Towards an Unwritten Contract of NVMe ZNS SSD

Nick Tehrany
VU, Amsterdam

Abstract

The standardization of NVMe Zoned Namespaces presents a unique addition to storage devices. Pushing the responsibility of data placement and garbage collection from SSD FTLs up the stack to the host allows for more optimal data placement, predictable garbage collection overheads, and lower device write amplification. However, with the current software stack there are numerous ways of integrating these devices into systems. In this work, we begin to systematically evaluate the performance implications of ZNS integration and present several initial rules for an unwritten contract of ZNS devices that is aimed to provide guidelines for developers to optimize ZNS performance.

Main findings show several implications on ZNS performance. Firstly, performance of the randomly writable space on ZNS devices achieves 94.2% higher bandwidth with sequential read workloads compared to write workloads. Secondly, larger I/O sizes are required to saturate the device bandwidth. Lastly, configuration of the I/O scheduler can provide workload dependent performance gains. Therefore, for application developers to optimize ZNS performance careful consideration of integration and configuration is required, depending on the application workload and its access patterns.

1 Introduction

Zoned Namespace (ZNS) devices are now standardized and commercially available [5, 24]. Requiring append-only writes while providing random reads, the new interface of these devices necessitates changes to the host storage stack. However, there is more than one way these devices can be integrated within the system. In a classical setup as shown in figure 1(a), at the bottom is a block device with a filesystem on top of it, and an application running on top of the filesystem. Integrating NVMe ZNS devices into this storage stack can be done in three different configurations. Firstly, integration at the block-device level (figure 1(b)), while keeping the rest of the stack above the same. An example of this setup is the

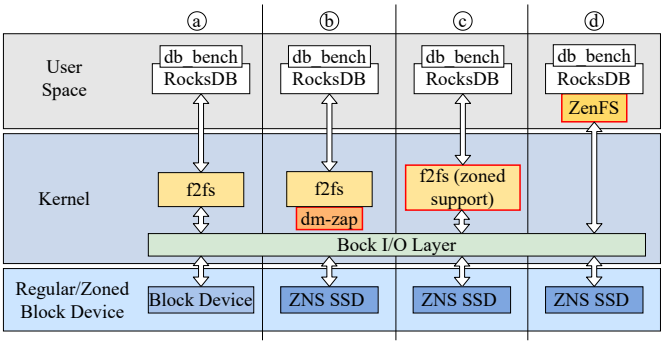


Figure 1: Integration of conventional block devices into the host software stack (a) compared to the three levels of integration for ZNS SSDs (b)-(d) into the software stack.

zoned device mapper project within the Linux Kernel [11]. This way of integration is the least intrusive one and requires the minimum amount of changes to anything running on top of the block device. Secondly, integration at the filesystem level (figure 1(c)). For this integration a filesystem is made aware of the zoned device characteristics such as zone capacity, number of active zones limits, and append limits. Example of such a project is the added ZNS support in f2fs [5]. With such integration the knowledge and required changes for ZNS devices are pushed higher in the stack, from block-device level to the filesystem level. Lastly, ZNS-aware application integration (figure 1(d)). In this case, there are ZNS-specific application-level changes at the very top of the storage stack. By pushing the customization higher up in the stack, the expectation is to deliver the best performance together with the best case application-specific customization and integration with ZNS devices. An example of such an integration is the ZenFS filesystem module for RocksDB and MySQL [5, 12].

With such configuration possibilities, it is not immediately clear which integration one should choose for their workload. In this research work, we aim to systematically understand the impact of the ZNS integration in the host storage system. This work is largely inspired by related work that has

provided unwritten contracts of storage devices for Optane based SSDs [25] and flash based SSDs [18]. We similarly aim to provide an initial set of rules for optimizing performance for the newly standardized NVMe ZNS devices. Due to the limited time for this study, we mainly focus on the raw ZNS performance, and leave exploration of the possible integration levels open as future work. In particular, we make the following contributions:

- We provide information on the newly standardized NVMe ZNS devices, how these devices work, how they are integrated into the host storage stack, and how existing applications were modified to support ZNS devices.
- We measure the raw ZNS device performance comparing it to conventional block devices in terms of achievable IOPs and bandwidth, and benchmark the possible scheduler configurations for ZNS devices, depicting their implications and limitations.
- We present pitfalls and failed experiments during this evaluation in an effort for others to learn from and to avoid the obstacles we encountered.
- We provide a set of initial rules for the unwritten contract of ZNS devices and propose several future work ideas to explore ZNS performance implications and expand the unwritten contract of ZNS devices.

2 Background

Applications and filesystems rely on the Linux block layer to provide interfaces and abstractions for accessing the underlying storage media. Originally designed to match the hardware characteristics of Hard Disk Drives (HDDs) the block layer presents the storage as a linear address space, allowing for sequential and random writes. SSDs however have different write constraints due to their architecture. At a high level, SSDs are organized in blocks, which contain a certain number of pages (typically 4KiB in size) [1]. Pages within a block have to be written sequentially, and prior to being able to rewrite data on an already written page, all pages within the block containing that page have to be erased. Furthermore, erase operations require substantially more time than read and write operations [17].

SSDs therefore include complex firmware, called the Flash Translation Layer (FTL), to provide the seemingly in-place updates of data and hide the device's sequential write constraints. It achieves this by invalidating the page(s) of the data to be rewritten, and writes the new data to free page(s) on the device. This requires the FTL to maintain a fully-associative mapping of host logical block addresses (LBAs) to physical addresses [17], in order to provide the LBAs of the page(s) that contain the valid data. Over time, as blocks contain an increasing number of invalid pages, the FTL has to move the still valid pages in the block to a new free block and erase the old block. This process is referred to as *garbage collection* (GC), which FTLs often run periodically to free up space.

However, garbage collection requires the device to maintain a certain amount of free space, called the *overprovisioning space*, such that the FTL is able to move valid pages. Most commonly used overprovisioning takes between 10-28% of the device capacity. While the FTL hides the write constraints of the device from the host, often applications have negative performance impacts due to the unpredictable performance from the device garbage collection [19, 26], large tail latency it causes [9], and increased write amplification [10]. The increased write amplification additionally reduces the device lifetime, since flash cells on SSDs have limited program/erase cycles.

2.1 Zoned Storage

The arrival of ZNS SSDs eliminates the need for on device garbage collection done by the FTL, pushing this responsibility to the host. This provides the host with more opportunity for optimized data placement, making garbage collection overheads predictable. The architecture of ZNS SSDs is the same as conventional SSDs, only the device firmware is modified. ZNS devices expose their address space with numerous *zones*, where data inside a zone must be written sequentially, and prior to overwriting existing data in a zone, the zone has be reset. The concept of exposing storage as zones is not new, as it was already introduced when Shingled Magnetic Recording (SMR) HDDs [15, 16, 23] appeared, which also enforce a sequential write constraint. The zoned storage model was introduced with the addition of Zoned Block Device (ZBD) support in the Linux Kernel in an effort to avoid the mismatch between the block layer and the sequential write constraint on devices.

The standards defining the management of SMR HDDs in the zoned storage model came through the Zoned Device ATA Command Set (ZAC) [7] and the Zoned Block Command (ZBC) [6] specifications. Since zones require sequential writing, the address of the current write is managed with a *write pointer*. The write pointer is incremented to the LBA of the next write only after a successful write. In order to manage zones, each zone has a state associated to it. These are to identify the condition of a zone, which can be any of the following; *EMPTY* to indicate a zone is empty, *OPEN* which is required for a zone to allow writes, *FULL* to indicate the write pointer is at the zone capacity, *CLOSED* which is used to release resources for the zone (e.g. write buffer on the ZNS device) without resetting a zone, this additionally does not allow writes to continue on the zone until it is transitioned to *OPEN* again, *OFFLINE* which makes all data in the zone inaccessible until the zone is reset, and *READ-ONLY*. The command sets provide the proper mechanisms to transition zones between any of the states.

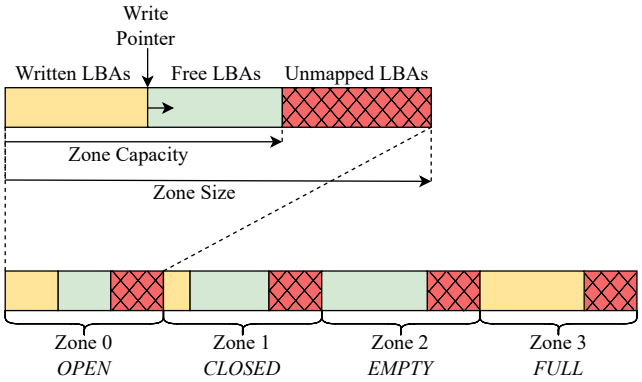


Figure 2: Layout of a ZNS SSD, depicting zone capacity, write pointer, and zone states associated to each zone. Adapted from [5].

While the majority of available space for both SMR and ZNS devices is utilized by sequential write required zones, they can also expose a limited amount of randomly writable area. This is mainly intended for metadata as this space only occupies a very small percentage of the device capacity. For example, the ZNS device used in this evaluation could expose 4 zones as randomly writable, equivalent to approximately 4GiB, compared to the total device capacity of 7.2TiB.

The newly introduced ZNS SSDs are standardized through the NVMe specification [24], which builds on the foundations established with ZBD support. While the zoned storage model aims to provide a unified software stack for all zoned devices (SMR and ZNS devices), the NVMe specification introduces several new concepts particular to ZNS devices. Firstly, it defines a *zone capacity* for each zone, stating the usable capacity of a zone, which is less than or equal to the size of the zone. Figure 2 shows an example layout of zones on a ZNS SSD and each zone's associated state. Zone capacity is kept separate from the zone size such that the capacity can be aligned to the erase unit of the underlying flash media, and such that the zone size can be kept as a power of two value. This is required for easier conversions between LBAs and zone offsets.

Secondly, the *active zone limit* specifies the maximum number of zones that can be active (in *OPEN* or *CLOSED* state) at any point in time. As the SSD requires to allocate resources for each active zone, such as the write buffers, it enforces a maximum on the active zones. Lastly, the NVMe specification introduces the *zone append* command, providing the option for the host to submit I/Os to a zone without requiring it to be at the write pointer. Rather the host issues the I/Os to the first LBA of the zone. The ZNS controller then writes the data at the write pointer, and returns the LBA of the write to the host. The use of append allows to issue concurrent writes to a zone without enforcing write ordering, which is especially beneficial if a large number of small writes are issued.

If applications do not utilize the append command, they require to ensure correct ordering among I/Os such that writes happen at the correct LBA, equal to the write pointer of the zone. ZNS devices additionally require the use of direct I/O, bypassing the page cache. This enforces to align with the sequential write constraint of zones, such that pages from the page cache are not written at out-of-order LBAs.

2.2 ZNS I/O Scheduling

Adhering to the sequential write requirement with ZNS devices under multiple outstanding I/O requests requires an appropriate scheduler to be set. It is responsible for ensuring writes are submitted at consecutive LBAs on the device [8]. However, requests can additionally be reordered on the device [24], making the host responsible for ensuring command ordering. For this the *mq-deadline* scheduler has to be enabled. It enforces that just a single write I/O is submitted and holds all subsequent I/Os until completion of the submitted I/O. This allows to submit numerous asynchronous write requests while adhering to the sequential write requirement and enforcing correct command ordering.

Additionally, the scheduler set to *none* can also be used with ZNS devices, bypassing the Linux Kernel I/O scheduler, however this does not enforce sequential write ordering or command ordering, as I/O requests are directly issued to the device. Hence, if this scheduler is set writes have to be issued synchronously and at the correct LBA. As ZNS devices only enforce sequential write ordering, reading can be done with both schedulers asynchronously with any number of outstanding I/Os, as long as it is not interleaved with write I/Os, at which point command ordering needs to be enforced again.

2.3 ZNS Application Integration

With the zoned storage model providing a unified software stack for SMRs and ZNS devices and support for the SMRs having been in numerous applications for some time, required changes for added support of ZNS devices was minimal. We focus primarily on f2fs [20] and ZenFS [5], a RocksDB storage backend for zoned storage devices.

f2fs. It had existing support for the ZBC/ZAC specification [3], making the changes for supporting ZNS devices minimal [5]. The changes included adding the zone capacity and limiting the maximum number of active zones. f2fs manages the storage as a number of segments (typically 2MB), of which one or more are contained in a section. Sections are the erase unit of f2fs, and segments in a section are written sequentially. The segments are aligned to the zone size, such that they do not span across zones. Since ZNS devices have a zone capacity, which is possibly smaller than the zone size, an additional segment type *unusable* was added in order identify segments outside the usable zone capacity. Partially usable segments are also supported with the *partial* segment type, in

order to fully utilize the entire zone capacity if a segment is not aligned to zone capacity.

The maximum active zones was already implemented by limiting the maximum number of open segments at any point in time. By default this is set to 6, however if the device supports less active zones, f2fs will decrease this at filesystem creation time. While f2fs supports ZNS devices, it requires an additional randomly writable block device for metadata, which is updated in-place, as well as caching of writes prior to writing to the zoned device.

ZenFS. ZenFS provides the file system plugin for RocksDB to utilize zoned block devices. RocksDB is an LSM-tree based persistent key-value store [14, 21] optimized for flash based SSDs. It works by maintaining tables at different levels in the LSM tree, of which the first level is in memory and all other levels are on the storage device. Writes initially go into the table in the first level, called the *memtable*, which gets flushed to the next level periodically or when it is full. Flushing will merge the flushed table with one from the next level, such that keys are ordered and do not overlap. This process is called *compaction*. Tables at lower levels than the memtable are referred to as Sorted String Tables (SSTs). SSTs are immutable, written sequentially, and erased as a single unit, hence making it a flash friendly architecture [5].

With zoned storage devices the RocksDB data files need to be aligned to the zone capacity for most efficient device utilization. ZenFS maps RocksDB data files to a number of *extents*, which are contiguous regions that are written sequentially. Extents are written to a single zone, such that they do not span across multiple zones, and multiple extents can be written into one zone, depending on the extent size and zone capacity. Selection of extent placement into zones relies on the provided lifetime hints that RocksDB gives with its data files. ZenFS places an extent into a zone where the to be written extent's lifetime is smaller than the largest lifetime of the other extents in the zone, such that it is not unnecessarily delaying the resetting of a zone. ZenFS resets a zone when all the files that have extents in that particular zone have been invalidated.

While RocksDB provides the option to set the maximum size for data files, data files will not have precisely this size due to compaction resulting in varying sized data files. ZenFS manages this by setting a configurable utilization percentage for the zones, which it fills up to said percentage, leaving space if files are larger than specified. While ZenFS requires at least 3 zones to run, of which one is for journaling, another is the metadata, and the last is a data zone, if the device supports more active zones, the active zones can be increased by setting a larger number of concurrent compactions in RocksDB. This can be up to the value of maximum active zones (minus the metadata and journaling zone), however performance gains for more than 12 active zones are insignificant [5].

3 Experimental Setup

The initial goal of this evaluation was to evaluate all possible integrations of ZNS devices. For this we established the following configuration:

f2fs. The f2fs (figure. 1(c)) parameters were largely kept at its default options, with the only change being the enforcing of 10% overprovisioning. Since f2fs requires a conventional device for its metadata, we expose 4 of the zones on the ZNS device as randomly writable space, which f2fs then uses for metadata and write caching. Since the ZNS device exposes a maximum of 4GiB randomly writable space. However, this required the space of f2fs on the zoned space to align with this size, i.e. the conventional space has to be able to contain all the metadata for the filesystem. Therefore, the resulting largest possible size that successfully mounts f2fs on the zoned namespace is 100GiB.

ZenFS. ZenFS (figure 1(d)) requires an auxiliary path for its metadata to store LOG and LOCK files for RocksDB at runtime. With this it additionally allows to backup and recover a filesystem through its command line options, however we do not utilize this option in our evaluation. For the auxiliary path we use the 4GiB conventional space exposed by the ZNS device and create a f2fs filesystem on it which is mounted for solely the ZenFS auxiliary path to be placed on.

Based on these configurations, the device is setup to provide a 100GiB experimental namespace, alongside the 4GiB randomly writable namespace, and all remaining capacity is left in an unused namespace. As the ZNS specification was integrated in Linux Kernel 5.9+, we use the later Kernel version 5.12.

4 Unsuccessful Experiments

Designing of experiments to evaluate the performance of the varying levels of integration proved challenging, as initial experiments did not provide insightful results. We provide the iterations of experimental design and why experiments failed in an effort for others evaluating ZNS performance to avoid these pitfalls. Section 4.1 provides the configurations of the various benchmark, followed by section 4.2 describing the failures of said benchmarks and pointing out possible causes for this. Lastly, we give additional lessons learned during this evaluation in section 4.3.

4.1 Benchmark Setup

With the established device configuration we run several benchmarks to evaluate different performance aspects of ZNS devices. While each of the evaluations relies on db_bench and RocksDB, they have slightly different benchmarking configurations. Below we describe the workload parameters for the different benchmarks.

4.1.1 Integration Level

We first benchmark the different possible integration levels, as shown in figure 1, using RocksDB and db_bench as the application that is running on each configuration. Benchmarks initially fill the database in random and sequential key order, followed by reading from the database in random and sequential key order. We additionally run an overwrite benchmark, which overwrites existing keys in random order, and an updaterandom workload that modifies values of random keys. The overwrite benchmark and the updaterandom differ in the way values are accessed on modified. Overwrite issues asynchronous writes to the key, whereas updaterandom uses a read-modify-write approach. Keys are 16B each, values are configured to be 100B, and we disable any data compression. As mentioned, ZNS devices require direct I/O and we therefore set appropriate db_bench flags to issue direct reads and use direct I/O for flushing and compaction.

4.1.2 ZenFS Benchmark

Next, to verify correctness of our setup we attempt to repeat an experiment depicted in [5], comparing db_bench performance on a configuration with f2fs to a configuration with ZenFS. The focus of this benchmark is write intensive workloads, as this is meant to trigger increased garbage collection and showcase gains of managing application-level ZNS integration with ZenFS. The benchmark is configured to run fillrandom and overwrite with a key size of 100B and value size of 800B. We additionally use data compression to compress data down to 400B. The original paper uses the entire device for its benchmark, however we scale it to the maximum possible that successfully fits into the namespace, which is equivalent to 50 million keys. Lastly, we set the target file size of SST files to be equal to the zone capacity, and utilize appropriate flags again for direct I/O.

4.1.3 Multi-Tenancy

Lastly, we run an experiment to evaluate how multiple concurrently running namespaces affect the performance of one another. For this we mount f2fs on the experimental 100GiB namespace and create an additional 100GiB namespace on which ZenFS with a db_bench benchmark is running. We compare the performance of running the ZenFS namespace alone, without any interference from another namespace, to running the f2fs namespace concurrently. The ZenFS namespace runs the same workload as describe in the previous benchmark (section 4.1.2). Initially, this namespace is run by itself, followed by repeating it and running the additional f2fs namespace, which is running an identical workload. The goal being that the f2fs namespace creates substantial device traffic through the write I/Os, especially during garbage collection from the overwrite benchmark, and thus show the performance impact on the ZenFS namespace.

4.2 Pitfalls to Avoid

Results of all experiments showed little to no performance difference in their benchmarks. Especially the ZenFS benchmark, where the original paper showed substantial performance gains with ZenFS, in particular on overwrite benchmarks where GC is being triggered heavily. We failed to reproduce these exact results and only had minor performance gains from ZenFS. The multi-tenant evaluation showed very similar results, which appear contrary to prior expectations. That is if one namespace fully utilizes the device, then another namespace attempting to use the same device will have some performance implications, as they would now be sharing the device resources.

As the prior illustrated proved ineffective in their evaluation, we propose the assumption that for performance differences to appear in the ZNS device it has to be largely utilized, such that LBA mappings are fully setup and more garbage collection is triggered. If the device is not fully utilized, LBA mappings are not fully setup and the device is able to provide peak performance without showing effects of garbage collection, as there is a large amount of free space it can utilize. Additionally, the device bandwidth has to be saturated. We therefore believe that with our setup we failed to produce enough device utilization in bandwidth and space, and thus evaluations showed that there are no performance differences. While we did not evaluate all configurations under increased garbage collection and device utilization, we believe it to have an effect on performance and thus propose its evaluation as future work (discussed in detail in section 8).

4.3 Lessons Learned

In addition to unsuccessful experiments we encountered several obstacles that were not immediately obvious to debug. Again, we provide our experiences in order for others to avoid these pitfalls. Firstly, by default the ZNS devices do not set a scheduler, neither do the applications such as f2fs or ZenFS, nor is there an error on an invalid scheduler being set. Thus setting up of the applications and formatting the ZNS device completes successfully, while as soon as writes were issued to the device I/O errors appear. Therefore, it is important to ensure the correct scheduler is always set, that is the *mq-deadline* scheduler, and every namespace requires to be set individually after creation. The majority of applications utilize multiple outstanding I/Os per zone, thus the *mq-deadline* scheduler is required, as it will hold back I/Os such that only one outstanding I/O is submitted to the device at any point in time. If the application would issue single I/Os synchronously, the scheduler could be left at the default configuration, however here again the application must enforce the sequential write constraint.

Secondly, device mapper support is not there yet. An existing implementation, such as dm-zoned [13] is ZBC and

	Optane	Samsung	ZNS
Media Size	260.8GiB	1.8TiB	7.2TiB
Usable Capacity	260.8GiB	1.8TiB	3.8TiB
Sector Size	512B	512B	512B
Zone Size	-	-	2048MiB
Zone Capacity	-	-	1077MiB
Number of Zones	-	-	3688

Table 1: SSD architecture of the three utilized devices during experimentation. ZNS information depicts the zoned namespace on the ZNS device.

ZAC compliant but not compliant to the new concepts of zone capacity from the ZNS specification. The dm-zap [11] device mapper aims to be ZNS compliant, however it is still a prototype and not fully functional with ZNS devices yet. As a result, we were not able to evaluate performance of one level of integration (figure 1(b)), requiring a future evaluation when the device mapper support is functional.

5 Adapted Experimental Setup

Based on the prior failed experiments, we adapt the experimental setup by committing the unused space with a cold file in an effort to fully utilize the ZNS device, instead of leaving the unused space empty. We additionally make use of two conventional SSDs, one of which is Samsung flash-based SSD and another Optane-based SSD. The conventional SSDs are used to provide a performance comparison between the conventional namespace on the ZNS device and regular SSDs. Detailed characteristics of each device are presented in table 1.

The conventional SSDs do not support multiple namespaces, therefore separate partitions are used, with the same 100GB of experimental space, and the remaining space serving as storage for cold data. While the ZNS device used during the experiments supports setting sector sizes of 512B and 4KiB, we set the device to 512B, since the used conventional SSDs only support 512B sectors and we aim to keep device configurations as similar as possible. Prior to running experiments devices were pre-conditioned to steady state performance by writing the entire space numerous times.

6 Raw Device Performance

Focusing on the raw performance of ZNS devices, we design several benchmarks using the fio benchmarking tool [2]. In particular, we evaluate the following aspects of ZNS performance:

- **ZNS I/O Performance for the Conventional Namespace (§6.1).** We establish the baseline performance of raw I/O on the ZNS device for its conventional namespace, which is exposed as a small randomly writable space. We measure the achievable throughput and compare it to performance of conventional SSDs. Main findings show that, unlike on conventional SSDs, read and write performance on the ZNS device is asymmetric and larger I/O sizes are required for achieving peak write bandwidth.

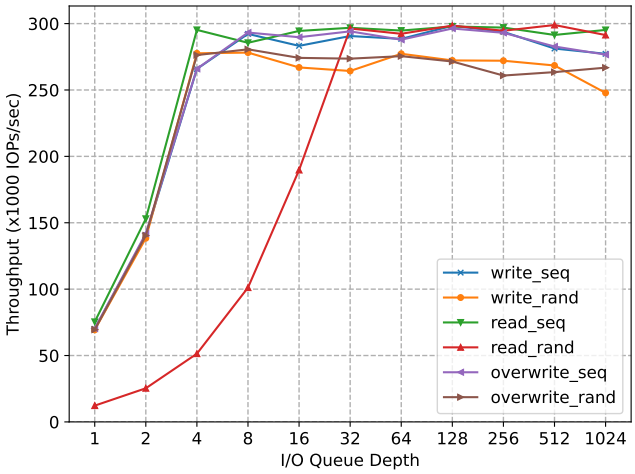
- **ZNS I/O Performance for the Zoned Namespace (§6.2).** We measure the performance of the ZNS device, benchmarking its zoned namespace. Specifically, we identify the performance of the *mq-deadline* and *none* scheduling under various read and write workloads. Results show that sequential write performance is higher with the *mq-deadline* scheduler, and read performance achieves lower median and tail latency with the scheduler set to *none*.

6.1 Conventional Device Performance

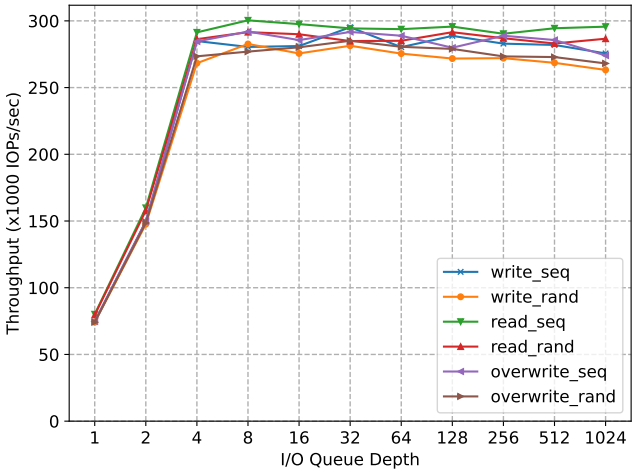
To measure the raw performance of the conventional namespace we run several fio workloads over the namespace and compare its performance to that of the Optane and Samsung SSDs. The workloads consist of a read benchmark with sequential accesses and another for random accesses, a sequential write benchmark, a random write benchmark, and a sequential and random overwrite benchmark. Overwrite benchmarks are utilized by filling the entire namespace with data and running a write benchmark on this full namespace. As the conventional device is accessible as any other block device, there are no write constraints allowing for random write access patterns. The benchmarks are run with an I/O size of 4KiB (referred to as the block size throughout this section) and varying I/O queue depths, indicating the outstanding I/Os to maintain against the file [4]. We retrieve the achievable I/O operations (IOPs) for the different benchmarks across all the devices.

Figures 3a and 3b show the performance of said benchmarks for the Samsung SSD and the Optane based SSD, respectively. Both devices show a stable peak performance in IOPs for small queue depths of 4, except for random reading which requires 16 outstanding I/Os to reach peak IOPs. The performance of the conventional namespace exposed by the ZNS device is shown in figure 4. It only reaches peak IOPs for read benchmarks at deeper queue depths of 8 and 64 for sequential and random reading, respectively. Additionally, showing contrasting performance in maximum achievable IOPs for read and write benchmarks. Write benchmarks reach peak performance at a shallow queue depth of 2, however performance is only 19% of the peak device throughput.

Next, we measure the achievable bandwidth for the device for larger block sizes by increasing said block size and maintaining a lower queue depth of 4. We again run the



(a) Peak throughput of the Samsung SSD.



(b) Peak throughput of the Optane SSD.

Figure 3: Peak throughput of the conventional SSDs under various fio workloads, with an increasing I/O queue depth and 4KiB block size.

same benchmarks with this configuration on the conventional namespace from the ZNS device. The resulting performance is depicted in figure 5, showing an increase to a peak write bandwidth of 1GiB for block sizes from 16KiB and larger. However, asymmetry in read and write performance is present, compared to the prior measured performance of the conventional SSDs, which have symmetric read and write performance. In addition, random reading requires larger queue depth to achieve the same peak performance as sequential reading.

Based on this evaluation we can identify that **(i)** read and write performance of the randomly writable space on the ZNS devices is asymmetric, where sequential reading achieves a 94.2% larger peak bandwidth, **(ii)** for achieving peak write bandwidth of the device larger block sizes ($\geq$ 16KiB) are

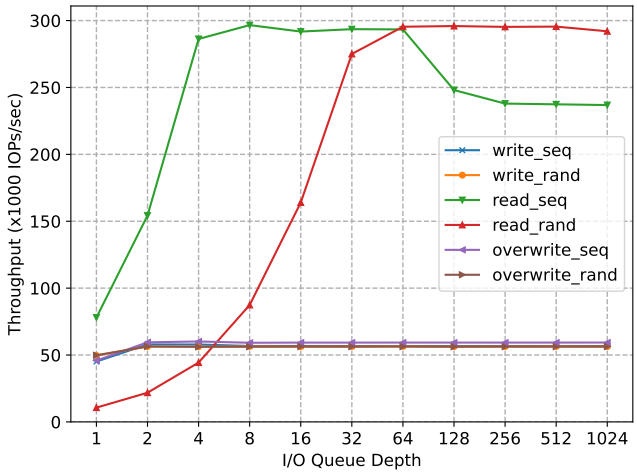


Figure 4: Peak throughput of the conventional namespace from the ZNS device under various fio workloads, with an increasing I/O queue depth and 4KiB block size.

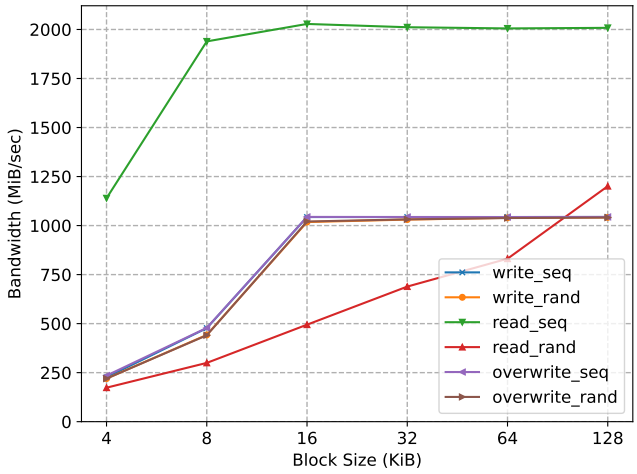


Figure 5: Maximum achievable bandwidth of the conventional namespace on the ZNS device with an I/O queue depth of 4 and an increasing block size.

required, **(iii)** and random writing requires a significantly larger I/O queue depth ($\geq$ 64) to achieve the device's maximal IOPs with 4KiB block size.

6.2 Zoned Device Performance

These benchmarks focus purely on the zoned namespace performance. In particular, since zones have a sequential write constraint, we only measure sequential writes within the zone. For this one can either utilize the *mq-deadline* scheduler and utilize a higher I/O queue depth, as it will hold back I/Os and only submit a single I/O at a time, or one could set the scheduler to *none*, however has to ensure the I/O queue depth

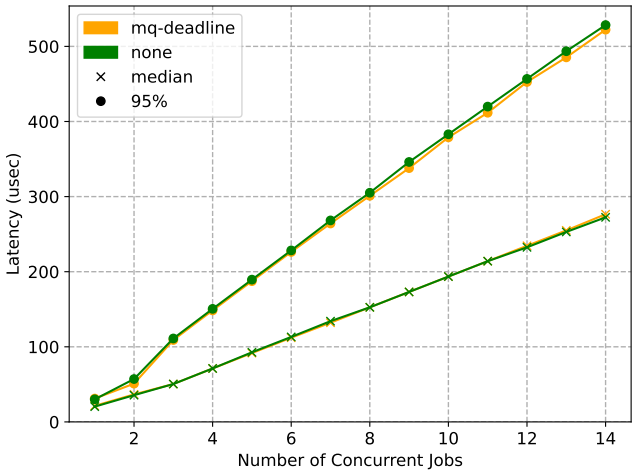


Figure 6: Latency of concurrently issuing 4KiB I/Os over an increasing number of active zones with an I/O queue depth of 1 under *mq-deadline* and *none* scheduler.

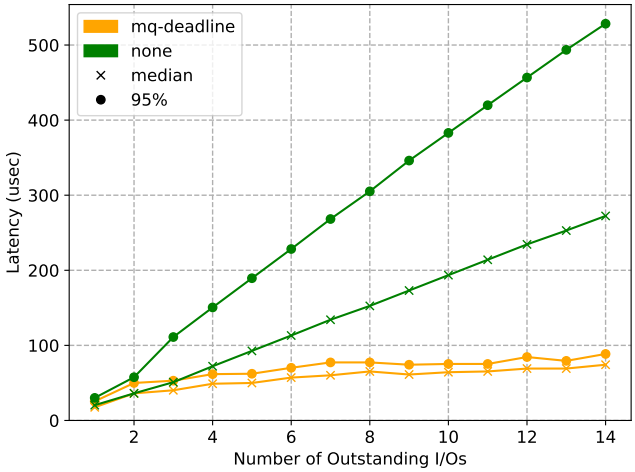


Figure 7: Latency of issuing 4KiB I/Os with increasing I/O queue depth for *mq-deadline* on a single zone, and I/O queue depth of 1 and increasing active zones for *none* scheduler.

is equal to one. We measure the differences in read and write performance for both schedulers.

Specifically, we measure the performance of both schedulers under the same workload, and both schedulers under similar workloads, where *mq-deadline* issues I/Os with a larger I/O queue depth and *none* utilizes a single I/O queue depth however concurrently issues I/Os over the number of active zones available. Figure 6 shows the comparison between both schedulers under the same workload of issuing 4KiB I/Os with an increasing number of active zones and a single outstanding I/O in each active zone. I/Os are submitted to active zones by concurrent threads, equal to the number of active zones. We depict the mean and tail latency, and as can be seen

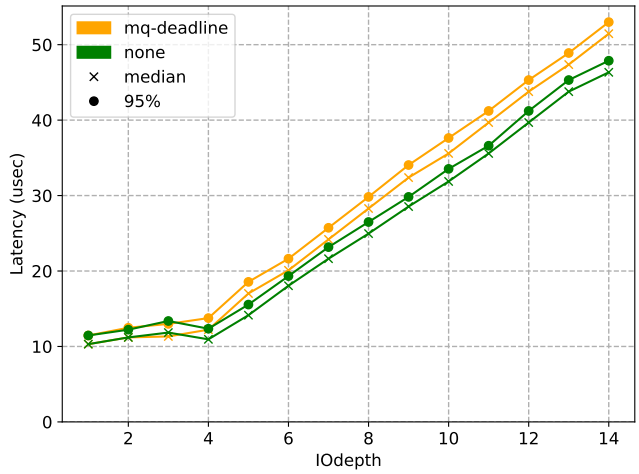


Figure 8: Latency of issuing 4KiB random I/Os in a single zone and increasing I/O queue depth with *mq-deadline* and *none* schedulers.

there are no significant performance differences between the two schedulers.

Next, we run the same benchmark for the *none* scheduler, but for the *mq-deadline* scheduler we instead use a single zone and increase the I/O queue depth. Figure 7 shows the resulting latencies. The lower latency for *mq-deadline* as the number of outstanding I/Os increases is due to the merging requests into larger I/Os, as all of the I/Os are at consecutive LBAs. This allows it to issue overall less I/Os than with *none* scheduling in this configuration. Based on this we can identify that if I/Os are consecutive one should utilize the *mq-deadline* scheduler and merge I/Os rather than splitting I/Os concurrently over multiple zones.

Similarly, we measure the performance of issuing random read requests with both schedulers. Fio is configured to issue random 4KiB I/Os in a single zone with an increasing I/O queue depth. Recall that ZNS devices only have a sequential write constraint, and there is no constraint on reading. Hence, the scheduler set to *none* can issue several read requests in a single zone without errors. Results presented in figure 8 show that the scheduler set to *none* achieves a lower mean and tail latency as the number of outstanding read requests per zone increases. This is due to the added overhead of having the *mq-deadline* scheduler compared to *none* bypassing the Linux I/O scheduler.

With this evaluation we can identify that **(i)** if the workload requires multiple outstanding write I/Os at once, utilizing the *mq-deadline* scheduler and merging I/O requests provides performance gains over splitting outstanding I/Os concurrently across active zones, and **(ii)** random read heavy workloads should avoid the Linux I/O scheduler by setting it to *none*, providing lower mean and tail latency.

7 Related Work

While ZNS has just recently been standardized [5], there have been several initial evaluations and discussions of ZNS devices. Bjorling et al. [5] present modifications made to RocksDB and f2fs to support ZNS devices, and showcase the performance gains for these devices. Stavrinos et al. [22] provide an initial discussion on the benefits of ZNS devices and possible improvements for applications on ZNS devices. However, there currently is no work that systematically studies the possible ways to integrate ZNS devices into the host software stack. We are the first to present a start at such a study.

While not focusing on ZNS devices, past work presented similar evaluations for conventional SSDs, where He et al. [18] provide an unwritten contract for flash-based SSDs, depicting numerous guidelines on performance improvements for such devices. Similarly, Wu et al. [25] present such an unwritten contract for Optane SSDs. Both of these unwritten contracts were inspiration for us to provide an unwritten contract for the new ZNS devices. Yang et al. [26] characterize performance implications of building log-structured applications for flash-based SSDs, showcasing the negated benefits of optimizing application data structures for flash caused by the device characteristics. Such an evaluation showcases application-level integration, which presents insightful results that should be reproduced on ZNS devices to further expand the unwritten contract of ZNS devices we provide.

8 Future Work

With ZNS devices having just been introduced, they leave a plethora of avenues to explore. In particular, as we showcased in figure 1, there are numerous possibilities of integrating ZNS devices into the host software stack. In this evaluation we focus mainly on the raw ZNS performance and provide several initial rules, which we define as the unwritten contract for ZNS devices. Expansion of this unwritten contract is left as future work by exploring the additional levels of integration. Specifically this includes evaluating the following aspects of ZNS devices:

- What is the performance of the varying levels of integrations for ZNS devices in terms of achievable IOPs, latency, transactions/sec, and additional instructions required? Under sequential and random, read and write workloads, are there performance implications of a specific integration, and what rules can be established when building applications for a particular integration?

- How is garbage collection influenced by the different levels of integration? Does building application specific garbage collection policies provide superior performance over other levels of integration? In particular how does an almost fully utilized device (e.g. 95% utilization) affect garbage collection performance at the different levels?

- Do multiple applications running on the same ZNS device interfere with each other? This includes a shared device with multiple namespaces, shared block-level interface with for example multiple concurrently running filesystems, and lastly a shared filesystem on ZNS with multiple concurrent applications. Does in each case an application's garbage collection affect the performance of concurrently running applications?

As we evaluated some of these aspects and failed to produce insightful results, we propose to evaluate them by taking into account our shortcomings and considering the assumptions we provide in section 4.2. An additional exploration that became apparent during this study was that there is currently no enforcing on the number of active zones at a time. The device only allows a maximum number of zones to be active at any point in time, however there is no managing of how active zones are split across namespaces on the device, or across applications running on the same namespace. Especially, as the device supports a larger number of namespaces to be created than active zones that can be open, a zone manager is required to assign zones across namespaces and applications, and provide fair resource sharing across namespaces and applications, enforcing that the maximum number is not exceeded, even if there are more concurrent applications running.

Lastly, we suggest the exploration of filesystem improvements for f2fs, and other ZNS specific filesystems, with grouping of files by creation time, death time, or owner as was suggested in [22] and similarly evaluated for conventional SSDs in [18]. This would provide the possibility for optimizing garbage collection and improving the device performance, at the filesystem integration level.

9 Conclusion

The newly standardized ZNS device present a unique new addition to the host storage stack. Pushing the garbage collection responsibility up in the stack to the host allows for more optimal data placement, providing predictable garbage collection overheads, compared to overheads from conventional SSDs. We provide one of the first systematic studies analyzing the performance implications of ZNS integration, and present an initial set of unwritten rules for ZNS devices.

While we mainly focus on the raw performance of ZNS devices, we additionally provide our unsuccessful experiments and pitfalls to avoid, and furthermore propose numerous future work proposals to evaluate ZNS device integration further and extend the unwritten contract provided in this work with further guidelines. Main findings in this evaluation show that read and write performance on the randomly writable space of ZNS device is asymmetric, with sequential reads achieving

almost double the peak bandwidth of writes, and larger I/Os are required for fully saturating the device bandwidth. Additionally, the selection of scheduler for the zoned space on ZNS devices can provide workload dependent performance gains.

Acknowledgments

I would like to thank Animesh Trivedi for supervision during this research work, and Matias Bjørling (Western Digital) and Dennis Maisenbacher (Western Digital) for providing extensive help on usage and benchmarking of ZNS devices, as well as providing access to the then private build of the dm-zap device mapper for zoned devices from Western Digital.

Availability

All the collected data during this evaluation, scripts for running benchmarks, as well as plotting results are made publicly available at https://github.com/nicktehrany/ZNS-Study. Instructions and commands for usage of ZNS devices and reproducing of this evaluation are additionally provided in the Appendix.

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[2] Jens Axboe. fio Flexible I/O Tester. https://github.com/axboe/fio. Accessed: 2022-01-21.

[3] Jens Axboe. [PATCH v8 0/7] ZBC / Zoned block device support. Patch, October 2016. Available at: https://www.mail-archive.com/linux-block@vger.kernel.org/msg01462.html.

[4] Jens Axboe. fio - Flexible I/O tester rev. 3.27. Documentation, 2017. Available at: https://fio.readthedocs.io/en/latest/fio_doc.html.

[5] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, July 2021.

[6] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Standard, American National Standards Institute, September 2014. Available from: https://www.t10.org/.

[7] INCITS T13 Technical Committee. Information technology – Zoned Device ATA Command Set (ZAC). Standard, American National Standards Institute, December 2015. Available from: https://www.t13.org/.

[8] Western Digital Corporation. Zoned Storage - Write Ordering Control. Documentation, June 2021. Available at: https://zonedstorage.io/docs/linux/sched.

[9] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, February 2013.

[10] Peter Desnoyers. Analytic Models of SSD Write Performance. *ACM Trans. Storage*, 10(2), March 2014.

[11] Western Digital. dm-zap. https://github.com/westerndigitalcorporation/dm-zap. Accessed: 2022-01-21.

[12] Western Digital. ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs. https://github.com/westerndigitalcorporation/zenfs. Accessed: 2022-01-21.

[13] Western Digital. dm-zoned Device Mapper Userspace Tool. https://github.com/westerndigitalcorporation/dm-zoned-tools, 2016. Accessed: 2022-01-21.

[14] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[15] Timothy R. Feldman and Garth A. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *login Usenix Mag.*, 38, 2013.

[16] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, Portland, OR, June 2011. USENIX Association.

[17] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. *SIGARCH Comput. Archit. News*, 37(1):229–240, March 2009.

[18] 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*, EuroSys '17, page 127–144, New York, NY, USA, 2017. Association for Computing Machinery.

[19] 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, February 2015. USENIX Association.

[20] 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, Santa Clara, CA, February 2015. USENIX Association.

[21] Facebook RocksDB. RocksDB: A Persistent Key-Value Store for Flash and RAM Storag, 2022. Available at: https://rocksdb.org/.

[22] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt 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, page 144–151, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Anand Suresh, Garth A. Gibson, and Gregory R. Ganger. Shingled Magnetic Recording for Big Data Applications. 2012.

[24] NVM Express Workgroup. NVM Express NVM Command Set Specification 2.0. Standard, January 2022. Available from: https://nvmexpress.org/specifications.

[25] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[26] 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 14)*, Broomfield, CO, October 2014. USENIX Association.

A General Information

Throughout this guide, commands and setup explanation contain names of the specific NVMe device which are depicting their configuration in our system. *nvme0n1p1* depicts the Samsung SSD, and *nvme1n1p2* the Optane SSD, both of which only support a single namespaces, requiring a partition to setup the 100GB experimental space. The conventional namespace of the ZNS device is *nvme2n1* and the zoned namepace is *nvme2n2*. We provide scripts for automation of all these benchmarks [1], as often numerous steps and retrieval of device specific information is required.

[1] Available at https://github.com/nicktehrany/ZNS-Study

B Device Setup

This section contains all required setup for devices, including namespace configuration, as well as the required applications for the different configurations used in this evaluation. To interact with ZNS devices, libnvme [2], nvme-cli [3], blkzone from util-linux [4], libzbd [5], and all their dependencies need to be installed. Note, ZNS integration is largely still new to applications used in this evaluation, therefore using the master branch is often required. Additionally, ZNS support was added to the Linux Kernel 5.9, therefore said version or newer is required.

B.1 ZNS Device Configuration

As we are using several namespaces on the ZNS device, one to expose a small amount of conventional randomly writable area, another of 100GiB (50 zones), and one for the remaining available space. The command for identifying the available size of the device is shown in listing 5, note that the ZNS device is configured to a 512B sector size. See Appendix C on how to retrieve the supported sector sizes for a device. Next, the creation of namespaces is depicted in listing 1. After creation of the namespaces, it is important to set the appropriate scheduler for all zones namespaces, as applications often do not do this automatically or check if the desired scheduler is set.

Listing 1: Setting up NVMe ZNS namespaces.

```
# Create a 100GiB namespace, size is
    given in 512B sectors
$ sudo nvme create-ns /dev/nvme2 -s
    209715200 -c 209715200 -b 512 --csi
    =2
# Repeat for all namespaces with
    according size
# Attach all namespaces to same
    controller (adapt -n argument with
    ns id)
sudo nvme attach-ns /dev/nvme2 -n 1 -c 0

# Set the correct scheduler for all
    zoned namespaces (adapt device path
    for each ns)
$ echo mq-deadline | sudo tee /sys/block
    /nvme2n2/queue/scheduler
```

[2] Available at https://github.com/linux-nvme/libnvme
[3] Available at https://github.com/linux-nvme/nvme-cli
[4] Available at https://github.com/util-linux/util-linux
[5] Available at https://github.com/westerndigitalcorporation/libzbd

B.2 f2fs Configuration

Setting up of f2fs requires f2fs-tools [6] to make the filesystem. Configurations of f2fs with a ZNS device require an additional regular block device that is randomly writable, due to f2fs using in place updates for metadata and the superblock. In addition, both devices have to be configured to the same sector size. The exact commands for creating of the f2fs filesystem and mounting it are shown in listing 2. The order of devices specified lists one or more zoned device, followed by a single conventional block device that is randomly writable. The location of the superblock is used for mounting, hence only the randomly writable device used for filesystem creating is provided in the mount command.

Listing 2: Creating and mounting of f2fs file system. Requires the ZNS device and an additional randomly writable block device.

```
# Format devices and create fs
$ sudo mkfs.f2fs -f -m -c /dev/nvme2n2 /
    dev/nvme2n1
$ sudo mount -t f2fs /dev/nvme2n1 /mnt/
    f2fs/
```

B.3 ZenFS Configuration

ZenFS [7] provides the storage backend for RocksDB to provide usage of ZNS devices. The ZenFS file system allows to be backed up and recovered to avoid data loss in failure events. Setting up of ZenFS requires the zoned device and an additional auxiliary path on another device with a filesystem, where it places the backup files, as well as any LOG and LOCK files required during RocksDB runtime. The command to setup ZenFS on a zoned device is shown in listing 3. The auxiliary path is placed on a conventional block device that is mounted with f2fs.

Listing 3: Creating of ZenFS file system. Requires an auxiliary path to place metadata.

```
$ sudo ./plugin/zenfs/util/zenfs mkfs --
    zbd=nvme2n2 --aux_path=/home/nty/
    rocksdb_aux_path/zenfs2n2
```

B.4 Namespace initialization

As mentioned previously, we utilize a 100GiB namespace (*nvme2n2*) for experiments and leave the remaining available space in a separate namespace (*nvme2n3*) to be filled with cold data. Listing 4 shows how this is achieved using fio [8].

For fio to be able to write the entire namespace on the device, it requires the block size to be a multiple of the zone capacity (see listing 5 on how to retrieve it). Similarly, the conventional devices (Optane and Samsung SSDs) also have their free space filled with cold data with the second shown command. The command is setup to write 2TiB however, fio will quit once the device is full. Additionally note that as mentioned earlier the conventional SSDs only support a single namespace and hence have separate partitions setup for the experimental and cold data space.

Listing 4: Filling namespace 3 with cold data.

```
# Fill ZNS free space with cold data
$ sudo fio --name=zns-fio --filename=/
    dev/nvme2n3 --direct=1 --size=$
    ((4194304*512*`cat /sys/block/
    nvme2n3/queue/nr_zones`)) --ioengine
    =libaio --zonemode=zbd --iodepth=8
    --rw=write --bs=512K

# Fill conventional SSD free space with
    cold data
$ sudo fio --name=zns-fio --filename=/
    dev/nvme0np2 --direct=1 --size=2T --
    ioengine=libaio --iodepth=8 --rw=
    write --bs=512K
```

C Getting ZNS Device Information

There are several attributes to the ZNS device that are required for later experiments, such as the zone capacity and the number of allowed active zones. Listing 5 illustrates how to retrieve these. Supported sector sizes can be checked with the provided command, and are presented in powers of 2. Hence, an *lbads:9* is equivalent to $2^9 = 512$ Bytes. We additionally retrieve the NUMA node at which the device is attached to, in order to pin workloads to this specific NUMA node.

Listing 5: Retrieving information about the ZNS device.

```
# Get the available device capacity in
    512B sectors
$ sudo nvme id-ctrl /dev/nvme2 | grep
    tnvmcap | awk '{print $3/512}'

# Get the zone capacity in MiB
$ sudo nvme zns report-zones /dev/
    nvme2n2 -d 1 | grep -o 'Cap:.*$' |
    awk '{print strtonum($2)
    *512/1024/1024}'

# Get maximum supported active zones
$ cat /sys/block/nvme2n2/queue/
    max_active_zones
```

[6] Available at https://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs-tools.git/about/

[7] Available at https://github.com/westerndigitalcorporation/zenfs

[8] Available at https://github.com/axboe/fio

```
# Get the supported sector sizes in
    powers of 2
$ sudo nvme id-ns /dev/nvme2n2 | grep -o
    "lbads:[0-9]*"

# Get NUMA node device is attached at
$ cat/sys/block/nvme2n1/device/numa_node
```

D Raw ZNS I/O Performance

This section contains the commands used to establish the baseline maximum performance of the device, as well as extracted metrics for comparison to later experiments. All experiments for this are using fio as benchmarking tool. Appendix D.1 shows the setup and commands for the provided evaluation in §6.1, and Appendix D.2 shows the same evaluation on the ZNS device, as presented in §6.2.

D.1 Conventional Device Performance

We run sequential and random writing and reading benchmarks for a block size (I/O size) of 4KiB under varying I/O queue depths to identify the maximum achievable IOPs of the devices. We run this on all three devices, since they all expose the regular conventional block device interface without write constraints, hence we only use the conventional namespace of the ZNS device in this section. The commands are shown in Listing 6.

The order of benchmarks is intentional such that randomwrite first runs, then the namespace is reset and written with the sequential benchmark. Overwrite benchmarks are done similarly with sequential and random writing after the entire namespace is filled with a write benchmark. Note that benchmarks are pinned to the NUMA node where the device is attached (see Appendix C for retrieval of this). All defined variables (indicated by the $ before a variable) are setup by our script, however for manual running of these commands require to simply be replaced by their associated value. The name and output argument depict naming for our plotting script to parse, however can simply be changed.

We additionally showed the bandwidth scaling for the conventional namespace on the ZNS device for a I/O queue depth of 4 and increasing block sizes, which are shown in Listing 7. Again, replace all defined variables with their respective value.

Listing 6: Establishing the peak IOPs for the conventional devices with 4KiB block size and varying I/O queue depths.

```
# We define several variables, replace
    these with values
    # DEV: device name (e.g. nvme2n1)
    # depth: I/O depth (from 1-1024 in
        powers of 2)
    # DEV_NUMA_NODE: NUMA Node of the
        device
    # SIZE: device size (e.g. 100G)

$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_randwrite_4Ki_queue-depth-
    $depth --output=
    $DEV_randwrite_4Ki_queue-depth-
    $depth.json --output-format=json --
    filename=/dev/$DEV --direct=1 --size
    =$SIZE --ioengine=libaio --iodepth=
    $depth --rw=randwrite --bs=4Ki --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95

# Reset the namespace between write
    benchmarks (only on namespaces, not
    partitions)
$ sudo nvme format /dev/$DEV -f

# Run remaining benchmarks
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_write_4Ki_queue-depth-
    $depth --output=$DEV_write_4Ki_queue
    -depth-$depth.json --output-format=
    json --filename=/dev/$DEV --direct=1
     --size=$SIZE --ioengine=libaio --
    iodepth=$depth --rw=write --bs=4Ki
    --runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_read_4Ki_queue-depth-
    $depth --output=$DEV_read_4Ki_queue-
    depth-$depth.json --output-format=
    json --filename=/dev/$DEV --direct=1
     --size=$SIZE --ioengine=libaio --
    iodepth=$depth --rw=read --bs=4Ki --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_randread_4Ki_queue-depth-
    $depth --output=
    $DEV_randread_4Ki_queue-depth-$depth
    .json --output-format=json --
    filename=/dev/$DEV --direct=1 --size
    =$SIZE --ioengine=libaio --iodepth=
    $depth --rw=randread --bs=4Ki --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95
```

```
# Namespace is still full so run
    overwrite benchs
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_overwrite-seq_4Ki_queue-
    depth-$depth --output=$DEV_overwrite
    -seq_4Ki_queue-depth-$depth.json --
    output-format=json --filename=/dev/
    $DEV --direct=1 --size=$SIZE --
    ioengine=libaio --iodepth=$depth --
    rw=write --bs=4Ki --runtime=30 --
    numa_cpu_nodes=$DEV_NUMA_NODE --
    ramp_time=10 --time_based --
    percentile_list=50:95
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_overwrite-rand_4Ki_queue-
    depth-$depth --output=$DEV_overwrite
    -rand_4Ki_queue-depth-$depth.json --
    output-format=json --filename=/dev/
    $DEV --direct=1 --size=$SIZE --
    ioengine=libaio --iodepth=$depth --
    rw=randwrite --bs=4Ki --runtime=30
    --numa_cpu_nodes=$DEV_NUMA_NODE --
    ramp_time=10 --time_based --
    percentile_list=50:95
```

Listing 7: Establishing the maximum achievable bandwidth of the ZNS convention namespace with I/O depth of 4 and increasing block size.

```
# We define several variables, replace
    these with values
     # DEV: device name (e.g. nvme2n1)
     # block_size: I/O size (from 4KiB to
         128KiB in powers of 2)
     # DEV_NUMA_NODE: NUMA Node of the
        device
     # SIZE: device size (e.g. 100G)

$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=
    $DEV_randwrite_$block_size_queue-
    depth-4 --output=
    $DEV_randwrite_$block_size_queue-
    depth-4.json --output-format=json --
    filename=/dev/$DEV --direct=1 --size
    =$SIZE --ioengine=libaio --iodepth=4
     --rw=randwrite --bs=$block_size --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95

# Reset the namespace between write
    benchmarks (only on namespaces, not
    partitions)
$ sudo nvme format /dev/$DEV -f

# Run remaining benchmarks
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_write_$block_size_queue-
    depth-4 --output=
    $DEV_write_$block_size_queue-depth
    -4.json --output-format=json --
    filename=/dev/$DEV --direct=1 --size
    =$SIZE --ioengine=libaio --iodepth=4
     --rw=write --bs=$block_size --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_read_$block_size_queue-
    depth-4 --output=
    $DEV_read_$block_size_queue-depth-4.
    json --output-format=json --filename
    =/dev/$DEV --direct=1 --size=$SIZE
    --ioengine=libaio --iodepth=4 --rw=
    read --bs=$block_size --runtime=30
    --numa_cpu_nodes=$DEV_NUMA_NODE --
    ramp_time=10 --time_based --
    percentile_list=50:95
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_randread_$block_size_queue
    -depth-4 --output=
    $DEV_randread_$block_size_queue-
    depth-4.json --output-format=json --
    filename=/dev/$DEV --direct=1 --size
    =$SIZE --ioengine=libaio --iodepth=4
     --rw=randread --bs=$block_size --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    time_based --percentile_list=50:95

# Namespace is still full so run
    overwrite benchs
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_overwrite-
    seq_$block_size_queue-depth-4 --
    output=$DEV_overwrite-
    seq_$block_size_queue-depth-4.json
    --output-format=json --filename=/dev
    /$DEV --direct=1 --size=$SIZE --
    ioengine=libaio --iodepth=4 --rw=
    write --bs=$block_size --runtime=30
    --numa_cpu_nodes=$DEV_NUMA_NODE --
    ramp_time=10 --time_based --
    percentile_list=50:95
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$DEV_overwrite-
    rand_$block_size_queue-depth-4 --
    output=$DEV_overwrite-
```

```
    rand_$block_size_queue-depth-4.json
    --output-format=json --filename=/dev
    /$DEV --direct=1 --size=$SIZE --
    ioengine=libaio --iodepth=4 --rw=
    randwrite --bs=$block_size --runtime
    =30 --numa_cpu_nodes=$DEV_NUMA_NODE
    --ramp_time=10 --time_based --
    percentile_list=50:95
```

D.2 Zoned Device Performance

We run various workloads on the zoned namespace of the ZNS device, and compare the performance of the scheduler set to *mq-deadline* and *none*. Listing 1 showed how to change the scheduler for a namespace. For all benchmarks we utilize fio configured to use 4KiB I/Os. The benchmarks are shown in Listing 8. The listing also shows the write benchmark that was used to produce Fig. 6. This benchmark writes a single 4KiB I/O to a zone and we increase the number of concurrent writes that are issued to the increasing number of active zones. Therefore, the benchmarks for the different schedulers have the exact same command, and only requires to change the scheduler in between iterations, when the number of concurrent jobs is increased. Note, we also use a 50 zone namespace and have a maximum of 14 active zones, which is why the *offset_increment* flag is set to 3z, such that each additional thread starts at an increasing offset of 3 zones and all 14 threads can still fit into the namespace when running concurrently. In addition, the write size of each thread is 3z.

Listing 8: Measuring write latency for the schedulers with increasing number of active zones and a single outstanding I/O per zone.

```
# We define several variables, replace
    these with values
     # DEV: device name (e.g. nvme2n2)
     # DEV_NUMA_NODE: NUMA Node of the
        device
     # scheduler: current scheduler
     # jobs: [1-14] number of active
        zones

# Write benchmark, change the scheduler
    between iterations
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$(echo "${DEV}_${scheduler}
    _4Ki_nummjobs-${jobs}") --output=$(
    echo "${DEV}_${scheduler}
    _4Ki_numjobs-${jobs}.json") --output
    -format=json --filename=/dev/$DEV --
    direct=1 --size=3z --
    offset_increment=3z --ioengine=psync
     --zonemode=zbd --rw=write --bs=4Ki
    --runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    group_reporting --numjobs=$jobs --
    percentile_list=50:95
```

The next write benchmark, as was shown in Fig. 7, runs *mq-deadline* with a single zone and an increasing I/O depth and *none* runs just as before, with a single I/O per zone and an increasing number of active zones. Listing 9 shows these specific commands. This benchmark defines the size as the total space of the device, since it runs a single thread.

Listing 9: Measuring write latency for the schedulers with increasing number of active zones and a single outstanding I/O per zone.

```
# In addition to the prior defined
    variables we also define
     # SIZE: device size (e.g. 100G)
     # depth: [1-14] I/O depth

# mq-deadline benchmark
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$(echo "${DEV}_mq-deadline_${BS
    }_iodepth-${depth}") --output=$(echo
     "${DEV}_mq-deadline_${BS}_iodepth-$
    {depth}.json") --output-format=json
    --filename=/dev/$DEV --direct=1 --
    size=$SIZE --ioengine=libaio --
    zonemode=zbd --rw=read --bs=4Ki --
    runtime=30 --numa_cpu_nodes=
    $DEV_NUMA_NODE --ramp_time=10 --
    iodepth=$depth --time_based --
    percentile_list=50:95

# none benchmark
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$(echo "${DEV}_none_4Ki_iodepth
    -${depth}") --output=$(echo "${DEV}
    _none_4Ki_iodepth-${depth}.json") --
    output-format=json --filename=/dev/
    $DEV --direct=1 --size=$SIZE --
    ioengine=libaio --zonemode=zbd --rw=
    read --bs=4Ki --runtime=30 --
    numa_cpu_nodes=$DEV_NUMA_NODE --
    ramp_time=10 --iodepth=$depth --
    time_based --percentile_list=50:95
```

Read performance was measured with random reading of 4KiB on a single zone and an increasing I/O depth for both schedulers. Recall, that ZNS devices only have a sequential write constraint, and hence read requests can be issued with increasing I/O depth under any scheduler and do not have to be at the write pointer of a zone. Listing 10 shows the commands for this particular benchmark. The benchmark increases only the I/O depth from 1 to 14, and commands are the same under both scheduler configurations, however the

correct scheduler has to be set before each iteration.

Listing 10: Measuring random read performance for 4KiB I/Os in a single zone with increasing I/O depth.

```
# Defined variables
    # scheduler: current scheduler
    # depth: [1-14] I/O depth

# Change scheduler between iterations
$ sudo numactl -m $DEV_NUMA_NODE fio --
    name=$(echo "${DEV}_${scheduler}
    _4Ki_iodepth-${depth}") --output=$(
    echo "${DEV}_${scheduler}
    _4Ki_iodepth-${depth}.json") --
    output-format=json --filename=/dev/
    $DEV --direct=1 --size=SIZE --
    ioengine=libaio --zonemode=zbd --rw=
    read --bs=4Ki --runtime=30 --
    numa_cpu_nodes=$DEV_NUMA_NODE --
    ramp_time=10 --iodepth=$depth --
    time_based --percentile_list=50:95
```