Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS)

Krijn Doekemeijer[*1], Nick Tehrany[*1,2], Balakrishnan Chandrasekaran[1], Matias Bjørling[3], and Animesh Trivedi[1]

[1]Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
[2]Delft University of Technology, Delft, the Netherlands
[3]Western Digital, Copenhagen, Denmark
{k.doekemeijer, n.a.tehrany, b.chandrasekaran, a.trivedi}@vu.nl, matias.bjorling@wdc.com

Abstract—The recent emergence of NVMe flash devices with Zoned Namespace support, ZNS SSDs, represents a significant new advancement in flash storage. ZNS SSDs introduce a new storage abstraction of append-only *zones* with a set of new I/O (i.e., append) and management (zone state machine transition) commands. With the new abstraction and commands, ZNS SSDs offer more control to the host software stack than a non-zoned SSD for flash management, which is known to be complex (because of garbage collection, scheduling, block allocation, parallelism management, overprovisioning). ZNS SSDs are, consequently, gaining adoption in a variety of applications (e.g., file systems, key-value stores, and databases), particularly latency-sensitive big-data applications. Despite this enthusiasm, there has yet to be a systematic characterization of ZNS SSD performance with its zoned storage model abstractions and I/O operations. This work addresses this crucial shortcoming. We report on the performance features of a commercially available ZNS SSD (13 key observations), explain how these features can be incorporated into publicly available state-of-the-art ZNS emulators, and recommend guidelines for ZNS SSD application developers. All artifacts (code and data sets) of this study are publicly available at https://github.com/stonet-research/NVMeBenchmarks.

Index Terms—Measurements, NVMe storage, Zoned Namespace Devices

I. INTRODUCTION

The emergence of fast flash storage in data centers, HPC, and commodity computing has fundamentally effected changes in every layer of the storage stack, and led to a series of new developments such as a new host interface (NVM Express, NVMe) [1], [2], [3], a high-performance block layer [4], [5], [6], [7], new storage I/O abstractions [8], [9], [10], [11], [12], [13], [14], and re/co-design of storage application stacks [15], [16], [17], [18], [19], [20], [21]. Today, flash-based solid-state drives (SSDs) can support very low latencies (i.e., a few microseconds), and multi GiB/s bandwidth with millions of I/O operations per second [22], [23], [24].

Despite these advancements, the conceptual model of a storage device remains unchanged since the introduction of hard disk drives (HDDs) more than half a century ago. A storage device supports only two necessary operations: write and read data in units of *sectors* (or blocks) [25]. Data can be read from and written to anywhere on the device, hence

supporting random and sequential I/O operations. Though this model works with conventional HDDs, it is not apt for flash-based storage devices as flash internally does not support overwriting data [26], [27], [28]. Flash devices offer the illusion of "overwritable" storage via the *flash translation layer (FTL)*, a software component that runs within the device. The FTL enables easy integration of flash devices (by allowing them to masquerade as fast HDDs), albeit it introduces unpredictability in performance [29], [30], [31], [32], [33], [34] and complicates device lifetime management [35]. These challenges are defined as the *unwritten contracts* of SSDs [26]. As data centers have largely transitioned to SSDs for fast, reliable storage [36], [37], and modern big data applications have high QoS demands [38], [39], there is a dire need to address these unwritten contracts.

Researchers and practitioners advocate for open flash SSD interfaces beyond block I/O [40] to address these challenges. Examples include *Open-Channel SSDs* (OCSSD) [41], *multi-stream SSDs* [9], and, more recently, *Zoned Namespaces* (ZNS) [11]. The focus of this work is on NVMe devices that support ZNS, which are commercially available today [42], [43]. ZNS promises a low and stable tail latency [11] and a high device longevity, and, hence, addresses the needs of modern big-data workloads. There is, unsurprisingly, a rich body of active and recent work on ZNS [44], [11], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56]. Despite this enthusiasm, there has *not* been a systematic performance and operational characterization of ZNS SSDs. The lack of an extensive characterization of ZNS SSDs severely limits the utilization and application of ZNS devices in big-data workloads. In this work, we bridge this gap by presenting the performance characterization of a commercially-available NVMe ZNS device.

We complement this characterization of a physical device with an investigation of emulated ZNS devices, since they are widely used in research [51], [57], [58], [55]. Emulated devices enable researchers to explore the ZNS design space without being constrained by device-specific characteristics. Such unconstrained explorations are crucial since ZNS is a new interface and the selection of available configurations in a real SSD is, unsurprisingly, quite limited. The research validity of all of these works hinge on an emulator's ability to mimic

---

*Equal contributions, joint first authors. Nick was with TU Delft during this work.

the performance characteristics of real hardware. In our investigation of the state-of-the-art emulators—FEMU [59] and NVMeVirt [60]—we reveal limitations in ZNS performance characterizations and discuss approaches to address them.

We summarize our key contributions as follows.

• We systematically characterize the performance (i.e., latency, bandwidth, and parallelism management) of a commercially available NVMe ZNS device, including its `append`, `read`, `write`, and zone management operations (i.e., `finish` and `reset`).

• We analyze the implications of interference from ZNS's unique `reset` and `append` operations on the `read`, `write` and `append` I/O performance.

• We reveal limitations in the performance models of the state-of-the-art ZNS emulators and discuss recommendations to address them.

• We share key recommendations for ZNS application developers using the insights from our characterizations.

• We publish our benchmarking software and data set at https://github.com/stonet-research/NVMeBenchmarks for encouraging reproducible research.

II. BACKGROUND

In this section, we review the background on flash-based storage and ZNS SSDs.

A. Flash storage

The storage area of flash-based devices is organized into flash *pages*, which is the unit of addressing and I/O operations [61]. A typical flash page is 4–16 KiB, which is always atomically written or programmed [27].

A flash page can not be overwritten; rather the page *must* be erased before it can be written again. Flash SSDs, nevertheless, provide an illusion of over-writable storage by storing data (i.e., the overwrite) in a new (flash) page, while marking the old (i.e., replaced) page as invalid or "garbage." After these garbage pages are eventually erased, the pages can be (re)used for writing data.

Erasing a page is a complex and time-consuming operation on flash storage. First, pages are physically grouped into *blocks*, and erasures only work at the block level. Second, since a block may contain both valid and garbage pages, valid pages have to be copied to a new block prior to erasing the old block. Third, pages always need to be sequentially written to blocks, complicating data management as random writes are not allowed. The erasing of a block and the associated book-keeping tasks (e.g., tracking valid and garbage pages) collectively constitute the *garbage collection (GC)* process (Flash-based SSDs, Chapter-44, [25]). Such GC processes may interfere with concurrent I/O operations and degrade I/O performance, since the GC process and user-issued I/O operations may need to access the same block or storage area [62]. Further on, flash memory has limited lifetime because it has a *limited* write or program/erase (P/E [63]) *endurance*. It is also prone to *read-write disturbs* that occur when many reads are issued to the same blocks—they cause
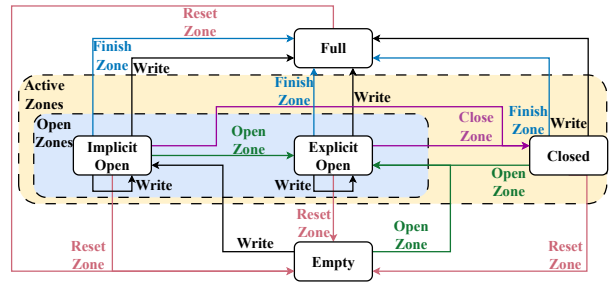

Figure 1: *Simplified overview of the ZNS zone state machine*

data loss. Limited storage lifetime exacerbates GC and other management (e.g., lifetime optimization) tasks [27], [64], [65], [66], [67], [68].

The onus of hiding these complexities of flash SSD management from applications running on the host resides with the *flash translation layer* (FTL), which is a part of the device firmware. It provides, for instance, the familiar page-addressable, any-address readable or writable storage media [69], [25]. Several prior work offer in-depth explanations on flash drives and FTLs [70], [71], [72], [62], [27].

B. Devices with Zoned Namespace

The storage area in ZNS devices is divided into regions known as *zones*. Zones themselves are divided into blocks, which represent the fundamental unit of I/O operations—the blocks are logically equivalent to pages in traditional flash storage. Blocks are addressed using a *logical block address (LBA)*. Zones only support sequential writes, matching the constraint of underlying flash pages, which can only be programmed sequentially. To meet this sequential-write constraint, within a zone, applications must issue page-write requests to LBAs in increasing order. SSDs, however, schedule I/O requests internally and may reorder them as required [7]. There can, hence, only be a single in-flight `write` request per zone so that the re-orderings, if any, do not violate the must-be-sequentially-written constraint of a zone. This design, consequently, limits `write` concurrency in a zone to one.

ZNS introduces the `append` operation [45] to alleviate the `write` concurrency limitation within a zone. The `append` idea is similar to *nameless writes* [73], writes that are not issued to an address, but return the address on completion. Unlike the `write` operation, which accepts the target block address, the `append` operation takes a *zone starting* LBA (ZSLBA) along with the data. Once an `append` is completed, the LBA is returned to the application. As a result it is safe to reorder `append`s in a zone, which enables applications to issue multiple outstanding `append` requests to the same zone.

ZNS SSDs offer explicit control over their GC process through the `reset` operation. A `reset` operation on a zone informs the device that the data in the zone can be discarded and the zone can be garbage collected. The `reset` operation does not, nevertheless, immediately force a block erasure [74]. The `reset` operation can be limited to FTL-metadata-mapping manipulations, which indicate to the device that the block can be erased later.

Zones of a ZNS device have states (Fig. 1), which dictate the allowed operations on a specific zone. Since each zone operation (e.g., `read`, `write`, and `append`) consumes SSD resources (e.g., internal buffers), there are limits on the number of zones that can be concurrently opened and used. These limits are defined as the *maximum open zone limit* and *maximum active zone limit*, respectively. Applications must abide by these constraints, and explicitly manage the zone states and transitions. An application must, for instance, open a zone *before* it accepts `writes` or `appends`. State transitions can be internal to a device and *implicit* (e.g., a `write` to an empty zone transitions it to an open zone in Fig. 1), or *explicit* as a response to a user request.

ZNS offers several explicit zone management operations, which include `open`, `close` and `finish`. We skip discussing the first two, whose names reveal their functionalities, and focus on the last. The `finish` operation transforms an *open* zone directly into a *full* zone. It releases all resources attached to the zone (to stay within the maximum open zone limit). Then, the device can either fill the zone with data or mark the unused capacity with mapping (metadata) updates in the "finished" zone (Fig. 1). Mapping updates would require extra metadata to keep track of partially-filled zones. The `finish` operation has implications for performance, and the costs of this operation varies from one ZNS SSD implementation to another.

In summary, ZNS devices support a rich I/O interface that includes operations beyond the simple `read` and `write` operations of traditional flash storage. It is, therefore, crucial to understand and characterize the performance of these operations as they (and their state-machine transitions) are now part of the Linux storage software stack.

C. Software support

ZNS devices are fully supported in Linux since kernel version `5.9` [75]. Currently there is a limited number of applications that use ZNS, and most that do, do not use all functionalities (e.g., no `finish` or `open`). Evaluating these applications would limit what ZNS properties we can measure and, therefore, in our work we use synthetic benchmarks to understand all of ZNS' facets first. The results of our benchmarks can then be used for application design. Here, we briefly mention several prominent ZNS applications in research to present an overview of what is currently available. Currently, applications have access to ZNS-friendly file systems F2FS [76], Btrfs [77] and Ceph [78]. There is also support for a swap system known as ZNSwap [49] and a RAID system known as ZRAID [79] Lastly, KV-store RocksDB has ZenFS as a ZNS-capable file system back-end [11].

III. EXPERIMENTS

In this paper, we characterize the performance and interference properties of the Western Digital Ultrastar DC ZN540 SSD, a large-zone ZNS SSD, using a series of controlled benchmarks. As of this writing, the number of commercially-available ZNS SSDs is limited, therefore, we focus our efforts

TABLE I: *Overview of the key insights*

Category	*Insight*
Append vs. write	`Write` operations have up to 23% lower latencies than `append` operations (§III-C)
Scalability	Prefer intra-zone scalability (§III-D, §III-E)
Zone transitions	`Finish` is the most expensive operation; it takes up to hundreds of milliseconds (§III-E)
I/O interference	NVMe ZNS offers $3\times$ higher read throughput under concurrent I/O operations than NVMe (§III-F)
I/O & GC interference	Reset latency increases by up to 78% under concurrent I/O operations, but `reset` operations themselves have no effect on `append`, `read` or `write` operations (§III-G)

on characterizing one SSD model and synthesize the performance questions to ask when evaluating a ZNS SSD. Tab. I summarizes our key findings.

A. Benchmarking setup

We use *fio* [80] for generating the workloads and benchmarking the ZNS device. We also employ custom *SPDK* benchmarks for benchmarking state transitions (§III-E) and `reset` interference (§III-G), since *fio* does not support them. We describe our benchmarking platform in detail in Tab. II.

We use two storage stacks for benchmarking: the Linux kernel block layer and the *SPDK* stack. The Linux block layer ships with the *mq-deadline* scheduler, which buffers multiple `write` operations to a single zone, merges writes to contiguous LBAs into one or multiple (larger) writes, and sequentially issues the merged requests. Applications can, hence, issue multiple `write` operations to a single zone. The *SPDK* stack, in contrast, is a bare-bones storage stack without any I/O scheduler. The rationale behind our storage stack selection is twofold. First, no storage stack currently supports all combinations of I/O and management operations that we aim to benchmark. We cannot, for instance, issue and benchmark `append` or zone management operations via *fio* and the Linux I/O stack. In a similar vein, we are restricted to issuing only one `write` per zone at a time with *SPDK*, since it lacks an I/O scheduler. Second, the selection enables us to compare the implications of state-of-the-practice—the Linux stack—and that of the state-of-the-art—*SPDK*—for ZNS application development.

We run experiments for 20 minutes and/or repeat them at least three times for deriving robust statistics. We pin our benchmarking code to the NUMA node containing the ZNS device. For the Linux storage stack, we use the *io_uring* engine with submission-queue polling enabled, following the recommended settings [14].

B. Performance metrics

We briefly describe the metrics we use to evaluate the performance of NVMe (ZNS) devices. Two indicators of I/O operation performance are throughput (i.e., the number of operations or bytes per second) and operation latency (i.e, the time each operation takes). We measure ZNS throughput in

TABLE II: *Details of the benchmarking environment*

Component	Configuration details
CPU	Dual socket Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 2 sockets, 10 cores/socket, hyper-threading disabled, with Spectre and Meltdown patches, `intel_pstate=disable`, `intel_idle.max_cstate=1`
DRAM	256 GiB, DDR4
ZNS	Western Digital Ultrastar DC ZN540 1TB (zone size: 2,048 MiB; zone capacity: 1,077 MiB; total number of zones: 904; max. active zones: 14)
NVMe	Western Digital Ultrastar DC SN640 960 GB
Software	Ubuntu (v22.04), kernel (v5.19, built from source), fio (v3.32; #db7fc8d), SPDK (v22.09; #aed4ece), nvme-cli tools (v2.0, #5a36bab)

(a) *Request sizes: 512 B for 512 B LBA format; 4 KiB for 4 KiB LBA format*

(b) *Request sizes: 4 KiB for* `writes`*; 8 KiB for* `appends`

Figure 2: *I/O latencies of* `append` *and* `write` *operations (queue depth, QD=1)*

I/O operations completed per second, referred to as IOPS, or in bytes written/read per second. We measure operation latency from the moment a request is submitted on the NVMe submission queue until a request is completed and visible on the NVMe completion queue. It is possible to send requests at a higher *queue depth* (QD)—QD measures the number of requests that can be concurrently in flight. When the queue depth is higher than 1, it is possible that multiple requests are submitted, but not yet in a completed state. This has implications for request latency as some requests will take longer to complete than others. We always, hence, mention the queue depth of an experiment.

C. `append` and `write` performance

`append` and `write` operations both write data to the device, albeit they differ in their approach (refer §II-B). The difference between `append` and `write` operations, fundamentally, lies in who is responsible for ensuring sequential writes to a zone—host (in case of a `write`) or device (in case of an `append`). Currently, there is no standard benchmark to make an educated decision on what operation to use. We perform, therefore, a quantitative analysis of the performance of both operations and facilitate making an informed decision about the use of these operations, taming the complexity and determining the changes required in the storage stack (see file system design for nameless writes [73]).

We evaluate `write` and `append` operations as follows. First, we study them under varying LBA formats (format of the NVMe namespace) with sector sizes 512 B and 4 KiB to verify whether `append` and `write` operations are both affected by the format. We then select the LBA format that results in the lowest latency. Second, we investigate the implications of the choice of I/O engine (i.e., *io_uring* and *SPDK*) and scheduler (i.e., *none* and *mq-deadline*) on the performance of these operations. This investigation helps to decide what engine to use for ZNS and whether to use a host scheduler or appends. Third, we evaluate the effect of request sizes on I/O latency. The evaluations are single-threaded and synchronous (QD=1) in order to evaluate the performance of requests in isolation. While prior work demonstrates that request size affects `write` latency on ZNS [44], we investigate if this observation is also apparent for `append` latency.
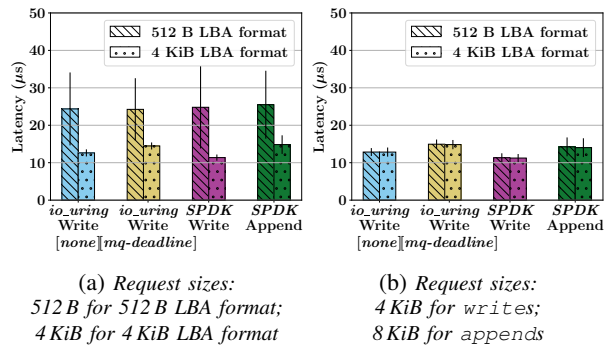
Observation #1: *The LBA format can have significant impact on both* `write` *and* `append` *latencies.* The LBA format affects both `append` and `write` operations (Fig. 2a). We show the operational latencies in microseconds along the Y-axis (lower is better) and software stack combinations with 512 B and 4 KiB formats along the X-axis. We set the request size to the same value as the block size of the respective LBA format. Latencies of the operations with a 4 KiB LBA consistently outperform that of a 512 B LBA, sometimes by as much as a factor of two (Fig. 2a). This difference between formats is highly dependent on the firmware, as firmware might not be optimised for small I/O, but highlights that it is important to consider what format to use for applications/benchmarks. Before running experiments we pick the optimal format size. We, hence, use the 4 KiB LBA format for all further experiments. We note, however, that the choice of an optimal block size depends on the ZNS device: It may be that the device was, for instance, explicitly optimized for 512 B accesses. We, therefore, need to consider what LBA format to use for both `write` and `append` operations.

Observation #2: *Using the SPDK storage stack results in the lowest latencies.* The Linux storage stack has, typically, higher overheads than the *SPDK* stack [81], and we report that it holds true for ZNS as well (see the 4 KiB format in Fig. 2a). *SPDK* has the lowest latency overheads for single outstanding I/O requests: `write` operations in *SPDK* (11.36 μs) have 9.98% lower latency than that in the kernel without a scheduler (12.62 μs). The *mq-deadline* scheduler, furthermore, adds non-negligible latency overheads (i.e., 1.85 μs out of 14.47 μs, or 12.81%). As the latency of raw flash storage access decreases, the scheduler's relative overheads will increase [82].

Observation #3: `Write` *and* `append` *throughput depends on the request size.* We plot the throughput (in terms of thousands of I/O operations per second) of `write` (in Fig. 3a) and `append` operations (in Fig. 3b) as a function of request size (in KiB). We observe that `write` operations experience the highest throughput in IOPS (i.e., 85 KIOPS) for request sizes of 4 KiB and 8 KiB, whereas the performance of `append` operations improves slightly—from 66 to 69 KIOPS—when we double the request size. The throughput in bytes per
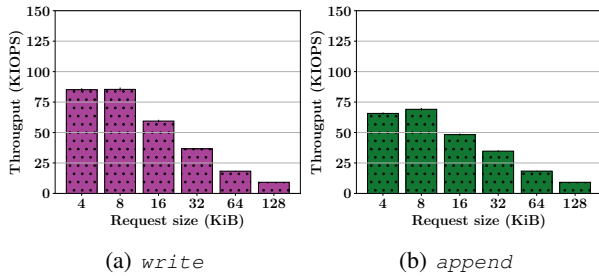
(a) `write` (b) `append`

Figure 3: *SPDK I/O throughput in IOPS (QD=1)*

seconds is highest for large requests (e.g. >=32 KiB, calculated as request size × IOPS). Note that we issue requests synchronously, hence, throughput is the inverse of request latency. The impact of request size on performance, hence, differs between `append` and `write` operations. That the `append` throughput is lower than `write` throughput is not inherent to the design of ZNS and dependent on the firmware. It is expected that `append` throughput will increase for newer ZNS devices. It is likely that the request size has an impact because of zone parallelism (i.e., zones mapped to multiple flash channels), similar to what is observed in [44]. We, therefore, recommend issuing large requests for maximal throughput; for the ZNS device used in our evaluation, we observe maximal throughput in IOPS with 4 KiB and 8 KiB for `write` and `append` operations, respectively, and maximum throughput in bytes for requests larger than 32 KiB.

Observation #4: `write` *operations have lower I/O latencies than* `append` *operations.* Across all configurations (Fig. 2a, Fig. 2b, Fig. 3a, Fig. 3b), we observe that the latency of `write` operations is lower than that of `append` operations, even if the request size is the same. In Fig. 2b, we use 4 KiB `write` and 8 KiB `append` operations, since these request sizes offered the lowest I/O latencies in prior experiments, and retain them as such for both the 512 B and 4 KiB block-size LBA formats. Our request sizes are now multiples of the block size, and they show, hence, fewer overheads. We achieve low `write` latencies of 11.36 μs (4 KB with *SPDK* `write`) and `append` latencies of 14.02 μs (8 KiB with *SPDK* `append`), and observe differences as large as 3.48 μs (or 23.42%) between `write` and `append` operations.

Recommendation #1: *Use* **`write`** *instead of* **`append`** *operations for low I/O latencies (differences between them can be as much as 23%), and use the SPDK storage stack since it delivers the lowest I/O latencies.*

D. Scalability: intra-zone versus inter-zone

A set of I/O requests such as `read`, `write`, and `append` operations can be distributed over either a single zone (intra-zone scalability) or multiple zones (inter-zone scalability). We define the maximum number of in-flight requests for both intra- and inter-zone as the *concurrency level*. Below, we analyze both of these approaches.

We measure the scalability of random `read`, sequential `write`, and sequential `append` operations in terms of IOPS

and bandwidth. In this setup, we always issue `read` and `append` operations via *SPDK*, but we issue `write` operations via *SPDK* for measuring inter-zone scalability, and use Linux with *mq-deadline* and *io_uring* for measuring intra-zone scalability. We use *SPDK* wherever possible in the evaluations owing to its low overhead (*Observation #2*) and `append` support. We rely on *io_uring* for issuing multiple `write` operations to the same zone, since *SPDK* (with no access to schedulers) does not support this functionality. Intra-zone benchmarks are single-threaded and inter-zone benchmarks use one thread for each concurrent zone.

We measure the throughput of sequential `write`, `append`, and random `read` operations using 4 KiB requests both within a single (Fig. 4a) and across multiple zones (Fig. 4b). We plot the throughput (in KIOPS) as a function of the level of concurrency in the system—in terms of queue depth for intra-zone and zones for inter-zone scalability measurements. Below we discuss three primary observations; all of them reveal that intra-zone parallelism fares better than inter-zone parallelism.

Observation #5: *Intra-zone parallelism achieve higher overall IOPS than inter-zone parallelism.* We observe that both `read` and `write` intra-zone operations (Fig. 4a) achieve higher IOPS than inter-zone requests (Fig. 4b) as we increase the level of concurrency.[1] We also note that inter-zone scalability is further constrained by the maximum open zone limit (§II-B): The number of concurrent zones when issuing `write` or `append` operations to multiple zones were, for instance, limited to a maximum of 14 zones, which was the maximum open zone limit for the device we evaluated. We prefer intra-zone to inter-zone parallelism if applications require higher scalability than permitted by the number of open zones.

Observation #6: *The* `append` *throughput, however, is agnostic to whether we use inter-zone or intra-zone requests.* Either scaling method offers similar throughput for `append` operations. The `append` throughput (in Fig. 4a and Fig. 4b) increases slightly until concurrency level 4 (~132 KIOPS), but does not improve afterwards. We hypothesize this limit is the device limit—not fundamental to the implementation or design of the ZNS device. That the `write` operations exhibit only marginal increase, if any, in throughput beyond 4 concurrent zones supports our hypothesis. The results indicate that distributing `append` operations across zones is a valid scaling strategy too, albeit it wastes additional open zones.

Observation #7: *In a single zone* `read` *operations scale the best (with queue depth), followed by* `write` *and* `append` *operations.* The `append` throughput reaches a maximum of 132 KIOPS at a queue depth of 4. The throughput of `write` operations with *mq-deadline*, in contrast, reaches 293 KIOPS at a queue depth of 32. However, the `read` throughput, reaches 424 KIOPS at a queue depth of 128 (although not shown in the figure). Until a queue depth of 4, within a single zone, `append` operations outperforms `write` operations, but at higher queue depths `write` operation outperforms `append`

---

[1]The `write` operations in Fig. 4a use `io_uring` with the `mq-deadline` scheduler, while the `write` operations in Fig. 4b use *SPDK*.
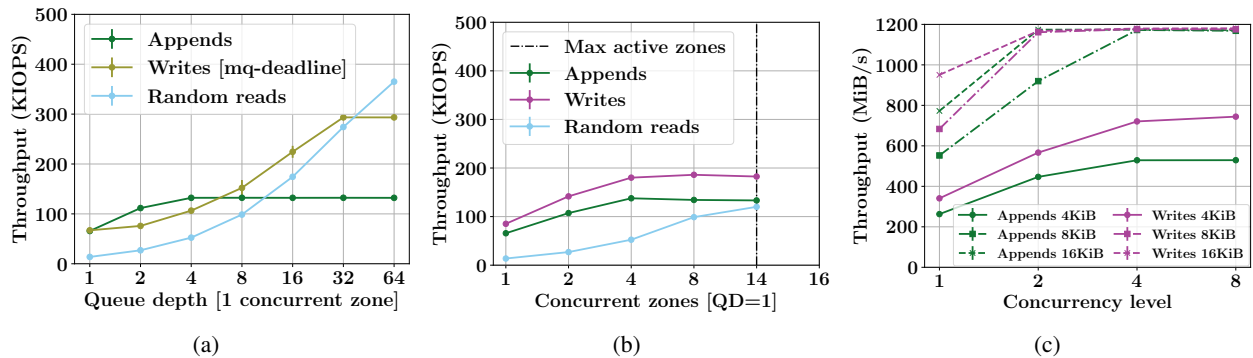
Figure 4: *(a) Intra-zone scalability in IOPS for 4 KiB requests (variable QD, 1 zone/thread); (b) Inter-zone scalability in IOPS for 4 KiB requests (QD 1, variable zones/threads); and (c) Inter-zone and intra-zone bandwidths using SPDK; concurrency level is queue depth for* `append` *operations, and concurrent zones for* `writes`

operations. This drastic performance improvement of `write` operations stems from the *mq-deadline* scheduler: The `write` operations to a zone must be issued sequentially, and the scheduler merges these sequential (4 KiB requests) into (fewer) larger `write` operations, thereby delivering higher throughput than those of the isolated `write` operations. At a queue depth of 16, for instance, the *fio* benchmark reveals that 92.35% of `write` operations were merged (into larger requests). The `write` throughput saturating at 186 KIOPS in the inter-zone scenario is reflective of the device performance, as merges are absent in this scenario; the `write` throughput is still higher than the inter- and intra-zone `append` throughput. Both `read` and `write` operations may also benefit from hardware acceleration, whereas the (first-generation of) `append` implementations may require extra support from the firmware. We, hence, expect `append` performance to improve as ZNS devices mature. Based on these observations, we recommend intra-zone parallelism for `write` operations (when using *mq-deadline*) for small (i.e., 4 KiB) I/O requests.

Observation #8: *For large (i.e., $>=8$ KiB) I/O requests, the performance of intra-zone* `append` *and inter-zone* `write` *operations reach the device limit and scale with concurrency levels in a similar manner.* We have already seen that request size has a large impact on throughput for requests at a concurrency level of one in §III-C. To further investigate the relation between request size and higher concurrency levels, we increase the request buffer size from 4 KiB to 8 KiB and 16 KiB (Fig. 4c), while retaining the setup intact as in prior experiments. We issue `append` operations to a single zone at variable queue depths and `write` operations concurrently (i.e., multiple threads) to multiple zones at a queue depth of 1. Inter-zone `write` operations initially offer better performance than the intra-zone `append` operations (the former likely benefits from optimized implementations), they converge quickly to offering similar throughput. The small (i.e., 4 KiB) requests fail to reach the device limit ($\sim$1.2 GiB/s), achieving a maximum throughput of 726.74 MiB/s for `write` operations, while the large (i.e., $>=8$ KiB) requests reach the limit when using 2–4 zones concurrently. Similar to §III-C we observe that

larger requests lead to higher throughput. `append` operations scale poorly compared to `write` operations as they require a higher level of concurrency to reach the device limit, and use of multiple zones concurrently benefits `write` more than `append`. Although not shown in the figure, when we use high queue depths when we issue `append` operations to multiple zones, performance degrades: We observe throughput reductions of up to 20 MiB/s with a queue depth of 4 for `append` operations issued to 4 zones concurrently. We urge use of either intra-zone or inter-zone `append` operations, but not both. For bandwidth-intensive workloads (i.e., with request sizes of at least 8 KiB), we recommend intra-zone `append` and inter-zone `write` operations.

Recommendation #2: *Prefer intra-zone to inter-zone parallelism; the former is ideal for* **append** *and* **read** *operations, while the latter is best suited for* **write** *operations. Issue I/O at large request sizes (i.e., $>=8$ KiB, close to the internal block size), as larger requests scale better with higher concurrency levels.*

E. The Zone State Machine Transition Costs

ZNS offers several unique and novel management operations for interacting with zones, including `open`, `close`, `reset`, and `finish` (refer §II and Fig. 1). These operations are issued explicitly by an application, but little is known about their performance implications. We also know little about the performance cost of implicit zone transitions. Quantifying the cost of these operations is, hence, crucial for designing performance-sensitive applications such as schedulers, file systems, and key-value stores. We analyze the performance of implicit and explicit operations, the cost of opening and closing zones, and the costs of `finish` and `reset` operations. Since *fio* does not evaluate or use all relevant state transitions, we evaluate state-transition costs with custom *SPDK* benchmarks.

Observation #9: *There is no performance difference between explicit and implicit zone open transitions, and the cost of opening/closing is marginal.* We can open zones either *explicitly* with the `open` or *implicitly* by writing to them (see Fig. 1). We measure the costs of opening a zone under
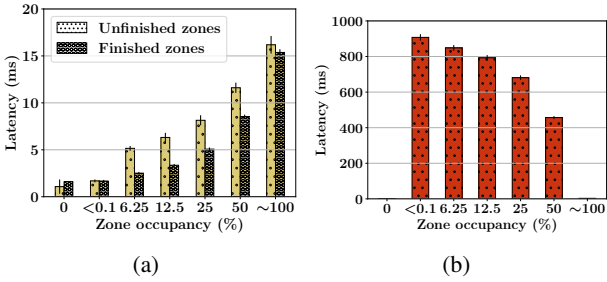
Figure 5: *(a)* `reset` *latency of partially-occupied zones and* *(b)* `finish` *latency of partially-occupied zones. The y-axis range differs significantly between the plots—up to 20 ms in (a) and 1,000 ms in (b).*

Figure 6: *(a)* `write` *throughput of concurrent workloads issuing random* `write` *operations and (b) random* `read` *throughput while concurrent workloads are issuing random* `write` *operations. The Y-axis ranges differ substantially between the plots—up to 1,300 MiB/s in (a) and just up to 3 MiB/s in (b).*

three different configurations: (1) *explicitly* using an `open` operation, (2) *implicitly* with `write`, and (3) *implicitly* with `append`. All `write` and `append` operations are issued at a request size of 4 KiB, since we know its latency characteristics (see Fig. 2b). After opening a zone, we fill it with either `append` or `write` operations, and we check for any difference in latency performance between *implicitly* and *explicitly* opened zones. We fill the zone to the second-last page, close the zone, and measure the latency of the `close` operation.

Our experiments show that it takes about 9.56 μs for opening a zone and 11.01 μs for closing it. The costs of the `close`, `append`, and `write` operations appear agnostic to how (i.e., *implicitly* or *explicitly*) we opened the zone. The first `write` and `append` operation to an *implicitly* opened zone, however, experiences some small (non-trivial) latency overhead—2.02 μs and 2.83 μs, on average, for `write` and `append` operations, respectively. To put in perspective, this is an overhead of about 17.38% and 19.32% for 4 KiB `write` and `append` operations respectively. These overheads are not surprising since zones must still be opened before a `write` (or an `append`) operation can be issued, even if the two operations are merged into one larger operation. We observe the costs of `open` and `close` operations to be marginal.

Observation #10: *Zone occupancy (or utilization) has a significant impact on the performance of both* `reset` *and* `finish` *operations.* Prior work demonstrates a positive correlation between zone size and `reset` latency [74], and we ask whether that correlation extends to zone occupancy (i.e., the count of written blocks in a zone) and applies to `finish` operations as well. We issue `reset` operations on 3,000 zones sequentially (over multiple runs) and change the occupancy of zones through various levels—0% (empty), 1 page (minimal), 6.25%, 12.5%, 25%, 50% and 100%. We use sequential 4 KiB `write` operations to fill a zone to the desired level. Once the desired occupancy level is reached, we pause for a second to let the device stabilize as `write` operations influence the `reset` performance (we defer that discussion until §III-G). We then issue either a `reset` operation to the zone, or a `finish` operation followed by a `reset` operation. The two approaches enable us to evaluate the latency of `reset`
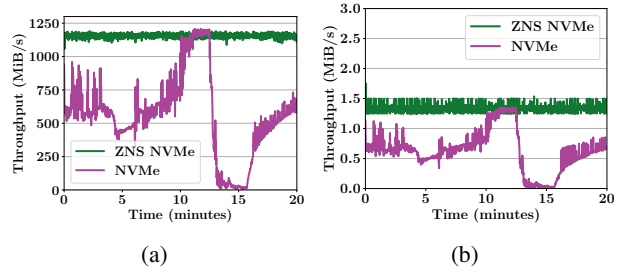
operations on both unfinished and finished zones.

Our experiments reveal that zone occupancy has a substantial impact on `reset`'s performance (Fig. 5a): `reset` operations incur an overhead of 11.60 ms on half-full zones and 16.19 ms on full zones, which is three orders of magnitude higher than `write` latencies (see §III-C). We posit that `reset` operations may require metadata updates to unmap the used pages in a zone. The *trim* operation on conventional NVMe SSDs, which hints to the SSD that it can reclaim a page, also incurs overheads due to metadata updates [49], [83]. We also observe that the latency of `reset` operations on zones depends on the zone being finished before it was reset. Finished zones take, for instance, less time to be `reset` than unfinished zones. Resetting a half-full zone takes, on average, 3.08 ms (26.58%) less time than resetting a zone that was first finished. Regardless of occupancy, a zone `finish` operation is, however, a very expensive operation.

We benchmark the performance of the `finish` operation in a manner similar to how we evaluated the `reset` operation. An occupancy of less than 0.1% (in Fig. 5b) indicates that we only filled one page, while ∼100% implies that we filled all except for one; since the standard does not permit us to issue a `finish` operation to a full or empty zone. The `finish` latency, unlike that of `reset` operations, decreases with occupancy (indeed linearly from < 0.1% to 25%). Average latency decreases by about 295 times—from 907.51 ms (almost one second!) to 3.07 ms—when we increase occupancy from <0.1% to 100%. Hence, when we add the costs of `finishing` and `resetting` a zone, the total cost can reach up to hundreds of milliseconds more than that spent in merely `resetting` the zone.

Recommendation #3: *Avoid the* `finish` *operation (more so than a* `reset`*), especially for partially written zones. Minimize the number of zones that need to be finished, hence, by leveraging intra-zone parallelism (thus, reducing the number of active zones, supporting the recommendations in §III-D).*

F. I/O interference: `write`, `append` and `read` interference

Below, we evaluate the interference from write operations (i.e., `write` and `append`) on read operations (i.e., `read`). Applications typically use workloads characterized by a mix of read and write operations, and it is crucial, hence, to measure the interference caused by read and write operations on each other. Latencies of `read` operations are, for instance, affected significantly by interference from other `read` or `write` operations on flash storage [84]. Garbage collection (GC) operations happening in the background on conventional SSDs further exacerbate interference, resulting in long `read` tail latencies. GC operations in ZNS, however, are triggered explicitly by the host—a widespread hypothesis is that this approach limits interference effects. In this section, we investigate this hypothesis.

We design an experiment in which we compare a conventional NVMe SSD with a ZNS SSD, both SSDs have the same hardware specifications. Fundamental in this evaluation is that GC is triggered inside the FTL firmware on the conventional SSD. Triggering GC is fundamental because GC is known to affect the `read` throughput negatively. In ZNS the GC process is managed by the software itself, and thus in this case the benchmark is responsible for the GC.

We evaluate the impact of garbage collection on SSD performance under diverse write workloads, controlled by explicitly rate limiting the workload's write bandwidths.

We base this limit on the peak ZNS NVMe and ordinary NVMe SSD write bandwidth, which we measured to be 1,155 MiB/s (see Fig. 4c). The write bandwidth is rate limited to values of 0, 250 (i.e., ∼25%), 750 (i.e., ∼75%) and 1,155 MiB/s (i.e., 100%) using *fio*. Concurrently we issue random `read` operations and we measure the write and `read` throughput over time. We sent `write`/`append` operations with four threads; each thread issues 128 KiB requests at a queue depth of 8. We pick this configuration as it puts significant pressure on the SSD, forcing intensive GC. We use random `write` operations on the conventional SSD, whereas on ZNS we utilize `append` operations on a set of random zones. Lastly, we issue `read` operations randomly at 4 KiB request size and use one thread (separate from the write threads).

Observation #11: *ZNS devices offer more stable read and write performance in the presence of concurrent write-triggered garbage collections than ordinary NVMe devices.* We observe that ZNS does not have the same write/read throughput fluctuations that we observe on the conventional NVMe interface. This observation also confirms earlier ZNS research on throughput stability for ZNS [11].

We report that both write and read throughput remains stable in all rate-limiting configurations (not shown). This observation confirms earlier ZNS evaluations [44]. On conventional SSDs, on the other hand, write and read throughput fluctuates for *all* configurations with concurrent writes. Especially, when `write` operations are rate limited to the peak bandwidth (i.e., 1,115 MiB/s). We only plot this scenario in Fig. 6a and
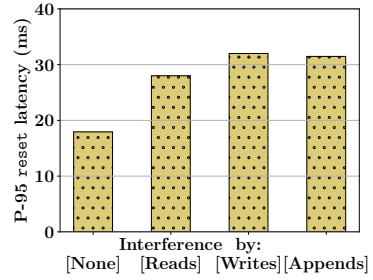

Figure 7: *Interference effect of* `read`, `write`, *and* `append` *operations on the 95-percentile* `reset` *latency*

Fig. 6b for `write` and `read` operations, respectively. On the x-axis we plot the time in minutes and on the y-axis we plot the throughput in MiB/s. The results are not surprising as the SSD needs to issue garbage collection (GC) operations in the background during `write`-heavy workloads, which leads to performance drops, while ZNS does not need this (as shown in Fig. 6a). `write` throughput fluctuates between a few MiB/s up to 1,200 MiB/s. ZNS also performs GC operations via `reset`, though the cost of resetting is ∼1% of the cost of filling the zone. Fig. 6b shows the impact of GC operations on random `read` latency (QD 32, 4 KiB). We use QD 32 as the performance saturates in the experiment at this point. `read` latency is affected significantly by concurrent `write` operations and GC operations. Furthermore, when `write` operations are rate limited to 1,115 MiB/s and `read` operations are issued at queue depth 1, `read` latency at the 95th percentile increases to 299.89 ms for the conventional SSD and 98.04 ms for the ZNS SSD (not shown in a figure). To put it in perspective, when only `read` operations are issued, the 95th percentile latencies are 81.41 μs for both conventional and ZNS, an increment by a factor of 1,000.

It is important to consider I/O interference when an application needs to scale, even if the interference effects are stable. For example, the observations made in §III-D do not consider concurrent I/O. Achievable throughput becomes limited if there is concurrent I/O. Intra-zone scalablity will, therefore, saturate at lower queue depth and inter-zone scalability at fewer concurrent zones. Coincidentally, an application that uses high intra- or inter-zone scalability reduces the throughput of other applications. Applications must hence take interference effects into account to achieve QoS targets. In short, inter- and intra-zone resources are shared between multiple applications/threads.

Recommendation #4: *Developers should measure the peak read/write performance of ZNS devices, and provision their application storage needs around them. There is no need to account for performance fluctuations because of GC operations.*

G. Reset interference

We observe that the `reset` operations take tens of milliseconds to complete (§III-E), *provided that we execute them in isolation*. In real workloads or applications, however, it is

likely that we issue `reset` operations concurrently with other I/O, e.g., as a separate garbage collection thread, and not in isolation. In this section we evaluate the effects of concurrent I/O on `reset` operations and the other way around. This evaluation is also important to support the observations made in §III-F. We evaluate with a custom benchmark using *SPDK*.

In this benchmark we use two concurrent threads. We use one thread solely for issuing `reset` operations on 100% occupied zones and one for issuing either `append`, `read`, or `write` operations. The intent is to ensure that there always is concurrent I/O and to measure if the latency alters during concurrent load. To prevent `reset` and `read` operations or writes to the same addresses, we issue `reset` operations to the first half (~400 zones) of the device and `read`, `write` and `append` operations to the second half. We issue `write` and `append` operations sequentially, but `read` operations randomly. We set the request size to 4 KiB. We finish the experiment once there are no more zones to `reset`, and repeat the experiment three times.

Observation #12: *`reset` operations do not interfere with `read`, `write` or `append` latency.* We discover no significant effect of `reset` operations on either `append`, `write` or `read`. Such behavior is likely to happen when the `reset` operation is limited to metadata operations on different resources than are used by I/O. Note that such behavior can be device-specific, but does give the impression that there is little need for schedulers to account for `reset` when managing QoS for `read`, `write`, or `append`.

Observation #13: *`read`, `write` and `append` operations interfere with the `reset` latency significantly.* Contrary to the previous observation, we do observe that `append`, `write` and `read` operations influence `reset` latency. In Fig. 7 we plot this interference effect. On the X-axis we plot the operation that runs concurrently with `reset` operations (if any), and on the Y-axis we plot the 95th percentile tail latency of `reset` operations. We can infer that the 95th percentile `reset` latency increases from 17.94 ms to 28.00 ms for concurrent `read` operations (56.11% increase), 32.00 ms for concurrent `write` operations (78.42% increase) and 31.48 ms for concurrent `append` operations (75.50% increase). All three I/O operations thus increase `reset` latency significantly. Therefore, it is likely that there is a form of resource contention, where I/O operations are prioritized over `reset` operations inside of the ZNS device.

Recommendation #5: *`reset` operations can be issued concurrently with I/O operations like `read`, `write` and `append`, since `reset` operations do not have an impact on I/O latency. Further on, while `reset` latency itself does increase significantly when they are issued concurrently with other I/O operations, `reset` operations themselves are issued at a large granularity (per-zone, 1 GiB), leading to sporadic `reset` operations (minimum time between two `reset` operations is: zone_size / write_bandwidth, on our device about one second).*

IV. OPEN CHALLENGES WITH ZNS EMULATION

So far we reported numbers for a specific device, the ZN540. However, the actual ZNS design/implementation space is large. Here, NVMe emulators can help: They have been used in previous research [51], [57], [58], [55]. The use of emulators can help verify observed trends and generalize our results. We observe, however, that none of the available emulators are ZNS-ready (yet). Emulators that are currently publicly available for ZNS include FEMU [85] and NVMeVirt [60]. The more recent ConfZNS [86], which we discuss only briefly, was not an open-source software at the time of our evaluations. We only consider the design/implementation of the emulators in this section and do not re-run the evaluations for the emulators: As of this writing, we could not run all of our evaluations in the emulators due to stablility or compatibility issues in the emulators (e.g., *SPDK* not working). We assume that as the software ecosystem stabilizes, these issues will be resolved. The goal of this section is, therefore, to address challenges in the design of emulators themselves.

We explain which of our ZNS performance observations the available emulators FEMU and NVMeVirt are currently unable to capture because of their design and explain what should be changed to support them. We do not consider observations #1 (LBA format), #2 (*SPDK* has the lowest I/O latencies), and #11 (ZNS is more stable than NVMe) to be relevant in this section as they do not represent essential behavior to emulate (e.g., not ZNS-specific).

FEMU currently makes no attempt at emulating ZNS SSD request latency, and requests are, therefore, as fast as the underlying hardware (i.e., CPU and DRAM) permits. The lack of a latency model leads to various reproducibility challenges. First, it is not possible to reproduce differences in I/O latency behavior of `append`, `write`, and `read` operations, since there is no inherent performance difference between these operations. There is also no difference in intra- and inter-zone scaling. As a result, FEMU fails to reproduce real-world empirical observations #3, #4, #5, #6, #7, and #8. Similarly, FEMU does not emulate zone transition latency, and as a result FEMU fails to also reproduce #9, #10, #12, and #13. For example, `finish` operations will become unrealistically fast as they are limited to metadata operations in DRAM. Therefore, FEMU, in its current state, cannot accurately reproduce *any* of our real-world observations and, consequently, cannot be used for evaluating the performance of real ZNS applications. We recommend FEMU to take an approach similar to NVMeVirt, as it has an explicit timing model. This timing model accounts for channel and NAND latency, and distinguishes between `read` and `write` latency.

NVMeVirt utilizes a latency model that is shown to be reasonably accurate for ZNS devices [60]. However, currently NVMeVirt uses the same latency model for both `append` and `write` operations and, hence, cannot represent observation #4 (`append` and `write` latency differ). This shortcoming introduces similar issues for intra- and inter-zone scaling, as `append` and `write` latencies are equal. This issue would

prevent #5 and #6 to be accurate. We argue that NVMeVirt should use a different model for both operations. For example, by using a different latency for `append` operations. Unfortunately, NVMeVirt also does not represent zone management operations correctly. It sets the latency of `reset` static and equal to NAND erasure latency (multiple milliseconds), which we have observed to not always be the case. Therefore, we opt that it should also be possible to use a dynamic cost where the latency depends on zone occupancy. A simple linear model would suffice here. Notably, NVMeVirt does not emulate timing for the other zone management operations at all. Since the `finish` operation is shown to be expensive, we argue that this operation should be implemented, preferably with a model based on zone occupancy. As it stands NVMeVirt fails to reproduce #9, #10, #12, and #13. In short, while NVMeVirt is accurate for `read` and `write` operations, it requires more refinement for `append` and zone transition operations to be a more accurate model.

More recently, Song et al. released ConfZNS, a ZNS emulator with an accurate latency model [86]. As of this writing, the code was not open-source; we do not, hence, describe the accuracy of this emulator in-depth and are unable to investigate which observations can be reproduced. The timing model of ConfZNS promises to lead to accurate latencies of `write` operations for inter-zone and `read` operations for inter- and intra-zone scalability, which should lead to results similar to §III-D and is the first step towards designing accurate ZNS emulators.

In short, while available open-source emulators have latency models for `write` and `read` operations, no current emulator has (or proposes) an accurate latency model for either `append` operations or zone transitions. Emulators should consider adopting both in order to be accurate.

V. RELATED WORK

A. On Zoned Namespace (ZNS) devices

There have been a number of (performance) characterizations for ZNS, but none consider the `append` operation, the ZNS state transitions and the interference of ZNS operations. Nevertheless, existing ZNS evaluations/implementations on various other properties, such as zone isolation, do exist and are complementary to this work. One of the early performance studies is by Shin et al. [44], where they verified the performance isolation properties of ZNS devices, and show that increasing I/O size decreases I/O latency, confirming our claims on interference and request size. Similar to our study, they investigated inter-zone scalability and the impact of request sizes. Bae et al. investigate the impact that zone size and ZNS internal parallelism have on the host I/O performance [50]. They identify that large zone sizes are preferred as they offer more opportunities to stripe and distribute I/O requests across multiple parallel channels and flash dies. We show that this is true for `append`s as well. However, large zones (in GiB/s) are also said to have large zone `reset` latencies that can significantly influence `read` latencies (pushing them to milliseconds and seconds). In our results we have only observed `reset` latencies in the milliseconds, but also showed that the cost is not static and largely depends on zone occupancy. We could not reproduce the effects of `reset`s on `read` latency. As a result of the aforementioned `reset` cost, Bae et al. advocate using small zones [50]. To improve the device performance and parallelism, the authors introduce a host-side inference tool to identify zone parallelism mappings by inter-zone interference measurements, and an accompanying I/O scheduler that can do mapping-aware I/O scheduling. Im et al. use small-zone SSD inter-zone parallelism for RocksDB [87]. Their results on inter-zone parallelism confirm our results and show similar trends for both small and large zone SSDs. We limit our evaluation to a large zone SSD in this evaluation. Jung et al. investigate various `reset` algorithms and show a correlation between zone size and `reset` latency [74]. Their work complements our findings on zone occupancy.

There is also a healthy amount of research on the ZNS specification and its new operations [11], [47], [55], [48]. Bjørling et al. present a comprehensive work on ZNS devices itself and discuss the design rationale and integration options [11]. The work also evaluates ZNS on the macro level, while we evaluate ZNS on the micro level. ZNS's unique `append` operation is discussed here [45]. Purandare et al. discuss the impact of ZNS devices on log-based data management systems, specifically log-based file systems, key-value stores (LSM tree), and database systems with logs [47]. They identify the ZNS `append` operation as a unique operation to leverage in the design of these systems. Despite much enthusiasm regarding the ZNS `append` operation, to the best of our knowledge, it has only been used in a handful of systems such as TropoDB [88], BtrFS [77] and ZNSwap [49].

There is also ample research in the application domain. Especially on the KV-store RocksDB and the ZenFS file system-backend is prevalent [51], [52], [54], [51], [89], [52], [87], [11]. ZNS application research has, as of now, mostly focused on improving garbage collection algorithms and new zone allocation policies. Such work can now also use the observations made in this device characterization. For example, by accounting for `finish` latency and zone occupancy for their garbage collection algorithms.

Naturally, this large body of work identifies that there is a big interest in accommodating and adapting ZNS-capable storage devices in the storage software stack. In all of these previous works, there has been a selective performance benchmarking and characterization using a mix of real-device [43], emulators [85], [60], or hardware support [55]. In this work, we primarily focus on aspects of performance of a commercially available ZNS device. In this process we have verified past published results as well as and reported new results.

B. On Performance Characterization

Due to the black-box nature of flash SSDs, multiple past studies have focused on empirical, stochastic, and analytical modeling of their operational characteristics [33], [90], [91], [92]. Such modeling is important for accurately predicting the access latency of a flash SSD for performance provisioning

and QoS-oriented scheduling. Past works have extended HDD-based blackbox analytical models to flash SSDs (e.g., linear, or regression based) at a broader workload-level granularity [91], [92]. SSDCheck studies the impact of write buffering and garbage collection by implementing various representative algorithms in SSD hardware, and verify/map their results to multiple blackbox SSDs [33]. In comparison, our work is focused on measurement-based study to establish the baseline performance without deconstructing the ZNS internals.

The modeling and impact of garbage collection algorithms have been one of the most studied areas with flash SSDs [93], [94], [35], [95], [96], [97]. Hu et al. analytically model the residual lifetime of a given SSD, in the presence of GC and enterprise workload [95]. Pletka et al. present the details of enterprise-grade latency, ECC, and GC flash algorithms [93]. Li et al. present a stochastic Markov chain model to model the I/O dynamics of an SSD with concurrent I/O and GC workloads [98]. Using this model, they design a randomized greedy algorithm that can be tuned to operate close to the optimal operational curve of the SSD. As write amplification, wear-leveling and the choice of GC algorithm are inextricably linked, Verschoren and Houdt analyze (e.g., simulation, and trace-based workloads) d-choice GC algorithms for their impact on flash wear-leveling and device lifetime [94], [96]. Lange, Naor and Yadgar take a broad approach to SSD performance modeling and consider the "SSD management" problem that includes block allocation, wear leveling, write amplification, and garbage collection from an algorithmic perspective [35]. They report on a series of analytical models which are verified with synthetic and trace-based workloads.

Before the emergence of ZNS devices, there has been work on modeling SMR device operations [99]. Shafaei et al. present an analytical model for a device-managed SMR drive (unlike ZNS which is host managed) [100], [101]. Chen et al. present one of the earliest systematic studies focused on flash SSD performance characterization [102]. They identified various undocumented performance anomalies due to flash fragmentation and establish a high correlation between access pattern and flash performance (previously thought to be unrelated). Jung and Kandemir provide a thorough and detailed empirical evaluation of six SSDs for their `read`, `write`, and `trim` (similar to the ZNS `reset` operation) interference from background activities (GC and buffer flush) performances [34]. Their results show unexpected influences of `reads` on the device lifetime (P/E cycle) and the influence of background activities on sustained SSD performance. In their work, they also recommend exposing flash firmware API to the host software, that ZNS does in a standard way. Our work follows the same spirit for a new generation of flash storage devices with unique operations.

VI. Threat to Validity

Much of our results in this work confirm the expected ZNS behavior that is hypothesized in ZNS's development. The confirmation of these results also opens up new directions of research where much of the previously published work on flash SSD is open for scrutiny—and perhaps can even become obsolete [46]. Our experiments and benchmarks are *empirically driven, user-observed* behavior for one specific type of ZNS device. Such selective benchmarking has risks. **We have consulted and verified our observations with Western Digital, the ZNS device manufacturer, for the particular ZNS device that we have tested.** However, we are aware that it is challenging to generalize our findings as many device internal details are confidential, and ZNS device capabilities are expected to improve in the future. Nonetheless, we believe that this paper makes strong contributions by performing a first-of-its-kind systematic performance sweep of a NVMe Flash Device with Zoned Namespaces, and providing specific workable recommendations to the developers. In order to ensure the long-term viability and repeatability of our research we have open-sourced all the scripts, tools, and data sets collected. We are also in the process of acquiring different ZNS models to extend our study.

VII. Conclusion

Zoned Namespace-capable NVMe devices represent a significant step in the evolution of flash hardware and software stack. They offer a rich interface (introducing operations such as `append`, `finish`, and `reset`) to the host software that allows fine-grained control over managing the flash storage. In this work, we systematically characterized the performance of a commercially available ZNS-capable NVMe device. To this end, we developed benchmarks and tools to characterize the performance the new I/O operation (i.e., `append`) and flash-management commands (i.e., `finish`, `reset`, `open`, and `close`). We analyzed the impact of operation interference (I/O and management) between conventional and ZNS-capable NVMe devices. We present five recommendations to ZNS application developers concerning `append` performance, inter-zone and intra-zone scalability, the cost of management operations, and I/O- and GC-level interference. We identify shortcomings in the state-of-art ZNS emulators, which are widely used in academic research, and outline the changes that they require to ensure a high fidelity emulation. We published all the artifacts of this study at https://github.com/stonet-research/NVMeBenchmarks. We hope our results and the publicly available artifacts encourage developers and researchers to apply and evaluate our recommendations in a wide variety of applications and expand this work with similar characterizations of other ZNS devices.

REFERENCES

[1] NVMe Consortium, "Everything You Need to Know About the NVMe® 2.0 Specifications and New Technical Proposals." https://nvmexpress.org/everything-you-need-to-know-about-the-nvme-2-0-specifications-and-new-technical-proposals/, Accessed: 2023-Aug-16.

[2] D. H. Walker, "A Comparison of NVMe and AHCI." https://sata-io.org/sites/default/files/documents/NVMe%20and%20AHCI_%20long_.pdf, Accessed: 2023-Aug-16.

[3] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and Their Implication on Real World Databases," in *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[4] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems," in *6th International Systems and Storage Conference*, SYSTOR 13, ACM, 2013.

[5] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O Stack: A Low-Latency Kernel I/O Stack for Ultra-Low Latency SSDs," in *USENIX Annual Technical Conference*, USENIX ATC 19, USENIX Association, 2019.

[6] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-Queue Fair Queueing," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, (USA), USENIX Association, 2019.

[7] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ: Device-Direct Fair Queueing for NVMe SSDs," in *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021* (M. K. Aguilera and G. Yadgar, eds.), USENIX Association, 2021.

[8] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "AutoStream: Automatic Stream Management for Multi-Streamed SSDs," in *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[9] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-State Drive," in *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, USENIX Association, 2014.

[10] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. 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, (New York, NY, USA), p. 471–484, Association for Computing Machinery, 2014.

[11] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. 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)*, pp. 689–703, USENIX Association, July 2021.

[12] Jens Axboe, "Efficient IO with io_uring, ." https://kernel.dk/io_uring.pdf, Accessed: 2023-Aug-16.

[13] Jonathan Corbet, "The Rapid Growth of Io_uring." https://lwn.net/Articles/810414/, Accessed: 2023-Aug-16.

[14] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and Animesh Trivedi, "Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and io_uring," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, (New York, NY, USA), p. 120–127, Association for Computing Machinery, 2022.

[15] A. Trivedi, N. Ioannou, B. Metzler, P. Stuedi, J. Pfefferle, K. Kourtis, I. Koltsidas, and T. R. Gross, "FlashNet: Flash/Network Stack Co-Design," *ACM Trans. Storage*, vol. 14, dec 2018.

[16] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A User-Space I/O Framework for Application-Specific Optimization on NVMe SSDs," in *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, (USA), p. 41–45, USENIX Association, 2016.

[17] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting Linux Storage Stack for Microsecond Latency and High Throughput," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 113–128, USENIX Association, 2021.

[18] A. Papagiannis, G. Saloustros, M. Marazakis, and A. Bilas, "Iris: An Optimized I/O Stack for Low-Latency Storage Devices," *SIGOPS Oper. Syst. Rev.*, vol. 50, p. 3–11, Jan 2017.

[19] A. Tai, I. Smolyar, M. Wei, and D. Tsafrir, "Optimizing Storage Performance with Calibrated Interrupts," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 129–145, USENIX Association, July 2021.

[20] N. Tehrany, K. Doekemeijer, and A. Trivedi, "A Survey on the Integration of NAND Flash Storage in the Design of File Systems and the Host Storage Software Stack," *CoRR*, vol. abs/2307.11866, 2023.

[21] K. Doekemeijer and A. Trivedi, "Key-Value Stores on Flash Storage Devices: A Survey," *CoRR*, vol. abs/2205.07975, 2022.

[22] J. Smart, "Optimizing Storage Performance for 4–5 Million IOPs." https://www.usenix.org/conference/vault19/presentation/smart, Feb 2020. Accessed: 2023-Aug-16.

[23] J. Kariuki, "What? 80 Million I/O Per Second with a Standard 2U Intel® Xeon® System!." https://spdk.io/news/2021/05/06/nvme-80m-iops/, May 2021. Accessed: 2023-Aug-16.

[24] Jens Axboe, "That's it. 10M IOPS, one physical core.." https://twitter.com/axboe/status/1452689372395053062, Accessed: 2023-Aug-16.

[25] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018.

[26] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The Unwritten Contract of Solid State Drives," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), p. 127–144, Association for Computing Machinery, 2017.

[27] N. Li, M. Hao, H. Li, X. Lin, T. Emami, and H. S. Gunawi, "Fantastic SSD Internals and How to Learn and Use Them," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, (New York, NY, USA), p. 72–84, Association for Computing Machinery, 2022.

[28] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proceedings of the USENIX 2008 Annual Technical Conference*, ATC'08, (Boston, Massachusetts), pp. 57–70, 2008.

[29] L. Bouganim, B. Jónsson, and P. Bonnet, "uFLIP: Understanding Flash IO Patterns," in *Fourth Biennial Conference on Innovative Data Systems Research, CIDR 2009, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, www.cidrdb.org, 2009.

[30] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 173–190, USENIX Association, Nov. 2020.

[31] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," *ACM Trans. Storage*, vol. 13, oct 2017.

[32] J. Kim, K. Lim, Y.-D. Jung, S. Lee, C. Min, and S. H. Noh, "Alleviating Garbage Collection Interference through Spatial Separation in All Flash Arrays," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, (USA), p. 799–812, USENIX Association, 2019.

[33] J. Kim, K. Choi, W. Lee, and J. Kim, "Performance Modeling and Practical Use Cases for Black-Box SSDs," *ACM Trans. Storage*, vol. 17, jun 2021.

[34] M. Jung and M. Kandemir, "Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, (New York, NY, USA), p. 203–216, Association for Computing Machinery, 2013.

[35] T. Lange, J. S. Naor, and G. Yadgar, "Offline and Online Algorithms for SSD Management," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, dec 2021.

[36] D. G. Andersen and S. Swanson, "Rethinking Flash in the Data Center," *IEEE micro*, vol. 30, no. 04, 2010.

[37] S. Han, P. P. Lee, F. Xu, Y. Liu, C. He, and J. Liu, "An In-Depth Study of Correlated Failures in Production SSD-Based Data Centers," in *FAST*, pp. 417–429, 2021.

[38] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[39] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *USENIX Annual Technical Conference*, pp. 753–766, 2019.

[40] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan, "The Necessary Death of the Block Device Interface," in *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, www.cidrdb.org, 2013.

[41] M. Bjørling, J. González, and P. Bonnet, "LightNVM: The Linux Open-Channel SSD Subsystem," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, (Santa clara, CA, USA), p. 359–373, 2017.

[42] Samsung, "Samsung Introduces Its First ZNS SSD With Maximized User Capacity and Enhanced Lifespan." https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan, Accessed: 2023-Aug-16.

[43] Western Digital, "Ultrastar DC ZN540." https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-zn540-nvme-ssd, Accessed: 2023-Aug-16.

[44] H. Shin, M. Oh, G. Choi, and J. Choi, "Exploring Performance Characteristics of ZNS SSDs: Observation and Implication," in *9th Non-Volatile Memory Systems and Applications Symposium, NVMSA 2020, Seoul, South Korea, August 19-21, 2020*, pp. 1–5, IEEE, 2020.

[45] M. Bjørling, "Zone Append: A New Way of Writing to Zoned Storage." https://www.usenix.org/conference/vault20/presentation/bjorling, Feb 2020. Accessed: 2023-Aug-16.

[46] 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*, HotOS '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[47] D. R. Purandare, P. Wilcox, H. Litz, and S. Finkelstein, "Append is Near: Log-based Data Management on ZNS SSDs," in *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*, www.cidrdb.org, 2022.

[48] U. Maheshwari, "From Blocks to Rocks: A Natural Extension of Zoned Namespaces," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, (New York, NY, USA), p. 21–27, Association for Computing Machinery, 2021.

[49] S. Bergman, N. Cassel, M. Bjørling, and M. Silberstein, "ZNSwap: un-Block your Swap," in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022* (J. Schindler and N. Zilberman, eds.), pp. 1–18, USENIX Association, 2022.

[50] H. Bae, J. Kim, M. Kwon, and M. Jung, "What You Can't Forget: Exploiting Parallelism for Zoned Namespaces," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, (New York, NY, USA), p. 79–85, Association for Computing Machinery, 2022.

[51] H.-R. Lee, C.-G. Lee, S. Lee, and Y. Kim, "Compaction-Aware Zone Allocation for LSM Based Key-Value Store on ZNS SSDs," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, (New York, NY, USA), p. 93–99, Association for Computing Machinery, 2022.

[52] Western Digital, "ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs." https://github.com/westerndigitalcorporation/zenfs, Accessed: 2023-Aug-16.

[53] Y. Zhang, T. Yao, J. Wan, and C. Xie, "Building GC-Free Key-Value Store on HM-SMR Drives with ZoneFS," *ACM Trans. Storage*, vol. 18, aug 2022.

[54] J. Jung and D. Shin, "Lifetime-Leveling LSM-Tree Compaction for ZNS SSD," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, (New York, NY, USA), p. 100–105, Association for Computing Machinery, 2022.

[55] K. Han, H. Gwak, D. Shin, and J. Hwang, "ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (A. D. Brown and J. R. Lorch, eds.), pp. 147–162, USENIX Association, 2021.

[56] M. Oh, S. Yoo, J. Choi, J. Park, and C.-E. Choi, "ZenFS+: Nurturing Performance and Isolation to ZenFS," *IEEE Access*, vol. 11, pp. 26344–26357, 2023.

[57] R. Liu, Z. Tan, Y. Shen, L. Long, and D. Liu, "Fair-ZNS: Enhancing Fairness in ZNS SSDs through Self-balancing I/O Scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[58] G. Oh, J. Yang, and S. Ahn, "Efficient Key-Value Data Placement for ZNS SSD," *Applied Sciences*, vol. 11, no. 24, p. 11842, 2021.

[59] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pp. 83–90, 2018.

[60] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "NVMeVirt: A Versatile Software-defined Virtual NVMe Device," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies (USENIX FAST)*, (Santa Clara, CA), February 2023.

[61] M. Cornwell, "Anatomy of a Solid-State Drive," *Communications of the ACM*, vol. 55, no. 12, pp. 59–63, 2012.

[62] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo, "Garbage Collection and Wear Leveling for Flash Memory: Past and future," in *2014 International Conference on Smart Computing*, pp. 66–73, IEEE, 2014.

[63] S. Yamada, Y. Hiura, T. Yamane, K. Amemiya, Y. Ohshima, and K. Yoshikawa, "Degradation Mechanism of Flash EEPROM Programming After Program/Erase Cycles," in *Proceedings of IEEE International Electron Devices Meeting*, pp. 23–26, IEEE, 1993.

[64] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 521–526, EDA Consortium, 2012.

[65] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1285–1290, EDA Consortium, 2013.

[66] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 551–563, IEEE, 2015.

[67] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu, "Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 438–449, 2015.

[68] C.-Y. Liu, Y.-M. Chang, and Y.-H. Chang, "Read Leveling for Flash Storage Systems," in *Proceedings of the 8th ACM International Systems and Storage Conference*, pp. 1–10, 2015.

[69] D. Ma, J. Feng, and G. Li, "A Survey of Address Translation Technologies for Flash Memories," *ACM Comput. Surv.*, vol. 46, jan 2014.

[70] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to Flash Memory," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, 2003.

[71] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND flash memories*. Springer Science & Business Media, 2010.

[72] S. Aritome, *NAND flash memory technologies*. John Wiley & Sons, 2015.

[73] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. 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'12, (USA), p. 1, USENIX Association, 2012.

[74] S. Jung, S. Lee, J. Han, and Y. Kim, "Preemptive Zone Reset Design within Zoned Namespace SSD Firmware," *Electronics*, vol. 12, no. 4, p. 798, 2023.

[75] Western Digital, "Zoned Storage Devices." https://zonedstorage.io/docs/introduction/zoned-storage, Feb 2022. Accessed: 2023-Aug-16.

[76] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, (USA), p. 273–286, USENIX Association, 2015.

[77] 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.

[78] J. Y. Ha and H. Y. Yeom, "zCeph: Achieving High Performance On Storage System Using Small Zoned ZNS SSD," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pp. 1342–1351, 2023.

[79] T. Kim, J. Jeon, N. Arora, H. Li, M. Kaminsky, D. G. Andersen, G. R. Ganger, G. Amvrosiadis, and M. Bjørling, "RAIZN: Redundant Array of Independent Zoned Namespaces," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 660–673, 2023.

[80] Jens Axboe, "Fio." https://github.com/axboe/fio, Accessed: 2023-Aug-16.

[81] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi, "Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, pp. 120–127, 2022.

[82] Z. Ren and A. Trivedi, "Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring," *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, 2023.

[83] C. Hyun, J. Choi, D. Lee, and S. H. Noh, "To TRIM or not to TRIM: Judicious triming for solid state drives," in *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles*, SIGOPS '11, ACM, 2011.

[84] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash = Local Flash," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (Xi'an, China), pp. 345–359, ACM, 2017.

[85] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The Case of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, (USA), p. 83–90, USENIX Association, 2018.

[86] I. Song, M. Oh, B. S. J. Kim, S. Yoo, J. Lee, and J. Choi, "ConfZNS: A Novel Emulator for Exploring Design Space of ZNS SSDs," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, pp. 71–82, 2023.

[87] M. Im, K. Kang, and H. Yeom, "Accelerating RocksDB for Small-Zone ZNS SSDs by Parallel I/O Mechanism," in *Proceedings of the 23rd International Middleware Conference Industrial Track*, pp. 15–21, 2022.

[88] K. Doekemeijer, "TropoDB." https://github.com/atlarge-research/tropodb, Accessed: 2023-Aug-16.

[89] G. Choi, K. Lee, M. Oh, J. Choi, J. Jhin, and Y. Oh, "A New LSM-Style Garbage Collection Scheme for ZNS SSDs," in *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'20, (USA), USENIX Association, 2020.

[90] J. Kim, J. Kim, P. Park, J. Kim, and J. Kim, "SSD Performance Modeling Using Bottleneck Analysis," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 80–83, 2018.

[91] S. Li and H. H. Huang, "Black-Box Performance Modeling for Solid-State Drives," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 391–393, 2010.

[92] H. H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance Modeling and Analysis of Flash-based Storage Devices," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, 2011.

[93] R. Pletka, I. Koltsidas, N. Ioannou, S. Tomić, N. Papandreou, T. Parnell, H. Pozidis, A. Fry, and T. Fisher, "Management of Next-Generation NAND Flash to Achieve Enterprise-Level Endurance and Latency Targets," *ACM Trans. Storage*, vol. 14, dec 2018.

[94] R. Verschoren and B. V. Houdt, "On the Endurance of the D-Choices Garbage Collection Algorithm for Flash-Based SSDs," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 4, jul 2019.

[95] J. Hu, H. Jiang, and P. Manden, "Understanding Performance Anomalies of SSDs and Their Impact in Enterprise Application Environment," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, (New York, NY, USA), p. 415–416, Association for Computing Machinery, 2012.

[96] B. Van Houdt, "A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-Based Solid State Drives," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, (New York, NY, USA), p. 191–202, Association for Computing Machinery, 2013.

[97] Y. Yang, V. Misra, and D. Rubenstein, "On the Optimality of Greedy Garbage Collection for SSDs," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, p. 63–65, sep 2015.

[98] Y. Li, P. P. Lee, and J. C. Lui, "Stochastic Modeling of Large-Scale Solid-State Storage Systems: Analysis, Design Tradeoffs and Optimization," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, (New York, NY, USA), p. 179–190, Association for Computing Machinery, 2013.

[99] A. Aghayev, M. Shafaei, and P. Desnoyers, "Skylight—A Window on Shingled Disk Operation," *ACM Trans. Storage*, vol. 11, oct 2015.

[100] M. Shafaei, M. H. Hajkazemi, P. Desnoyers, and A. Aghayev, "Modeling Drive-Managed SMR Performance," *ACM Trans. Storage*, vol. 13, dec 2017.

[101] M. Shafaei, M. H. Hajkazemi, P. Desnoyers, and A. Aghayev, "Modeling SMR Drive Performance," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, (New York, NY, USA), p. 389–390, Association for Computing Machinery, 2016.

[102] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, (New York, NY, USA), p. 181–192, Association for Computing Machinery, 2009.