Karlsruhe Institute of Technology

What does the Flash Translation Layer do? Exploring Modern NVMe SSD Performance Characteristics

Bachelor's Thesis of

Peter Simon Bohner

At the KIT Department of Informatics
ITEC-OS

First examiner:	Prof. Frank Bellosa
Second examiner:	Prof. Wolfgang Karl
First advisor:	M.Sc. Peter Maucher
Second advisor:	Dipl.-Inform. Thorsten Gröninger

9. January 2025 – 09. June 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 09.06.2025

. . . . . .

(Peter Simon Bohner)

Abstract

In the last decade, NVMe SSDs have become the standard medium for high performance storage. Despite their ubiquity, their internal workings are poorly understood and guidance for efficient use of SSDs is lacking [1]. In this work, we develop a benchmarking suite based on xNVMe [2] capable of reporting individual request latencies, and use it to analyze performance characteristics of multiple consumer NVMe SSDs in variety of synthetic benchmarks. This leads to the discovery of several cases of pathological SSD performance in simple workloads. We issue general guidance for performant SSD use and conclude that the devices tested implement few optimizations and perform vastly different to what their specifications would suggest.

Contents

1. Introduction

In the last decade, solid state drives (*SSDs*) have displaced hard disk drives (*HDDs*) as the dominant storage medium for performance-critical applications, with modern SSDs squeezing extraordinary performance from a less than perfect medium (NAND, see Section 2.1.2) by leveraging ever more sophisticated firmware [1]. However, using SSDs in a way which results in predictable and high performance, is an ongoing challenge facing the industry [1, 3, 4].
As with any device, optimizing for SSDs necessitates detailed knowledge of their behavior [1, 4]. Despite this need, there is little publicly accessible documentation about the inner workings or performance characteristics of SSDs. This is in stark contrast to most other computer components, for example CPUs, where developers have long had access to detailed programmer's and optimization manuals, published by manufacturers [5–7]. Without such first party information, we rely on outdated information and guesses about the structure of NVMe devices [1, 8], and have to treat SSDs as black boxes [4]. From this follows, that much of our understanding and existing guidance for efficient usage of SSDs is over a decade old and derived devices now considered obsolete [9–11].
By analyzing the performance of NVMe SSDs in a variety of custom benchmarks, we seek to improve our knowledge about SSD behavior and aim to gain some insight into the operation of modern flash translation layers. We will use the knowledge obtained to validate commonly cited [1, 4, 8] performance recommendations [4, 9].
We start in Chapter 2, by introducing the key components of NVMe SSDs and the NVMe protocol itself, followed by recapitulating existing performance guidance and other related works in Chapter 2.4. Chapter 3 describes and motivates the experiments performed, whose implementation and validation is covered in Chapter 4. We evaluate their results in Chapter 5 and, finally, conclude by summarizing our findings and discussing future work in Chapter 6.

2. Background

This chapter serves to provide the background information necessary to understand NVMe SSDs. In Section 2.1, we describe the structure of SSDs and highlight the challenges posed by the NAND medium. Continuing in Section 2.2, we enumerate the tasks performed by the SSD controller. Finally, Section 2.3 describes NVMe, the interface used by modern SSDs to communicate with their host.

2.1. Solid State Drives

Solid State Drives (*SSDs*) are devices providing persistent block storage. This means that SSDs retain their data upon power-loss (persistent) can be addressed at block granularity. A block is generally 512bytes [12, 13], although most SSDs do support larger block sizes, such as 4KiB [14, 15].
Almost all SSDs are backed by NAND flash, although other technologies such as *3D XPoint*(sold under the brand name "Intel Optane") do exist. Independent of the storage technology, an SSD contains a **controller**, and optionally volatile memory used as cache.

2.1.1. NAND flash

At the lowest level, NAND flash stores data in memory cells. A cell consists of a floating-gate MOSFET *(FGMOS)*, a transistor with an electrically-isolated gate, on which electrical charge may be trapped. [16] The amount of charge stored, and thereby the voltage measured when reading the cell, determines the information stored. Storing N bit per cell is achieved by using 2^N different voltage levels. The terms single-level cell *(SLC)*, multi-level cell *(MLC)*, triple-level cell *(TLC)* and, quad-level cell *(QLC)* refer to a cell holding one, two, three, or four bits respectively [16, 17].
Storing charge in flash is called *programming*, and clearing charge is known as *erasing* [16, 18].
Cells are organized into *pages*, which are grouped further into *blocks*. Multiple blocks form a *plane*, multiple planes make up a *die*. A physical flash package contains one

or multiple stacked dies. [16] Pages are the smallest unit that can be programmed, whereas blocks are the smallest unit that may be erased. These blocks are sometimes referred to as *erase blocks*, to diffrentiate them from the *logical blocks* exposed over the block storage interface. Each die can perform I/O-commands independently, and an I/O-command may act on multiple planes in parallel [16, 17, 19, 20].
While the logical organization remains the same, modern SSDs almost exclusively use V-NAND (also called 3D-NAND), which arranges cells in three dimensions, allowing for greater density and further performance scaling, than traditional (planar) NAND. In addition, different types of transistors, such as charge trap flash, may be used [16, 18].

2.1.2. Limitations

NAND flash has multiple serious limitations, that SSD controllers need to work around.
Firstly, each program-erase cycle physically degrades the cells, leading to limited write endurance. This varies between flash types, with SLC historically having 2–3 orders of magnitude higher endurance than MLC NAND [21]. Unfortunately, no datasheets for current VNAND cells are publicly available, therefore we are unable to provide precise endurance figures. From the total endurance rating of the Samsung SSD tested in this work (see Section 3.2.1), we can however extrapolate that modern TLC NAND can be rewritten at least 600 times[1].
Secondly, programming a block can only be done sequentially, with each programming operation taking an order of magnitude longer than a read. Erase operations are even slower, on the order of milliseconds. Both read and programming times approximately double for each additional bit per cell stored [17, 21].
For these reasons, MLC/TLC/QLC flash can usually operate in SLC mode, which is used by SSDs to improve read/write speeds by acting as a cache. This is typically accomplished by writing data to blocks in SLC mode, then later rewriting them in the native mode when the number of allocated blocks exceeds a threshold [17, 21].
With increasing flash density, NAND reliability significantly worsens. Read and write disturbances as well as bit errors occur more frequently, and data retention shortens [17]. These issues are also faced by DRAM, where they lead to the *RowHammer* vulnerability [22]. Modern NAND therefore likely also necessitates sophisticated error correction techniques, and spare space in case of failure. This leads to extra capacity at each layer of organization [17, 21, 22].

[1] $\frac{\text{rated write endurance}}{\text{capacity}} = \frac{150\text{TB}}{250\text{GB}} = 600$ [12]

2.2. Controller

SSD controllers are the microprocessors responsible for providing a block storage interface backed by the flash storage they control, and managing how data is stored on flash. Communication with the host on modern devices is usually handled though the *NVMe* protocol, which we introduce in Section 2.3, while older devices typically used the *SATA* interface, which is not covered in this work. Through the NVMe interface, the SSD controller presents individually numbered read and writable blocks (LBAs) to the host. Because of the limitations described in Section 2.1.2, LBAs can not be feasibly mapped to individual flash blocks by identity. Instead, the SSD needs to translate these addresses to flash blocks in a way that works around these issues. This functionality is implemented in the flash translation layer, a major part of the SSD controller's firmware, described in Section 2.2.1. Managing flash storage also necessitates several background processes, such as *garbage collection*, *wear leveling*, or moving data from the SLC cache to slower NAND regions. [8, 9, 23] Because of these, an SSD not currently processing commands may still be reading from and writing to flash [8, 17].
To be able to performantly complete their tasks, SSD controllers require a substantial amount of memory, which in most cases is provided by a DRAM chip on the device [1, 8, 16]. This cache may be used to hold parts of the FTL mappings, to cache reads or to buffer writes. Because of the latter, SSDs usually have a set of capacitors storing enough power to write out the DRAM contents to flash in the event of power loss [8]. To save costs, some NVMe SSDs may utilize a part of system RAM instead, using the host memory buffer HMB feature described in Section 2.3.1 [24].

2.2.1. Flash Translation Layer

The flash translation layer *(FTL)* maps LBAs to individual flash pages, while working around the limitations of the storage medium. Very little is known about the FTL implementations in use in current commercial SSDs [1], but lots of FTL designs have been described in academia. Basic types of FTL include block and page level mapping, as well as log based mapping. Page level mapping uses a single table indexed by the LBA, where each entry contains the physical page address. [8] While this mapping scheme is simple, it uses a lot of space. Block level mapping reduces the granularity of mapping to erase blocks, trading allocation flexibility for space savings. These space savings allow more of the mappings to fit into memory, which is essential for lookup performance. Log based mapping sequentially records all data written, thereby converting random writes into sequential writes, but requires iteration to determine the latest block data [8, 9]. Hybrid mapping implementations

combine multiple strategies, for example using a log to record incoming writes, then later moving the data to block level mapping [8, 23].
All of these schemes still need separate data structures to look up free pages, which are also part of the FTL. It is important to note, that lookups in most FTLs do not take constant time, and the FTL itself needs to be stored on flash as well. Similarly to the TLB in memory controllers, SSD controllers also heavily utilize caching for (parts of) the FTL data structures [9, 23]. As such, the we believe the FTL is responsible for much of the variability in latency of SSD operations.

2.2.2. Wear-leveling

It is almost certain, that some parts of a storage device are going to be written more frequently than others, for example regions containing file system data structures (like ZFS superblocks or the File allocation table on FAT file systems). However, since NAND flash only has limited write endurance and only a small amount of spare capacity is available, these writes need to be distributed evenly across the flash, to avoid destroying certain blocks, while others are still fresh [8].
We differentiate between two kinds of wear-leveling, *dynamic* and *static*. Dynamic wear leveling refers to distributing new data being written to different blocks on flash. Static wear leveling refers to cycling data to different parts of the chip, even if the data is not modified [8].
The latter may seem pointless or even detrimental at first glance, as it leads to additional flash writes, thus introducing additional wear, however it is required for device longevity. Imagine a device, which is filled to 80% by unchanging ("cold") data, while 10% of the blocks are frequently rewritten ("hot"). In that case, dynamic wear leveling can only distribute the hot data on 20% of the device (the unallocated and hot parts), which would eventually lead to the device failing with 80% of its NAND almost unused.

2.2.3. Garbage collection

As data can not be updated in place, device writes, explicit deallocation (sometimes called *trimming*), as well as wear leveling lead to programmed pages which no longer hold any valid *"live"* data, only *"garbage"*. The process of reclaiming these pages as programmable space is known as *garbage collection*. [8] This typically involves first locating erase blocks with no or few live pages, then relocating any live pages to other blocks. Afterward the now free block can be erased, which is followed by marking the block as available for writing in the FTLs data structures [8].

2.2.4. Write Amplification

In order to write a block to flash, potentially multiple blocks need to be written, as the page may need to be assigned in the FTL or other data pages moved to make space. The amount of extra writes needed is characterized by the *Write Amplification Factor (WAF)*. A higher WAF inversely correlates with device performance and longevity. A WAF of 1.0 is the ideal case, where for every block the host writes only one flash page is written [8].

2.3. NVM Express

NVM Express (NVME) is a specification for communicating with non volatile memory *(NVM)* devices. It is primarily used for SSDs, but also has extensions for other storage types, such as rotational media (HDDs). Most commonly, NVMe is used over the *PCI Express* bus (referred to as the *transport*), which is commonly exposed through the M.2 and U.2 connections [8, 13].
NVMe is designed to enable maximum parallelism, by using an asynchronous communication model built on queues residing in shared memory. The host submits commands by placing them into a *submission queue*. After the request is completed, the result is written to the corresponding *completion queue* by the NVMe device [13, 25]. The amount of items in a queue is referred to as the *queue depth*. An NVMe SSD typically has a single *admin queue* used for managing the device (for example creating or formatting namespaces or setting up I/O queues) and multiple *I/O queues* used for submitting interacting with the storage [25].
An NVMe SSD is logically divided into *domains*, *endurance groups*, *sets*, and finally *namespaces*. Each domain has a controller (which is accessed on Linux through the `/dev/nvmeX` device file). Endurance groups and sets are not used by consumer SSDs. Each *namespace* (represented by the `/dev/nvmeXnY` device file on Linux), is an independent block addressable region of storage [13, 26]. Each namespace uses exactly one *I/O command set*, which describes the I/O operations available on it. While others such as the *Key Value* command set do exist, for us the only relevant one is the *NVM* command set used for block storage [13, 27].

2.3.1. Host Memory Buffer

Host Memory Buffer (HMB) is an extended capability of NVMe that allows devices to use part of the host's memory as their own [13]. As accesses to HMB memory require traversing the transport, such memory has much higher latency than local

DRAM. HMB can be used to replace dedicated DRAM on SSDs. Such devices are called DRAM-less SSDs, which trade performance for lower cost (see Section 3.2.1).

2.3.2. Health Information Log

The Health Information Log, commonly known as the *SMART log* (from the *Self-Monitoring, Analysis, and Reporting Technology* introduced for hard drives) is a log information page which can be read through the admin command queue. It contains metrics about the state of the drive, including as the current device temperature, and the number of media errors encountered [13].

2.4. Related Work

The black-box nature of SSDs and their performance characteristics has long concerned researchers and developers alike[1], leading to multiple angles of research being pursued, the most relevant of which are described in the sections below.

2.4.1. The Unwritten Contract of Solid State Drives

The Unwritten Contract of Solid State Drives by Jun He, et al. [9] analyses SSD performance of multiple workloads atop different file system (specifically the databases LevelDB, RocksDB and SQLlite atop EXT4, XFS and F2FS). The paper uses the results to formulate the following contract SSD clients should adhere to in order to maximize SSD performance.

1. Clients should issue either large i/o requests or many smaller concurrent requests, in order to achieve peak performance.
2. Clients should access data with temporal and spacial locality, in order to minimize FTL cache misses. Temporal locality means acessing the same blocks in close sucession (though this should rarely be required), and spacial locality refers to accessing data that is close-by, ideally sequential to the previous data accessed.
3. Clients should aim to write data with high alignment and sequentially, in order to have high sustained performance.
4. Pages with similar death (deallocation or overwrite) times should be grouped together.
5. The lifetime of pages should be equal.

With the first point being by far the most important.
The authors also show, that in practice random writes perform better than expected and log-based data storage does not perform as good as suggested.

2.4.2. Fantastic SSD Internals and How to Learn and Use Them

This paper by Li, et al. [4] introduces *Queenie* a user-mode is a tool to detect SSD internal parameters via probing, capable of detecting 10 attributes: *page size*, *page type* (SLC, MLC, TLC), *chunk size*, *stripe width*, *channel/chip layout*, *read and write buffer capacities*, *write parallelism*, *internal flush window* and, *read performance consistency* (qualitatively) [4]. It does this by performing a number of read/write benchmarks on the SSDs, similar to our methodology. In the paper, 21 different SSDs were tested, the result set can be found under the name *Kelpie*.

2.5. SSD Simulators

Attempts to model SSD performance are not new and numerous SSD simulators, such as MQSim (focused on modelling the parallel nature of multi-queue SSDs) [28], WiscSim [9], and SSDSim [20] have been created, though their simulation accuracy varies wildly and none of them have been validated with many devices [9, 20, 28]. Each of the simulator claimed high simulation accuracy, only to be disproven by the subsequent simulator [9, 20, 28]. This makes them unsuitable to draw performance guidance from, however their code could have served as a source of possible FTL implementations to aid in designing tests.

2.6. FTL Designs

There exist numerous patents, e.g. [29–31], mostly from SSD manufacturers about FTL designs, but it is almost impossible to know if or where these designs are actually in use. The same issue applies to novel FTL designs proposed in papers, such as [32–34]. These published FTL designs could have served as a valuable aid in designing the tests to discriminate SSDs, as it is likely that firmware authors are referencing some of these publications.

2.6.1. Open-channel SSDs

Contrary to traditional SSDs, *open-channel SSDs* expose raw flash operations to the operating system. This allows the FTL and background tasks, like garbage-collection, to be managed in software [17]. The open-channel subsystem in the Linux kernel was called *LightNVM* [17]. This approach allows for much greater transparency into the operation of SSDs, thereby enabling developers to better optimize their applications and allows SSD behavior to be adapted to specific workloads. Open-channel SSDs

have been used to implement new FTL designs [35]. While the openness allows for unparalleled insight into SSD operations, we can not be sure if LightNVM's design bears any resemblance to modern commercially available SSDs. Nevertheless, LightNVM could have been used to validate benchmarks designed to discriminate FTL implementations. Open-channel SSDs did not see commercial adoption, and LightNVM has been removed from the Linux kernel in 2021 [36]. They seem to have been superseded by Zoned Namespaces.

2.7. F2FS

F2FS is a file system designed from the ground up for flash storage. It aims to align filesystem data structures to functional units of the FTL and spread random writes over the LBA space, while maintaining spacial locality. At it's core, F2FS is a log-structured filesystem with multiple logs, which are used to separate hot and cold data [37, 38] and to avoid overwriting the same addresses frequently. F2FS incorporates some key performance guidance for SSDs and claims a more than 2x performance improvement over ext4 in specific workloads [37]. Despite this, F2FS does not enjoy widespread adoption and later results show F2FS only performing on-par or worse than traditional file systems like ext4 and XFS in similar workloads [9, 39]. Analyzing the efficacy of F2FSs in modern SSDs would have served as a good case study for exploring the change in SSD behavior since it's inception in 2012.

3. Experiments

In this chapter, we motivate and describe the experiments performed for this work. We start in subsection 3.1 by explaining our benchmarking setup, then dive into each experiment individually.

3.1. Benchmark Design

To analyze SSD performance, we wish to benchmark the SSDs in a variety of synthetic workloads.
On a high level, each benchmark feeds a list of operations into the SSD and measures the time it takes each one to get completed. Therefore, we need to time the submission and completion of each operation. We also want to periodically collect SMART data (see Section 2.3.2), to measure temperatures and write amplification.
Based on these requirements, we decided on the following high-level architecture for our benchmarks.
Each benchmark uses three threads, the main thread is responsible for feeding I/O requests to the device and polling for completions. A second thread is used to periodically request the *health information page*. Both of these threads send data to a third logging thread. The main thread buffers the precise submission and completion times until either enough time has elapsed or a certain number of measurements have been taken. It then stops issuing new requests, waits until all in-flight requests have been completed, calculates latency statistics which it sends to the logging thread, then resumes issuing requests.
Based on this architecture, we can factor out a lot of common functionality from all benchmarks. Each benchmark first sets up the requested amount of device queues at the specified queue depth, allocates shared buffers used for request data, then starts the auxiliary threads. The main thread then takes measurements until specified, after which it waits for the auxiliary threads to join.
From this, we can specify each benchmark through just three functions.

1. A function to determine the LBA to issue the next command to
2. An optional function that prepares the data buffer, for write benchmarks
3. A function responsible for submitting an I/O command to the LBA from function 1 with the data from function 2.

3.2. Experiment setup

This is a short overview of the setup used to perform the experiments presented in this chapter. The system used to benchmark the drives uses an AMD Ryzen 7 7700 with 32GiB of DDR5 main memory, running Fedora Linux 41 with kernel 6.13.10. All overclocking and power-management features, including EXPO, PBO and ASPM were disabled, in order to have consistent performance.

3.2.1. Devices Under Test

Four SSDs were evaluated, which were chosen to represent a variety of current consumer SSDs. All devices were running their latest firmware, at the time of benchmarking and they have the following specifications:

Manufacturer	Samsung [12]	Kingston [15]	Gigabyte [14]	Intel [40]
Name	SSD 970 EVO Plus 250GB	NV2 PCIe 4.0 NVMe SSD	NVMe SSD 256GB	Memory M10 Series 16GB
Model Number	MZ-V7S250BW	SNV2S250G	GP-GSM2NE 3256GNTD	PHBT722 3024J016D
Recording Technology	TLC V-NAND	unknown V-NAND	unknown NAND	3D X-Point
PCIe link (throughput[2])	3.0 x 4 (32GT/s)	4.0 x 4 (64GT/s)	3.0 x 4 (32GT/s)	3.0 x 2 (16GT/s
capacity[3]	250GB	250GB	256GB	16GB
DRAM cache/ HMB[4]	512MiB DDR4	none/HMB	none/HMB	none

In order to aid the readability of the charts, the manufacturer's names are sometimes used to refer to the specific devices in the following chapters.
We selected low-capacity models primarily to reduce benchmarking time on benchmarks utilizing the entire available space. Additionally, the lower cost of these devices made them even more attractive.
Unfortunately, we only found out after receiving them, that none of the SSDs supported querying the amount of data written to the media. This hindered us from directly measuring write amplification as discussed in subsection 3.1.
Luckily, most of the drives seemed to behave nicely during benchmarking. We only encountered two device quirks. Firstly, the Samsung SSD, which despite reporting *mdta_bytes* as 2097152, which equals a maximum I/O request size of 4096 512B

[2] 8GT/s results in approximately 1GB/s usable bandwidth
[3] 1GB = 10^9 Byte ≈ 0.93 GiB
[4] Host memory buffer, see Section 2.3.1

blocks, writes at this size would fail. We limited our requests to half that, 2028 blocks. Secondly, The Gigabyte SSD once became completely unresponsive during a benchmark and required a hard reboot to work again.

3.3. Experiments Performed

This chapter motivates and describes each of the experiments performed, the results of which can be found in Chapter 5.

3.3.1. Parallelism

On NVMe SSDs, there are two dimensions which control the amount of parallel work submitted to the SSD: the queue depth and the number of queues in use. In this experiment, we seek to determine the degree to which SSD performance scales with increased parallelism. For this, we chose to benchmark the operations which individually take the smallest amount of time possible. We believe this to be reading blocks in sequence, as reads are faster than writes and operating on fewer pages should always be faster than more pages.
We tested all combinations of the following parameters on SSDs which were sequentially filled.

parameter	values
number of queues	1,2,4
queue depth	1,2,4,8,16,64
read size [blocks]	1,2,3,4

In general, we found that drive performance scales with the queue depth, but not the amount of queues used. For results, see Section 5.1.

3.3.2. Locality

This set of benchmarks aims to determine, how SSD performance is affected by the locality of accesses, i.e., if the pattern is sequential or random, and how this scales with different request sizes. For this, we prepared the DUT by sequentially filling them with 0xff (see Section 3.3.6 for why this does not matter). We then read an 8GiB section of each disk sequentially and in a randomly shuffled order, with all supported read sizes that are a power of two.
All of our experiments depending on randomness use a pseudorandom number generator initialized with a fixed seed, to ensure comparability between executions.

This experiment found that for sequential reads, performance scales very well with increasing request size. This was not always the case for random reads, where in some cases, smaller read sizes (16 blocks) outperformed larger ones. Sequential reads also performed more than 10 times better than random reads.
See Section 5.7 for the detailed results.

3.3.3. Permutation of reads

As seen in by the results of the previous section (see Section 5.7), sequential operations perform significantly better than random operations. NVMe SSDs have the ability to reorder operations, as the device operates on command queues. SATA HDDs make use of this feature through **native command queuing**. This benchmark aims to determine, if SSDs reorder commands in a similar manner.
For this, we compare the random read and sequential read speeds to permuted read speeds. For a permuted read, the sorted sequence of LBAs to be accessed is split into small (smaller than the queue depth) subsequences. These subsequences are individually randomly shuffled, but the order of the subsequences is preserved. For example, reading the LBAs 1-9, with a window size of 3, could result in the read order (3,1,2,4,6,5,9,7,8).
If the SSD is able to reorder the operations, the permuted performance should be more similar to the sequential than the random performance.
Reads were performed with read size 1, shuffle window sizes 2, 4, and 8, with double the window size used as the queue depth. We found that one device performed significantly better with this than random reads. The results can be found in Section 5.8.

3.3.4. Alignment

In CPUs, unaligned memory accesses frequently come with severe performance penalties [5, 6]. We were curious, if unaligned reads from SSDs also pose performance issues. Once again, we performed sequential reads (on sequentially written SSDs), with maximum write size and queue depth, but this time offsetting all reads by powers of two up to half of the request size. For results, see Section 5.5.

3.3.5. Random Fill Performance

Up until now, all writes were performed sequentially. In this experiment, we seek to find out, how the SSDs perform under a worst-case load. We filled the SSDs one block at a time in a random order.

This resulted in some devices writing less than 1MiB/s, with the benchmark taking multiple days to complete. The details can be found in Section 5.9.

3.3.6. Data Dependence of Writes

Since our SSDs return zero-filled pages for never-allocated/empty blocks, we asked ourselves if there is an optimization in place for writing all-zero pages. More generally, we wanted to see if there was any correlation between the contents of the data being written, with the write duration. For this, we performed a sequential write test with maximum queue depth 64 on trimmed disks, with the following patterns: entire pages filled with 0x00, 0xff, and 0xaa (alternating bits), pages with the first 8 bytes being the LBA, the rest being 0xff. The NVMe *write zeroes* command was also tested, if it is faster at writing zeroes both on a full and an empty device.
We found that the data written did not have any influence on the write performance, and using *write zeroes* yielded high and predictable performance on most devices.
The results of this experiment are presented in Section 5.10.

3.4. Reset behavior

SSD controllers can be reset through issuing a reset command [13], however it is unclear from the specification what, if any of the device caches will be cleared by this reset command. If resetting cleared all caches, it would make obtaining consistent benchmark results easier. To test what this command does, three identical sequences of random reads are issued to the device, and the total time taken to perform all is measured. The first set serves to populate any device caches with these LBAs. After the second and before the third run, `xnvme_controller_reset` gets called.
No differences were observed between the second and third run, however performing this test directly after removing power from the computer yielded slower results. This experiment is discussed in Section 5.6.

3.5. Experiments planned but not performed

We started to implement, but did not manage to in time, perform experiments examining the following:

1. The effect of grouping I/O operations by type versus interleaving of reads and writes
2. Discarding *(deallocating)* a block before overwriting it

3. The differences between secure erasing, formatting, and trimming an entire namespace.
4. The order in which the SSDs prefer to complete work on multiple queues
5. The performance of stratified reads and writes

4. Implementation

This chapter discusses design decisions made and issues encountered during the creation of the benchmarking system.

4.1. Implementation language

We chose C++ to implement the benchmarking suite in for four main reasons. Firstly, the language needed to not have any garbage collector or other runtime, as these can introduce hard to predict interruptions to our control flow, which would be detrimental to accurate timing. This left four choices which we were familiar with: C, C++, Rust, and Zig.
Secondly, we did not want to implement basic data structures (vectors, maps, etc.) ourselves, which eliminated C as a language choice. Thirdly, the language needed to make interoperating with C libraries simple, as xNVMe and SPDK are written in C. One of Zig's primary features is its great C interoperability. For Rust, there do exist low-level (unsafe) bindings for both libraries (the `xnvme-sys` and `spdk-rs` crates), however because of their unsafe nature, these bindings negate many of the advantages the Rust language provides. Creating proper bindings would have been a significant project in itself, which is why Rust was eliminated as a candidate.
Finally, C++ was chosen over Zig, because we have much more experience writing software in it.

4.2. Interfacing with NVMe SSDs

There are multiple ways of interacting with the NVMe drives on a modern Linux system. This chapter introduces them and explains the rationale for ultimately choosing xNVMe with the SPDK backend to implement our benchmarks in.

4.2.1. Linux NVMe Subsystem

Communicating with devices is part the core functionality of operating systems. Our system of choice, Linux, primarily provides a block device interface to access storage devices, through the `/dev` file system, where a file for each namespace of each device is created at `/dev/nvme<drive number>n<namespace number>`. This interface provides lots of features needed in a modern system, most importantly I/O scheduling and block level caching [26].
This, however, leads to the block device interface being too high level for our benchmarking needs. We need to directly issue commands to the drives. Issuing commands can be accomplished through the Linux kernel, which also provides this ability through the `ioctl` system call on the same files. The consumer of this interface cannot control in what NVMe queue the request is placed in, nor does the user have the ability to control the number or depth of queues used. Linux always uses[5] the maximum amount of queues, at their maximum queue depth [41]. Existing benchmarking tools utilizing this interface limit the amount of work they submit to the kernel in order to benchmark writes at specific queue depths [42].
Another downside of using ioctls each request requires a system call to be issued, which incurs a high overhead. This can be mitigated by the `io_uring` interface, which allows a program to submit I/O requests to the kernel, including the aforementioned ioctl, by writing them to a circular buffer. The kernel then completes these requests asynchronously and writes the results into another circular buffer, thereby avoiding the user-kernel transition responsible for the system call overhead [26].
We considered writing a kernel module for benchmarking, which would then have had the ability to directly communicate with the SSDs or call into functions only exposed as kernel symbols, such as enqueuing requests on a specific queue. Writing kernel-mode code could also avoid some timing difficulties discussed in subsection 4.3, as it would have direct access to timing information and not be interrupted by the scheduler. However, writing a kernel module is much harder. By limiting us to writing C or Rust, both without the standard library and a bug can take down the entire system, possibly requiring a hard reboot, hindering remote development.

4.2.2. SPDK

It is possible to bypass the kernel's NVMe subsystem entirely, by using the **Storage Performance Development Toolkit** (SPDK) [25]. SPDK describes itself "a development kit to build high performance storage applications" and is a C library consisting of many optional components, including NVMe driver, iSCSI drivers, a

[5] unless a specific device quirk `NVME_QUIRK_QDEPTH_ONE` is enumerated

blob storage implementation, and much more. For our purposes, the only relevant part of SPDK is the NVMe driver. SPDK uses `vfio-pci`, a Linux driver ment for passing through PCIe devices to virtual machines, to expose the PCIe interface of NVMe drives directly to our user-mode program. The SPDK NVMe driver then implements the NVMe protocol, providing an API to interface with NVMe controllers, such as creating queues, and submitting commands through them. [25] Using SPDK allows for full control of the NVMe device, while removing the complexity associated with implementing the NVMe protocol or writing kernel-mode code.
Because of this, SPDK was at first chosen as the interface for use in our benchmark suite. Using it in a C++ project should have been simple, however installing required source-based compilation and manual installation of system dependencies, which proved difficult to automate. Linking against SPDK is also complicated enough to warrant it's own documentation page [43]. During initial implementation the build system was responsible for multiple hard to debug bugs (due to out-of-date objects not being rebuilt), and making the `ccls` language server correctly find includes was difficult. These frustrations led us to explore other possibilities, including xNVMe.

4.2.3. xNVMe

`xNVMe` [2] is a cross-platform library to interface with NVMe devices in user space, providing a single API (similar to SPDK's) through several different back-ends for different operating systems. Importantly for us, on Linux it uses either `io_uring` or the SPDK NVMe driver as a back-end. After validating that using SPDK through xNVMe does not introduce significant overhead compared to using SPDK directly (see Section 4.4.2), the implementation was ported to xNVMe. This enabled using a simpler build system, and the ability to compare performance of the Linux back-ends with the same benchmarks.

4.3. Accurate Timing

As described in subsection 3.1, our benchmarks require the measurement of operations that take on the order of microseconds (e.g. A single 512B read has an average latency of about 14 us on the Intel Optane SSD, see section 3). There are two major issues with accurate measurements in user space, the kernel interrupting our measuring process, and getting accurate timing information.

4.3.1. Minimizing Interrupts

In an ideal world, our measurement thread would run on a dedicated CPU core, which would not be interrupted during the benchmark. Unfortunately, Linux is not a real-time operating system that can truly facilitate this, but by using the techniques outlined below, we could minimize the interruptions to one approximately every second.
Firstly, using the `sched_setaffinity` system call, we can specify our benchmarking processes' CPU affinity mask, which is the set of logical CPUs the process can be scheduled on. Using this also leads to other processes (which are not using the same affinity mask) to not be scheduled on those threads [26].
Secondly, we can disable servicing most interrupts on the benchmarking CPU by specifying the interrupt's CPU affinity mask in the `proc` file system to not include the CPUs the benchmarking thread is using. This is done writing the affinity mask to `/proc/irq/*/smp_affinity`, in our case "ff55" to reserve threads 1,3,5,7 for benchmarking [26]. This choice of threads is such, that no two are hyperthreads of the same physical core. Beyond that, this choice arbitrary, as our CPU only has one core complex die (CCD), so all cores should perform the same.
These three things eliminate most interrupts, but the scheduler still interrupts periodically, even if there is only one runnable task, our benchmarking process, on the CPU. To change this, the kernel must be compiled with `CONFIG_NO_HZ_FULL=y` and this feature explicitly enabled by setting the `nohz_full=` kernel command line parameter to the CPUs we want to receive fewer scheduling interrupts 1,3,5,7) [26]. This leaves about 1 interrupt/second required by the kernel to calculate some metadata, which cannot be disabled. The measurements affected by these need to be filtered out later, during data analysis.
Finally, we need to ensure that the benchmarking process itself behaves predictably and does not issue system calls during the measurement process. For this, it is essential that no memory allocation is performed while the benchmark is running. Contrary to some other languages[6], our implementation language C++, specifically data structures in the standard library, can perform hidden memory allocations, such as when inserting into a `std::vector`. To avoid this, we pre-allocated the maximum capacity needed for all data structures (by, for example, using `std::vector::reserve`) before starting any measurement.
We verified that our benchmarking tool does not perform system calls or memory allocations during the measurement loop by using GDB to set breakpoints on `malloc()`[7] and on any system call.

[6] like zig [44]
[7] `operator ::new()` calling `malloc()` is implementation defined behavior on gnu C++

4.3.2. Accurate Timing Information

Obtaining high accuracy data timing data is usually done by using the `rdtscp` (Read Time Stamp Counter and Processor id) instruction to read out a timestamp [7]. This timestamp used to be a counter incremented every clock cycle, but this is no longer the case. Modern CPUs dynamically change the frequencies of their CPU cores, which is known as "boosting". This behavior would make comparing timestamps across CPU cores impossible, so newer CPUs have an **invariant TSC**, which will increment in a fixed interval on all CPUs, though this interval itself can only be determined at runtime. The determination itself requires reading two **model-specific registers** (MSRs) `APERF`, `MPERF` , and using them to calculate the actual frequency of the CPU [7]. This can only be done in kernel mode and the Linux kernel does implement this, but does not expose the calibration data to user space. Luckily, Linux provides the `clock_gettime(CLOCK_MONOTONIC_RAW)` clock source, which uses this calibration internally [45]. Even though `clock_gettime` is implemented in the virtual Dynamic Shared Object (`vDSO`), a shared object loaded into every process for implementing system call like functionality without a system call, calling `clock_gettime` still takes too long for our measurement loop.
Our solution to accurate and low overhead timing was to perform our own calibration based on `clock_gettime` before every benchmark. For this, `rdtscp` and `clock_gettime` were called immediately before and after a spin-loop, from which the real-world duration of a TSC increment d_{tsc} was calculated as follows: $d_{\text{tsc}} = \frac{\Delta_{\text{tsc}}}{\Delta_{\text{clock\_gettime}}}$. On our system this turned out to equal approximately $\frac{1}{3.8}$ ns, meaning our CPUs invariant TSC runs at about 3.8 GHz. The benchmarks measure time by saving timestamps immediately after submission and after completion, which are later converted into wall-time.

4.4. Validation

In order to have confidence in the benchmark results, the benchmarking implementation and setup needs be checked against known data points, and against foreseeable issues.

4.4.1. Usage of Identical Buffers

In order to simplify read benchmarks, thereby reducing benchmark overhead, having the ability to issue multiple concurrent requests to the same destination buffer would

be preferable to managing buffers for each request.[8] In theory, parallel writes to the same destination buffer could be optimized away by the SSD controller, by completing all requests simultaneously while only performing one. If this were the case, using identical buffers would result in a significant speed up. It is also possible that writing to the same memory through is slower because of contention in some part of the system.
To test if sharing a destination buffer has an effect on SSD behavior, we compared sequential reads with small read sizes ($1 * 512B$ blocks) either all to the same buffer, or to dedicated buffers for each item in the queue (with depth 8). This benchmark was chosen, because of these are the fastest requests for an SSD to complete while still transferring data to the destination buffer.

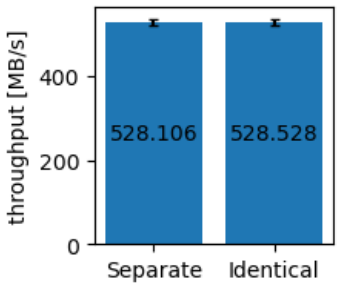

Figure 4.1: Average (n=100), throughput while reading 10000 512b blocks from the Intel Optane SSD. Error bars indicate 1%-99% interval

No significant performance difference was observed on any of the DUT in this test. We can therefore use one shared buffer in read benchmarks.

4.4.2. xNVMe overhead

Check if the overhead of the xNVMe SPDK back-end over native SPDK involved porting the sequential read benchmark from SPDK over to xNVMe. This benchmark specifically was selected again for the same reason as in Section 4.4.1; Sequential reads should be one of the fastest operations SSDs to perform, leading to the interfaces having the highest possible contribution to the runtime. Then the total time for reading 10 GiB of unallocated sectors was performed at queue depth 1 with both implementations. These parameters minimize the time for the SSD to complete each request and maximize the number of operations the interface library needs to perform.

[8] Our benchmarks do not validate the data returned by reads, as this would be difficult to implement without high overhead. However none of our devices have a quirk specified in the Linux NVMe driver, so it is reasonable for us to believe they return data correctly [41].

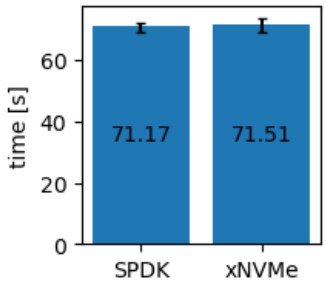

Figure 4.2: Average (n=1000) total time of reading 10GiB of data from the Intel Optane SSD at queue depth 1 and read size of 1 block. Error bars indicate 1%-99% interval.

On average, a relative performance difference of xNVMe being 0.5% slower was observed over 1000 iterations, which we deemed acceptable.

4.4.3. Performance

To rule out unexpected performance problems with our implementation, we compared our sequential read performance benchmarks with both the device specifications and the commonly-used [17, 25, 35] benchmark tool **fio** [42].

We selected sequential reads because they are one of the few metrics where both benchmark tools and specifications for all drives under test exist[9]. For all drives, we chose a "best case", by using the maximum read size, maximum queue depth supported by each device, and with a sequentially written drive.[10] The test were executed for approximately 60s each.

Device	Specification [MB/s][11]	fio io_uring [MB/s]	ours [MB/s]
Samsung	3500	3594	3560
Kingston	3000	2281	2304
Gigabyte	1700	1508	1701[12]
Intel	900	929	926

Table 4.2: Sequential read performance under ideal conditions

This test showed that all drives could perform close to their claimed performance, and our software performs similarly to fio. Based on this test, we conclude that our benchmarking tool is fit for purpose.

[9] Notably, the Kingston [15] drive does not specify any IOPS figure

[10] Using fully empty (trimmed) drives resulted in far higher read speeds

[11] 1MB = 10^6Bytes

[12] This SSD slows down significantly after about 15s, which is why this test was shortened to about 10s. See subsection 5.3

4.4.4. Concurrent benchmark execution

Running benchmarks on all four SSDs in the system simultaneously could reduce the duration of all benchmarks by significantly, and is therefore highly desirable. To validate that concurrent benchmarking does not affect the results, again our sequential read benchmark, using the same parameters as in Section 4.4.3 was performed both sequentially and concurrently on all SSDs.
No statistically significant difference in latency or throughput were observed. Because of this, we concluded that we may run benchmarks concurrently, and have done so for all benchmarks presented in this work.

4.4.5. Thermal

For our benchmarks to be reproducible, they need to be independent of the prior state of the system. As such, we need to ensure the SSDs are not negatively impacted by high temperatures and are reducing their performance (this behavior is known as thermal throttling). For this, we periodically recorded the data from the SSDs internal temperature sensor during all our benchmarks. This data is provided by the SSD through the health log page (Section 2.3.2).

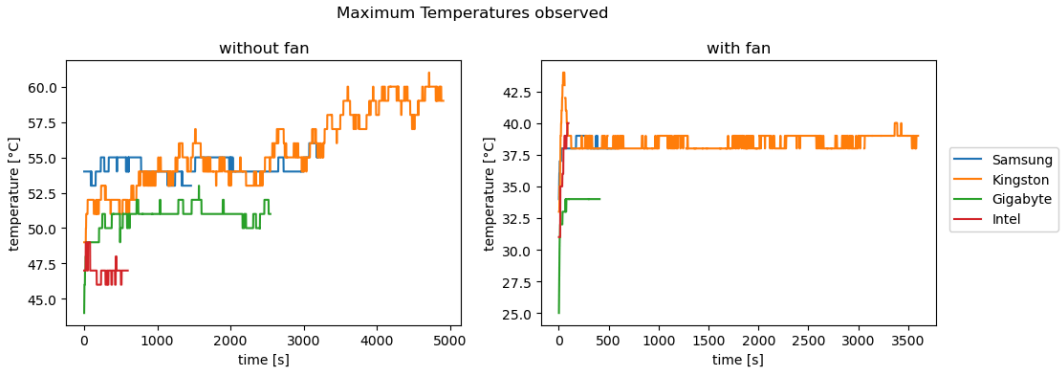

Figure 4.3: Maximum temperatures observed during any benchmark with and without the fan.

The maximum recorded temperature during our initial benchmarks reached 61°C in a 1.2h run on the Kingston SSD, which is below the warning and critical temperatures reported by the SMART log. This should mean that no thermal throttling occurred. Despite this, we elected to install a 140mm fan blowing directly on the SSDs. This further reduced the maximum temperatures to a maximum temperature observed by 19°C, to a maximum of 43°C.

5. Evaluation

This chapter presents the results of the experiments described in Chapter 3.

5.1. Parallelism

This section presents the result of the parallelism experiment (Section 3.3.1) performed on sequentially filled SSDs. Due to the high number of permutations ($3 * 4 * 6 = 72$), only one iteration of each parameter combination was performed, however each iteration averages measurements from reading the entire SSD.
In general, we notice that throughput improves on all devices with higher queue depths, so all SSDs are able to fulfill requests concurrently. However each device shows unique characteristics. The Samsung SSD prefers 512B (it's native block size) over larger reads[13]. As expected, the DRAM-less drives the Kingston and Gigabyte scale well with increasing queue depths.
The Intel Optane SSD reaches its maximum performance already at queue-depth 4, and outperforms all other devices, despite having the lowest maximum read performance (see Section 3.2.1). It is the only device that even approaches its maximum speed at small read sizes.
Interestingly, the Gigabyte and Samsung perform slightly worse with 4 queues and maximum queue depth. While this could possibly be a result of our single-threaded I/O submission, we should have seen this effect more pronounced in the higher-performing Intel SSD, but there we did not see it at all.
In summary, SSDs typically perform better at higher queue-depths, however a single queue suffices to achieve maximum performance. This is good news, as when not using SPDK, the operating system decides how many queues are used and which I/O request is placed on which queue. The reason for having multiple queues is that multiple CPUs using one queue each (the way Linux divides up queues [41]) can utilize an SSD without synchronization. As this benchmark however showed, using multiple CPUs is not necessary to achieve maximum performance on these consumer SSDs. This finding matches rule 1 of the *unwritten contract* (Section 2.4.1).

[13] As we found this very surprising, we re-tested this separately and got the same results.

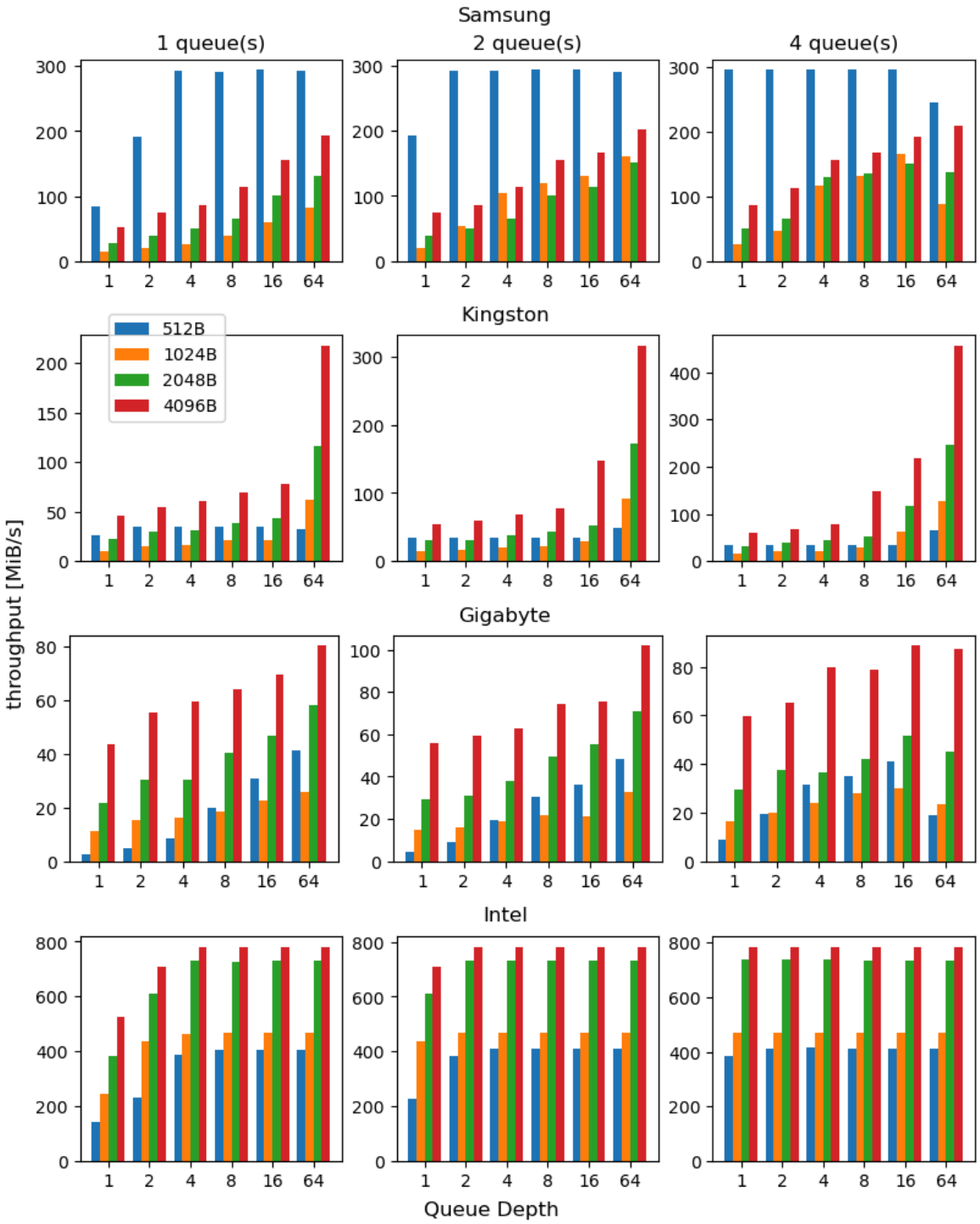

Figure 5.4: Sequential read throughput of each DUT broken down by the number of queues, queue depth and read size. The device was patterned with sequential writes.

5.2. Effects of Write Pattern

The data in subsection 5.1 was gathered with the SSDs sequentially written. Repeating this experiment on drives filled randomly yields the following results: While Samsung's and Intel's performance remains unaffected, Kingston's and Gigabyte's performance significantly deteriorates. From this, we can conclude that the FTLs of these two groups significantly differ. The former group has an FTL which seems unaffected by the order the drives were written to, like page or block mapping, whereas the latter group utilize FTLs which are sensitive to the order they are written.
From this, we make the general recommendation that SSDs should be written sequentially to improve sustained throughput afterward.

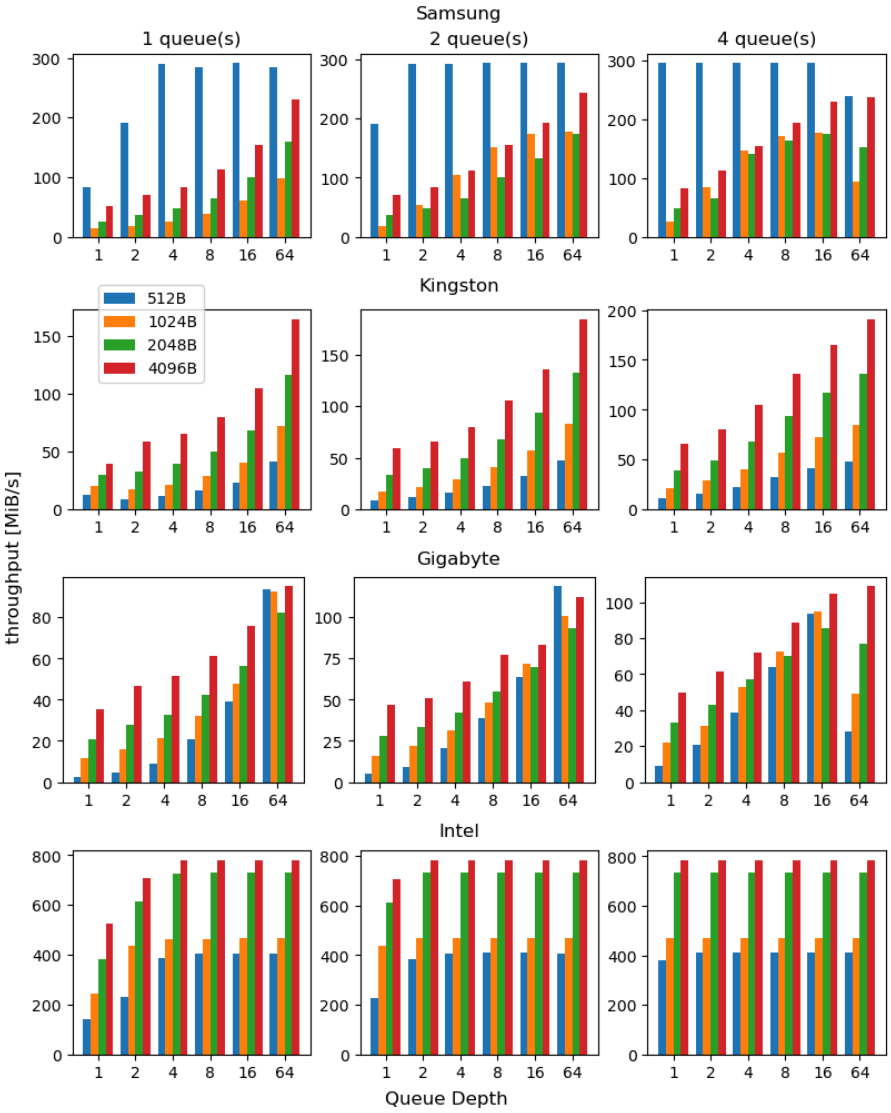

Figure 5.5: Sequential read performance of each DUT broken down by the number of queues, queue depth and read size. The device was patterned with random writes.

5.3. Sequential Read Performance

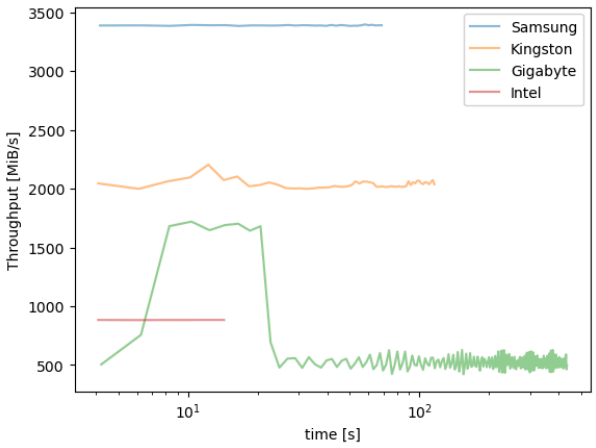

Figure 5.6: Sequential read speed over time, for a sequentially written SSD

This experiment, which originally was just part of the validation, measures the maximum performance while sequentially reading the SSDs. The SSDs were once again patterned by the sequential write, with the maximum write size supported by each device.

We assumed that read performance should be consistent across time, however this is not the case for the Gigabyte SSD, which is not capable of sustaining its read speed in this best-case scenario. We have no explanation for this. The behavior is repeatable, and not caused by thermal throttling, as the device temperature maxed out at 36°C during the test.

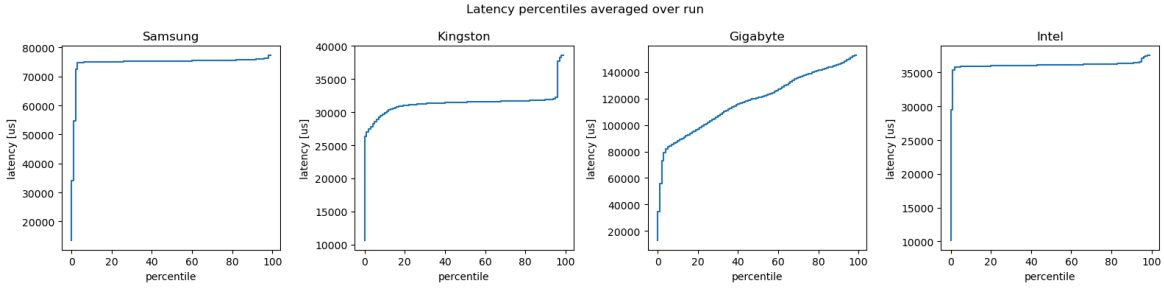

Figure 5.7: Averaged distribution of request latency by percentile

During the above test, the latency distribution of the requests shows that Samsung and Intel perform consistently, with the vast majority of requests ($\approx [5, 95]$th percentiles) taking approximately the same amount of time. The Kingston shows a gradual increase in latency from the 5th to the 20th percentile, while the Gigabyte's latency increases mostly linearly. Especially the last one correlates with essentially random accesses to the underlying flash. Combined with the result from subsection 5.2, we believe that the Gigabyte SSD does not store data written sequentially in continuous regions of flash. Figure 8 shows certain latency percentiles over the course of this test. Once again, we can see that Samsung and Intel perform consistently.

The 5th and 25th percentiles of the Kingston fluctuate, which mirrors the gradual increase by percentile we see in Figure 5.7.

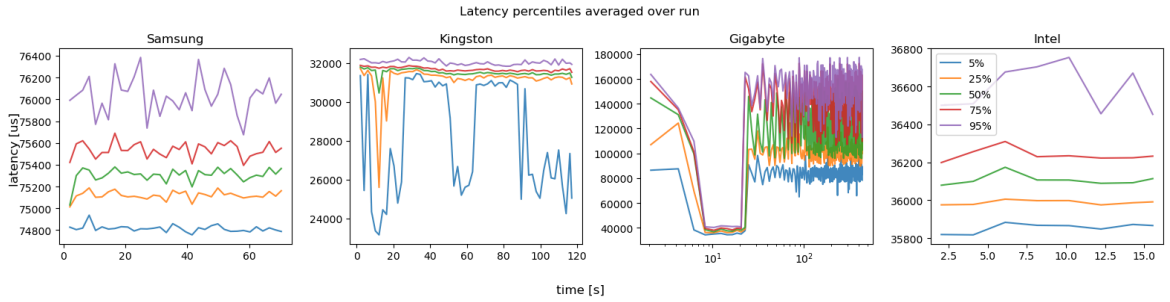

Figure 5.8: Request latency by percentile over time

The only SSD with inconsistent behavior over time is the Gigabyte SSD, which requires about 7 seconds to reach maximum performance and only holds this for a short time. This behavior can also be seen in Figure 5.6 and Figure 5.9.

5.4. Sequential Write Performance

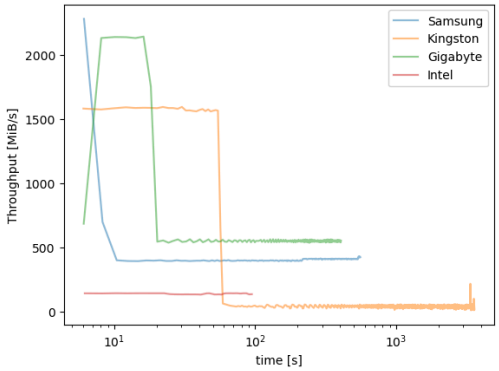

Figure 5.9: Sequential write performance over time, during a

The benchmark shown in Figure 5.9 was performed in a best-case scenario, with the data being written at maximum queue depth, maximum request size onto a fresh SSDs. Again, each device behaves differently. While the Intel Optane drive maintained low but constant performance (matching the datasheet [40]), all others started out writing quickly (meeting their claimed performance targets, see Section 3.2.1), and deteriorated after some time. This falloff in performance occurred at 10s, 14.7GiB written for Samsung, after 56s and 42.5GiB written for Kingston, and after 20s and 14.5GiB written for Gigabyte. We believe this happened after the SLC cache used has been exhausted. If that is the case, then the SLC cache size is approximately 5%, 15%, and 5% for Samsung, Kingston and Gigabyte respectively.

Based on this, we recommend that *SSD clients* expect write performance to drop after writing only 5% of the drive's capacity at once.

It is possible, that the Samsung SSD achieved higher performance in the very beginning, as the 512MiB of DRAM, if used for write caching, filled up before the first measurement was taken (t=2s).

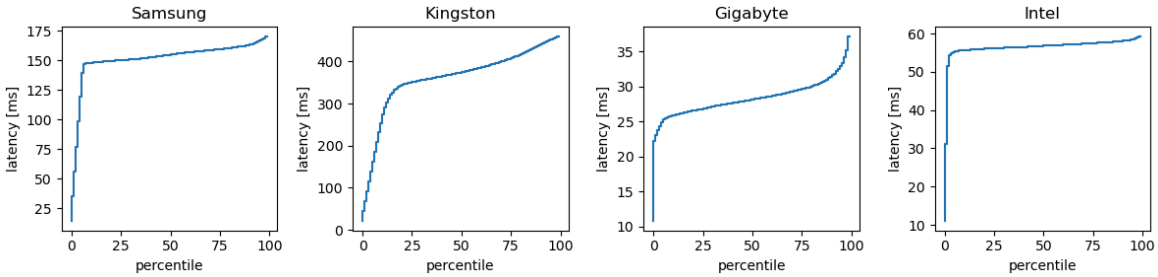

Figure 5.10: Averaged latency percentiles over run

During the sequential write, we note a mostly linear latency percentile curve on all devices, with once again, the Intel device being the flattest.

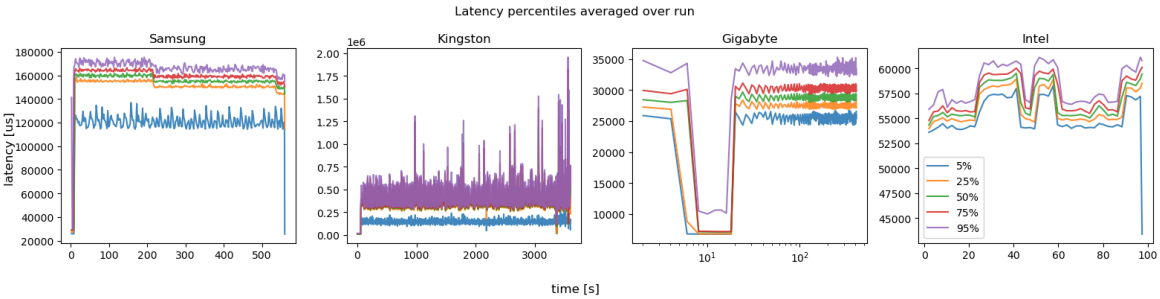

Figure 5.11: Latency percentiles over time

This also translates into the latency development over time, as the latencies of the Samsung, Gigabyte and Kingston drives remained consistent. Interestingly, the 25th and higher percentiles of the Samsung SSD improve slightly (along with the throughput) at approximately 200s into the test. We are not sure what causes this. The Kingston drive has the least predictable latency, with especially the 95th percentile varying by over four times, peaking at almost two seconds. We can not conceive of any reason for such latency for writing $64 * 512B = 32$KiB of data, other than needing to move (and thereby erase) multiple blocks, during one request. This is also indicative of this SSD not storing our sequentially written data sequentially on disk.

5.5. Read Alignment

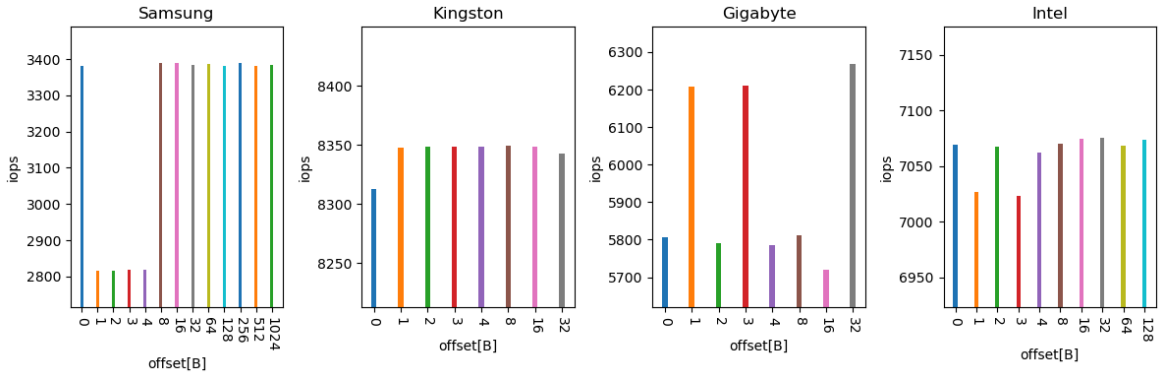

Figure 5.12: Averaged I/O operations per second (higher is better) while reading at the maximum supported read size at offsets

In this benchmark, we evaluated the effect of alignment on read performance. Surprisingly, unaligned accesses performed better on the Gigabyte SSD than aligned accesses. For all others, either alignment made almost no difference, or in the case of the Samsung SSD offsets not a multiple of 8 lead to a significant ($\approx 15\%$) performance drop. This likely means that some part of the SSD uses a $8 * 512B = 4\text{KiB}$ block size, despite not exposing 4k sectors over NVMe.

5.6. Reset behavior

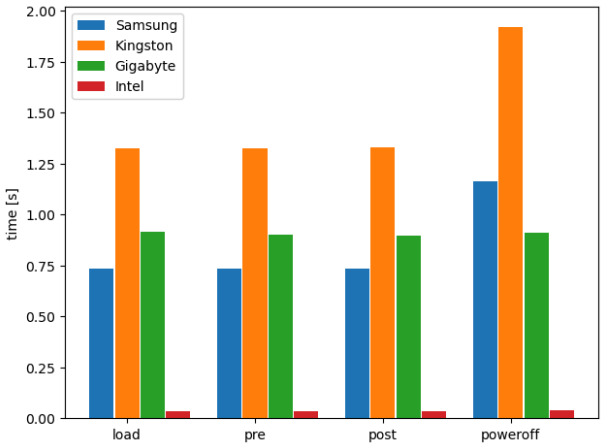

Figure 5.13: Total benchmark runtime for loading (baseline), pre-reset, post-reset and the loading after power has been removed from the device

The experiment described in Section 3.4, showed that the reset command does not have any effect on benchmark runtime. The total time to read all blocks was within margin of error of the baseline (load) on every iteration.

The Samsung and Kingston devices, however, perform slightly worse on the very first load (labeled *poweroff* in Figure 5.13) after each power on.
There are two possible interpretations of this data. We can either conclude, that a controller reset does not clear any FTL/read caches, only removing power from the device does. This calls into question the usefulness of the reset command. Alternatively, no relevant caching is performed on the SSDs, but performing the very first operations after power-on takes additional time, even if the SSD controller presents itself as ready.
Neither of these options seem particularly convincing, but based on both the overall performance of the Samsung drive, the large discrepancy between sequential and random performance (see subsection 5.3 and subsection 5.7) and the fact that it has a DRAM chip, it seems more likely to us that it does perform caching, which is simply not cleared by the reset command.

5.7. Locality

This experiment sought to characterize the effects of locality of consecutive accesses. For this, the throughput of random and sequential reads were compared. Each data point is derived from reading 8 GiB worth of data, with all SSDs using their preferred sector size.

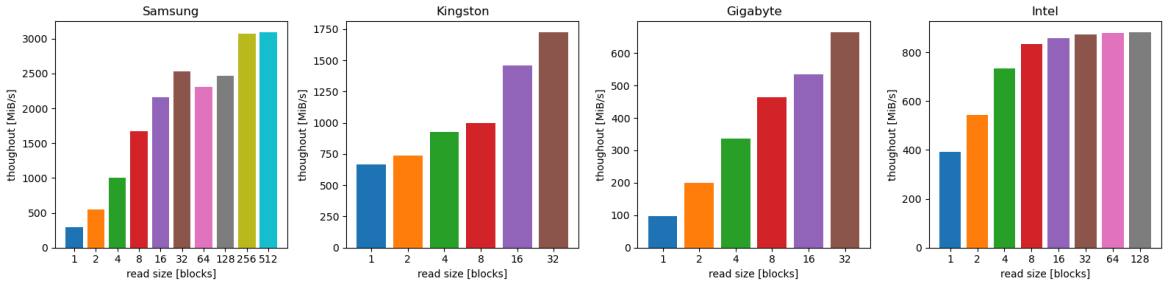

Figure 5.14: Sequential read throughput at different request sizes

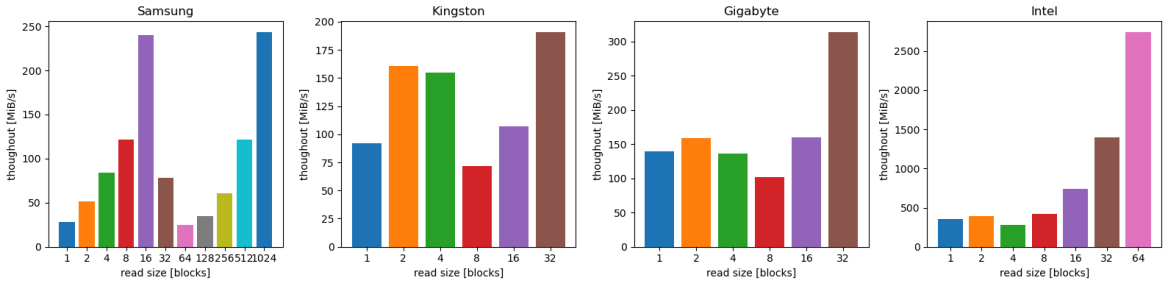

Figure 5.15: Random read throughput at different request sizes

While the other measurements seem plausible and consistent with other benchmarks as well as the data sheet, we can not explain the random read results for the Intel SSD. The values for read sizes 32 and 64 are far higher than seen in any other

benchmark, and are multiples of the maximum claimed in the datasheet. We wiped and re-filled the SSD thrice with different data, yet got the same results.
This anomaly reduces our confidence in the following statements somewhat, however we still believe them to be generally accurate.
Sequential read performance scales with request size, as such, *SSD clients* should use the largest possible request sizes when reading data sequentially. Contrary to this, random read performance does not always scale with increasing request size, which is why we can not recommend a specific best size. As such, SSD clients should perform tests on their specific device to determine optimal request size for random access. In general, random accesses should be avoided if at all possible, since sequential access can be over 10 times faster in some cases.

5.8. Permuted Reads

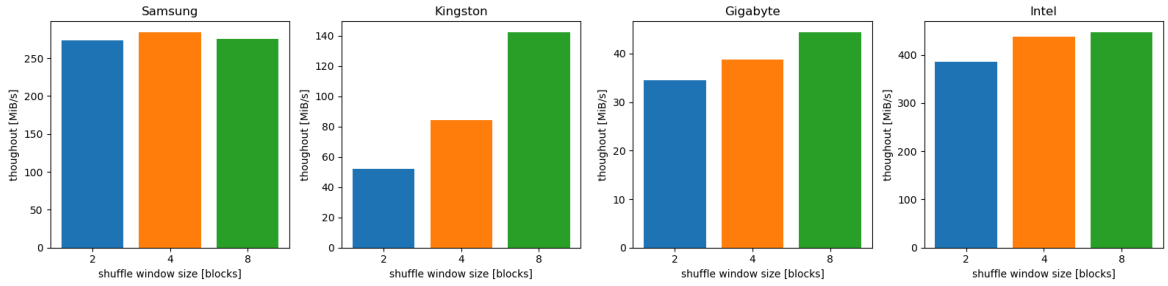

Figure 5.16: Read throughput of permuted reads by window size

Comparing the data with the data in subsection 5.7, we see that the Samsung SSD performs much closer to its sequential read performance than to the random read performance (looking at read size=1), while Kingston seems to slightly outperform random reads at window size 8. Intel seems to perform very similarly in these cases, while Gigabyte seems to perform worse in this test than with random reads. We refrain from making direct numerical comparisons, because those tests used a higher queue depth of 64. It is however clear, that at least some SSDs can take advantage of such permuted requests. This also means that these SSDs perform at least some operations truly in parallel.

5.9. Random Fill Performance

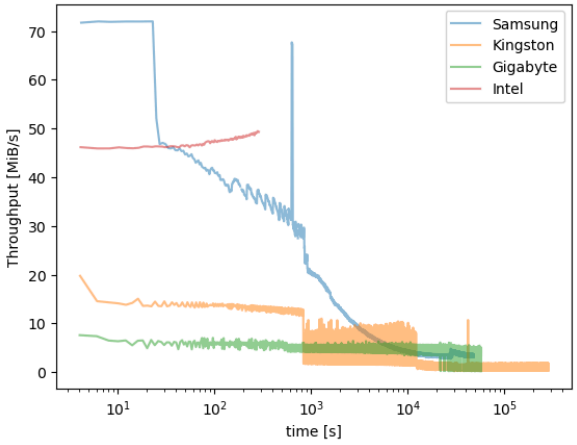

Figure 5.17: Random order fill write throughput over time

This benchmark aimed to capture a worst-case scenario for the SSDs, by filling them in a random order one block at a time. Unsurprisingly, random writes are slower than sequential writes, however the scale of the slowdown is extreme. The Kingston drive took 79.2 hours to complete this test, averaging 876KB/s. While the performance of the Samsung drive slowly decayed, the Kingston and Gigabyte drives performed badly from the start. Only the Intel Optane SSD could provide consistent performance, albeit only at almost $\frac{1}{3}$ of it's rated write speed. We suspect it would have performed much better with a queue depth of 4, as it is likely that with its low latency it spent a significant time waiting for new work.

We believe the bad performance of the NAND flash based drives can directly be attributed to the need for rewriting erase blocks. Figure 5.18 shows the proportion of requests that take longer than 1ms over the course of the benchmark. Since programming times should be an order of magnitude lower than that, we believe this graph shows the percentage of requests that result in a block needing to be erased.

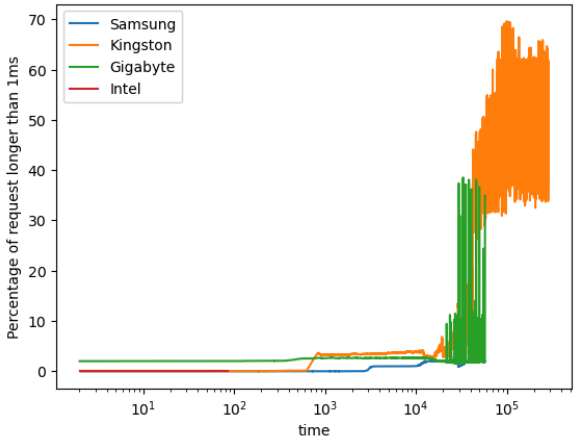

Figure 5.18: Percentage of I/O requests taking more than 1ms over time, 100 measurement rolling average

5.10. Data Dependance of Writes

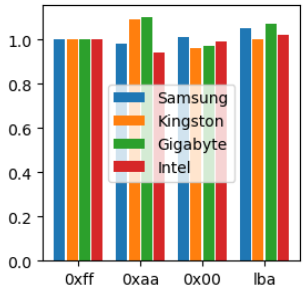

Figure 5.19: Relative total write duration of different data patterns

All drives were erased then rewritten four times, with the different data patterns. No performance difference was observed between any of the runs. From this, we conclude that none of the SSDs have optimizations based on the data being written.

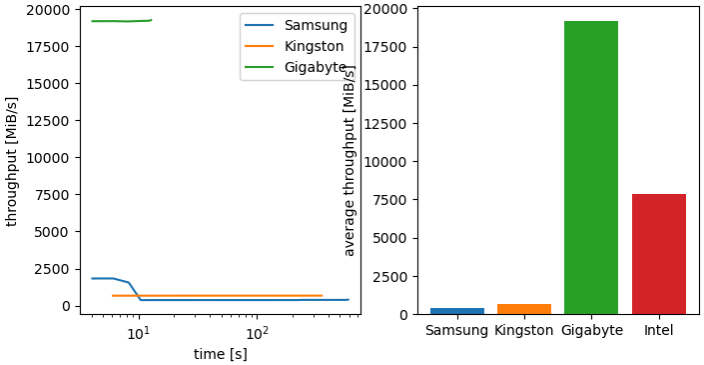

Figure 5.20: Write Zeroes at queue depth 64 on an empty drive

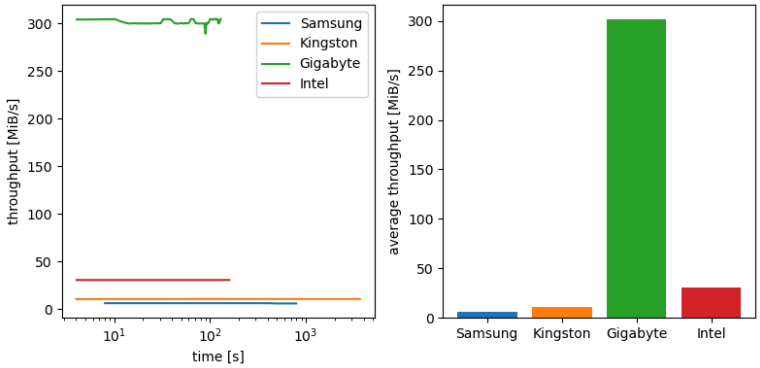

Figure 5.21: Write Zeroes at queue depth 1 on a full drive

Using the write zeroes interface performed very consistently and much better than regular writes, both on full and on empty drives.

Therefore, *SSD clients* should utilize the *write zeroes* interface where available.

6. Conclusion

The purpose of our thesis was to improve our understanding of how SSDs behave and to validate if existing performance guidance holds up today.
Our initial plan was to conduct benchmarks, which would allow us to draw conclusions about the inner workings of SSD FTLs, hence the title of this work. Aside from being able to determine the write cache size (see subsection 5.4), we did not succeed in this regard.
We did however successfully build and validate software capable of benchmarking SSDs in detail not available by other tools like fio [42].
Due to time constraints, we did not build all benchmarks we set out to make, and the benchmarks we performed were sometimes only run at smaller sample sizes then we had hoped. Our findings from the benchmarks we did analyze do not clash with the existing guidance from [4, 9], but we were not able to test each aspect mentioned in those previous works.

6.1. Findings

To summarize our findings from the pervious chapter:

1. SSDs scale well with increasing queue depths, although multiple queues are not required to achieve maximum performance. Therefore, *SSD clients should submit as much work as possible simultaneously.*
2. While sequential workloads benefit greatly from larger I/O requests, highly random workloads do not. Which is why, *SSD clients should conduct testing on the device used before deciding on the correct I/O size for random workloads.*
3. SSDs may significantly slow down during prolonged writes, which is why *SSD clients should expect write performance to drop after writing only 5% of the drive's capacity at once.*
4. In order to sustain good read performance, *SSD clients should write data sequentially.*
2. The Intel Optane outperforms other devices in workloads involving lower queue depths or random accesses. It is also the most consistently performing device tested.

6.2. Future work

Due to time constraints, not all data collected in this thesis has been fully analyzed, which would form the beginning of a follow-up work.
In particular, the anomalous performance of the Intel SSD in subsection 5.7 and the as well as the cause for the power-cycle behavior shown in subsection 5.6 in should be investigated.
The next steps directly related to the benchmark suite described in this work, could involve performing these benchmarks on more devices, to generate a corpus similar to *Kelpie* [4]. Comparing our low-end consumer devices to bleeding-edge SSDs could enable the generalization of guidance applicable to all SSDs, which is important for software (such as embedded databases) designed to scale well.
By using SSDs that enable the direct measurement of write amplification (see Section 3.2.1), we believe more insight could be gained into the behavior of SSDs under write workloads. This measurement could also allow the detection of background processes (wear-leveling), which could be used to determine the time an SSD needs to "recover" after sustained use.
We originally planned to monitor performance degradation over time, which we did not manage to implement. This would also be interesting to test.
Lastly, further work should also include evaluating SSD using the *zoned namespaces (ZNS)* and the *flexible data placement (FDP)* APIs [13, 27], as these seem to be gaining industry adoption and behave vastly differently to conventional SSDs.

Bibliography

1. Zuck, A., Gühring, P., Zhang, T., Porter, D.E., Tsafrir, D.: Why and How to Increase SSD Performance Transparency. In: Proceedings of the Workshop on Hot Topics in Operating Systems. pp. 192–200. ACM (2019). https://doi.org/10.1145/3317550.3321430.
2. Lund, S.A.F., Bonnet, P., Jensen, K.B.A., Gonzalez, J.: I/O interface independence with xNVMe. In: Proceedings of the 15th ACM International Conference on Systems and Storage. pp. 108–119. Association for Computing Machinery, Haifa, Israel (2022). https://doi.org/10.1145/3534056.3534936.
3. Schroeder, B., Lagisetty, R., Merchant, A.: Flash reliability in production: the expected and the unexpected. In: Proceedings of the 14th Usenix Conference on File and Storage Technologies. pp. 67–80. USENIX Association, Santa Clara, CA (2016).
4. Li, N., Hao, M., Li, H., Lin, X., Emami, T., Gunawi, H.S.: Fantastic SSD internals and how to learn and use them. In: Proceedings of the 15th ACM International Conference on Systems and Storage. pp. 72–84. ACM (2022). https://doi.org/10.1145/3534056.3534940.
5. Corporation, I.: Intel® 64 and IA-32 Architectures Optimization.
6. Processor Programming Reference (PPR) for AMD Family 19h Model A0h, Revision A2 Processors Volume 1 of 6, https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/57228.zip, (2024).
7. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, (2025).
8. Trivedi, A.: Storage Systems Course, Vrije Universiteit Amsterdam.
9. He, J., Kannan, S., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: The Unwritten Contract of Solid State Drives. In: Proceedings of the Twelfth European Conference on Computer Systems. pp. 127–144. ACM (2017). https://doi.org/10.1145/3064176.3064187.
10. Gühring, P.: The missing Samsung EVO 840 - 250 GB SSD Repair Manual, (2016).
11. Zhang, L., Hao, S.-g., Zheng, J., Tan, Y.-a., Zhang, Q.-x., Li, Y.-z.: Descrambling data on solid-state disks by reverse-engineering the firmware. Digital

Investigation. 12, 77–87 (2015). https://doi.org/https://doi.org/10.1016/j.diin.2014.12.003.
12. Samsung Electronics Co., L.: 970 EVO Plus NVMe M.2 SSD 250GB, https://www.samsung.com/de/memory-storage/nvme-ssd/970-evo-plus-nvme-m-2-ssd-250gb-mz-v7s250bw/.
13. NVM Express® Base Specification, Revision 2.2, https://nvmexpress.org/specification/nvm-express-base-specification/, (2025).
14. Gigabyte Technology Co., L.: GIGABYTE NVMe SSD 256GB, https://www.gigabyte.com/SSD/GIGABYTE-NVMe-SSD-256GB/sp.
15. Co., K.T.: NV2 PCIe 4.0 NVMe SSD, https://www.kingston.com/datasheets/SNV2S_en.pdf.
16. TP Electronics, I.: NAND Flash 101.
17. Bjørling, M., González, J., Bonnet, P.: LightNVM: the Linux open-channel SSD subsystem. In: Proceedings of the 15th Usenix Conference on File and Storage Technologies. pp. 359–373. USENIX Association, Santa clara, CA, USA (2017).
18. Kang, D., Jeong, W., Kim, C., Kim, D.-H., Cho, Y.S., Kang, K.-T., Ryu, J., Kang, K.-M., Lee, S., Kim, W., Lee, H., Yu, J., Choi, N., Jang, D.-S., Ihm, J.-D., Kim, D., Min, Y.-S., Kim, M.-S., Park, A.-S., Son, J.-I., Kim, I.-M., Kwak, P., Jung, B.-K., Lee, D.-S., Kim, H., Yang, H.-J., Byeon, D.-S., Park, K.-T., Kyung, K.-H., Choi, J.-H.: 7.1 256Gb 3b/cell V-NAND flash memory with 48 stacked WL layers. In: 2016 IEEE International Solid-State Circuits Conference (ISSCC). pp. 130–131 (2016). https://doi.org/10.1109/ISSCC.2016.7417941.
19. Jung, M., Kandemir, M.: Revisiting widely held SSD expectations and rethinking system-level implications. SIGMETRICS Perform. Eval. Rev. 41, 203–216 (2013). https://doi.org/10.1145/2494232.2465548.
20. Hu, Y., Jiang, H., Feng, D., Tian, L., Luo, H., Zhang, S.: Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In: Proceedings of the international conference on Supercomputing. ACM (2011). https://doi.org/10.1145/1995896.1995912.
21. Micron Technology, I.: Technical Note TN-29-19: NAND Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product.
22. Mutlu, O., Kim, J.S.: RowHammer: A Retrospective. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 39, 1555–1571 (2020). https://doi.org/10.1109/TCAD.2019.2915318.
23. Chen, F., Lee, R., Zhang, X.: Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture. pp. 266–277 (2011). https://doi.org/10.1109/HPCA.2011.5749735.
24. Kim, K., Lee, E., Kim, T.: HMB-SSD: Framework for Efficient Exploiting of the Host Memory Buffer in the NVMe SSD. IEEE Access. 7, 150403–150411 (2019). https://doi.org/10.1109/ACCESS.2019.2947350.

25. Yang, Z., Harris, J.R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V., Paul, L.E.: SPDK: A Development Kit to Build High Performance Storage Applications. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 154–161 (2017). https://doi.org/10.1109/CloudCom.2017.14.
26. The Linux Kernel Documentation, https://www.kernel.org/doc/html/v6.15/.
27. NVM Command Set Specification, Revision 1.1, https://nvmexpress.org/specification/nvm-command-set-specification/, (2025).
28. Tavakkol, A., Gómez-Luna, J., Sadrosadati, M., Ghose, S., Mutlu, O.: MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In: FAST (2018).
29. al., C. et: Flash Translation Layer with lower Write Amplification.
30. al., S. et: Hierarchical Flash Translation Layer.
31. al., Q. et: VM-Aware FTL Design for SR-IOV NVME SSD.
32. Chen, F., Luo, T., Zhang, X.: CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In: 9th USENIX Conference on File and Storage Technologies (FAST 11). USENIX Association, San Jose, CA (2011).
33. Ma, D., Feng, J., Li, G.: LazyFTL: a page-level flash translation layer optimized for NAND flash memory. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. pp. 1–12. Association for Computing Machinery, Athens, Greece (2011). https://doi.org/10.1145/1989323.1989325.
34. Ma, C., Wang, Y., Shen, Z., Chen, R., Wang, Z., Shao, Z.: MNFTL: An Efficient Flash Translation Layer for MLC NAND Flash Memory. ACM Trans. Des. Autom. Electron. Syst. 25, (2020). https://doi.org/10.1145/3398037.
35. Litz, H., Gonzalez, J., Klimovic, A., Kozyrakis, C.: RAIL: Predictable, Low Tail Latency for NVMe Flash. ACM Trans. Storage. 18, (2022). https://doi.org/10.1145/3465406.
36. Hellwig, C.: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ea9b9c48387edc101d56349492ad9c0492ff78d.
37. Lee, C., Sim, D., Hwang, J., Cho, S.: F2FS: A New File System for Flash Storage. In: 13th USENIX Conference on File and Storage Technologies (FAST 15). pp. 273–286. USENIX Association, Santa Clara, CA (2015).
38. WHAT IS Flash-Friendly File System (F2FS)?, https://www.kernel.org/doc/html/v6.15/filesystems/f2fs.html.
39. Shaikh, S.: Billion-files File Systems (BfFS): A Comparison, https://arxiv.org/abs/2408.01805.
40. Corporation, I.: Intel® Optane™ Memory M10 Series 16GB, M.2 80mm PCIe 3.0, 20nm, 3D XPoint™, https://www.intel.com/content/www/us/en/products/sku/135581/intel-optane-memory-m10-series-16gb-m-2-80mm-pcie-3-0-20nm-3d-xpoint/specifications.html.

41. The Linux Kernel Source Tree, NVMe Subsystem, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.
42. Axboe, J.: FIO - Flexible I/O Tester, https://github.com/axboe/fio.
43. Linking SPDK applications with pkg-config, https://spdk.io/doc/pkgconfig.html, (2025).
44. The Zig Programming Language Homepage, https://ziglang.org/, (2025).
45. The Linux Kernel Source Tree, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.