SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Zoned Storage and Non-Sequential Write Patterns

Mohamed Mehdi Gharam

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Zoned Storage and Non-Sequential Write Patterns

Zoned Storage und nicht-sequentielle Schreibmuster

Author:	Mohamed Mehdi Gharam
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Simon Ellmann, M.Sc.
Submission Date:	October 15, 2024

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, October 15, 2024 Mohamed Mehdi Gharam

Abstract

Traditional flash-based SSDs have long suffered from significant overheads introduced by the flash translation layer (FTL), leading to unstable performance, higher complexity, and faster wearout. Zoned namespaces (ZNS) SSDs have recently emerged as a solution to this problem. By eliminating garbage collection and shifting the responsibility of data placement to the host, ZNS SSDs promise quicker and cheaper storage than conventional SSDs. Hosts are then free to either adhere to ZNS constraints or to build their own host-side FTL for smarter data placement that minimizes write amplification.

In this thesis, we extend vroom, a user-space driver written in Rust, with zoned namespace support. Additionally, we provide our own host-side FTL implementation to support non-sequential writes through a new interface. We found that vroom's performance benefits still hold up when using zoned storage. We also evaluate the decisions we made when building a host-side FTL and share some insights that would prove beneficial to anyone attempting the same. Overall, we found our interface's overheads to be acceptable and that it has the potential to outperform conventional SSDs with the right garbage collection strategy.

Contents

1 Introduction

As the demand for high-performance, low-latency storage solutions continues to grow, particularly by data centers and cloud service providers, solid-state drives (SSDs) have become increasingly critical to meeting these needs, as they provide much better performance than hard disk drives (HDDs) and are increasingly more affordable.

Most SSDs expose a block interface like HDDs, which exposes a flat address space where all logical blocks can be written to or read from. However, this differs quite significantly from flash-based SSD internals, which have several write constraints. In order to expose a conventional interface, flash media rely on the Flash Translation Layer (FTL) and Garbage Collection (GC), which have a significant performance overhead.

Zoned Namespaces (ZNS) [15, 1] is a new specification that allows flash-based SSDs to circumvent the overhead of exposing a block-interface by shifting the burden of adhering to flash media constraints to the host. They achieve this by dividing the address space into zones, which represent the erase unit and in which only sequential writes are allowed. Hosts can then either adhere to ZNS constraints or build their own host-side FTL, which they can tailor to their specific needs.

Zoned storage comes with several advantages [21]. Unlike conventional SSDs, ZNS SSDs no longer need garbage collection due to the interface they expose, and the FTL is significantly less complex (only zone-to-flash block mappings are needed) and requires less on-board memory. ZNS SSDs also don't have to dedicate as much space for overprovisioning, which is used in conventional SSDs to mitigate the effects of garbage collection. This means that ZNS SSDs effectively expose most of their space to the user (some is still needed to replace bad flash blocks). Thanks to this, ZNS SSDs are typically cheaper and faster than conventional drives. There are also more subtle advantages to zoned storage. The block interface abstracts too many of the write constraints that are imposed by the nature of flash media, making it harder to optimize according to application-level information that the SSD can't normally know about or use. By shifting the burden of garbage collection to the host, ZNS SSDs permit fine-tuned data placement that minimizes write amplification.

ZNS drives are increasingly seeing more support. Several file systems, such as btrfs [18] and f2fs [20], were easily made to support ZNS drives, thanks to their log-structured approach that is well suited for the ZNS sequential write constraint. RocksDB, a key-value store, also offers native support for zoned storage and even saw major performance gains with ZNS drives [1].

In this thesis, we extend vroom, a user-space NVMe driver written in Rust [17], with zoned namespaces support. We also build our own host-side FTL implementation to support non-sequential writes. As vroom was particularly built to make use of Rust's memory safety guarantees, we stay faithful to this purpose, as we rely on Rust's thread safety guarantees in our random write interface implementation in section 4.2.

2 Background

In this section, we first introduce the NVMe driver that we're extending. We then discuss flash media internals and zoned storage. This is particularly important because we later use ZNS SSDs in section 4.2 to mimic the way flash-based SSDs work. In the end, we introduce our programming language of choice, Rust, and what we gain from it.

2.1 vroom

vroom is a userspace NVMe driver written in Rust. Its goal is to achieve SPDK-like performance with simpler code and with the memory safety guaranteed by Rust, and it largely succeeds [17]. However, vroom currently only supports a subset of the NVM I/O command set and fails to identify other supported I/O command sets during initialization.

Like all NVMe drivers, vroom uses submission and completion queues that are implemented as ring buffers [14]. Commands are submitted to the submission queue, and after processing them, the controller submits a completion entry in the completion queue, which contains error codes if any occurred and any values that were potentially returned by the command (e.g., the zone append command returns the first LBA that was written to). Concurrency is then achieved by assigning each thread its own queue pair.

2.2 Flash Media

Conventional flash-based SSDs are organized into *pages* (typically 4-16KiB) that represent the read/write unit. Due to the physical nature of flash storage, pages must be erased before they can be overwritten. However, erases can only occur for *blocks*, which are composed of several pages. Despite this, flash SSDs usually adhere to the conventional block interface that was made with HDDs in mind. This interface exposes to the host a flat address space that doesn't have any of the previously mentioned restrictions. This is realized through the *Flash Translation Layer* (FTL).

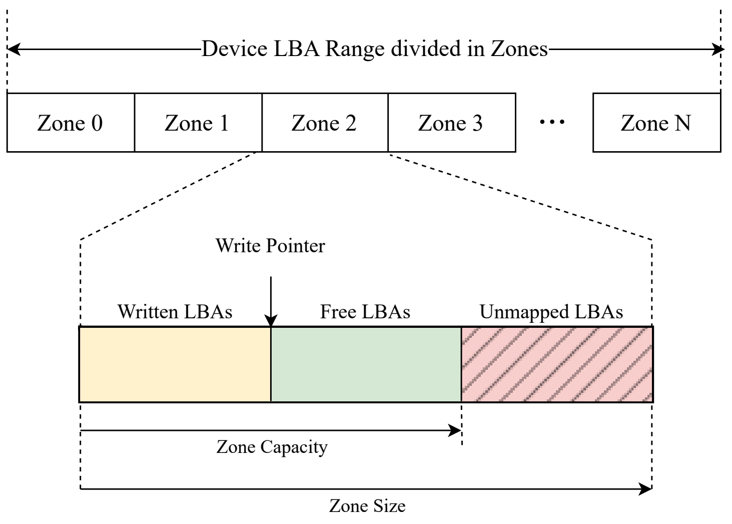
Logical block addresses (LBAs) are mapped to pages. When the host attempts to overwrite a page, it will be marked as invalid, and the new data will be added to a new page. The SSD then updates its internal mapping to make the written LBA point to the new page. Once blocks fill up, a *victim block* is picked, and its valid pages are copied to a new empty block. Afterwards, the victim block is erased. This process is known as *Garbage Collection* (GC), and it unfortunately causes *write amplification*, where a single write to the logical address space actually results in multiple writes due to copying valid pages. Other than the obvious impact on performance, write amplification also causes flash media to wear down faster since it has a limited lifetime. Furthermore, a portion of free space, referred to as *overprovisioning space*, must be set aside for copying so that garbage collection can function effectively. This typically requires reserving 10-28% of the device's total capacity.

Victim selection algorithms for garbage collection have been studied extensively in order to achieve minimal write amplification [19, 24, 13]. While picking the block with the lowest amount of valid pages is the simplest and most intuitive strategy, it suffers from high write amplification in real-world applications. Instead, research has shown that careful thought must be put into data placement so that data that is invalidated at the same time is grouped together.

2.3 Zoned Namespaces

Zoned Namespaces (ZNS) SSDs [4, 22] were introduced to overcome the drawbacks that are commonly associated with flash storage by exposing an interface that is more faithful to how the underlying media works. A ZNS SSD is partitioned into several zones of fixed size. Within a zone, only sequential writes are allowed. This also means that overwriting blocks is not allowed. Instead, zones can be erased with a reset operation. This closely mimics the way flash media works, where blocks are the unit of read/write, and zones are the unit of erasure. To ensure the sequential write constraint, zones maintain a write pointer to the next free LBA that can be written to.
Figure 2.1 illustrates the layout of a ZNS SSD. Note that zones usually have unusable logical blocks at the end that can't be written to. Writable blocks are defined by the zone capacity attribute. This is needed to align zone capacity with the size of flash erase blocks while maintaining the same zone size for all zones.

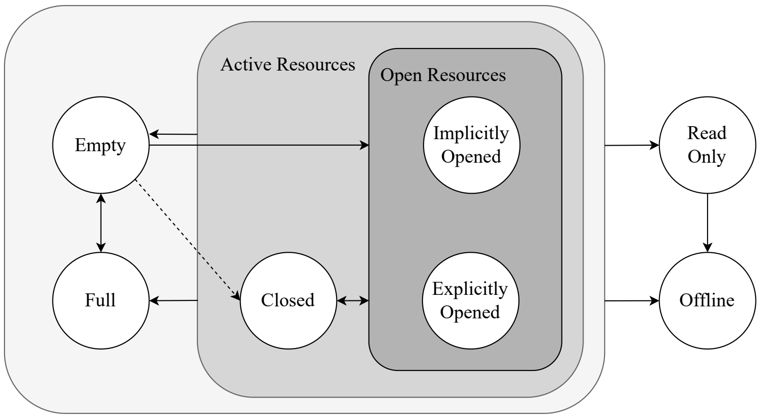
Figure 2.1: Layout of a ZNS SSD. Adapted from [1]



The ZNS command set [15] introduces several new commands and zone operations. One particular shortcoming of ZNS is that writes result in high lock contention for the zones they occur in, as concurrent requests are normally reordered to maximize performance. This effectively means that the host cannot guarantee the sequential write constraint introduced by ZNS when issuing concurrent writes, resulting in completion errors. To avoid this, the ZNS command set introduces the zone append command. Unlike the write command, zone append doesn't specify an offset and only needs to know the zone start LBA, which allows concurrent zone append commands. It's worth noting that append commands perform significantly worse than regular write commands [6].

Each zone is associated with a state machine, which is shown in Figure 2.2. Due to the need for ZNS SSDs to allocate resources, such as write buffers, for each open zone, and considering the characteristics of flash media, such as program disturbs [3, 1], zoned storage drives typically specify limits on the number of zones that can be in an open or active state. These limits are defined by the *maximum open resources* and *maximum active resources* fields.

Figure 2.2: Zone State Machine. Adapted from [15]



2.4 Rust

Rust is a low-level systems programming language that is steadily increasing in popularity. It aims to deliver performance that is comparable to C while ensuring memory and thread safety thanks to its unique ownership system.

One of the Rust slogans is *Fearless Concurrency*. Thanks to the borrow checker, it tries to prevent data races by ensuring exclusive ownership of mutable data. This is especially helpful for our host-side FTL implementation in subsection 4.2.3. As we attempt to synchronize concurrent readers, writers, and garbage collection threads, we found it helpful to have some guarantees about data races being prevented. Nevertheless, Rust is not fully capable of ensuring a bug-free implementation, as it cannot fully prevent deadlocks or race conditions from occurring [26].

3 Related Work

3.1 SPDK

The Storage Performance Development Kit (SPDK) provides several tools and libraries that are used in high-performance storage applications. This includes a poll-based user-space NVMe driver that performs far better than other storage APIs [5], thanks to the elimination of kernel context switches and interrupt handling [25]. As of version 20.10[1], SPDK's NVMe driver also supports the ZNS command set.
Although it is the de facto standard for user-space NVMe drivers, it remains highly complex and prone to memory safety vulnerabilities due to being written in C.

3.2 Host-Side FTLs

There have been several attempts at building host-side flash translation layers. *dm-zoned* is a Linux device mapper that provides a conventional block interface for zoned storage devices by supporting random writes, similar to what we try to achieve in section 4.2. However, it is only ZBC and ZAC compliant, predecessors to the ZNS specification for Shingled Magnetic Recording (SMR) HDDs [7, 8]. For example, it does not recognize zone capacities and relies on the existence of special *conventional* zones that already support non-sequential writes out of the box. These conventional zones are not defined in the latest ZNS specification at the time of writing.

dm-zap is another device mapper that aims to fully support ZNS SSDs. However, it is only a prototype and is not fully functional at the time of writing. Salkhordeh et al. [19] implement a large variety of victim selection algorithms within *dm-zap* and analyze their overhead and impact on write amplification.

It's important to mention that the idea of delegating device management to the host didn't originate with zoned namespaces SSDs. *Open-channel SSDs* [2] are a predecessor to ZNS SSDs that do not implement a flash translation layer and delegate it to the host. *pblk* [2] is a host-side FTL that provides a random write interface for open-channel SSDs.

[1] `https://spdk.io/release/2020/10/30/20.10_release/`

4 Implementation

4.1 Zoned Namespace Command Set

We first extend vroom to support the Zoned Namespace command set. This is concretely achieved through:

1. Adding support for other command sets during initialization by correctly setting the controller configuration register of the NVMe drive and then identifying the namespaces that support the zoned namespace command set.
2. Implementing the commands that were added in the zoned namespace command set. These are the zone management send, zone management receive, and zone append commands.

Zone management send is used to transition zones to different states (Figure 2.2), including resetting them. On the other hand, zone management receive is used to retrieve information about zones, such as their size, capacity, current state, and current write pointer. Zone append was introduced in section 2.3. Note that no changes to the current write command implementation are needed, as the sequential write constraint imposed by ZNS is solely the host's responsibility.

We also add some utility functions. One that is particularly helpful is the zone report function, which is inspired by a function of the same name from the nvme-cli utility [16]. As shown in Listing 4.1, the output illustrates the current state of zones in the NVMe drive and is simply a visualization of the zone descriptors returned by the zone management receive command.

Listing 4.1: Excerpt from the zone report output

```
nr_zones: 1905
SLBA: 0x00000 WP: 0x43500 Cap: 0x43500 State: Full Type: Seq. Write Required Attrs: 0x0
SLBA: 0x80000 WP: 0x9f0b4 Cap: 0x43500 State: Imp. Open Type: Seq. Write Required Attrs: 0x0
SLBA: 0x100000 WP: 0x100000 Cap: 0x43500 State: Empty Type: Seq. Write Required Attrs: 0x0
SLBA: 0x180000 WP: 0x180000 Cap: 0x43500 State: Empty Type: Seq. Write Required Attrs: 0x0
SLBA: 0x200000 WP: 0x200000 Cap: 0x43500 State: Empty Type: Seq. Write Required Attrs: 0x0
```

4.2 Supporting non-sequential write patterns

One of our goals is to provide an interface for ZNS drives that supports non-sequential write patterns, similar to the block interface that is provided by conventional (non-ZNS) SSDs. Since conventional SSDs internally follow constraints that are largely similar to those imposed by zoned storage, we can take inspiration from them as we build our own host-side flash translation layer.

We achieve this through the `ZNSTarget` struct (Listing 4.3) that takes a backing ZNS drive and maintains various data structures (Listings 4.2 and 4.3) to simulate an FTL. The `ZNSZones` struct maintains information about the backing device's zones and classifies them into free zones that can still be written to, full zones that are marked to be garbage collected, and overprovisioning (op) zones that are reserved exclusively for the GC process, just like conventional SSDs. The total number of writeable blocks that we expose is therefore $zone\_capacity * (nr\_zones - op\_zones)$.

As shown in Table 4.1, `ZNSTarget` provides a similar interface to the one that vroom offers for conventional SSDs. For concurrency, each thread in vroom can have its own submission and completion queue pair. These are then used by `read_concurrent` and `write_concurrent` to avoid lock contention on the `NvmeDevice` struct, just like vroom's `submit_io` function. A mutable reference to `ZNSTarget` is therefore not required.

Name	**Zero-Copy**	**Concurrency**
`read()`	Yes	No
`write()`	Yes	No
`read_copied()`	No	No
`write_copied()`	No	No
`read_concurrent()`	Yes	Yes
`write_concurrent()`	Yes	Yes

Table 4.1: I/O methods provided by `ZNSTarget`

In the upcoming subsections, we will delve into the specifics of our host-side FTL, justify some of our design decisions, and outline some assumptions.

4.2.1 Mapping

One of the first decisions we were faced with was how to implement the mapping in our host-side FTL. Conventional FTLs can have page-level or block-level mappings, and each approach has its own advantages. In fact, most modern FTLs use a hybrid approach that combines the best of both worlds [12, 9]. Unfortunately, unlike pages

Listing 4.2: Host-side FTL mappings

```
struct ZNSMap {
    l2d: Vec<u64>, //Logical to device mapping
    d2l: Vec<u64>, //Device to logical mapping, needed for GC
    invalid_bitmap: Vec<bool>
}
```

within a flash block, logical blocks within a zone must be written sequentially. This is an additional constraint that many SSDs do not have, and it forces us to reject zone-level mapping, the ZNS equivalent to block-level mapping. The reason is that zone-level mappings rely on having constant offset for the LBAs, which means that we must be able to write to all logical blocks within a zone.

While a hybrid-level approach might still be possible, it is significantly more complicated than an LBA-level approach, the ZNS equivalent to page-level mapping. Furthermore, the significant memory overhead of page-level mapping is not as severe for a host-side FTL as it is for a traditional FTL, as it is considerably cheaper for the host to use more memory.

For these reasons, we decide to use an LBA-level mapping. We achieve this through a `ZNSMap` struct that is shown in Listing 4.2. This struct maintains both logical-to-device and device-to-logical mappings, as the latter is needed when copying valid blocks during garbage collection. Additionally, a bitmap for invalid blocks is maintained. Since logical and device block addresses start from 0, we can use them as indexes in vectors instead of maintaining a hashmap.

4.2.2 Garbage Collection

`ZNSTarget` implements two methods for garbage collection: `reclaim` and its concurrent counterpart `reclaim_concurrent`, which are used by the respective I/O methods from Table 4.1. Both of them function similarly to the internal GC that can be found in conventional SSDs:

1. Pick a victim zone and an overprovisioning (op) zone.
2. Copy all the valid blocks in the victim zone to the op zone while updating the mapping.
3. Reset the victim zone, mark it as an op zone, and mark the op zone as a free zone.
4. Update the victim selection metadata of both zones accordingly.

Picking a victim is implemented in a way that could support multiple victim selection algorithms. However, only the greedy strategy, which picks the zone with the highest amount of invalid blocks, is currently implemented. Other strategies, such as Cost-Benefit (CB) [10], add a timing factor to victim selection by considering the time since the last modification of a zone alongside the number of invalid blocks. It is important to mention that some of these strategies have some significant performance and memory overheads. For example, the FeGC policy [11] needs to maintain several variables per page alongside several heaps, resulting in a large memory footprint. Careful analysis of the drawbacks and the impact on write amplification is therefore needed.

Furthermore, the second copying step makes a couple of assumptions. First, all zones must have the same zone capacity. Second, we assume that the zone variable capacity flag is not set for the backing ZNS drive [15]. This flag means that zone capacity may change after a reset operation. These proved to be fairly safe assumptions in our experience that we made for simplicity's sake. Nonetheless, adapting the implementation to support drives that violate these assumptions shouldn't be difficult and most likely won't have any impact on performance.

The base NVMe specification recently introduced the simple copy command that allows the host to issue device-managed data copy operations. The command was made with the host-side garbage collection use case in mind and has the advantage of not consuming any host resources or bandwidth, allowing for similar GC performance to conventional SSDs [23]. Although we did implement the copy command, it was unfortunately not supported by the drive we used, and we had to replace it with read/write commands and reserve a DMA buffer for it. Nonetheless, we recommend using the copy command for any drives that support it.

4.2.3 Concurrency

A variety of synchronization primitives are used to allow concurrent reads and/or writes to our interface. In the following, we explain how some conflicts are prevented:

- **Concurrent reads:** Aside from concurrent access to mapping data, which is locked behind a mutex, concurrent reads don't interfere with each other.
- **Concurrent writes:** Writers take out a free zone from the `ZNSZones` struct. Since they own the zone that they're writing to, no other thread can write to the same zone. Concurrency is therefore achieved in an inter-zone manner, as each thread writes to a different zone. For this reason, we also separate zone metadata (`MapperZoneMetadata`) and zones (`MapperZone`). Zone metadata can be modified

by everyone and should always be accessible, while ownership of `MapperZone` implies that only one writer thread can write to that zone.
An intra-zone approach where all writer threads write to the same zone using zone append would've also been possible. Benchmarks by Doekemeijer et al. [6] show that zone append performance remains unaffected by whether inter-zone or intra-zone operations are used. We therefore chose this approach for simplicity, as intra-zone parallelism would require further synchronization for writes that would end up filling up the zone, causing concurrent writes to fail (write or append operations over multiple zones are not allowed in the ZNS specification).
One drawback of the inter-zone approach we chose is that the number of concurrent writer threads is limited by the *Maximum Open Resources* (MOR) field, introduced in section 2.3. However, we generally expect that there will be a performance drop from having too many concurrent threads before this limit is reached.

- **Concurrent reads and reclaim:** Reclaiming zones during garbage collection can interfere with reads, in case we're reading from the victim zone that is being copied while mapping information isn't fully updated yet. For this reason, we define `reclaim_locks` (Listing 4.3): a vector of Reader-Writer locks with one lock for each zone. After picking the victim zone, reclaim would lock the corresponding RWLock with exclusive write access, blocking reader threads until it's freed. We chose an RWLock instead of a mutex because readers don't need exclusive access.

- **Concurrent writes and reclaim:** If a writer thread fills up a zone, it adds it to the list of full zones and notifies the reclaim thread through the `reclaim_condition` condition variable, after which the latter starts garbage collection if certain conditions are met. We arbitrarily chose that there must be more full zones than free zones for GC to start, though more complicated conditions can be used instead, depending on the use case.

4.2.4 Reads and Writes

Due to several constraints, reads and writes to the interface need to be internally split into several commands to the drive. In the following, we distinguish between logical blocks, which are exposed by our interface, and device blocks, which are the LBAs exposed by the backing zoned device.

As mentioned in subsection 4.2.3, read and write commands are generally not allowed to specify blocks that belong to more than one zone. Although some drives do offer an exception for read commands, a split according to zone boundary is still needed to properly synchronize concurrent reads with garbage collection. Furthermore, sequential logical blocks are not necessarily mapped to sequential device blocks, for example, if they were written at different times by different threads. This means that we need to split commands to the interface so that we can issue the biggest possible commands to the device. For reads, we also have to account for unmapped logical blocks and split issued reads to the interface to handle these blocks separately. For writes, overwriting previously written logical blocks requires invalidating them and updating their corresponding zone's metadata, requiring further splits.

Listing 4.3: Host-Side FTL Data Structures

```
struct MapperZoneMetadata {
    //Victim selections algorithms data should be here
    invalid_blocks: u64,
    zone_age: u64, //unused
}

struct MapperZone {
    zslba: u64,
    zone_cap: u64,
    wp: u64
}

struct ZNSZones {
    free_zones: Vec<MapperZone>,
    full_zones: Vec<MapperZone>,
    op_zones: Vec<MapperZone>
}

pub struct ZNSTarget {
    pub backing: Mutex<NvmeDevice>, //Backing ZNS device
    pub max_lba: u64,
    exposed_zones: u64,
    pub ns_id: u32,
    pub block_size: u64,
    pub zns_info: NvmeZNSInfo,
    map: Mutex<ZNSMap>,
    victim_selection_method: VictimSelectionMethod,
    zones: Mutex<ZNSZones>,
    zones_metadata: Vec<Mutex<MapperZoneMetadata>>,
    reclaim_buffer: Dma<u8>,
    reclaim_locks: Vec<RwLock<()>>,
    reclaim_condition: Condvar,
    pub end_reclaim: AtomicBool
}
```

5 Evaluation

We evaluate vroom's performance when used directly with zoned namespaces SSDs and then through the host-side flash translation layer interface that we built. In the process, we attempt to characterize vroom's zoned storage performance and compare it to conventional SSDs and other I/O engines.

5.1 Setup

All benchmarks are run on a system with an AMD EPYC 7713 64-Core processor and 1 TiB of RAM running on Ubuntu 24.04 LTS using a 4TiB ZNS SSD.

For our benchmarks, we use a similar setup to the one that was first used to evaluate vroom. For reads, we use a full ZNS drive, as NVMe controllers can be aware of reads to unwritten blocks and process them faster, leading to much better performance. Similarly, we only perform reads within the writable section of every zone (i.e. within the zone capacity), since we found that reads to the unmapped LBAs section from Figure 2.1 are also processed faster and would skew results. For consistency's sake, we also run write benchmarks on completely empty drives. However, unlike conventional SSDs, we found ZNS write performance to be constant with time since the drive does not have to perform garbage collection, and write amplification is kept minimal by design. For this reason, we have decided to run 60s sequential write workloads, a notable decrease from the 900s write workloads that were used to initially benchmark vroom, as throughput was found to vary with time. Similarly, we also use 60s random read workloads, as read performance is also constant with time. All benchmarks are run with I/O unit sizes of 4KiB.

In section 5.2, we compare vroom's performance against other I/O engines: SPDK as another user-space NVMe driver and the linux storage engines: `io_uring`, `libaio` and the file I/O API `pread/pwrite` (`psync`). We achieve this by using the flexible I/O tester (`fio`). However, its support for ZNS drives is fairly limited. Although it is possible to force sequential writes and cross-zone constraints, write workloads are limited by default to the regular write command, making it difficult to compare zone append performance. SPDK's `fio` plugin, however, does offer more extensive ZNS support, with which we could also benchmark zone append commands using the configuration

Listing 5.1: Zone Append benchmark using fio and SPDK

```
[global]
ioengine=spdk
thread=1
group_reporting=1
direct=1
time_based=1
ramp_time=5
runtime=60
size=128z
bs=4k
rw=write
iodepth=1
zonemode=zbd
zone_append=1
max_open_zones=13
initial_zone_reset=1
filename=trtype=PCIe traddr=0000.c6.00.0 ns=2
```

in Listing 5.1. For vroom, we use our own workloads that are equivalent to `fio`'s jobs. These can be found in our repository.

5.2 Zoned Namespaces

In this section, we evaluate the performance of ZNS drives with vroom and compare it to other I/O engines. First, we attempt to characterize general ZNS performance as we use our observations to justify some of the decisions we took with our host-side FTL implementation in section 4.2. We then run similar performance benchmarks to the ones that were first used to evaluate vroom's performance on a conventional non-ZNS SSD [17] to see whether vroom's performance benefits still hold when using zoned storage.

5.2.1 Throughput

We examine the throughput of both write and append commands under different queue depths and thread counts. Because of zoned storage restrictions (section 2.3), write commands are limited to queue depth 1, as they may be reordered by the controller and cause completion errors. Furthermore, there are different ways to parallelize zone append commands: an **intra-zone** approach where concurrent commands are issued to the same zone and an **inter-zone** approach where concurrent commands are issued to multiple zones. Due to the sequential write constraint, writes are limited by their nature to inter-zone concurrency. Additionally, the *Maximum Open Resources* field limits

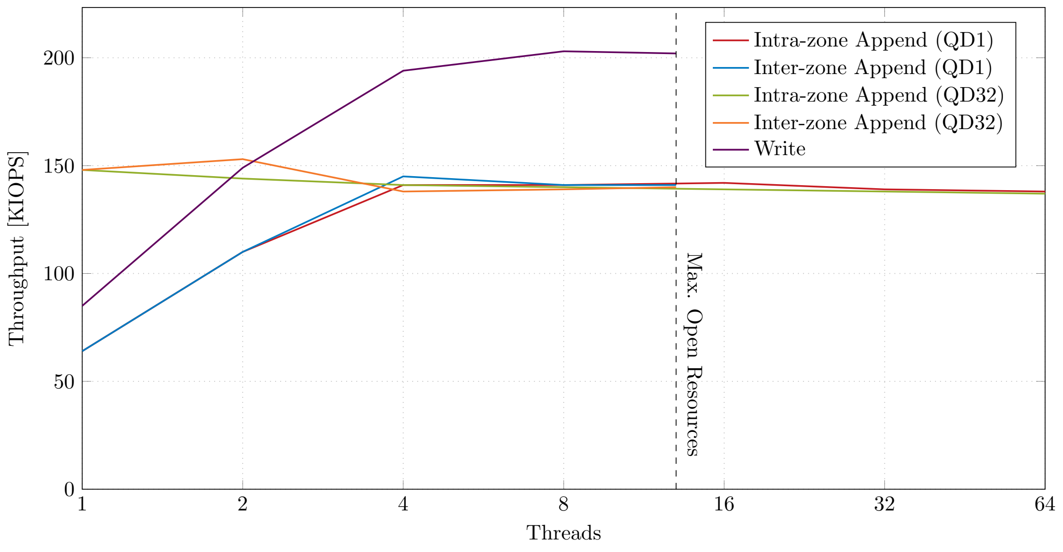


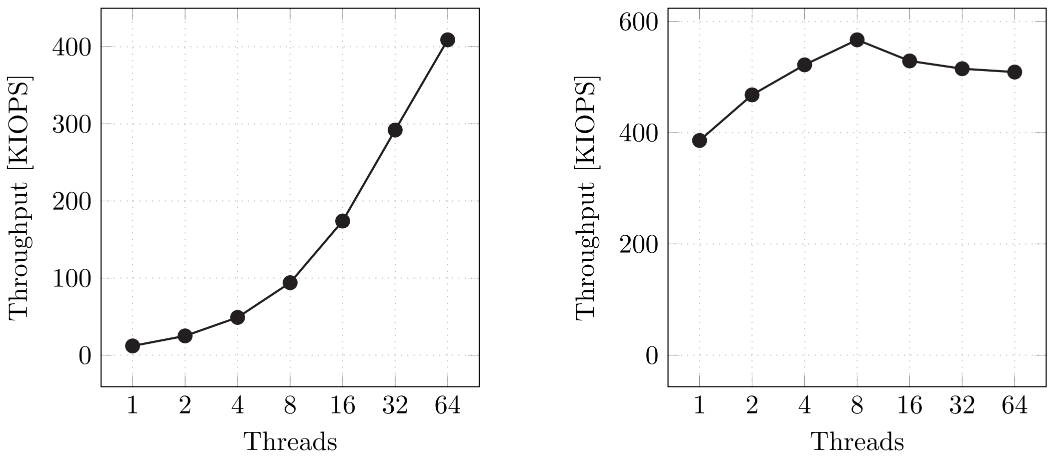
Figure 5.1: Threads vs. IOPS; vroom direct writes to a ZNS drive

the number of zones that can be used concurrently, therefore limiting the number of threads in an inter-zone scenario.
We visualize our findings in Figure 5.1 and observe the following:

- Write commands have around 25% higher throughput than zone append commands: write commands have a maximum throughput of about 200k IOPS, while zone append commands seem to peak around 150k IOPS. Despite being limited to a queue depth of 1, write commands almost always perform better.
- The choice between an intra-zone or inter-zone approach does not affect the performance of the zone append command.
- Zone append throughput seems to be limited to about 150K IOPS, and further increases to queue depth or thread count after reaching that number seem to have no effect.
- Throughput for write and zone append commands reaches its peak and stagnates long before we're limited by the *maximum open resources* field.

These observations seem to align with current research characterizing ZNS SSD performance [6].

Next, we examine the read command throughput of ZNS drives, focusing first on how the number of threads affects performance at two different queue depths (1 and 32), as shown in Figure 5.2. For a queue depth of 1, throughput constantly increases, reaching a maximum of around 400k IOPS at 64 reader threads. In contrast, at a queue depth of 32, throughput steadily increases in the beginning until it peaks at about 570k IOPS for 8 reader threads. Then, it gradually declines as the number of threads increases. This suggests that higher queue depths have a greater impact on read performance than increasing the number of concurrent readers. We further verify this by plotting throughput for various queue depths with a single reader thread in Figure 5.3. As expected, throughput rises with increasing queue depth, peaking at approximately 910k IOPS beyond a queue depth of 128.



(a) Queue Depth 1 **(b)** Queue Depth 32

Figure 5.2: Threads vs. IOPS; vroom direct random reads to a ZNS drive

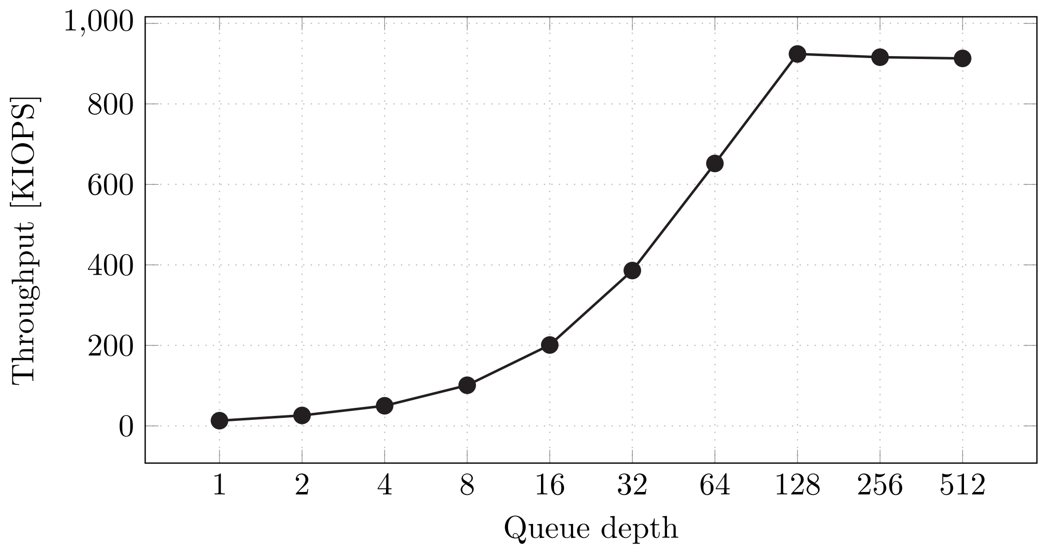
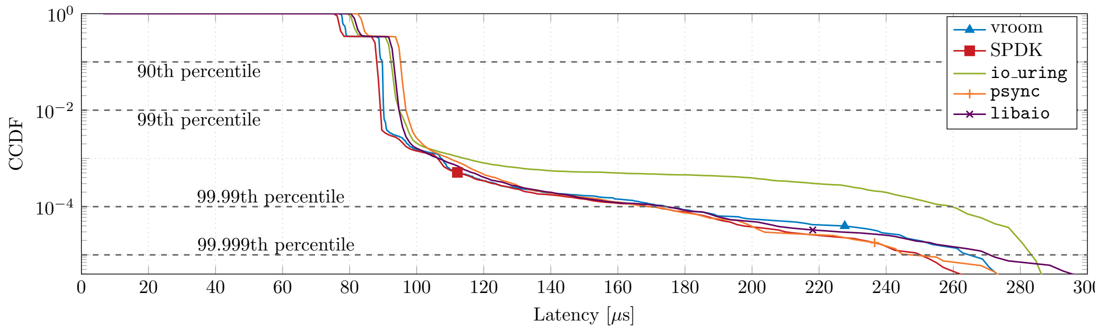


Figure 5.3: Queue Depth vs. IOPS; vroom direct reads to a ZNS drive (1 reader)

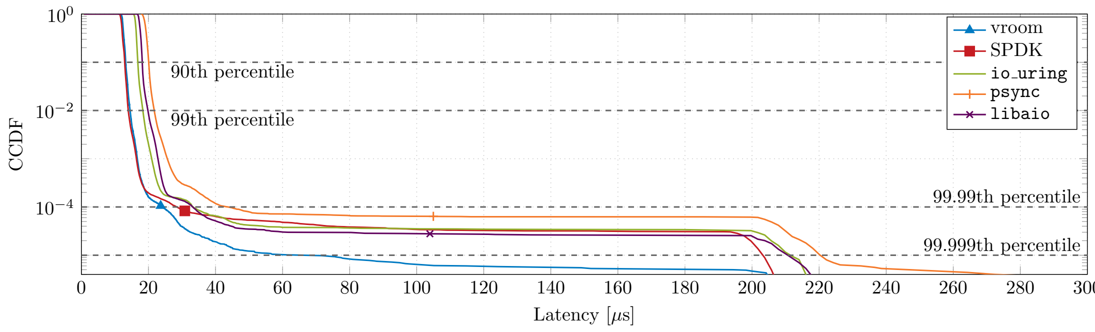
5.2.2 Latency

In Figure 5.4, we present the tail distributions of I/O latencies for random read, sequential write, and zone append operations on a ZNS drive using various I/O engines. Note that we can only compare vroom's append latency with SPDK, as `fio` currently lacks support for zone append benchmarks across other storage engines.

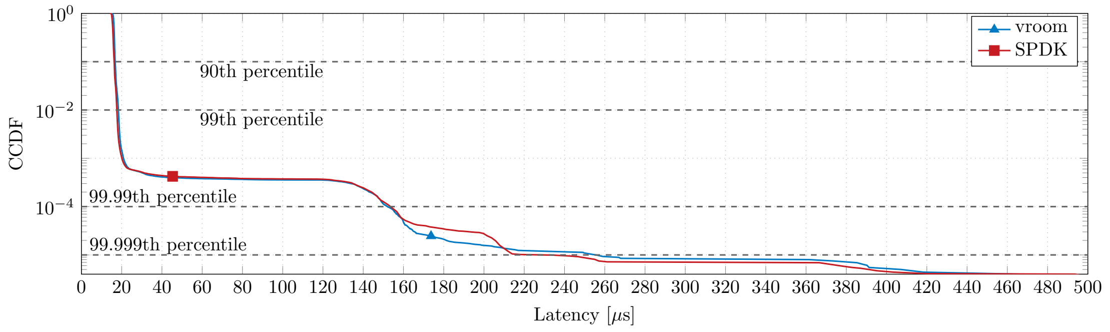
Our driver, vroom, and SPDK generally outperform the competition, demonstrating lower latencies than all other I/O engines in all operations. For sequential writes, the latency patterns are similar across all engines, with Linux I/O APIs having a higher offset of about 10 μs due to system call and interrupt overhead, which is eliminated entirely by SPDK and vroom. Overall, SPDK and vroom deliver comparable performance, though vroom occasionally has a slight advantage, which may be due to `fio`'s small inherent overhead when used for SPDK benchmarking. These results are consistent with vroom's performance compared to other I/O engines when using a conventional SSD [17]. The plots also align with our observations from subsection 5.2.1. On average, zone append latency is 25% higher than sequential write latency.



(a) Random read



(b) Sequential write



(c) Zone Append

Figure 5.4: Tail Latencies; direct operations to a ZNS drive

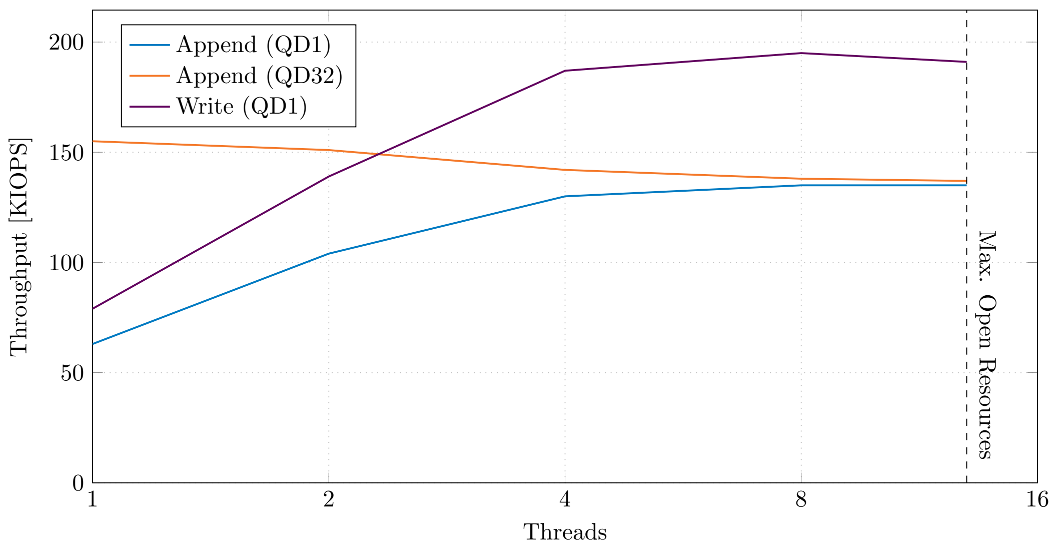


Figure 5.5: Threads vs. IOPS for different host-side FTL write implementations

5.3 Random Write Support

We now evaluate read and write performance through our host-side FTL and compare it with direct reads and writes to the ZNS drive. One side-effect of evaluating write performance on empty drives is that we do not get to evaluate our garbage collection implementation. However, we have decided not to do this for a couple of reasons. First, our ZNS drive does not support the copy command, which causes garbage collection to be much slower and also interfere significantly more with concurrent reads and writes, as they share device resources. Another reason is that performance would depend on the specific victim selection strategy that is used, as these tend to seek a balance between write amplification, memory overhead, and speed. For these reasons, we felt that including garbage collection would potentially cause write performance metrics to be misleading.

5.3.1 Throughput

In Figure 5.5, we compare the throughput of two different implementations of the `write_concurrent` function from Table 4.1: one utilizing the write command, the other using the zone append command. As expected, the results align with those shown in Figure 5.1, where we discussed the throughput of the zone append and write commands for a standard ZNS drive. Note that our host-side FTL is restricted to *inter-zone*

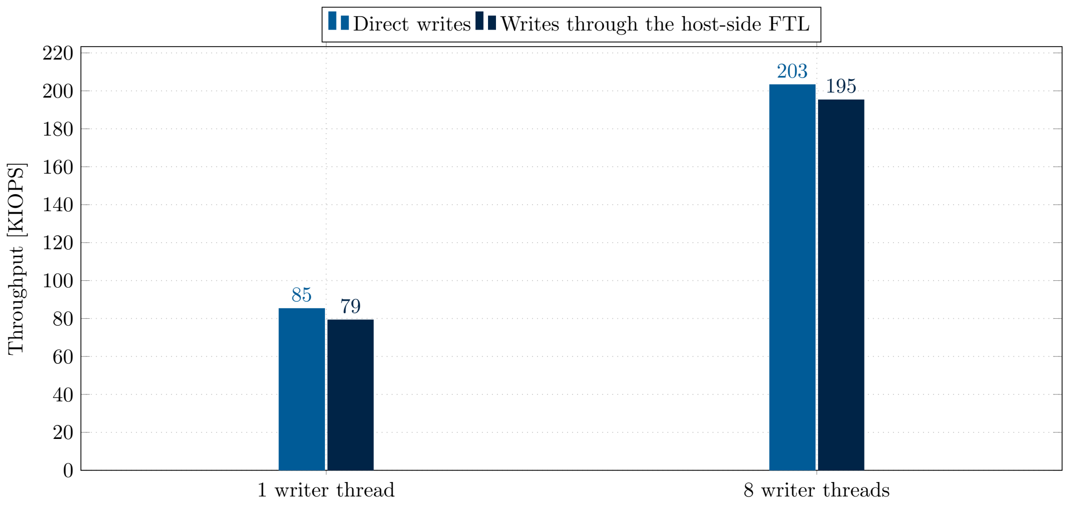


Figure 5.6: Random write throughput for different thread counts; Queue Depth 1

concurrency, as mentioned in subsection 4.2.3. We therefore summarize that a random write function implementation is most efficient when using the regular write command, despite the restriction on queue depth. We also confirm the hypothesis stated in subsection 4.2.3 that the *Maximum Open Resources* field does not restrict our random write performance. We also notice that our interface's overhead on write throughput is fairly minimal, regardless of the method used. We confirm this by comparing the throughput of direct writes to a ZNS SSD to the throughput of writes that go through our host-side FTL in Figure 5.6. Although we are technically comparing sequential write throughput for the ZNS SSD against random write throughput for our host-side FTL, it actually does not make any difference. Sequential writes usually perform better than random writes because they simplify garbage collection, as data that is grouped together is more likely to be invalidated together. Since we do not include garbage collection in our benchmarks, it does not make any difference for our host-side FTL whether we use random or sequential writes.

We now direct our attention to read performance. As we did in subsection 5.2.1, we consider how queue depth and concurrent readers affect random read IOPS. We plot the throughput for different queue depths in Figure 5.7 and for different thread counts under QD1 and QD32 in Figure 5.8. We also visualize the plots from Figure 5.2 and Figure 5.3 again to compare random reads through our interface with direct reads to the drive.

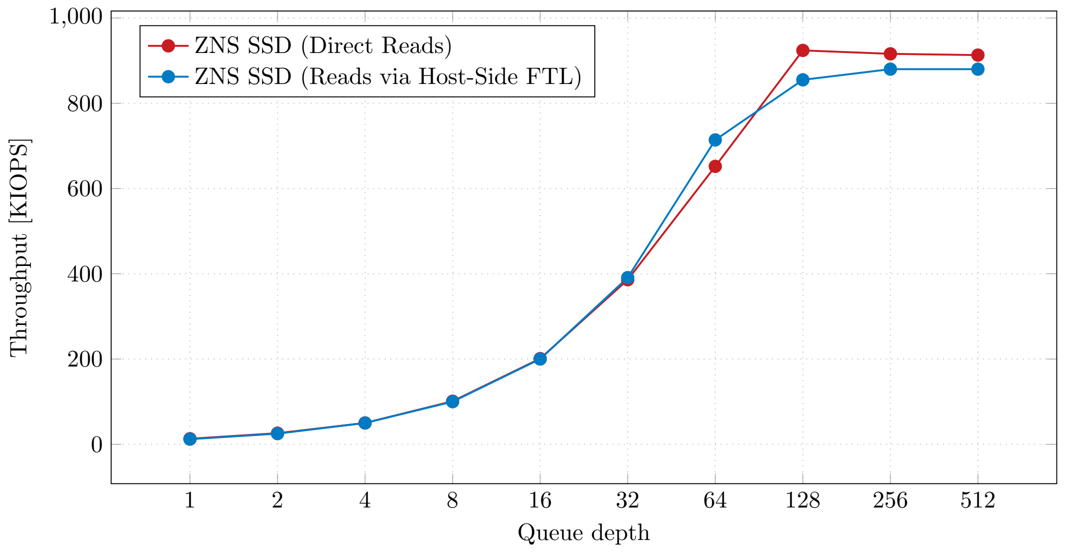
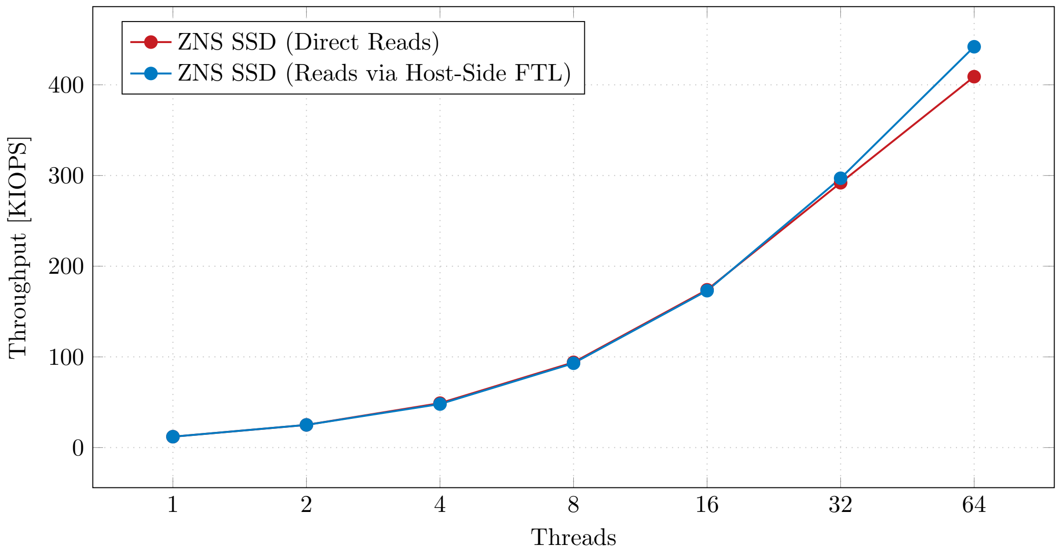


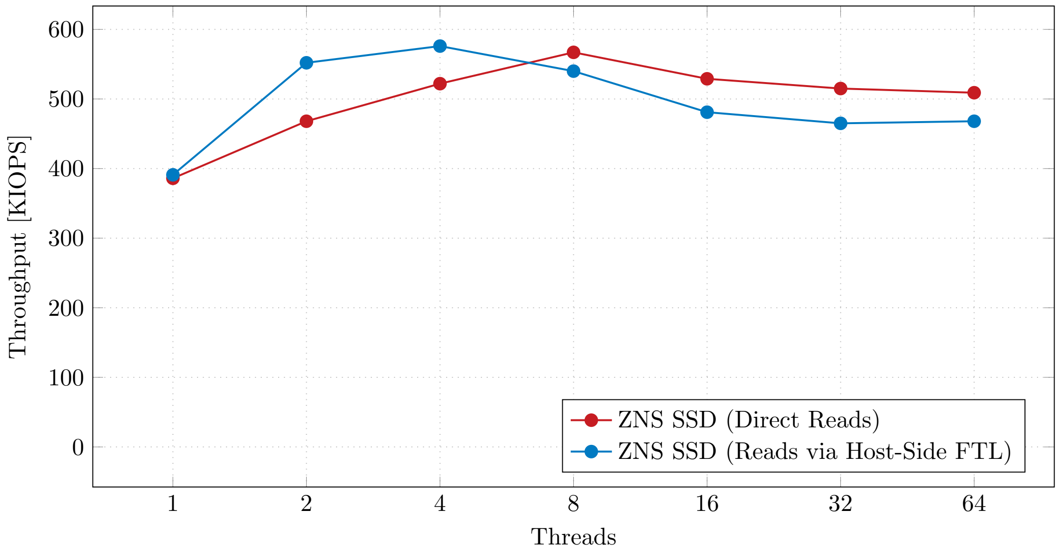
Figure 5.7: Queue Depth vs IOPS; vroom random read (1 reader)

We note the following observations:

- Read performance is barely affected by a host-side FTL, especially compared to write performance. Writes via the host-side FTL update zone metadata and need constant synchronized access to mapping data, as they both read it at the start and update it at the end. On the other hand, reads only access mapping data once at the beginning and don't interfere with each other otherwise.
- Similarly to direct reads, reads through a host-side FTL benefit much more from a higher queue depth than a higher thread count. This is especially evident when looking at Figure 5.7.
- As concurrency levels (both queue depth and thread count) increase, there is a brief time when host-side FTL reads outperform direct reads under the same conditions. This is particularly noticeable in Figure 5.8b. It is hard to determine the exact cause, but we surmise that synchronized access to mapping data may pace requests to the controller in a way that better minimizes resource contention within the SSD.

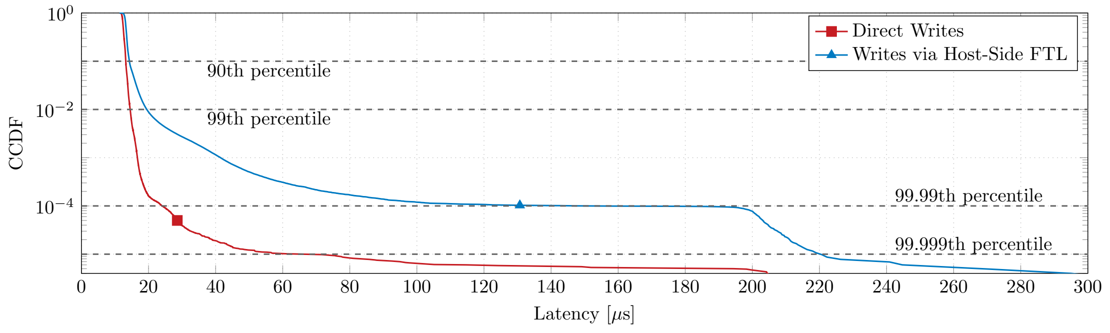


(a) Queue Depth 1

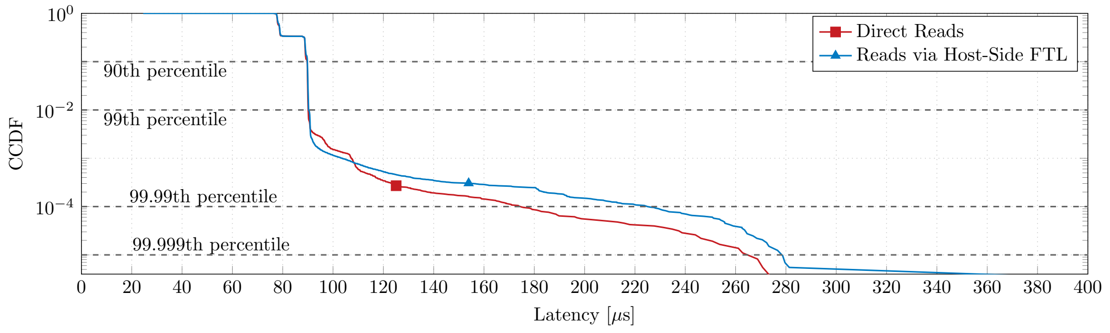


(b) Queue Depth 32

Figure 5.8: Threads vs. IOPS; vroom random read



(a) Random write



(b) Random read

Figure 5.9: Tail Latencies

5.3.2 Latency

We now investigate our observations from the previous subsection by plotting tail latencies for random reads and writes via the host-side FTL in Figure 5.9. We additionally plot in the respective vroom I/O tail latencies from Figure 5.4.

For writes, we notice that both plots are largely similar up to the 90th percentile, after which writes via the host-side FTL start to slow down significantly in comparison. Overwriting previously written blocks has a slightly higher overhead than writes to unwritten LBAs, which explains the higher tail latencies. Synchronization overhead is another factor.

Host-side FTL reads, on the other hand, generally perform the same as direct reads up to the 99.9th percentile. Similarly, the higher tail latencies can be explained by the synchronization overhead. This aligns with our observations from subsection 5.3.1.

6 Conclusion

In this thesis, we examined zoned namespace SSDs and extended vroom, a user-space NVMe driver, to support the zoned namespace command set. Additionally, we built our own host-side flash translation layer to support non-sequential write patterns. We evaluated vroom's performance with a ZNS drive, both to characterize ZNS performance and to ensure that vroom's performance improvements over other I/O engines persisted when using zoned storage. We then evaluated the ZNS drive's performance when using our host-side FTL and compared it against regular ZNS performance. In the process, we also considered different host-side FTL implementations and used our findings to improve them.

We found our ZNS performance characterization to be consistent with the latest research. Write performance was consistently better than zone append performance despite being limited to a queue depth of one. We also concluded that it is better for host-side FTLs to be implemented using the write command. Furthermore, we found that our host-side FTL had acceptable overheads, although our evaluation could not consider garbage collection due to the copy command not being supported.

Future Work Our host-side FTL implementation is still a prototype and has several shortcomings that should be fixed. Mapping data is not persisted and is kept in memory, representing a significant memory overhead when using large drives. It also cannot be reconstructed in the case of a sudden crash or power outrage. Future work should ensure that metadata is properly persisted in a conventional drive and that only a partial copy of the mapping data is kept in memory. Additionally, victim selection during garbage collection is quite basic, and there are several more advanced algorithms that should be implemented and evaluated to minimize write amplification. Finally, we believe that a comprehensive study of the interface's performance during garbage collection is needed to properly evaluate its potential.

List of Figures

List of Tables

Listings

Bibliography

[1] 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: *USENIX Annual Technical Conference*. 2021. URL: https://api.semanticscholar.org/CorpusID:235818273.

[2] M. Bjørling, J. Gonzalez, and P. Bonnet. "LightNVM: The Linux Open-Channel SSD Subsystem." In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 359–374. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.

[3] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai. "Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation." In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 123–130. DOI: 10.1109/ICCD.2013.6657034.

[4] W. D. Corporation. *Zoned Storage Documentation*. 2021. URL: https://zonedstorage.io/ (visited on 10/06/2024).

[5] 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*. SYSTOR '22. Haifa, Israel: Association for Computing Machinery, 2022, 120–127. ISBN: 9781450393805. DOI: 10.1145/3534056.3534945. URL: https://doi.org/10.1145/3534056.3534945.

[6] K. Doekemeijer, N. Tehrany, B. Chandrasekaran, M. Bjørling, and A. Trivedi. "Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS)." In: *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 2023, pp. 118–131. DOI: 10.1109/CLUSTER52292.2023.00018.

[7] T. Feldman and G. Gibson. "Shingled magnetic recording: Areal density increase requires new data management." In: *; login:: the magazine of USENIX & SAGE* 38.3 (2013), pp. 22–30.

[8] G. Gibson and G. Ganger. "Principles of Operation for Shingled Disk Devices." In: *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*. Portland, OR: USENIX Association, June 2011. URL: `https://www.usenix.org/conference/hotstorage11/principles-operation-shingled-disk-devices`.

[9] A. Gupta, Y. Kim, and B. Urgaonkar. "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings." In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: Association for Computing Machinery, 2009, 229–240. ISBN: 9781605584065. DOI: `10.1145/1508244.1508271`. URL: `https://doi.org/10.1145/1508244.1508271`.

[10] A. Kawaguchi, S. Nishioka, and H. Motoda. "A Flash-Memory Based File System." In: *USENIX 1995 Technical Conference (USENIX 1995 Technical Conference)*. New Orleans, LA: USENIX Association, Jan. 1995. URL: `https://www.usenix.org/conference/usenix-1995-technical-conference/flash-memory-based-file-system`.

[11] O. Kwon, K. Koh, J. Lee, and H. Bahn. "FeGC: An efficient garbage collection scheme for flash memory based storage systems." In: *J. Syst. Softw.* 84.9 (2011), 1507–1523. ISSN: 0164-1212. DOI: `10.1016/j.jss.2011.02.042`. URL: `https://doi.org/10.1016/j.jss.2011.02.042`.

[12] Y. Luo and M. Lin. "Flash translation layer: a review and bibliometric analysis." In: *International Journal of Intelligent Computing and Cybernetics* 14.3 (2021), pp. 480–508.

[13] L. Nagel, T. Süß, K. Kremer, M. U. Hameed, L. Zeng, and A. Brinkmann. *Time-efficient Garbage Collection in SSDs*. 2018. arXiv: `1807.09313 [cs.PF]`. URL: `https://arxiv.org/abs/1807.09313`.

[14] NVM Express, Inc. *NVM Express Base Specification Rev. 2.1*. 2024. URL: `https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf` (visited on 10/06/2024).

[15] NVM Express, Inc. *Zoned Namespace Command Set Specification Rev 1.1*. 2021. URL: `https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-1.1-2021.06.02-Ratified-1.pdf` (visited on 10/06/2024).

[16] *nvme-cli*. URL: `https://github.com/linux-nvme/nvme-cli` (visited on 10/06/2024).

[17] T. Pirhonen. "Writing an NVMe Driver in Rust." BA thesis. Technical University of Munich, 2024. URL: `https://db.in.tum.de/~ellmann/theses/finished/24/pirhonen_writing_an_nvme_driver_in_rust.pdf` (visited on 09/25/2024).

[18] M. Rybczyńska. *Btrfs on zoned block devices*. 2021. URL: `https://lwn.net/Articles/853308/` (visited on 10/06/2024).

[19] R. Salkhordeh, K. Kremer, L. Nagel, D. Maisenbacher, H. Holmberg, M. Bjørling, and A. Brinkmann. "Constant Time Garbage Collection in SSDs." In: *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 2021, pp. 1–9. DOI: `10.1109/NAS51552.2021.9605386`.

[20] D. Seo, P.-X. Chen, H. Li, M. Bjørling, and N. Dutt. "Is Garbage Collection Overhead Gone? Case study of F2FS on ZNS SSDs." In: *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. HotStorage '23. Boston, MA, USA: Association for Computing Machinery, 2023, 102–108. ISBN: 9798400702242. DOI: `10.1145/3599691.3603409`. URL: `https://doi.org/10.1145/3599691.3603409`.

[21] 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. Ann Arbor, Michigan: Association for Computing Machinery, 2021, 144–151. ISBN: 9781450384384. DOI: `10.1145/3458336.3465300`. URL: `https://doi.org/10.1145/3458336.3465300`.

[22] N. Tehrany and A. Trivedi. *Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices*. 2022. arXiv: `2206.01547 [cs.OS]`. URL: `https://arxiv.org/abs/2206.01547`.

[23] "Towards Copy-Offload in Linux NVMe." In: 2021. URL: `https://www.snia.org/educational-library/towards-copy-offload-linux-nvme-2021` (visited on 10/06/2024).

[24] M. Wu and W. Zwaenepoel. "eNVy: a non-volatile, main memory storage system." In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. San Jose, California, USA: Association for Computing Machinery, 1994, 86–97. ISBN: 0897916603. DOI: `10.1145/195473.195506`. URL: `https://doi.org/10.1145/195473.195506`.

[25] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. "SPDK: A Development Kit to Build High Performance Storage Applications." In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017, pp. 154–161. DOI: `10.1109/CloudCom.2017.14`.

[26] Z. Yu, L. Song, and Y. Zhang. *Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software*. 2019. arXiv: `1902.01906 [cs.PL]`. URL: `https://arxiv.org/abs/1902.01906`.