10%.*

Active Zone Limits. ZenFS must respect the active zone limits specified by the zoned block device. To run ZenFS, a minimum of three active zones are required, which are separately allocated to the journal, WAL, and compaction process. To improve performance, the user can control the number of concurrent compactions. Our experimentation has shown that by limiting the number of concurrent compactions, *RocksDB can work with as few as 6 active zones with restricted write performance*, while more than 12 active zones does not add any significant performance benefits.

Direct I/O and Buffered Writes. ZenFS leverages the fact that writes to SST files are sequential and immutable and performs direct I/O writes for SST files, bypassing the kernel page cache. For other files, such as the WAL, ZenFS buffers writes in memory and flushes the buffer when it is full, the file is closed, or when RocksDB requests a flush. If a flush is requested, the buffer is padded to the next block boundary and an extent with the number of valid bytes is stored in the journal. The padding results in a small amount of write amplification, but it is not unique to ZenFS, and is similarly done in conventional file system.

5 Evaluation

To evaluate the benefits and compare the performance of ZNS SSDs, we utilize a production SSD hardware platform that can expose itself as either a block-interface SSD or a ZNS SSD. As such, we enable an apples-to-apples comparison between the two interfaces. The block-interface SSD is formatted to have either 7% OP or 28% OP, and supports streams. ZNS SSD-specific details are shown in Table 3.

Our evaluation of ZNS is constructed around evaluating the following aspects:

- **Raw device I/O performance (§5.1).** We perform an apples-to-apples comparison between the block-interface SSDs and the ZNS SSD. We find that the ZNS SSD achieves higher concurrent write throughput and lower random read latency in the presence of writes.
- **End-to-end application performance (§5.2).** We compare the performance of RocksDB on a block-interface SSD running the XFS and f2fs file systems, with the performance of RocksDB when running on a ZNS SSD using our ZenFS backend. We find that RocksDB achieves up to 2× higher read and write throughput, and an order of magnitude lower random read tail latency when running over the ZNS SSD.
- **End-to-end performance comparison with SSD Streams (§5.3).** We compare the performance of the ZNS SSD with a Streams-enabled block-interface SSD, by running RocksDB on XFS and f2fs. We find that RocksDB achieves up to 44% higher throughput on ZNS and up to half the tail latency compared to the Stream SSD.

SSD Interface	Conv.	Conv.	ZNS
Media Capacity	2 TiB	2 TiB	2 TiB
Host Capacity	1.92 TB	1.60 TB	2 TB
Over-provisioning	7%	28%	0%
Placement Type	None	None	Zones
Max Active Zones	N/A	N/A	14
Zone Size	N/A	N/A	2048 MiB
Zone Capacity	N/A	N/A	1077 MiB

Table 3: Feature summary of the evaluated SSDs

All experiments are performed on a Dell R7515 system with a 16-core AMD Epyc 7302P CPU and 128 GB of DRAM (8×16GB DDR4-3200Mhz). The baseline system configuration is an Ubuntu 20.04 distribution, updated to the 5.9 Linux kernel—the first version that includes our contributions as described in §4.1. RocksDB experiments were carried out on RocksDB 6.12 with ZenFS included as a backend storage plugin.

5.1 Raw I/O Characteristics

To ensure that the same workload is applied to both the block-interface SSD and ZNS SSDs, we divide the block-interface SSD address space into LBA ranges that have the same number of LBAs as the zone capacity exposed by the ZNS SSD. These zones, or LBA ranges, respect the same sequential write constraint as ZNS, but the zone reset is simulated by trimming the LBA range before writing. We measure performance after the SSD has reached its steady-state with a given workload.

Sustained Write Throughput. We evaluate SSD throughput to show the internal SSD garbage collection impact on throughput and its ability to consume host writes. The experiment issues 64KiB write I/Os to 4 zones. When one zone

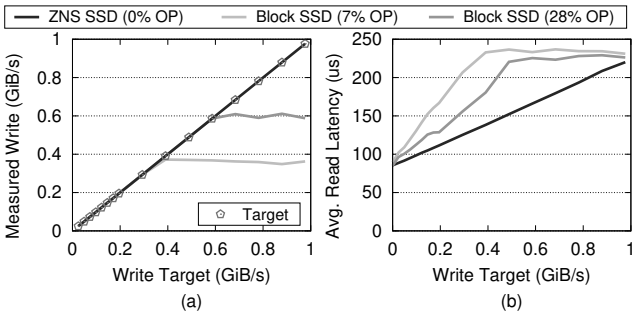

Figure 5: Measured (a) write throughput and (b) average random read latency during rate-limited writes.

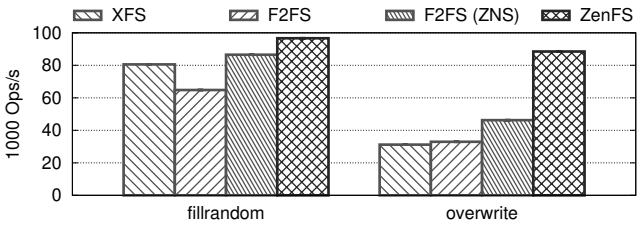

Figure 6: Throughput of RocksDB with write-heavy benchmarks— *fillrandom* followed by *overwrite* using the block-interface SSD with 28% OP and the ZNS SSD.

is full, a new zone is chosen, reset, and written sequentially. While running the experiment, we measure the drive's ability to reach a specific host write throughput target ranging from 0 to 1GiB/s. Figure 5 (a) shows the achieved write throughput for each write target. The block-interface SSDs sustain target writes up to 300MiB/s (0% OP) and 500MiB/s (28% OP), whereas the ZNS SSD sustains 1GiB/s. This aligns with the measured steady-state throughput shown in Figure 1, which shows that the block-interface SSDs achieve 370MiB/s (7% OP) and 590MiB/s (28% OP), respectively. The ZNS SSD with 0% OP, however, reaches 1010MiB/s. This is $1.7-2.7\times$ higher write throughput, and $7-28\%$ more storage capacity which is no longer required by the device-side garbage collection.

The ZNS SSD achieves higher throughput and lower write amplification as it can align writes onto distinct erase blocks and avoid garbage collection. The block-interface SSD mixes data onto the same erase block, which is ultimately garbage collected at separate points in time, increasing garbage collection overhead.

Random Reads and Writes. To demonstrate the reduced latency of random reads on a ZNS SSD, a second process is added to the previous experiment, which simultaneously performs random 4 KiB read I/Os across the SSD. We measure its average read latency while gradually increasing the host write target. Figure 5 (b) shows the average random read latency of the block-interface SSDs and the ZNS SSD. With no ongoing writes, both block-interface SSDs report a 4KiB read latency of 85µs. Compared to the SSDs with 7% and 28% OP, the random latency of the ZNS SSD is 19% and 5% lower at 50MiB/s write throughput, 45% and 16% lower at 150MiB/s write throughput, and finally 64% and 27% lower at 300MiB/s write throughput. The ZNS SSD's throughput continues to increase linearly with the write target, unlike the block-interface SSDs.

5.2 RocksDB

Next, we demonstrate the performance improvements achieved by RocksDB when it runs on a ZNS SSD accessed either through f2fs with ZNS support added, or on top of our end-to-end optimized storage backend ZenFS. We compare these two setups against RocksDB running on XFS and f2fs using the block-interface SSD. f2fs supports both storage interfaces, allowing us to compare the impact of a file system running of top of a ZNS SSD. Furthermore, ZenFS is optimized for ZNS, showing the benefits of integrating data placement into an I/O intensive application.

Five workloads are executed for each setup. *fillrandom* is a write-intensive workload pre-conditioning the SSDs for follow-on benchmarks by issuing writes to fill drive capacity several times. Second, *overwrite* measures the overall performance when overwriting. Third, *readwhilewriting* performs random reads while writing. Fourth, *random reads* performs as described. Finally, *readwhilewriting* performs as described with the addition of write rate-limiting. The last three benchmarks are ran three times each and their results are averaged.

The benchmarks are scaled to 80% of the SSD. As a result, the actual over-provisioned space is 27% and 48% with respect to 7% OP and 28% OP. In the interest of space, we show mainly the 28% results, but show the performance difference for the write-heavy benchmarks and the rate-limited *readwhilewriting* benchmark. A 128 MiB target SST file size is used in all benchmarks except for the ZenFS workloads where the target file size is configured to align with the zone capacity.

Write Throughput. We first study the write throughput improvements, using two write-heavy benchmarks on the SSD with 28% OP and the ZNS SSD. First, we populate the database using the *fillrandom* benchmark with 3.8 billion 20B keys and 800B values (compressed to 400B). Then, we randomly overwrite all values using the *overwrite* benchmark.

Figure 6 shows the average throughput of the two benchmarks over the four setups. In the *fillrandom* benchmark, we report the average of write operations per second (ops/s) for each run and is executed from a clean state. Thus, the result is impacted by the lower garbage collection overhead until the SSD reaches a steady-state. As a result, the full write amplification impact can be first seen in the *overwrite* benchmark. Both benchmarks show the XFS and f2fs setups perform lower than f2fs (ZNS) and ZenFS. The most significant impact is seen in the *overwrite* benchmark, in which the garbage collection overhead heavily impacts the overall performance.

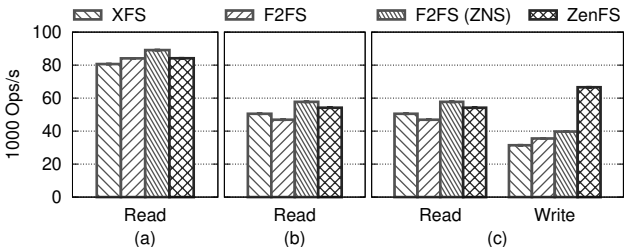

Figure 7: Throughput of RocksDB reads during (a) the *randomread* benchmark, (b) the *readwhilewriting* benchmark with writes rate-limited to 20 MiB/s and (c) the *readwhilewriting* benchmark with no write limits using the block-interface SSD with 28% OP and the ZNS SSD.

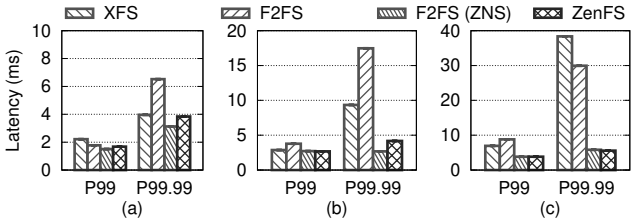

Figure 8: Latency of RocksDB reads during the (a) *randomread* benchmark, (b) the *readwhilewriting* benchmark with writes rate-limited to 20 MiB/s and (c) *readwhilewriting* benchmark with no write limits using the block-interface SSD with 28% OP and the ZNS SSD.

ZenFS is 183% faster than XFS, and f2fs (ZNS) performs 42% better than XFS and 33% better than f2fs.

While the performance of f2fs ZNS increases, the garbage collection overhead associated to large section size requires more data to be moved upon cleaning, which ultimately impacts the host-side garbage collection. To confirm, we measure the write amplification factor on the block-interface SSD, which is reported as 2.0× for XFS, and 2.4× for f2fs, whereas no device-side garbage collection (i.e., 1.0×) occured for f2fs (ZNS) and ZenFS.

Read While Writing. As we have already shown (§5.1), the ZNS SSD achieves lower read latency during concurrent writes because there are no in-device garbage collection operations to interfere with reads during writes. We now demonstrate how this affects the latency of RocksDB read throughput using two read-intensive benchmarks: *randomread* and *readwhilewriting*. The first initiates 32 threads that perform a random reads of key-value pairs on a drive that has been pre-conditioned by the write-intensive benchmarks. The second initiates an extra thread that performs random overwrites, in addition to the 32 threads performing reads.

First, we run the *randomread* benchmark, next we run *readwhilewriting* while rate-limiting writes to 20 MiB/s (i.e., 25.6 Kops/s), and finally we run *readwhilewriting* with no rate-limiting of writes. We run each of these benchmarks for 30 minutes, two times, again in the same four setups as before,

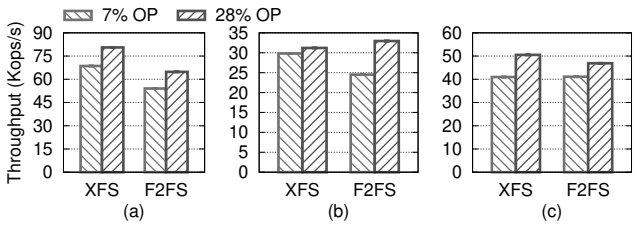

Figure 9: Throughput of RocksDB writes during (a) *fillrandom*, (b) *overwrite*, and (c) *readwhilewriting* benchmark, using the block-interface SSD with 7% OP, as well as 28% OP.

and report read/write throughput in Figure 7 and the average and tail read latencies per operation in Figure 8.

For the *randomread* benchmark, we observe that each combination achieves similar number of write ops/s and average latency, which is expected as there is no writes ongoing. However, the 99th percentile (P99) latencies show that ZenFS is 25% lower than XFS, and 6% lower than f2fs, while f2fs (ZNS) has 32% lower latency than XFS and 16% lower than f2fs.

Next, we study the throughput when performing rate-limited writes. We first notice that only ZenFS is able to sustain 20MiB/s writes while reading, with the other file systems are doing 15% less writes. f2fs (ZNS) performs the most read ops/s, followed by ZenFS, f2fs and XFS. Furthermore, we observe that ZenFS and f2fs (ZNS) have the lowest average latencies, and are able to achieve P99.9 read latencies that are 2× and 4× lower than for XFS and f2fs, respectively.

Finally, we remove the write rate-limit, and then measure the overall impact. Removing the write limit allows ZenFS to achieve 2× the write ops/s compared to f2fs and XFS on consumer SDDs at higher level of read ops/s. f2fs (ZNS) achieves the highest amount of read ops/s and significantly higher write throughput than XFS and f2fs. The P99.99 read tail latencies stand out for ZenFS and f2fs (ZNS) which are an order of magnitude lower than f2fs and XFS.

Impact of Capacity Over-Provisioning. We also evaluate the impact of SSD capacity over-provisioning on RocksDB performance. Whereas block-interface SSDs thrive on unused capacity, as garbage collection is improved and thereby lowering its write amplification factor. The ZNS SSD do not exhibit this behavior and can use all available capacity, while still maintaining a write amplification factor of $\sim 1\times$.

Figure 9 shows the measured number of operations per second for XFS and f2fs when executing the *fillrandom*, *overwrite*, and rate-limited textitreadwhilewriting benchmarks. The benchmarks are executed with 7% OP and 28% OP, respectively. For the *fillrandom* benchmark, an improvement between 14% and 16% in overall operations is seen, whereas in the *overwrite* benchmark, XFS only sees a modest 4% improvement, while f2fs is improved by 25%. Finally, in the *readwhilewriting* benchmark we see an improvement between

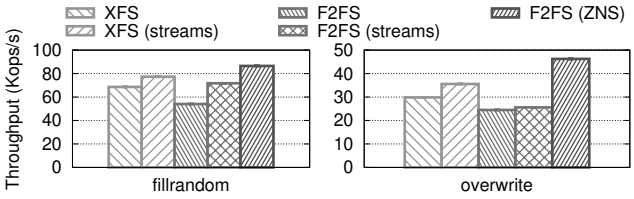

Figure 10: The throughput of RocksDB during the *fillrandom* and *overwrite* benchmarks when executing on an block-interface SSD with 7% OP and stream support.

12% and 18% for f2fs and XFS. While the overall throughput is important, the latency is as well. For XFS, the tail latency improves by 30% and by 23% for P99 and P99.9. Thus, if latency is more important than throughput, then one must budget with addition over-provisioning to lower the overall latency caused by extra garbage collection.

5.3 Streams

Finally, we evaluate the performance of a block-interface SSD with stream support against the performance of a ZNS SSD.

Figure 10 shows the throughput of the *fillrandom* and *overwrite* benchmarks executing on top of a block-interface SSD with 7% OP and with streams enabled or disabled on top of XFS and f2fs. RocksDB on XFS shows higher throughput by 11% and 16% with Streams on *fillrandom* and *overwrite* respectively. RocksDB on f2fs shows higher throughput by 24% and 4% with Streams on *fillrandom* and *overwrite* respectively. f2fs (ZNS) achieves 17% and 44% higher throughput compared to f2fs on a block-interface SSD with Streams for *fillrandom* and *overwrite* respectively.

Furthermore, the P99 latency for f2fs drops from 9,435µs to 6,529µs with Streams support. This is very close to ZenFS's P99 latency during the same benchmark of 3,734µs. We therefore find that RocksDB achieves up to 44% higher throughput on ZNS and close to half the tail latency compared to a block-interface SSD with streams [56].

6 Related Work

This section covers related work on host-device collaboration, research platforms, and key-value store designs. This section covers work that was not already covered as necessary background (§2.2).

Host-Device Collaboration. Significant research has gone into optimizing the storage interface between host and SSD. Josephson et al. implement DFS [30], a host-side file system tightly integrated with the underlying hardware. SDF [46] exposes individual flash dies to the host through a custom-designed storage controller, and FlashBlox [26] uses dedicated channels and dies for each application to improve isolation. Application-Managed Flash [38] defines an interface based on fixed-size segments that can be written sequentially and reset by a trim operation. ZNS does not fix the number of erase blocks per zone, which allows the SSD implementation to map erase blocks dynamically to zones. Zhang et al. examine sharing the responsibility of data placement by enabling the SSD to notify the host upon data relocation events [67]. ZNS provides a path to have both high performance and low cost, while leaving the reliability and coarse-grained management to the device.

Research Platforms. Various SSD hardware [14, 37, 58] and software [11, 39, 66] platforms have been developed over the years to expose the characteristics inherent to SSDs [10, 23]. This enables key research on storage interfaces [30, 54, 63] and makes it possible to improve upon the block device interface [27, 62]. Each of these works target specific optimizations and platforms, enabling the storage community to build on them. ZNS is built upon these advancements, each of which has enabled further exploration of the block interface design space.

Key-value Store Designs. Previous work introduces multiple key-value store designs for non-block interfaces. LOCS [61] takes end-to-end design to the extreme and modifies LevelDB to leverage the channel parallelism in OCSSDs. Unlike LOCS, which is tied to a specific OCSSD platform, ZenFS targets the general zone interface and thereby enables RocksDB to run on both ZNS SSDs and HM-SMR HDDs. Similarly, SMRDB [49], GearDB [65], and BlueFS [2] target strictly HM-SMR HDDs and are therefore unable to take the advantage of ZNS extensions and run on ZNS SSDs.

Conclusion

ZNS enables higher performance and lower-cost-per-byte flash-based SSDs. By shifting responsibility for managing data organization within erase blocks from FTLs to host software, ZNS eliminates in-device LBA-to-page maps, garbage collection and over-provisioning. Our experiments with ZNS-specialized f2fs and RocksDB implementations show substantial improvements in write throughput, read tail latency, and write-amplification when compared to conventional FTLs running on identical SSD hardware.

Acknowledgments

We thank the 22 anonymous reviewers, and our shepherd Ethan Miller for their help improving the presentation of this paper. We thank the members and companies of the PDL Consortium: Amazon, Facebook, Google, Hewlett Packard Enterprise, Hitachi Ltd., IBM Research, Intel Corporation, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital for their interest, insights, feedback, and support. This work has been made possible in part by gifts from VMware's University Research Fund and from the NetApp University Research Fund.

References

[1] Abutalib Aghayev and Peter Desnoyers. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, USA, February 2015. USENIX Association.

[2] Abutalib Aghayev, Sage Weil, Greg Ganger, and George Amvrosiadis. Reconciling LSM-Trees with Modern Hard Drives using BlueFS. Technical Report CMU-PDL-19-102, CMU Parallel Data Laboratory, April 2019.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design Tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, volume 57. Boston, USA, 2008.

[4] Naohiro Aota. btrfs: Zoned block device support. https://lwn.net/Articles/836726/, 2020.

[5] Bart Van Assche. fio: Add zoned block device support. https://github.com/axboe/fio/pull/585, 2018.

[6] Jens Axboe. Flexible I/O Tester. git://git.kernel.dk/fio.git, 2016.

[7] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[8] Matias Bjørling. NVM Express Zoned Namespaces Command Set 1.0, June 2020. Available from http://www.nvmexpress.org/specifications.

[9] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.

[10] Luc Bouganim, Bjorn Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the Int'l Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, 2009.

[11] John S Bucy, Gregory R Ganger, et al. *The DiskSim simulation environment version 3.0 reference manual*. School of Computer Science, Carnegie Mellon University, 2003.

[12] Keith Busch, Hans Holmberg, Ajay Joshi, Aravind Ramesh, Niklas Cassel, Matias Bjørling, Damien Le Moal, and Keith Busch. Linux kernel Zoned Namespace support. https://lwn.net/ml/linux-block/20200615233424.13458-6-keith.busch@wdc.com/, 2020.

[13] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 49–60. IEEE, 2017.

[14] Yu Cai, Erich F Haratsch, Mark McCartney, and Ken Mai. FPGA-based Solid-state Drive Prototyping Platform. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 101–104. IEEE, 2011.

[15] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 123–130. IEEE, 2013.

[16] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[17] Peter Desnoyers. Analytic Modeling of SSD Write Performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[18] DRAMeXchange. NAND Flash Spot Price, September 2014. http://dramexchange.com.

[19] Tim Feldman and Garth Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *;login*, 38(3), June 2013.

[20] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, April 2011.

[21] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. *ACM SIGPLAN Notices*, 44(3):229, 2009.

[22] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, Santa Clara, CA, 2016. USENIX Association.

[23] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the twelfth European conference on computer systems*, pages 127–144. ACM, 2017.

[24] Hans Holmberg. dm-zap: Host-based FTL for ZNS SSDs. https://github.com/westerndigitalcorporation/dm-zap, 2021.

[25] Hans Holmberg. RocksDB: ZenFS Storage Backend. https://github.com/westerndigitalcorporation/zenfs/, 2021.

[26] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, February 2017. USENIX Association.

[27] Ping Huang, Guanying Wu, Xubin He, and Weijun Xiao. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proceedings of the Ninth European Conference on Computer Systems*, page 22. ACM, 2014.

[28] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., September 2014. Available from http://www.t10.org/.

[29] INCITS T13 Technical Committee. Information technology - Zoned Device ATA Command Set (ZAC). Draft Standard T13/BSR INCITS 537, American National Standards Institute, Inc., December 2015. Available from http://www.t13.org/.

[30] William K Josephson, Lars A Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14, 2010.

[31] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme. *ACM Transactions on Embedded Computing Systems*, 9(4):1–41, March 2010.

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

[33] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189, Santa Clara, CA, 2015. USENIX Association.

[34] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A Space-efficient Flash Translation Layer for CompactFlash Systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, May 2002.

[35] D. Le Moal, Z. Bandic, and C. Guyot. Shingled File System Host-side Management of Shingled Magnetic Recording Disks. In *2012 IEEE International Conference on Consumer Electronics (ICCE)*, pages 425–426, 2012.

[36] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[37] S. Lee, K. Fleming, J. Park, Ha K, Adrian Caulfield, Steven Swanson, Arvind, and J. Kim. BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs. In *Proceedings of the 2010 Workshop on Architectural Research Prototyping*, January 2010.

[38] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, February 2016. USENIX Association.

[39] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST)*, pages 83–90, 2018.

[40] Wojciech Malikowski. SPDK Open-Channel SSD FTL. https://spdk.io/doc/ftl.html, 2018.

[41] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[42] Damien Le Moal. f2fs: Zoned block device support. https://www.spinics.net/lists/linux-fsdevel/msg103443.html, 2014.

[43] Damien Le Moal. f2fs: Zoned block device support. https://www.spinics.net/lists/linux-fsdevel/msg103443.html, 2016.

[44] Damien Le Moal. dm-zoned: Zoned Block Device device mapper. https://lwn.net/Articles/714387/, 2017.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[46] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 471–484, New York, NY, USA, 2014. Association for Computing Machinery.

[47] Adrian Palmer. SMRFFS-EXT4—SMR Friendly File System. https://github.com/Seagate/SMR_FS-EXT4, 2015.

[48] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, August 2008.

[49] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Aravind Ramesh, Hans Holmberg, and Shin'ichiro Kawasaki. fio zone capacity support. https://www.spinics.net/lists/fio/msg08752.html, 2020.

[51] Aravind Ramesh, Damien Le Moal, and Niklas Cassel. f2fs: Zoned Namespace support. https://www.mail-archive.com/linux-f2fs-devel@lists.sourceforge.net/msg17567.html/, 2020.

[52] Hannes Reinecke. Support for zoned block devices. https://lwn.net/Articles/694966/, July 2016.

[53] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[54] Mohit Saxena, Yiying Zhang, Michael M Swift, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *11th USENIX Conference on File and Storage Technologies (FAST)*, pages 215–228, 2013.

[55] Seastar. Ceph SeaStore optimizes for NVMe SSDs. https://docs.ceph.com/en/latest/dev/seastore/, March 2020.

[56] Samsung Semiconductor. Performance Benefits of Running RocksDB on Samsung NVMe SSDs, 2015.

[57] Ryan Smith. SSD Cost Pricing Calculator. https://www.soothsawyer.com/ssd-cost-pricing-calculator/, 2019.

[58] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openSSD: A PCIe-based open source SSD platform. *Proc. Flash Memory Summit*, 2014.

[59] Theano Stavrinos, Daniel Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS XVIII)*. ACM, 2021.

[60] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree based Key-value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.

[61] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[62] Yeong-Jae Woo and Jin-Soo Kim. Diversifying Wear Index for MLC NAND Flash Memory to Extend the Lifetime of SSDs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 6. IEEE Press, 2013.

[63] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-Tail Flash: Near-perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Transactions on Storage (TOS)*, 13(3):22, 2017.

[64] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log on My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[65] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 159–171, Boston, MA, February 2019. USENIX Association.

[66] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choil, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual Machine-based SSD Simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2013.

[67] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2012.