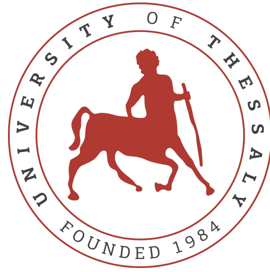


UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

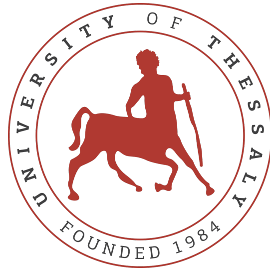
A study on Zoned Namespace SSDs' and their applications.

Diploma Thesis

Nikolaos Aridas

Supervisor: Athanasios Fevgas

Volos 2021



UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

A study on Zoned Namespace SSDs' and their applications.

Diploma Thesis

Nikolaos Aridas

Supervisor: Athanasios Fevgas

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δίσκοι στερεάς κατάστασης, η περίπτωση της αποθήκευσης σε ζώνες, μελέτη και εφαρμογές.

Διπλωματική Εργασία

Νικόλαος Αριδάς

Επιβλέπων: Αθανάσιος Φεύγας

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Athanasios Fevgas**

Laboratory teaching staff, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos D. Antonopoulos**

Associate Professor, University of Thessaly

Member **Michael Vassilakopoulos**

Associate Professor, University of Thessaly

Date of approval: 26-2-2021

Acknowledgements

To my family

DISCLAIMER ON ACADEMIC ETHICS
AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Nikolaos Aridas
24-2-2021

Abstract

In the last few years, more and more people are using technology for work or entertainment, which has lead to an astounding increase in the amount of data being created. For that reason, the data storage community has to keep up with the rise of created and requested data by attempting to offer more available storage, with better performance. The first common storage drive type was the cassete tape, followed by the Hard Disk Drives (HDDs) and then the Solid State Drives (SSDs). This thesis will study the performance of the most latest type of SSD, Zoned Namespaces, as well as its predecessor, the Open Channel SSD, by analyzing scientific papers and performing its own evaluation tests. The concept of Zoned Storage is a relatively recent development that refers to both HDDs and SSDs that enable the host system and the storage devices to cooperate to achieve better performance. The experimental evaluation of the drives is performed on FEMU (Flash EMUlator), which is an accurate NVMe SSD emulator and is exposed to the Guest operating system, which in this case is Linux. The tests will be performed with FIO (Flexible I/O tester) which is an open source disk I/O tool, created to be used for benchmark purposes. My evaluation tests confirmed the superior ZNS performance, resulting on average 4 times better write performance compared to the traditional and Open Channel SSDs.

Περίληψη

Τα τελευταία χρόνια, όλο και περισσότεροι άνθρωποι χρησιμοποιούν την τεχνολογία για δουλειά ή διασκέδαση, με αποτέλεσμα να έχει αυξηθεί ραγδαία η ποσότητα των δεδομένων που δημιουργούνται. Η επιστημονική κοινότητα που ασχολείται με αποθήκευση δεδομένων πρέπει να ανταποκριθεί σε αυτή την αύξηση των δεδομένων, προσφέροντας συσκευές αποθήκευσης με μεγαλύτερες χωρητικότητες και καλύτερες επιδόσεις. Η πρώτη συσκευή αποθήκευσης δεδομένων ηταν η ταινία κασέτας, για να ακολουθήσουν οι σκληροί δίσκοι (HDD) και τέλος οι δίσκοι στερεάς κατάστασης (SSD). Σε αυτή τη διπλωματική εργασία μελετήθηκε η απόδοση του τελευταίου τύπου δίσκων στερεάς κατάστασης που αποθηκεύει τα δεδομένα σε ζώνες (Zoned Storage), καθώς και των δίσκων ανοικτού διαύλου (Open Channel), με ανάλυση επιστημονικών άρθρων και πραγματοποίηση μετρήσεων. Η ιδέα της αποθήκευσης δεδομένων σε ζώνες είναι μια σχετικά πρόσφατη εξέλιξη που σχετίζεται τόσο με τους σκληρούς δίσκους, όσο και με τους δίσκους στερεής κατάστασης, η οποία επιτρέπει τη συνεργασία του κεντρικού συστήματος με τις συσκευές αποθήκευσης δεδομένων για καλύτερη απόδοση. Η πειραματική αξιολόγηση πραγματοποιήθηκε σε έναν προσομοιωτή μνήμης, το FEMU. Το FEMU είναι ένας ακριβής προσομοιωτής δίσκων τύπου NVMe, για το λειτουργικό σύστημα Linux. Τα τεστ πραγματοποιήθηκαν με το FIO, το οποίο ειναι ενα εργαλείο ανοιχτού κώδικα, που παρέχει την δυνατότητα πειραματικής αξιολόγησης των επιδόσεων μια συσκευής αποθήκευσης. Οι μετρήσεις που πραγματοποιήθηκαν επιβεβαίωσαν τα πλεονεκτήματα των δίσκων που αποθηκεύουν τα δεδομένα σε ζώνες, δείχνοντας 4 φορές καλύτερη απόδοση σε σύγκριση με τους παραδοσιακούς δίσκους στερεάς κατάστασης και τους δίσκους ανοιχτού διαύλου.

Table of contents

Acknowledgements **9**

Abstract **11**

Περίληψη **13**

Table of contents **15**

List of figures **17**

List of tables **19**

Abbreviations **21**

1 Introduction **23**

1.1 Storage Drive overview 23

1.2 Thesis organization 27

2 Open Channel SSDs **29**

2.1 Introduction 29

2.2 Open Channel SSDs 29

2.3 LightNVM 31

2.3.1 PBLK 32

2.4 QBLK 33

2.5 Baidu 35

2.5.1 SDF 35

2.5.2 LOCS 37

2.6 Alibaba Dual-Mode 37

2.7 Multi-tenancy 40
2.8 OC-Cache for Multi-Tenant Systems 40
2.9 SSW 42

3 ZONED NAMESPACES **43**
3.1 Introduction 43
3.2 SMR drives 44
3.3 Hardware 44
3.4 Software 45
3.4.1 Zone Append 47
3.4.2 Zone Random Write Area 48
3.4.3 Simple copy 49
3.5 Linux ecosystem 49
3.6 Use cases 50
3.7 SALSA, with Radian Zoned SSD case study 50
3.8 LSM-style case study 51
3.9 Sequential Read Ahead 52

4 TESTS on FEMU **53**
4.1 QEMU 53
4.2 FEMU 54
4.2.1 FEMU installation 55
4.3 Evaluation tests 57
4.3.1 ZNS, OCSSD and Traditional SSD comparison 58
4.3.2 ZNS tests 60

5 Conclusion **63**
5.1 Conclusion 63
5.2 The future of storage 63

Bibliography **65**

List of figures

1.1 Inside of an HDD [1] 23
1.2 NAND Flash Die Layout [2] 24
1.3 SATA SSD [3] 26
1.4 NVMe SSD [3] 26
1.5 Host and device new responsibilities on OCSSD [4] 27

2.1 LightNVM architecture 31
2.2 A picture of an SDF [5] 36
2.3 The SDF architecture [5] 36
2.4 The overall LOCS architecture [6] 38
2.5 Traditional Open Channel software stack [7] 38
2.6 The proposed user-space storage software stack [7] 39
2.7 SSW architecture [8] 42

3.1 the Ultrastar DC ZN540 [9] 44
3.2 Conventional SSDs and ZNS SSDs internal data placement [10] 46
3.3 The zone states [11] 47
3.4 Zone write [12] 48
3.5 Zone append [12] 48

4.1 Open Channel vs Femu performance [13] 55
4.2 Random Write (MB/s) 58
4.3 Random Read (MB/s) 58
4.4 Sequential Read (MB/s) 59
4.5 Sequential Write (MB/s) 59
4.6 ZNS Sequential Writes (MB/s) 60

4.7 ZNS Random Reads/Writes (MB/s) 61
4.8 ZNS vs OC Sequential Reads/Writes (MB/s) 61
4.9 ZNS Sequential Reads/Writes on the remote VM (MB/s) 62

List of tables

Abbreviations

etc.	etcetera
ZNS	Zoned Namespaces
OC	Open Channel
SSD	Solid State Drive
HDD	Hard Disk Drive
QoS	Quality of Service
FTL	Flash translation layer
OP	Overprovision
PPA	Physical Page Address
PBLK	Physical Block
L2P	Logical to Physical
PU	Parallel unit
LSM	Log-structured merge
KV	Keu-value
LOCS	Log-structured merge tree-based KV store on Open Channel SSD
API	Application programming interface
GC	Garbage collention
ZRWA	Zone random write area
SC	Simple Copy

Chapter 1

Introduction

1.1 Storage Drive overview

An SSD (Solid State Drive) is a non-volatile storage device, which consists of multiple NAND flash memory chips. Its low latency and fast access times, compared to its predecessor HDD (Hard Disk Drive), made the SSD the go-to drive for storing and using data efficiently.

On the other hand, HDD is a mechanical storage device mainly using a moving actuator arm connected to a fast rotating disk on where the data is placed [14] (Figure 1.1). HDDs dominated the market for a long time for their reliability, low-price and average speed combination.

Figure 1.1: Inside of an HDD [1]

Nowadays, SSDs are replacing old HDDs and there are a lot of reasons for that change. First of all, the lack of moving parts makes the SSDs damage and shock proof, more silent and faster. There is a significant improvement in start-up times, sequential access performance and also there is no penalty for random reads/writes, as each flash cell can be addressed directly. Moreover, hard drives are not suitable for mobile devices as the disk motor is very

power hungry and the fragile mechanical components can break, so flash storage is dominating this market. Despite that, HDDs are still a reliable option for data retention as it can store data for a very long time even without even being used.

NAND flash memory is the type of memory (non-volatile) that does not lose the stored data after its power is removed. It is organized in dies, planes, blocks and pages (Figure 1.2). Flash exhibit two levels of internal parallelism, at package and plane levels respectively. Package level parallelism refers to the ability of different dies executing different commands at the same time. On the other hand, plane level parallelism regards the concurrent execution of the same command on different planes of the same die. A block is the smallest amount of data which can be erased at once and usually its size is 4-8 MBs [15]. These blocks consist of pages which are the smallest writable entities, with 8-16 KB size. With NAND memory it is only possible to erase data in block and read or write in page units, but the pages must be erased before data can be written sequentially within the same (erased) block. The SSD controller is the one responsible for dealing with the previously mentioned rules. [4]

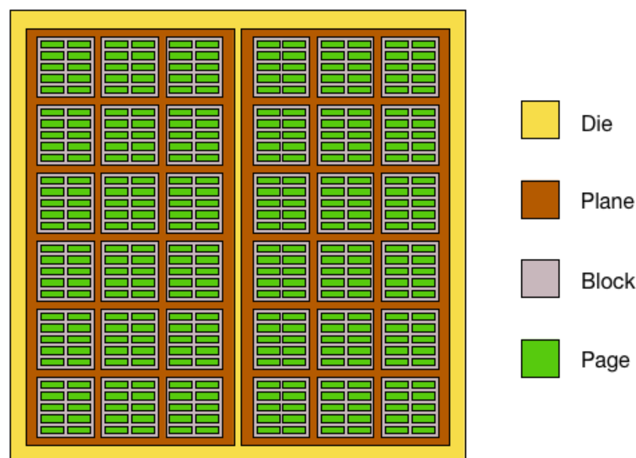


Figure 1.2: NAND Flash Die Layout [2]

However, NAND Flash memory tends to discharge if not being used for some time and will start losing data after a couple of years on average, so it is not optimal to use it for archiving purposes.

After an SSD is assembled, the SSD manufacturer reserves an additional percentage of the total drive capacity for over-provisioning during firmware programming. Over-provisioning improves performance and often increases the endurance of the SSD, helping the drive last longer due to the SSD Controller having more Flash NAND storage available to alleviate NAND Flash wear over its useful life [16].

One of the most important units regarding the overall performance of an SSD is the controller, a processor that connects the flash memory components to the SSD i/o interfaces, executing firmware-level software [17]. The flash memory is cheap but also it wears out. If flash gets rewritten many times, some logical block addresses (LBAs) will become useless over time. To mitigate this problem, the controller has a component called Flash Translation Layer (FTL) which maps the logical address from the host to the physical address on the flash, by spreading writes across all blocks. This way, the SSD stays healthy for a longer period. The FTL is responsible for all data placement, so the host software does not get involved in any of those functions and treats the SSD like a black-box. Most of the SSD's internal mechanisms are hidden in order to create the illusion of a block device, but software does not have control over the I/O performance when it reaches the device. It is clear that in many cases embedded FTLs are not optimal for utilizing the available resources.

In order for the host to communicate with the device via the PCIe or SATA interface, a host-to-controller protocol is needed (the AHCI protocol was used for the SATA interface). However, with the coming of PCIe, the SSD vendors provided different personalized protocols that made the transition to PCIe more difficult and time-consuming and the Non-Volatile Memory express (NVMe) standard was created to help the universal adaptation of this protocol. It was created from scratch, aiming to achieve better cooperation with PCIe and achieved better hardware interface (typically 600MB/s with SATA and 4 GB/s with x4 PCIe lanes), shorter hardware data path (reduce to data access latency) and a simplified software stack (improved access latency). SCSI, SATA and Serial Attached SCSI (SAS) are legacy storage protocols, designed mainly for supporting HDDs and they are connected with the host via an external Host Bus Adaptor (HBA). This technology had enough resources and capabilities to support the speed and performance required for HDDs, but the newer SSDs speed and demands surpassed the supported bandwidth. They were way faster, had lower I/O times and the HBA was a crucial factor for limiting the SSD's optimal performance. This bottleneck created the need for a newer technology, the PCIe (PCI express) [18].

Despite the huge performance improvement from HDDs to SSDs, the rapid reproduction of Artificial Intelligence, Cloud, Big Data etc created even higher demands for better performance and the ability for easy customization. The users do not have the flexibility to customize a traditional SSD, because standard hardware should apply to certain specifications. Being able to customize the SSD should not be considered a luxury but a necessity,

because enterprises demand much higher performance than standard users and this can only be achieved by working with application specific needs. These high-demand applications have extremely different I/O patterns and Quality of Service (QoS) targets and they are in need of distinct features from storage devices [4]. Figures 1.3 and 1.4 show a SATA and an NVMe SSD. The demand for cooperation between the host software and the SSD, led to the introduction of Open Channel SSD technology.

Figure 1.3: SATA SSD [3]

Figure 1.4: NVMe SSD [3]

The Open-Channel SSDs do not have an embedded FTL and they expose their parallelism to the host, which is now responsible for doing the FTL's work. Now, the host knows and manages the SSD, transforming it into a white-box and bringing a lot of benefits, like predicting how Input/Output (I/O) is submitted to the SSD and isolating I/O. In order to accomplish this big feature, some functionalities that the SSD firmware was responsible for performing, got moved to the host (Figure 1.5). However, the Open Channel SSDs did not get standardized, because now the drive exposed too many details to the software, thus several adaptation problems came up.

The NVMe working group proposed Zoned Namespaces (ZNS), an alternative storage organization of flash memory: divide the address space into zones and write on them sequen-

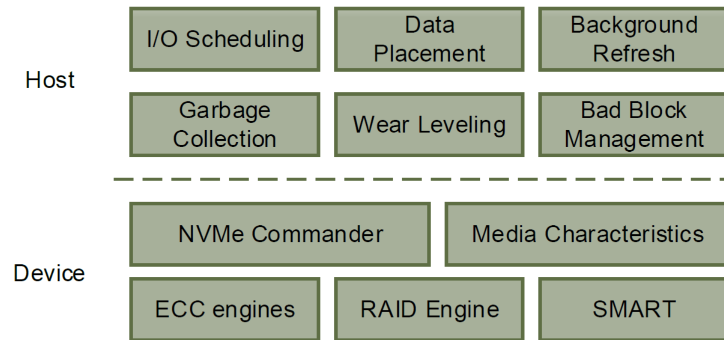


Figure 1.5: Host and device new responsibilities on OCSSD [4]

tially. This method has been used in Shingle Magnetic Recording (SMR) hard drives, but the older HDD technology created a limitation to its performance. However, the zoned organization turns out to be a good fit for the structure of the SSD. ZNS takes the load from the device and makes the host in control for write amplification, offers reduced overprovision, reduced DRAM (which is the second most expensive component, after NAND itself) and improved latency.

In this thesis I studied the NAND flash memory, as well as the use traditional SSDs. Moreover, I studied the Open Channel SSDs and the newer technology, Zoned Namespaces and their applications. Finally, I performed some evaluation tests on a Flash emulator called FEMU, which is able to emulate traditional SSDs, Open Channel SSDs and Zoned Namespaces.

1.2 Thesis organization

This paper is organized as follows. Chapter 2 presents the Open Channel SSDs 2. Chapter 3 analyses and reviews the Zoned Namespaces 3. Chapter 4 presents the evaluation results 4. Chapter 5 is talking about the conclusions of this study and then the future of storage industry.

Chapter 2

Open Channel SSDs

2.1 Introduction

With the arrival of the SSDs, we are experiencing a lot of improvements in the overall performance compared to the previous technology, HDDs. The demand for more storage is growing more than 50% annually the last years [19], thus companies are trying to find better solutions to deal with the rising cost and complexity of storage. It turns out that traditional SSDs, despite being very successful, have some inefficiencies that prevent the enterprises from relying on them, like unpredictable I/O latency and suboptimal use of the available resources.

2.2 Open Channel SSDs

Nowadays, Non-Volatile Memory technology provides the available resources and technology (very fast NAND chips inside the SSD, with predictable behavior), but we lack the right means to make the most out of them. Moreover, there is one very important obstacle that was difficult to overcome and that is the host's inability to manage the device directly because the host should first communicate with the Flash Translation Layer (FTL). The FTL is located at the flash memory controller and its role is to map the host LBAs to the physical address of the flash memory, which is of course a very important and necessary operation. However, with FTL the hardware (HW) is fully responsible for making all the decisions concerning placing data, scheduling, OP and GC which hurt performance [20]. That happens because the HW does not know anything about the SSD's geometry and basically makes as-

sumptions about the application workload and thus, there are a lot of missed optimizations. Not only did the FTL seemed to be an extra step that causes performance issues, but also it has its own bugs, and when embedded on the drive itself, it makes the handling and correction of the bugs more and more difficult to deal with [21]. As it turned out, the removal of the FTL is the first step towards better overall performance and host-device communication.

The Open-Channel SSD is the first proposed solution to the above problems, which does not include a Flash Translation Layer on the device but instead the SSD exposes its internals and allows the host to manage data placement [22]. So, the Open-Channel SSDs finally enable the device and host cooperation by sharing the responsibility of managing the SSD.

As I mentioned before, an SSD is made of many storage chips, which only support sequential operations and they are parallelly wired to a controller via channels [22] [23]. There are not any interferences between different channels, but there are usually between chips that belong in the same channel (e.g. on one chip, a write request might have to wait until a read request on another chip completes).

The Open Channel SSD shares its geometry with the host, like the number of parallel units (PUs), planes etc. The host has to know the exact dimensions and organization of the device in order to work together as planned. Some more important information that the host needs to know is the device performance, like typical and maximum read, write or erase times[22].

According to [24], there are three ways to build an Open Channel SSD storage system. The first method is to implement the flash translation layer inside a device drive and expose an I/O block interface, which allows legacy applications to use Open Channel SSDs with no modifications. The second one, is to place the FTL inside a file system and then expose it to a POSIX interface, which allows to eliminate some duplicated functions between the FTL and file system. The third, is to implement FTL functionalities inside user-space applications, which allows the users to get predictable latencies and more application-specific control of the device.

The usability of Open Channel SSDs, can be seen through some case studies. First, LightNVM was the first open subsystem for Open-Channel SSDs and host-based SSD management[22]. QBLK is an open-source driver, who further exploits the parallelism of OCSSDs and it was developed on LightNVM[24]. In addition, Baidu created its own customized SSD drive, called SDF, which was successfully used in cooperation with LevelDB[6]. Another proposition was made from Alibaba, where software and hardware were co-designed in order to

achieve better performance[7]. Finally, there were studies concerning the multi-tenancy attribute that Open-Channel SSDs can effectively provide [20] [25].

2.3 LightNVM

LightNVM is a Linux Open-Channel SSD subsystem that is organized in three main layers: NVMe Device Driver, LightNVM Subsystem and High-level I/O interface.

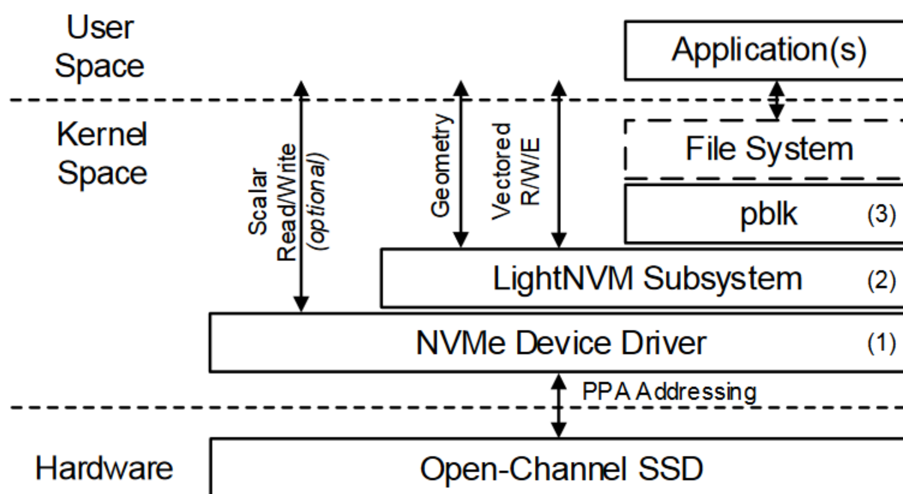


Figure 2.1: LightNVM architecture

As we can see from Figure 2.2, the kernel space can communicate with the SSD via the Physical Page Address (PPA) I/O interface. The PPA interface basically exposes the geometry of the SSD (physical or logical organization), performance metrics and controller functionalities (i.e. read, write, suspend support etc.) [26]. The dies are laid out one next to each other on this address space and instead of exposing all the linear space like on the traditional SSDs, each die is exposed separately. This way, the high throughput is achieved because when the user issues a write, he can now access multiple dies in parallel. To enable this functionality, there is a new interface through which the user can access multiple address spaces in parallel, for example instead of doing four separate reads or writes requesting an LBA, all four can be requested and received all together and, therefore, reduce four I/Os into one [22].

2.3.1 PBLK

The Physical Block Device (pblk) is an FTL on the kernel space of LightNVM. It is very important for the performance and usage of the OpenChannel SSD, as it does the mapping of logical to physical addresses, caching, garbage collection and error handling [22].

Write buffering

The NAND flash gets written in pages, but the host transfers data in sectors. When a host sector size is smaller than the NAND page size, it leaves some space unused, for example [22], a 16KB flash page with a 4KB sector size write. This created the need for a cache. Sector writes are buffered in the cache until there is enough data to match the page size. Then, potential read requests on those data are being sent in the cache, until the data is written to the drive. This process is called write buffering.

The write buffer is a circular buffer and is divided into two buffers, one for data entries (sector size granularity) and the other for keeping the metadata for each entry. Inside this buffer, the user can input I/Os in order to activate the mapping process from logical to physical addresses. If the input is already in the buffer, it gets updated [22]. When the buffer has enough data to fill a flash page or in the case of a flash command, a thread processes the entries and starts the mapping process. The parallel units (PUs) help with this process and there is always a certain number of active PUs, which are managed with a round robin logic.

The next question is: is it the host or the device the optimal place for the cache to reside? Following the general OpenChannel SSD logic, the cache should be located on the host side. This way, there is less interference (which is comparably slow) between the host and the device and if the cache was on the device side, then we will lose the predictable I/O. The only problem is that in the case of a power failure, the cache would empty and the data will not be saved, so error handling techniques should be available. In general, a read will be marked as failed, when all data retrieving techniques have tried and failed to do so. To recover a write, it is mandatory that the whole block should be retrieved and, if it is partially corrupted, then usually the buffer space should be enough to restore the lost data. If this does not happen, the whole block is considered unusable.

Logical to Physical table recovery

The Logical to Physical (L2P) mapping table needs to be reliable, so being able to recover from possible failures is a critical attribute for a drive. There are some ways to recover from a failure. Having a “copy” of the table in the form of a snapshot by keeping a log at different occasions is one way. Other methods are keeping the metadata on a block granularity, or saving a section of the mapping table in the Out Of Band (OOB) area (a cell area used for marking bad blocks on NAND memories[27]).

Error Handling

PBLK only has to manage write and erase errors, because at read fails the device is unable to do something about it and the system has to deal with it. The erase errors are dealt simply by marking the block that failed to erase as bad, as there are no data that need to be saved. Dealing with write fails is more complex than erase ones. The sectors that gave an error are being located and re-sent to the device without being put in the write buffer. The previous writes from the failed write should complete their process and the following ones should be paused until the problem is fixed. If the block that hosts the failed sectors is indicated as bad then it is sent for garbage collection [22].

Garbage Collection

As mentioned before, PBLK uses logging to save some data and so it is necessary to do some garbage collection. The physical blocks are responsible for keeping track of how many pages are valid on every block and the one with the fewest is sent for GC. The problem is that the GC may be interrupted by I/O sent from the user and in order to protect against this meddling, PBLK puts a lid on the number of those I/Os. It manages to achieve that, with the following logic: the more empty spaces are available on the write buffer, the higher priority is given to the user I/Os. If the buffer becomes full, I/Os will not get accepted at all until some space is freed[22].

2.4 QBLK

QBLK is an open-source driver which further exploits the parallelism of OCSSDs and is implemented based on the LightNVM subsystem. The team that proposed QBLK observed

that PBLK is not reaching the full potential of the performance it can achieve[24]. One reason it that PBLK has a simplified design of the translation map and write back functions, because it only uses one ring buffer to either cache or merge write requests. The problem is that when there are a lot of threads, this ring buffer becomes a bigger bottleneck. Moreover, PBLK uses a spin-lock to protect the whole mapping process despite the fact that different entries of the translation map do not need to be accessed one at a time, which also leads to increased overhead on heavy workloads.

QBLK tries to solve these problems, by offering multi-queue based buffering, per-channel based address management, lock-free address mapping and fine-grained draining.

First, QBLK adopts the ring buffer based write caching, but uses multiple ring buffers instead of one. QBLK allocates a ring buffer for each CPU, which has M times less capacity than the single ring buffer of PBLK, where M is the number of CPUs. The writing thread uses the ring buffer of the CPU that is running on. This technique reduces the competition among threads and thus offers better performance scalability. The only drawback for multi-queue based buffering are when intense I/Os are issued by only a few number of application threads, and it is not possible to utilize buffers bound with CPUs that are not running such threads. In this case QBLK does not perform well, but its a rare situation[24].

QBLK differs from PBLK on the physical address management subsystem by using multiple kernel threads taking data from ring buffers, instead of a line-based address management, because QBLK has multiple clients for the physical address management subsystem. So now in order to select the write back channel, each queue has an atomic pointer which points to its write back channel and every time a thread reads from the channel, the pointer moves to the next one.

QBLK deals with address mapping, by getting rid of the spin-lock and taking advantage of the atomic memory access primitives, that can transfer data to a map entry in an atomic way.

The fine-grained draining method suggests that when writing back pages from a ring buffer, it's better to submit more pages inside a request. By doing so, we can decrease the amount of total requests, diminishing software overhead.

These optimizations showed a 97% performance improvement compared to PBLK [24].

2.5 Baidu

Baidu is the biggest Internet search company in China and one of the biggest AI companies in the world. Baidu created a customized SSD, the SDF, and properly modified it to work effectively with LevelDB. LevelDB is an open-source key-value store implementing LSM-trees, originated from Google's BigTable (a high performance data storage system, designed to scale to a very large size) [28]. The SDF turned out to be a successful adaptation of Open Channel SSDs, and more than 3.000 SDFs were deployed in the production systems [6].

A log-structured merge (LSM) tree is an efficient way of eliminating random writes and keeping a viable read performance, using key-value (KV) stores. The KV stores are widely used by many internet services from Google, Facebook, Amazon, Github etc. Basically, they can be perceived like a hash table that maps a set of keys to their according values. KV stores are used for managing data, mainly for Internet services and they offer better performance and scalability than traditional relational database management systems (RDBMS). LSM trees and KV stores should be modified in order to take full advantage of the parallelism and multi-channel architecture of the SSD. Thus, Baidu proposed LOCS (Log-structured merge tree-based KV store on Open Channel SSD) which exploits a customized SSD that exposes its internals to the applications and they adapted it with LevelDB. Their tests showed a four times better performance than a traditional SSD.

2.5.1 SDF

As mentioned before, SDF is the customized SSD from Baidu exposing its internal channels to the applications, utilizing a customized controller. In SDF, the access to internal flash channels is open and applications can exploit this unique feature to fully utilize the drive's bandwidth. In order to take full advantage of this some customization should be applied to LevelDB. SDF uses a channel engine in order to provide some FTL functionalities, like wear leveling, bad block management and block-level address mapping and contains 44 independent channels. Moreover, the applications can view the SDF channels one by one independently (dev/ssd0 – dev/ssd43) and not as a single block device. This feature offers the ability to take full advantage of the raw flash performance and utilize it for data schedule and organization.

Figure 2.2: A picture of an SDF [5]

From a hardware perspective, an SDF consists of a customized flash device with a layer of software, responsible for translating the hardware parallelism into an improved throughput for the applications. The SDF controller has a total of five FPGAs. The first FPGA is a Xilinx Virtex-5, which works as a data path controller to enable the PCIe direct memory access. Virtex-5 is connected to four Xilinx Spartan-6 FPGAs (Figure 2.3), each of those having 11 channels with one FTL per channel. Each flash chip (8GB capacity) has two planes that support parallel access. So, the SDF has 4 FPGAs with 11 channels, each with 2 planes of 8 GBs, giving a 4*11*2*8 = 704 GB storage capacity [5]. A picture of an actual SDF can be seen on Figure 2.2.

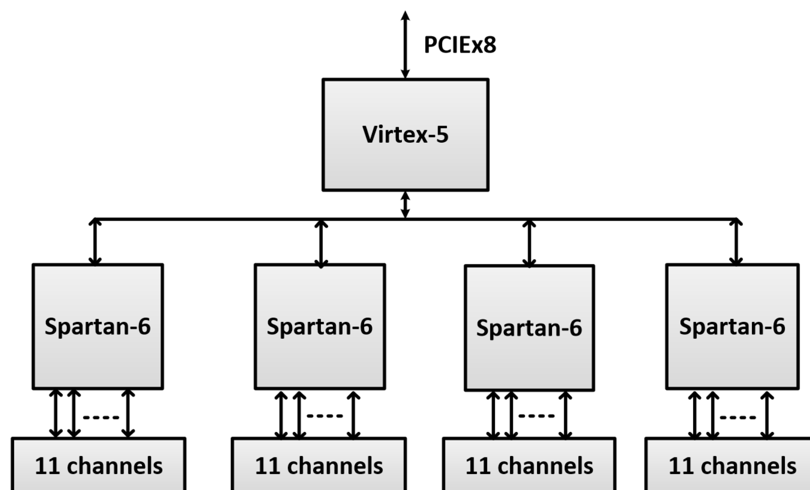


Figure 2.3: The SDF architecture [5]

SDF also does not give attention to small random writes in order to keep the ability of random reads, because it matches the LSM-tree access pattern. This happens because the write unit size is increased to 2MB, which is the flash erase block size and imposes that writes are aligned in blocks. Consequently, write amplification is reduced to almost zero because the

blocks have either valid or invalid pages during the GC. On the other hand, reads happen at 8KB (flash page size), in order for the SDF to maintain the random read attribute. The erase process on traditional SSDs is a painful and time-consuming process. This happens because the application is not aware of the erase operations as they are handled by the controller. As a result, we have the problem of unpredictable quality of service (QoS). With the SDF, the erase process is available as a new command for the user. So, the software responsible for managing and scheduling the erases can issue block erases before the block itself gets overwritten and thus offering better throughput and minimal delay.

Finally, the I/O stack on the Linux kernel can take up to 12µs and with the simplified and specifically made SDF I/O stack, it takes from 2-4µs. This happens because layer of I/O stack becomes a bottleneck for the high performance expectations of these days. On the other hand, the customization makes most file system functionalities useless and are bypassed with the ioctl interface (a system call for device-specific I/Os) to communicate with the SDF driver.

2.5.2 LOCS

LOCS is a system that works with the customized SSD (SDF) in order to cooperate with the LSM-tree. As we can observe from figure 2.4, its software is divided in: LevelDB, scheduler, storage API and the SSD driver. The proposed scheduler is designed only to accept requests from LevelDB (in contrast with the OS scheduler which manages every request) in order to schedule, dispatch and distribute them to the available SDF channels. Then, the requests call the respective application programming interface (API), based on the operation needed (write, read, delete). For the next step, they are being sent from the driver (software level) to the controller (hardware level) and then to the right channel(s), following on the information with which the scheduler provides them with.

2.6 Alibaba Dual-Mode

Another case study for OpenChannel SSDs is the Alibaba dual-mode SSD. Alibaba Group is a Chinese technology company and one of the world's biggest online commerce companies [7]. Dual-mode SSD supports two modes: the first one is a proposal for a hardware/software co-design implementation, on the hardware equipped with an Open Channel SSD and on the software their own Full-User-Space Storage Software Stack with a custom FTL (Figures 2.5

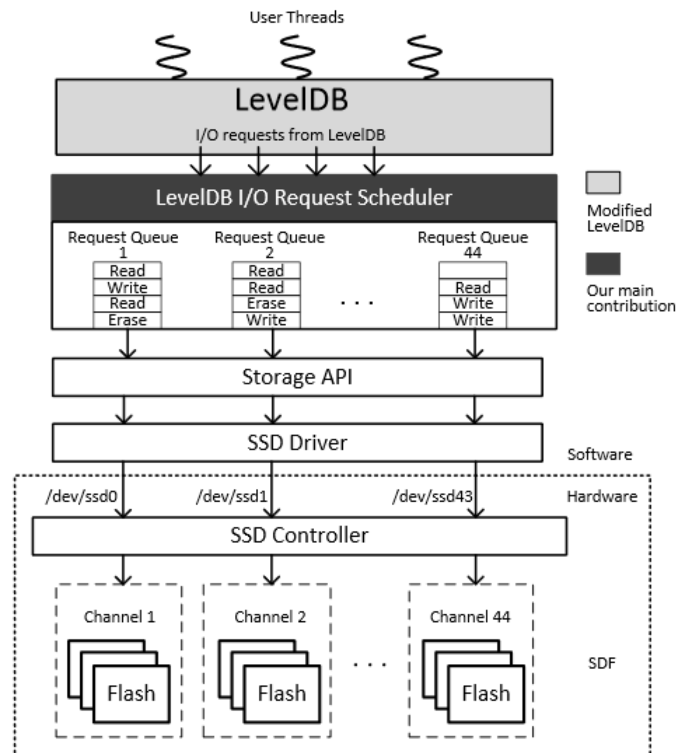


Figure 2.4: The overall LOCS architecture [6]

and 2.6). The second one is the standard NVMe SSD, mainly for compatibility reasons. The switch between modes can occur at runtime, only with an update of the firmware. Metadata needed from both modes should reside on the same system area, so both modes can have access to them.

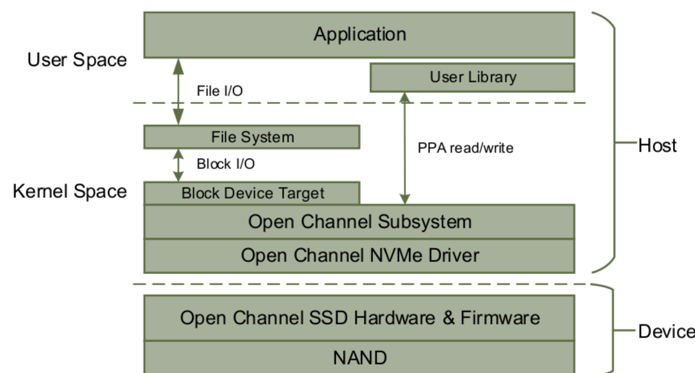


Figure 2.5: Traditional Open Channel software stack [7]

When the SSD works on the Open Channel (OC) mode, the drive basically operates like a traditional NVMe device, with the difference that is uses other commands for input/output. The roots of the software model is its library, called Dual-Mode Base Library which offers great stability on OC works and providing easier development on custom FTLs and incorpo-

ration with the applications. To underscore the importance of the library, here are some of its responsibilities: OC command processing, error handling, media management attributes and device identification.

As mentioned before, customized FTLs is the way to move forward to the hardware / software cooperation, as the Alibaba group suggested. FTLs are designed in order to match the specific application's input/output pattern, but the optimized results can be achieved in those specific cases and thus more work is needed for each case.

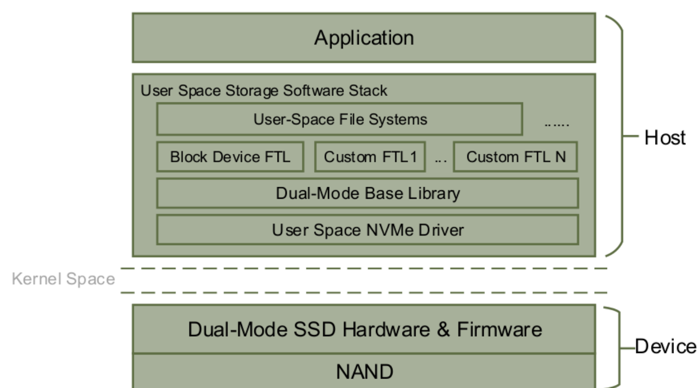


Figure 2.6: The proposed user-space storage software stack [7]

Regarding the thread management in the I/O stack, with the proposed model there are two options: the first one is the Run-to-Completion model, where each application is responsible to schedule all the input/outputs and as a result the overhead is greatly reduced, but further customization according to the specific application is needed. The other option is the pipeline model where a service process handles the requests and background tasks. This has the opposite effects with the Run-to-Completion model, as there is easier adoption with each application but there is greater overhead between communications of the same thread.

This case study also proposes a new I/O scheduler. First, the precise priority based I/O scheduling is responsible for labeling requests with critical, high, medium and low priority tags. The medium priority is the newly introduced one, compared to the traditional NVMe mode, aiming to take over GC operations. Critical priority is suitable for online read requests, high priority for offline data process and low for background task input/output traffic. The second logic is the coordinated garbage collection scheduling. During this process, it is very usual to observe write-burst phases, meaning that there is way more traffic than the average, which if not taken into account, can cause serious performance implications and deteriorate QoS. In this co-design, GC requests can be scheduled from the storage software stack ac-

cording to their input/output pattern, to minimize the performance impact that GC causes. The modifications made for garbage collection are simple: tagging GC requests as medium priority, disabling GC during burst-write phase and in rare cases promoting GC requests, if it is necessary.

2.7 Multi-tenancy

Multi-tenancy is a software architecture where the same software can serve multiple tenants. Achieving an effective multi-tenant environment requires a fair resource distribution by the host system and especially when we are talking about storage, the host is responsible for managing latency and throughput demands for each tenant. The proposed solution from [29] is build above LightNVM and manages to achieve half the latency times to I/O requests without modifying the applications. On LightNVM, dedicated pblk instances are used to provide I/O isolation, created on top of the physical LUNs which exclusively own.

More specifically, the idea is to provide I/O isolation between different applications. To guarantee that each tenant can only issue I/O to specific physical parallel units of the device (LUNs) and thus different application I/Os will never collide when reaching the physical media. But to achieve that, first there should be a fair (not necessarily equal) distribution of the parallelism parts of the device between tenants, depending on the usage and the demands of each application. Moreover, the bandwidth of each tenant should be limited to the number of LUNs that it uses, under the assumption that SSDs have the ability to provide the required bandwidth to the tenants with just their dedicated LUNs. For high-bandwidth demanding applications, throughput is worth trading in order to achieve I/O isolation on a multi-tenant setup.

2.8 OC-Cache for Multi-Tenant Systems

Nowadays multi-tenant systems are becoming very important with the rise of big data and cloud computing. In multi-tenant cloud systems, every tenant has a dedicated virtual machine, while other different VMs run on the same server to achieve a better resource utilization. The data is usually stored on a lot of HDDs because of its reduced cost compared to SSDs[30]. In order to bridge the performance gap between the main memory and the storage system, SSDs

have the role of an HDD cache to improve I/O performance, with the goal of maximizing SSD utilization and minimizing performance interference among different tenants [31]. Finally, it is important that the shared cache should be isolated to each tenant, but the problem is that data from different tenants are usually striped across the whole SSD.

The Open Channel Cache (OC-Cache) proposal from [25] is a framework to achieve both of these goals. First, OC-Cache calculates the tenant's cache space demand and creates an almost optimal cache partition to minimize overall miss ratio. Also it assigns channels to the Open Channel SSD, either as dedicated for the specific tenant or as shared between every tenant. Then, it proceeds to determine where to place a tenant's data according the access pattern.

As far as the architecture of this cache is concerned, OC-Cache launches a module called cache manager on the device mapper layer to manage its allocated cache space transparently to the tenant. The cache space allocator is a central module which resides on the user level and allocates cache space to each tenant. Every I/O request that is sent to a tenant's virtual disk (which is created by each tenant's cache manager) ends up on its own cache space and thus the requests from different tenants are separated without modifying the virtual software stack. The channel groups of the OC-Cache are divided into the ”isolated bucket” and ”shared pool” categories. On the isolated bucket category, one or more channels are allocated to a single tenant in order to ensure great performance isolation between tenants, while in the shared pool category the channels are shared among channels to improve the SSD cache utilization.

Block entries are managed from a multi-queue, which maintains a set of lists that are stacked into multiple levels. HDD block entries are placed at level 0 and they are moved to a higher level when they are accessed. That means that in the top levels reside the more accessed entries and on the bottom the less accessed ones. There is also a cache scheduler, which manages data migrations among HDDs and makes decisions for cache allocation according to the tenants' workloads. After a tenant finished its work and leaves the system, the cache allocator recycles free cache space in order to be used in the future. Finally, during the runtime of the OC-Cache, the tenants who have isolated bucket are separated from other tenants and performance isolation is achieved. In order to minimize the performance interference, the shared pool should not be accessed intensively.

2.9 SSW

The Strictly Sequential Writing (SSW) is a method that tries to take the most out of fully controlling the hardware resources of the Open Channel SSD. SSW is implemented in Light-NVM and adds a new layer between the hardware and the file system (Figure 2.7) which cooperates with FTL [8].

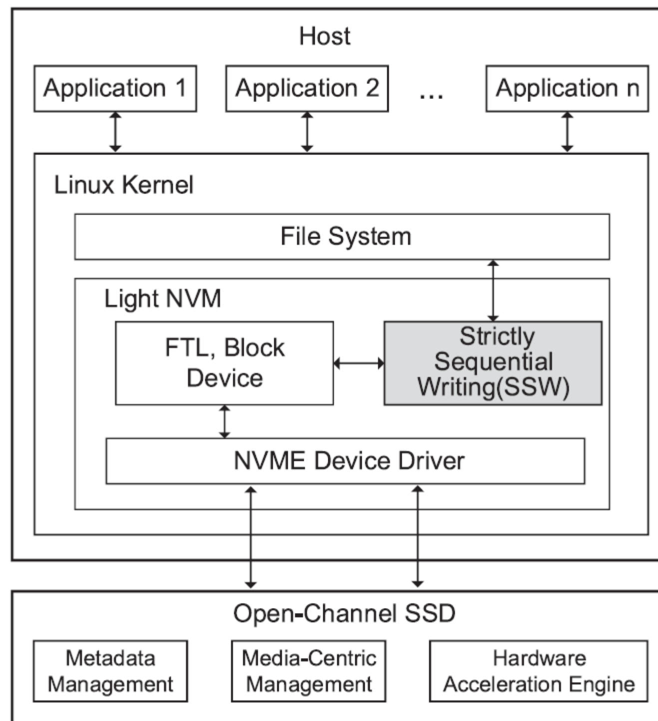


Figure 2.7: SSW architecture [8]

On the host side, the file system serves multiple data stream from the applications to the user space and then the data does not get processed by FTL, instead is optimized by SSW. SSW is able to be fully aware of the optimization that the read or write process need, because it knows the information from the metadata of the file system. The FTL is also located in the host and thus it helps SSW controlling the data allocation, which was almost impossible on typical SSDs.

Chapter 3

ZONED NAMESPACES

3.1 Introduction

The Open Channel SSD was the first alternative to the block storage abstraction and despite the fact that it provided I/O isolation, predictable latency, data placement and I/O scheduling, it did not get standardized. The main reason was that some of the earlier standards exposed too many details to the software and required to handle differences between different vendors' flash memory, or even between SLC, MLC, TLC or QLC [11]. This led the OCSSDs to put many requirements on the host software and that delayed adoption. Most current OCSSD projects are discontinued in favor of ZNS. Only a few Open Channel SSD implementation attempts were actually used, the Alibaba Dual-Mode SSD, Baidu's SDF and Denali's Open Channel SSD. Denali collaborated with CNEX, to optimize the workloads of a wide variety of cloud applications [10].

On November of 2020, Western Digital released the first ZNS SSD, the Ultrastar DC ZN540. This drive has a 96-layer 3D TLC NAND memory, with a controller supporting ZNS Command Set 1.0 specification [9]. The ZN540 comes in a U.2 factor, to provide a simple adaptation with existing servers. So far, this drive is shipped only to select customers of Western Digital, who run software that supports ZNS, in order to be further tested [32] (Figure 3.1.).

Radian also released its first Zoned Namespace SSD, the RMS-350. It is also a U.2 factor SSD, and it gives the user the opportunity to choose between Zoned Namespaces (ZNS), Symphonic API, or Open-Channel 2.0 interface [10]. The RMS-350 SSDs that are compliant with this NVMe ZNS specification are available to select development partners for evaluation

as well as cloud providers [33].

3.2 SMR drives

SMR is a predecessor to Zoned Namespaces. It was an approach to increase storage density on HDDs, by overlapping a portion of the previous magnetic track [34]. To achieve that, it was necessary to group tracks into zones and write into these zones sequentially. This happened because it was not possible to perform random I/O, without damaging the overlapping tracks. This technique did not work well because of random write problems, but inspired the idea of adapting this method to flash memory. It was proved that zoned storage cooperates well with the way flash drives work [11].

Figure 3.1: the Ultrastar DC ZN540 [9]

3.3 Hardware

From a hardware perspective, what are the required adjustments to the SSD so that it can support Zoned Namespaces? Surprisingly, only a firmware update from the manufacturer is needed to enable ZNS on most SSDs, although in this way, users will not get all the promised features and performance. ZNS feature reducing OP and the amount of DRAM on an SSD drive.

Regarding the benefits on the endurance of the drive, having more extra space (overprovision) available helps redistributing writes through every cell available. This way we avoid

to have cells that are written more frequently than others, which impairs the drive's lifetime. The typical overprovision (OP) percentage is 7% for read intensive and 28% for write intensive applications. So despite OP being extremely important for an SSD, not being able to use more than a quarter of the paid storage is not to be neglected. On the other hand, ZNS need approximately 10 times less OP, according to Western Digital [11]. This space becomes directly available to the user without compromising performance or endurance. This occurs because in zoned storage, write amplification can be avoided in the majority of the cases. Although, some overprovisioning is still necessary for managing and controlling NAND flash wear.

Moreover, DRAM is getting significantly reduced with zoned storage. Typically, there is 1 GB of DRAM for every 1 TB of SSD storage because traditional SSDs manage pages of 4KB, in order to store the flash translation table. Zoned storage on the other hand, is programmed to manage whole zones, which are several MBs each (usually 512MB) and the flash translation table has less requirements.

3.4 Software

Since there are not many hardware changes necessary (or none at all) to enable zoned namespaces, it is obvious that there should be way more software modifications. On ZNS, the host software has more responsibilities for the data management.

There are new concepts introduced with ZNS to help host software deal with the new constraints. The term Zone Capacity indicates the amount of logical block addresses (LBAs) available for use and is not the same for each drive. Instead, the size must be aligned with the size that the drive erases blocks in order to optimize the mapping of zone storage according to the drive's characteristics. Therefore, the vendors are not obliged to make power of 2 zone sizes.

Another important information that the host software has to know in order to be able to manage and control data placement, is the state of each zone. ZNS zone state logic is the same as in host-managed SMR drives, containing seven zone states in total. Figure 3.2 showcases the organization of data placement on regular and ZNS SSD.

The first two zone states are the empty (ZNE) and full (ZSF). As their name suggests, ZNE implies that the zone does not have any LBAs written and ZSF means that the zone

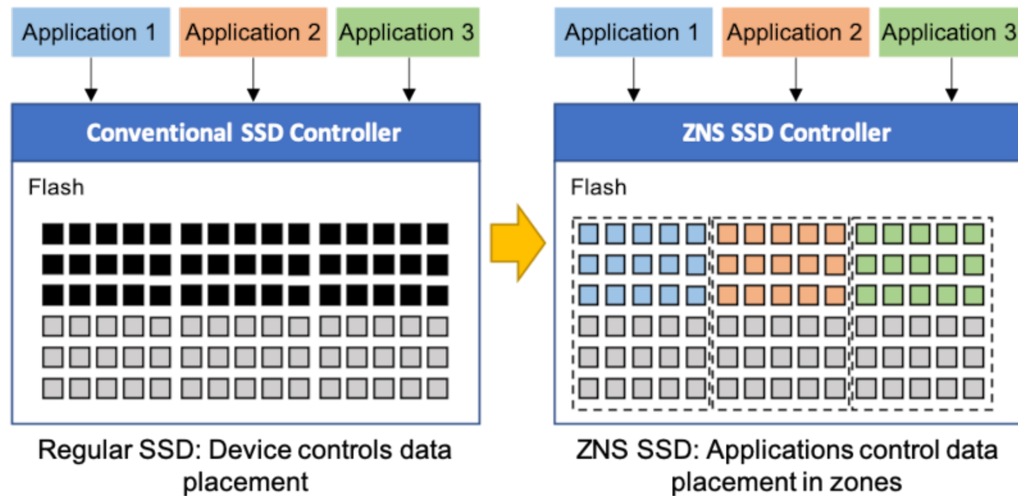


Figure 3.2: Conventional SSDs and ZNS SSDs internal data placement [10]

is full and not able to store anything else. ZNS SSDs still have to do wear leveling despite the fact that write amplification is decreased. Thus, when the drive exhibits high error rates, the offline (ZSO) and read-only (ZSRO) states are used. In ZSO the user can neither read, write nor change an offline zone into a usable state, even with format. On read-only states, the same rules apply but the user can only read from them. A zone is typically put on ZSRO state when a defective write occurs. When a zone is put in either conditions, the zonefs I/O error recovery is provoked and usually means that these zones are not usable any more [35].

Moreover, there are the implicitly (ZSIO) and explicitly (ZSEO) opened states, as well as closed ones (ZSC). Their common characteristic is that they all concern active zones. Active zones have a write pointer which informs the user on how much information is stored in each zone and the location of the next write on the zone. This extra information is very helpful and necessary for the drive, but comes with a cost. Drives do not allow the user to have unlimited open states because they need to manage this write pointer, which results in keeping track of more information. ZSIO and ZSEO also have some differences. First of all, ZSIO are opened from a write command. The SSD controller has the permission to close one implicitly open zone (ZSIO) if needed. On the other hand, ZSEO can open from a zone management command, but only the host software is able to close it. Also, in the case that all possible zones are explicitly opened it is not possible to open another one, but if there are some ZSIO, then, the controller will close some of them to make space for the new ones.

Below, there is a Figure (3.3) of those zone states.

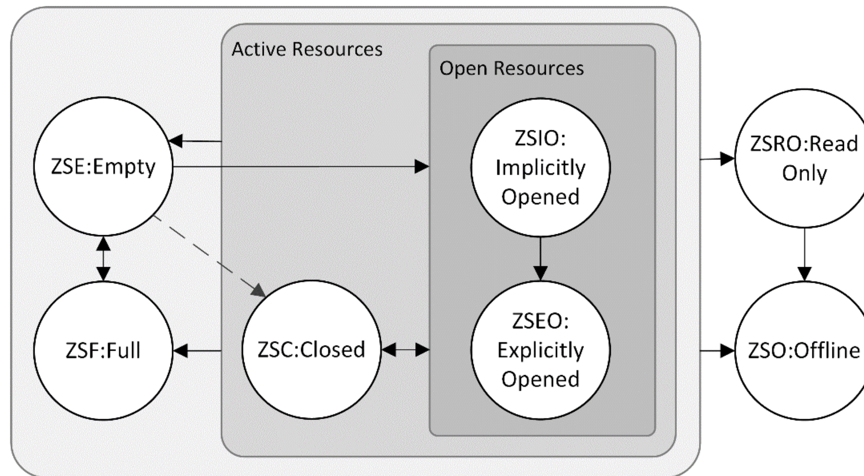


Figure 3.3: The zone states [11]

3.4.1 Zone Append

At SMR drives if the user wanted to have multiple writers attempting to write on a zone, they had to serialize them (send one write, wait for it to complete and then send the next one). This method, using a write depth queue of one, did not create any problems and had similar performance with traditional HDDs. On SSDs we have the potential to achieve higher throughput. However, serializing writes creates a bottleneck for large zones, although it is not a major problem for small zones.

The NVMe specification allows the host to submit multiple sequential write commands, but it does not guarantee to execute them in the received order. So a new command, Zone Append, allows the host not to specify a particular LBA on the zone but instead send several appends to a given zone. This was possible only by removing the queue depth constraint (length of one) and letting the host handle data placement. In the end, the same data has been written but not particularly in the same order as would be written with the simple write command. This new command improved the performance on both the host and the drive itself because it supports multiple outstanding writers to a zone.

For example, let us assume the user wants to send 4k, 8k and 16k writes with a 4k sector size [12].

On the traditional zone write, we have a zone queue depth of one, so the host first has to serialize I/Os (Figure 3.4). Then, the host has to start with the 4k write, increase the write pointer by one, do the 8k write and increase the write pointer by two and finally the 16k write and increase the pointer by 4. This serialization overhead is not very impactful when using

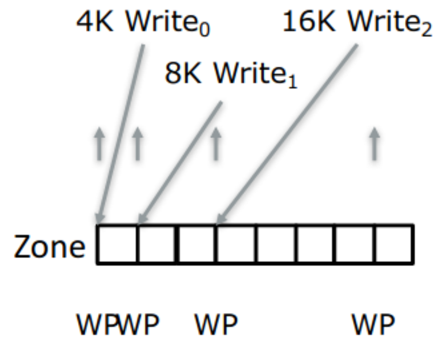


Figure 3.4: Zone write [12]

HDDs but harms SSD's performance.

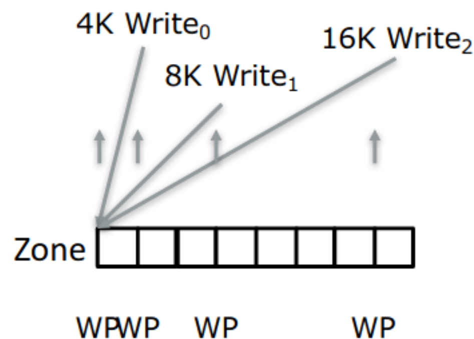


Figure 3.5: Zone append [12]

On the zone append case, the user sends all three of the write requests with a depth queue of 3 on the same place, because now the ordering is not a problem (Figure 3.5). So first the 8k write can be placed, then the 4k and then the 16k. This method is also scalable for both HDDs and SSDs.

3.4.2 Zone Random Write Area

The demand for sequential within a zone creates some difficulties for programmers. Thus, an extension on Zoned Namespaces is introduced, the Zone Random Write Area (ZRWA), which exposes the write buffer of the device to the host and allows the user to handle it. This area has no write pointer constraint and thus can receive in-place writing and over-writing of data, while they are still in the SSD's cache. Then, the ZRWA can be committed explicitly by a command and send the new data into the zones, or implicitly when the user tries to write more data than the size of the ZRWA. This extension has a lot of benefits because it is

"breaking" the write pointer restriction and so the write amplification factor is reduced and the user is able to send several I/Os in the device.

3.4.3 Simple copy

Another ZNS extension that is being standardized as a global NVMe command is the Simple Copy (SC) command. On traditional SSDs the device controller is responsible for the copy, but now the host is handling the metadata. So the host would have to read the data through the PCIe, transfer it all the way through the CPU route and write them on the destination, which is complex and time-consuming. With SC command, the user selects LBA ranges even from different sources and then points to a destination target using LBA and data length.

3.5 Linux ecosystem

The whole Zoned Namespace ecosystem was build based on the SMR drive ecosystem, and it is called Zoned Block Framework. This is a huge gain and ZNS adoption is going fast in terms of ecosystem because the framework does not need the amount of work that the Open Channel SSD needed, which took 2 – 3 years just to push the basics [36]. F2fs already supports the zoned devices, but there are some changes that have to be done. SMR and ZNS drives' main difference is that SMR drives have conventional zones, a number of zones that the user can randomly write to. All file systems are using these conventional zones to store the metadata. On the other hand, the ZNS purpose is not to have such zones at all and rather they exploit log-structures for every metadata. It is also needed to expose the framework to the applications and to add support for ZNS-specific features, like the ZRWA and Simple Copy. The framework lives on top of the block layer, which basically understands that the LBA spaces are partitioned zones [37].

There is also xNVMe, which is a library with the purpose to offer an interface for non-blocking devices and it is placed across different file systems [37]. One of the problems with ZNS is that the user does not deal with a block device anymore, so suddenly there is an application that needs to be partly modified. This modification part is the main problem with new implementations like ZNS or Open Channel SSDs, because programmers modify their application based on the device or framework they want to use. This has hindered adoption

because it is time-consuming to adapt to other frameworks [36]. So this library is created in order to make the re-write of the application significantly easier and to provide file-system semantics on block and non-block raw devices, without impacting the performance of the application.

So far, only a few applications have been released with support to ZNS SSDs. Most of the drive manufacturing companies have supported and studied zoned storage, with Western Digital being one of the top supporters of ZNS [11]. Western Digital has already developed a zoned storage backend for the RocksDB KV database, which was also used by Ceph, an open-source software storage platform. Samsung has also released a cross-platform library that accesses NVMe devices that support ZNS. Radian Memory has cooperated with IBM to publish a case study with an existing software-based log-structured storage system to run on Radian's non-standard zoned SSDs and they came out with significant results, regarding the throughput, quality of service and WAF.

3.6 Use cases

Zoned namespaces can be used for archival purpose, obviously. There are some specific configurations for ZNS in order to store cold data. The term cold data refers to data that is not frequently accessed, compared to the opposite "hot" data [38]. For this purpose, the zone size should be as large as possible, so there are less metadata and mapping tables [36] and thus the device management is going to be minimized [37].

Having big zones opens up the idea of treating them not only as very large data of QLC memory, but also using this mechanism to split the device and give place different applications in different parts of the device. This way, the applications are not only isolated between different zones, but also there is the new possibility of having multi-tenant applications with predictable I/O, because the I/O scheduling is managed by the host [37]. These former features are very important to cloud service companies.

3.7 SALSA, with Radian Zoned SSD case study

IBM Research developed a purpose-built storage software stack combined with Radian's Zoned Flash SSD and they claimed "dramatic" performance improvements, even without

huge development efforts [39]. SALSA (SoftwAre Log Structured Array) is a translation layer residing on the host, able to be configured to adapt to specific workloads and to expose a Linux block device in order to be used by unmodified applications.

SALSA is responsible mainly for data placement while ZNS in this case takes care of lower-level media management, like error handling, geometry emulation, wear leveling etc. Radian's SSD can perform decoupled wear leveling and NAND maintenance, meaning that it can internally perform wear leveling.

Almost every SSD loses zone capacity size over time because of the SSDs NAND failure characteristics. More specifically, in the case where the host is writing on a zone and a sudden write error happens, then this zone will reject any new incoming writes. This event triggers a process called zone excursion, where the host will select another zone and continue writing data there. Moreover, despite the fact that the host software issues the write requests sequentially, the NVMe standard does not guarantee ordering, so these requests may not arrive in order at the zoned storage device. The Radian SSD solved the problem by supporting out of order LBA arrivals to the same zone. Finally, SALSA is a log-structured software and in control of the SSD's management, so it inherently level wears new writes. However, in some cases it might not be enough so the SSD will perform light background processes which do not interfere with other I/O accesses and do not hurt performance [39].

3.8 LSM-style case study

Despite the fact that ZNS requires reduced garbage collection, some naive implementations can cause long latency because of the way bigger zone sizes. So a team studied a new Log-structured merge style GC, implemented and it turns out that performance is improved by almost 2 times on average [40].

The first implementation for zone GC works like that: selects the zone which is not being utilized so much, copies the valid blocks into a new zone and then resets the selected zone. This is the way GC is implemented on traditional FTL based SSDs, where GC manages 2MB or 4MB segments, however on ZNS we are dealing with 512MB or 1GB zones. This huge size difference creates a big GC latency on ZNS.

The new proposed approach (LSM_ZGC) is basically the division of a zone into smaller segments and then the management of each one separately. More specifically, this division

allows the GC execution with a detailed segment unit. LSM_ZGC does not read only the valid blocks but also the invalid ones, with 128KB I/O size because this way it is possible to exploit the internal parallelism of the SSDs. Finally, by dividing the zones into segments makes it possible to isolate hot from cold data and merge them into a different zone.

3.9 Sequential Read Ahead

Sequential Read Ahead (SRA) is the process anticipating a read request before it happens and thus having already read the sequential data. Since we are writing in zoned namespaces, the data related to each other holds a granularity of a single zone, so the idea is to implement sequential read ahead where the data is read ahead of time. The data is pre-fetched from the flash memory and is kept in the SRAM cache, in order to achieve quicker access. This way, a cache hit on every requested data occurs and the performance is greatly improved (70 MB/s on traditional vs 275 MB/s after SRA [41]).

In order to achieve that, we are keeping track of the states of the SRA function, which are basically the ’enabled’ SRA and ’disabled’ (default) SRA states, and this transition can happen under some pre-defined conditions [41]. After the transition from enabled to disabled state, the cache entry (CE) teardown process is called, which removes all data from the cache to free up space.

Chapter 4

TESTS on FEMU

4.1 QEMU

First of all, FEMU is a QEMU-based open-source Flash EMUlator which encourages software and hardware SSD research.

QEMU is a fast machine emulator that uses an original portable dynamic translator in order to emulate several CPUs on several hosts. It is composed of several subsystems, like the CPU emulator, a user interface (UI) and generic devices. QEMU offers a lot of usages. It is able to run one operating system on another (e.g. Linux on Windows) and it offers debugging, because QEMU is programmed to be easily stopped, in order to inspect its current state. Moreover, it can also simulate embedded devices by adding new machine descriptions and new emulated devices [42].

QEMU can operate in multiple modes, depending on the use case [42]. In the user-mode emulation, QEMU runs single Linux programs that originally were compiled for a different instruction set and its main targets are fast cross-compilation and cross-debugging. In system emulation mode, QEMU can emulate a full computer system, including peripherals in order to virtually host several computers on a single computer, while supporting multiple instruction sets (x86, ARM, MIPS etc). Moreover, QEMU uses a dynamic translator, which converts the target CPU instructions into the host instruction set at runtime and then storing the binary code in a cache, for reuse purposes. QEMU makes the transfer of the translator from one host to another very simple, by linking pieces of machine code generated by the C compiler. If not, the whole code generator must be written again, which is a very painful process.

QEMU supports two virtual disk image formats, QCOW or QCOW2 (Qemu Copy On

Write), optimized in order to take up as much disk space as the guest OS uses. A very important feature is that the QCOW2 format allows the user to delete the overlay, use a previous disk image in order to revert the emulated disk's contents to an earlier state in case there are some errors and to render the guest system unusable [42].

4.2 FEMU

FEMU is implemented in QEMU v2.9 [42] which acts like a virtual block device to the Guest OS (Linux) and the typical stack for SSD research is: Application, Host OS and FEMU. FEMU's challenge is to offer scalability, accuracy and increased extensibility, by offering 32 parallel channels, having a 0.5-38% variance as a replacement of OCSSD and supporting internal-only and split-level SSD research respectively.[43].

In addition, FEMU team claims that most of the scalability bottlenecks of QEMU exist because of two factors. First, to emulate I/Os, the Guest OS' NVMe driver notifies the QEMU (device) that new I/Os have been inserted in the device queue, and this notification will also be sent upon completion. The problem is the notification itself, because it is a Memory-mapped I/O (MMIO) operation, which uses the same address space to address both I/O devices and memory. This way of mapping is responsible for a very expensive transition from the GuestOS to QEMU (VM-exit). Now, FEMU creates a dedicated thread in the shared memory between GuestOS and QEMU, in order to poll the status of the device queue. This logic helps to avoid these VM-exits and still transfer the control to QEMU, and to enable that, the user only has to delete/comment out one line of code from the Linux NVMe driver!

The second bottleneck is the fact that QEMU uses asynchronous I/Os (AIO) in order to perform the read write byte transfer to the backing image file. This AIO overhead is crucial when the storage backend is an image that has been RAM-backed, however, AIO is needed to eliminate the blocking of QEMU by slow I/Os. FEMU's solution was to create a customized RAM-backed storage in QEMU's heap space and not to use the virtual image file. Then, the data gets transferred to this heap while the Guest OS is not aware of this process [13].

But how is FEMU able to emulate the delays and produce accurate results? Well, when an I/O arrives, a direct memory access read or write command is issued from FEMU. Then, the I/O is labeled with an emulated completion time, called Tendio (Time end I/O) and, finally, it is put on a specific queue (sorted based on completion time) where all completed I/O re-

side. Tendio is calculated through a queueing model, which represents a single-register and a uniform page latency model. Although this model will work fine for research, some programmers wanted to emulate the specifics of modern hardware and the FEMU model got extended in order to work with Open Channel SSDs. Open Channel mode use double-register planes, each owning the data and cache registers, so a NAND read or write in a plane can happen at the same time as a data transfer via the channel to the plane and, consequently, more parallelism is exploited. In addition, a non-uniform page latency model is used by Open Channel mode, where pages that reside to the upper bits of MLC cells, perform with higher latencies than of those residing to the lower bits. So for reference [13], if a lower page can read with 48μs latency, an upper page with 64μs.

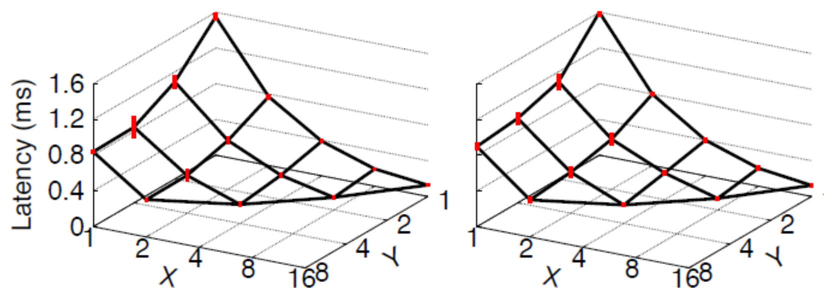


Figure 4.1: Open Channel vs Femu performance [13]

On figure 4.1, X represents the number of channels, while Y the number of planes per channel.

As we can see, both figures have a very similar graph based on the latency.

FEMU can be run in blackbox, whitebox, ZNS and no-ssd mode. Blackbox mode emulates the traditional SSDs, whitebox the Open Channel SSDs and ZNS supports Zoned Namespaces. The no-ssd mode gives out the best results possible for FEMU. One of the problems with FEMU is that it is DRAM-backed, so it can not emulate SSDs with big capacity.

4.2.1 FEMU installation

In order for FEMU to work, QEMU should be installed. QEMU installation is very simple, as only a single command should be executed on a Linux terminal.

I used FEMU from a (pre-installed on Linux Mint 20.1) program called Virtual Machine Manager (VMM). I needed to install a package called virtual manager in order to be able to

use VMM.

As for the FEMU installation now, the FEMU's github page [13] has most of the instructions I needed to follow for the proper installment. After installing all the libraries and dependencies from the given instructions, I run the shell script file (.sh) and the whole compilation and installment of FEMU began.

On the github page, they provide us with their own VM image, which is a .qcow2 image file, so we can skip the "building a new virtual machine" part and proceed faster onto testing. However, they provide a guide to help users build their own VM image.

To start the emulator, we need to open the VMM program, add a new virtual machine and follow the steps in order to load the downloaded image named u20s.qcow2 (as an ububtu retribution). After the virtual machine has booted, a text-based VM login shows up, where the credentials have been sent to the user via e-mail, along with the requested VM image. After logging in, a terminal is shown with only a folder named "nvme-cli".

Upon booting the VMM, the user can install and run the "openssh-server" inside the VM in order to use the VM through the user's host mach terminal . I found this to be the most effective way of using the VM. After using the ssh command, the user has to go to the "$HOME/femu/femu-scripts" path and run one of the following .sh files: "./run-blackbox.sh", "./run-whitebox.sh", "./run-zns.sh" or "./run-nossd.sh" . This command is programmed to load the specific u20s.qcow file from the "$HOME/images" folder, so if the user places the downloaded image somewhere else, they should modify the path in the .sh files. After the compilation of FEMU, a binary file is created called qemu-system-x86_64 at "$HOME/femu/femu-scripts/x86_64-softmmu", which has the "femu device name" loaded, so in order to run the scripts, the command used is x86_64-softmmu/qemu-system-x86_64, instead of just qemu-system-x86_64 which is for QEMU.

4.3 Evaluation tests

The testing system is equipped with an Intel Core i7-5500 @2.4GHz, with 8GB DDR3 RAM memory running on Linux Mint(64-bit), with Kernel 5.10.0, on a 500GB SSD and 2 CPU cores.

IOPS (input/output operations per second) is the unit of measurement for the maximum number of reads and writes to storage locations.

Bandwidth is shown on MB/s.

This is an example of a .fio test file. This script creates and writes (4kB reads) randomly (rw label) on a 10GB (size label) file (on the current directory), with filename fio-rand-write and a queue depth of 16 (iodepth label).

```
[global] name=fio-rand-write
filename=fio-rand-write
rw=randwrite
bs=4K
direct=0
numjobs=4
[file1]
size=10G
ioengine=libaio
iodepth=16
```

4.3.1 ZNS, OCSSD and Traditional SSD comparison

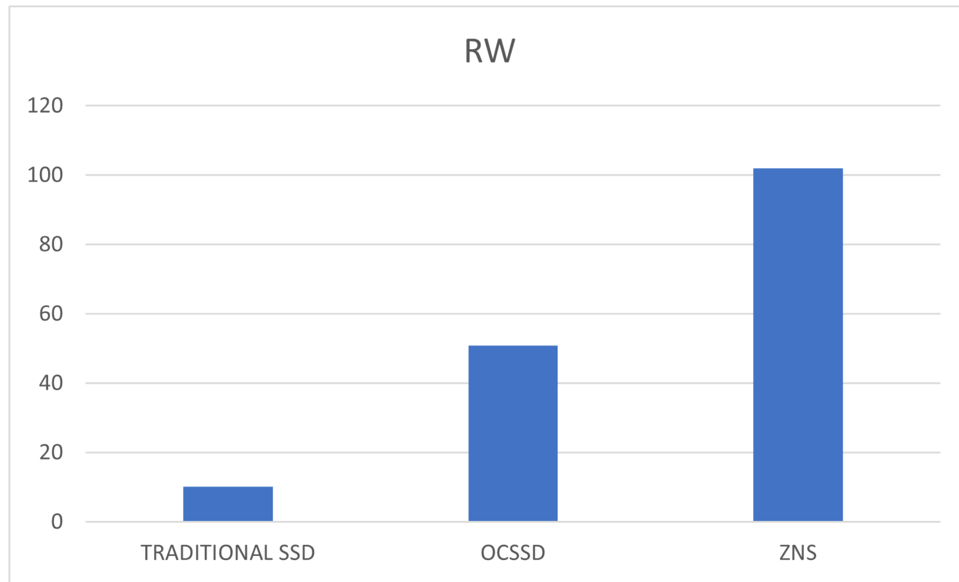


Figure 4.2: Random Write (MB/s)

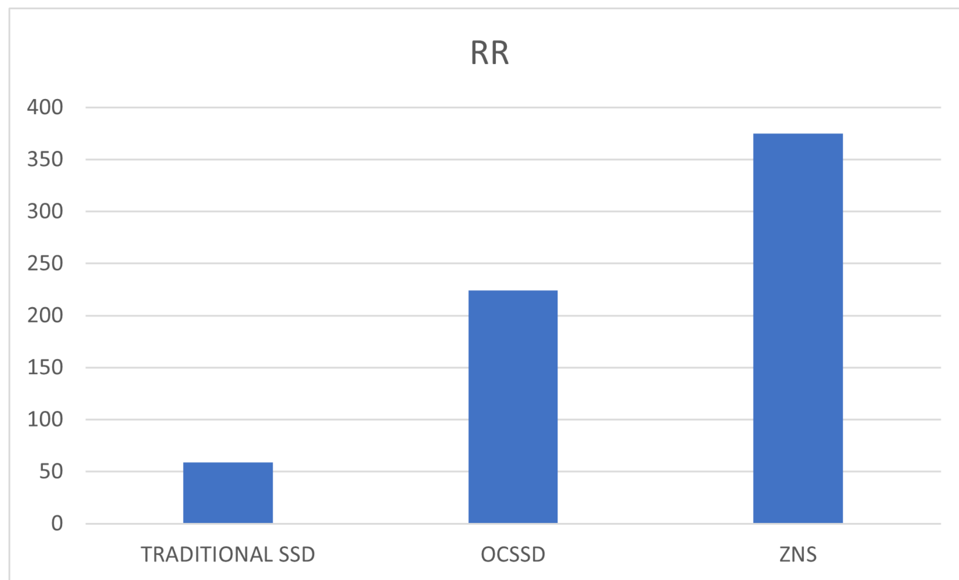


Figure 4.3: Random Read (MB/s)

We compare a traditional SSD with Open Channel and ZNS SSD respectively. The tests were performed with only one thread issuing the I/O operations. In all test cases, ZNS outperforms both the Open Channel and the conventional SSD.

The first experiment regards random writes and random reads (Figures 4.2 and 4.3). ZNS improves up to 90% compared to the traditional SSD and 50% compared to the Open-Channel one in random writes. The results are also similar in the random read test case.

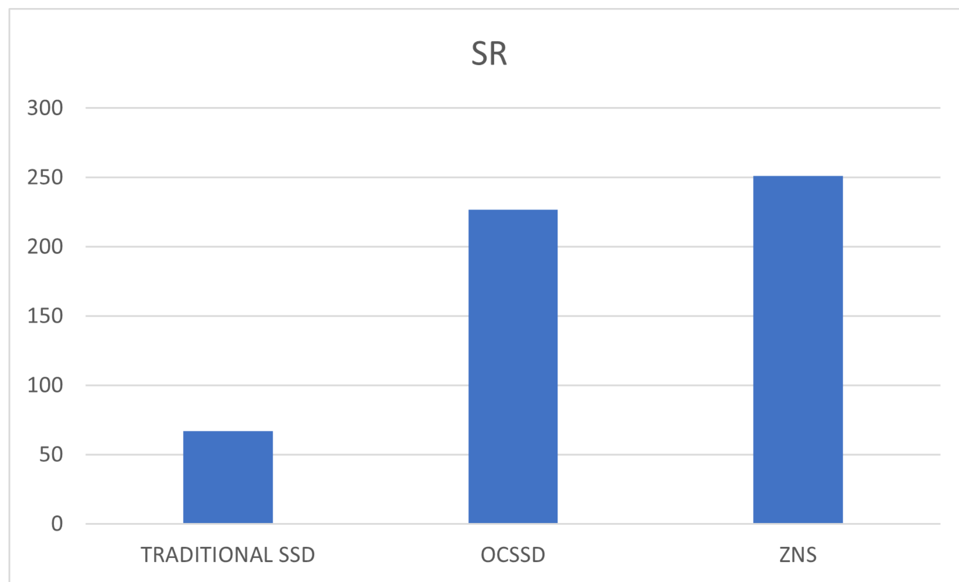


Figure 4.4: Sequential Read (MB/s)

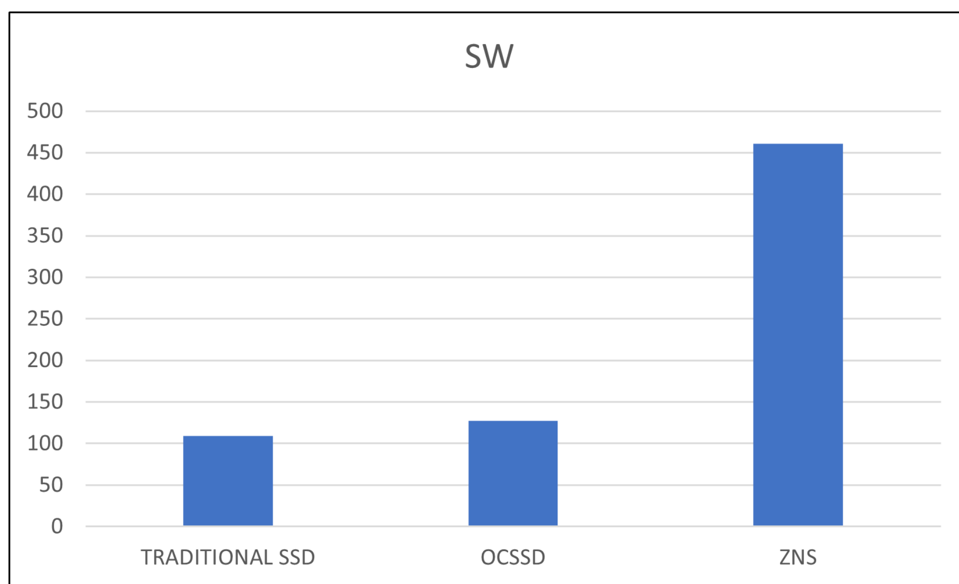


Figure 4.5: Sequential Write (MB/s)

Since ZNS write sequentially, we are most interested in the sequential performance, and ZNS shines there. Thus, the next experiment uses a 10GB sequential write workload. ZNS achieves 461 MB/s, which is 3.6 times faster than OCSSD and 4.2 times faster than traditional SSDs (Figure 4.5). On the 10GB sequential read test (Figure 4.4), the Open-Channel SSD was keeping up with the performance of ZNS (227 vs 251 MB/s respectively), but the traditional SSD was once again experiences the worst performance, achieving 67 MB/s.

4.3.2 ZNS tests

The ZNS tests were performed with an queue depth of 16, meaning writing 16 4KB pages simultaneously, so we are expecting to see the ZNS performance being improved compared to the the previously acquired results (Paragraph 4.3.1). These experiments test the performance of ZNS, using different number of Parallel Units in each run.

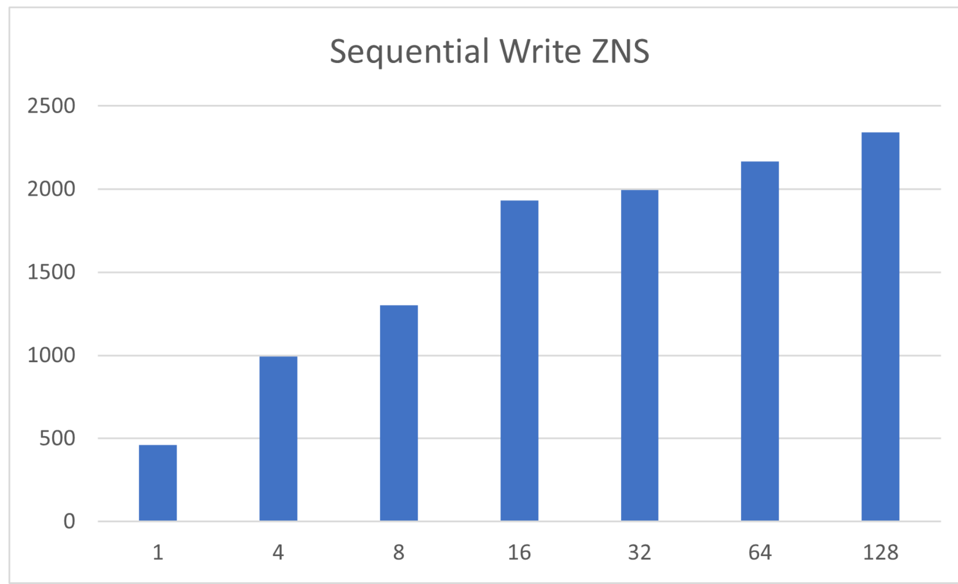


Figure 4.6: ZNS Sequential Writes (MB/s)

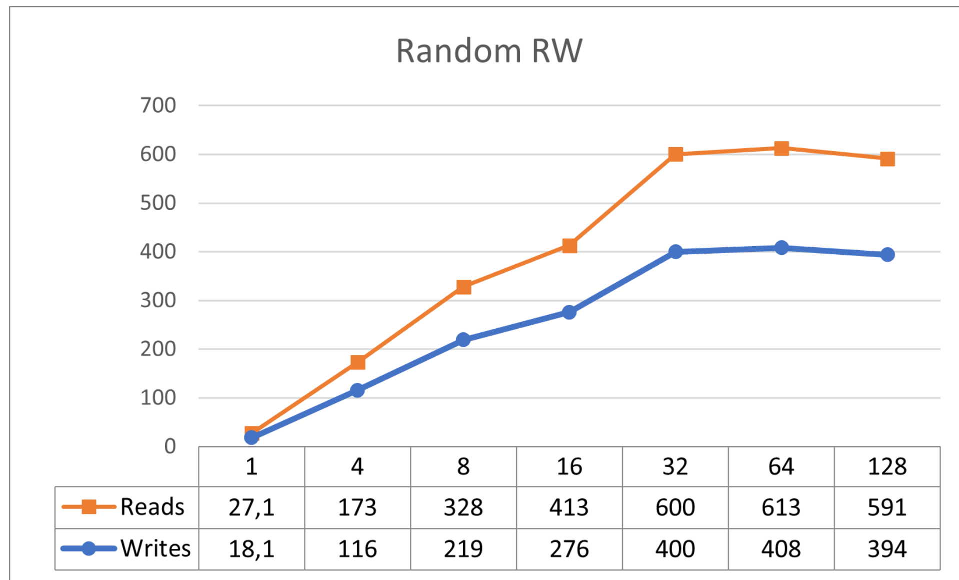
Figure 4.6 shows us that the more parallel units we provide, the more write speed we get. With 128 PUs available, we can reach up to 2344 MB/s, which is more than 5 times faster compared to the performance that we get from 1 PU.

However, real workloads mix reads and writes. Figures 4.7 and 4.8 present the results of a mixed workload with 40% writes and 60% reads. Performance is improved as the number of PUs increases.

The random read and write test (Figure 4.7) reaches 27 MB/s read and 18 MB/s write times when having only 1 PU available. Performance peaks at 64 available PUs, where we can both read and write 22 times faster.

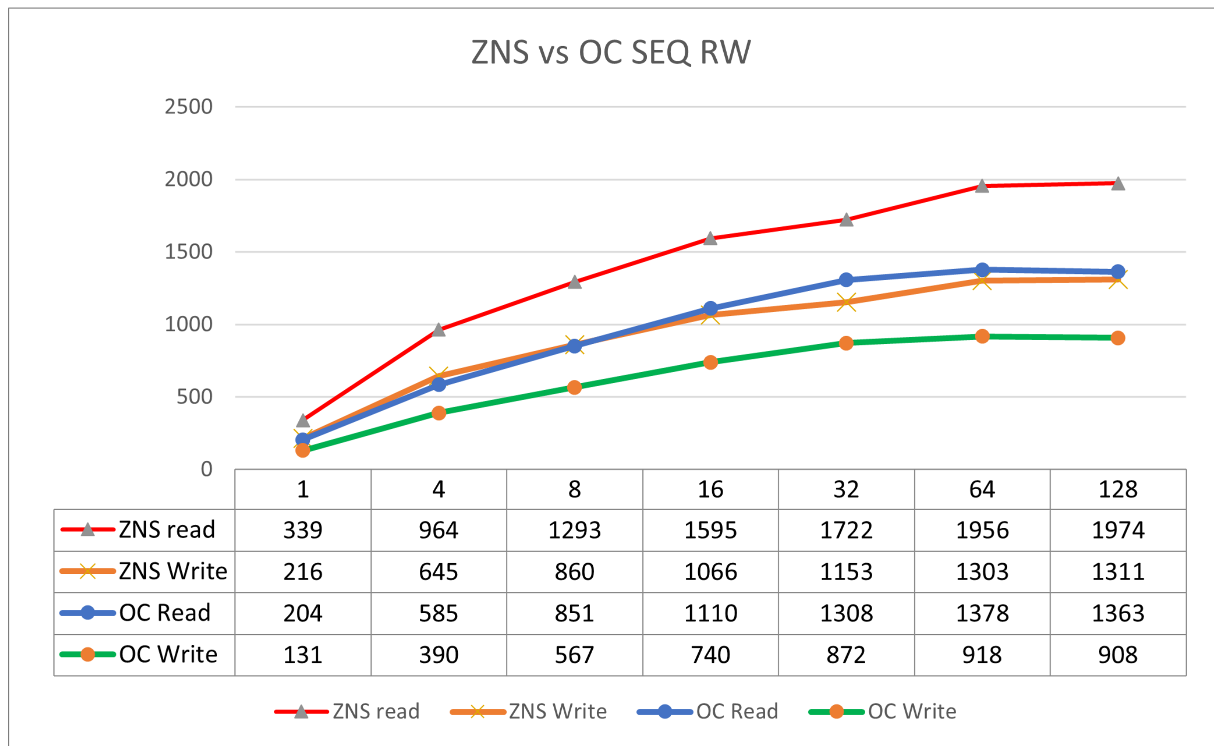
Figure 4.8 demonstrates the supreme performance of ZNS, when mixing sequential reads and writes. The read and write speed combination is by far the best even compared to OCSSD, which performed this test (with 1 PU available) with 204 MB/s sequential read and 131 MB/s write speeds, vs the 339 MB/s and 216 MB/s speeds of ZNS.

Finally, another set of experiments involving sequential reads and writes, was performed



	1	4	8	16	32	64	128
Reads	27,1	173	328	413	600	613	591
Writes	18,1	116	219	276	400	408	394

Figure 4.7: ZNS Random Reads/Writes (MB/s)



	1	4	8	16	32	64	128
ZNS read	339	964	1293	1595	1722	1956	1974
ZNS Write	216	645	860	1066	1153	1303	1311
OC Read	204	585	851	1110	1308	1378	1363
OC Write	131	390	567	740	872	918	908

Figure 4.8: ZNS vs OC Sequential Reads/Writes (MB/s)

on a remote Virtual Machine (VMWare ESXi 6.5), running Ubuntu Linux with a 5.10.0 kernel, 32GB RAM, 20 CPU cores and 100GB HDD.

We can observe from figure 4.9 compared to 4.8, that the overall performance is worse on the remote VM, but it scales a lot faster when the number of PUs is 32. At maximum performance, we can reach to almost 3000 MB/s read and 2000 MB/s write speeds.

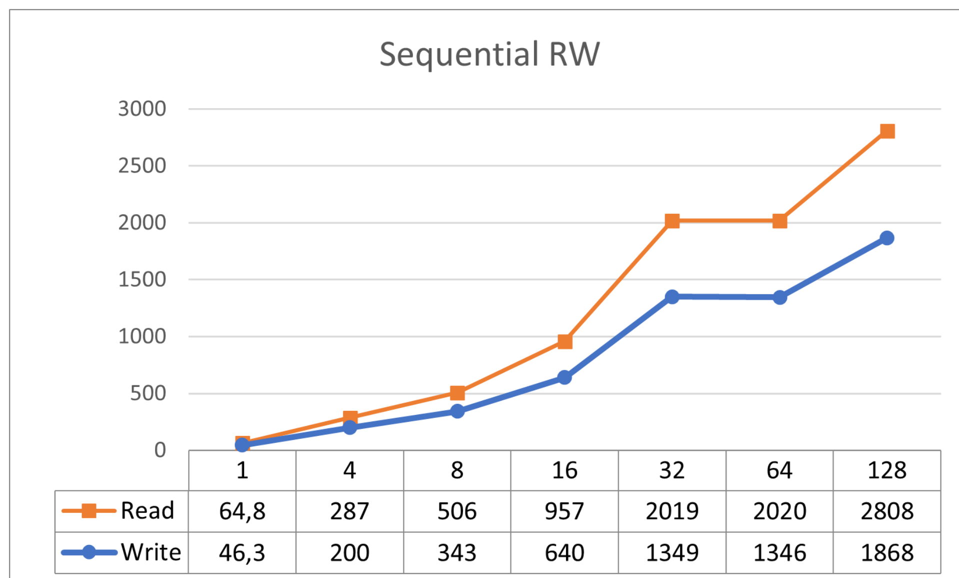


Figure 4.9: ZNS Sequential Reads/Writes on the remote VM (MB/s)

Chapter 5

Conclusion

5.1 Conclusion

To sum up, this thesis studied the performance of Zoned Namespaces through different use cases and evaluation tests. The testing showed the exceptional performance of ZNS in every category, compared to the previous technologies, the traditional SSD and the Open Channel SSD.

5.2 The future of storage

What does the future hold for the storage industry?

To try and answer this question, we should first examine how past successful technologies evolved through the years. The fact that tape drives are still being used in 2021 seems remarkable to me. The tape drive is the cassette-like tapes that we all know and this storage technology was first introduced way back in the 1950s. It still holds up to the market because it is well suited for backup, as the tape can hold on the data for up to 30 years![44] The tapes are more useful for storing cold data though, because with this older technology we are losing performance.

This leads us to assume that the HDD technology is going to be used for a lot of years in the future. According to the CFO of Seagate, HDDs are approximately 20 years from being displaced from newer technology and recent researches have figured that by 2025 their capacity will rise to 100TB. There is still a lot of development for new HDD technologies, for example Western Digital figured that if they manage to replace the air from inside the

drives with helium, which was 1/7th of the density of the air, they can fit the maximum 6 disks into a standard 3.5” HDD. This experiment worked well and despite the HDDs being more expensive to buy, a lot of datacenters started using them.

Now to get into perspective of how much data we humans need, I am going to showcase some interesting numbers and statistics: During 2020, every person created 1.7MB of data every second on average, totaling 2.5 quintillion bytes of data each day worldwide. Moreover, YouTube alone requires 1 million GB of new hard drive capacity every day [45]. That leads to the fact that 90% of the world’s data has been created on the last two years alone [46][47], and these numbers are only going to rise exponentially.

According to Google, cloud storage services will very soon be responsible for most of the capacity in use [45]. The problem is that so far, hard drives were not originally made for cloud storage usage, so the drives had to be optimized in order to be used in cloud centers. What drive manufacturers should do is to recognize that data center disks are part of a large collection of disks on which performance improvement should be focused rather than standalone usage.

Bibliography

[1] How to choose between an internal hard drive and an external hard drive. https://getprostorage.com/blog/how-to-choose-between-an-internal-hard-drive-and-an-external-hard-drive/, 2016. Accessed on 17.2.2021.

[2] Understanding flash: Blocks, pages and program / erases. https://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases/. Accessed on 6.1.2021.

[3] https://www.samsung.com/.

[4] Alibaba Group. In pursuit of optimal storage performance: Hardware/software co-design with dual-mode ssd.

[5] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 471–484, 2014.

[6] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[7] What is alibaba? https://graphics.wsj.com/alibaba/. Accessed on 5.2.2021.

[8] Yazhi Du, Jihua Gu, Zhongzhe Xiao, and Min Huang. Ssw: A strictly sequential writing method for open-channel ssd. *Journal of Systems Architecture*, 109:101828, 2020.

[9] Zoned storage for the zettabyte era. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/product-brief/product-brief-ultrastar-dc-zn540.pdf. Accessed on 19.2.2021.

[10] Rms-350. https://www.radianmemory.com/u-2-ssd-rms-350/. Accessed on 19.2.2021.

[11] Nvme zoned namespaces explained. https://www.anandtech.com/print/15959/nvme-zoned-namespaces-explained. Accessed on 17.12.2020.

[12] Bjorling Matias. From open-channel ssds to zoned namespaces. In *USENIX VAULT*, 2019.

[13] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 83–90, 2018.

[14] Hard disk drives. http://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf. Accessed on 7.2.2021.

[15] Understanding flash: Blocks, pages and program / erases. https://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases/.

[16] Understanding ssd over-provisioning (op). https://www.kingston.com/unitedstates/en/ssd/overprovisioning/. Accessed on 6.2.2021.

[17] Ssd controller. https://www.storagereview.com/ssd-controller. Accessed on 27.1.2021.

[18] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. http://www.cs.binghamton.edu/~tameesh/pubs/systor2015.pdf, 2015.

[19] Lite-on open channel ssd. https://www.youtube.com/watch?v=okz--PP489M. Accessed on 13.12.2020.

[20] Matias Bjørling et al. Open-channel solid state drives. *Vault, Mar*, 12:22, 2015.

[21] Jonathan Corbet. Taking control of ssds with lightnvm. https://lwn.net/Articles/641247/, 2015.

[22] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[23] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel ssd (what is it good for). In *CIDR*, 2020.

[24] Hongwei Qin, Dan Feng, Wei Tong, Jingning Liu, and Yutong Zhao. Qblk: Towards fully exploiting the parallelism of open-channel ssds. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1064–1069. IEEE, 2019.

[25] Haitao Wang, Zhanhuai Li, Xiao Zhang, Xiaonan Zhao, Xingsheng Zhao, Weijun Li, and Song Jiang. Oc-cache: An open-channel ssd based cache for multi-tenant systems. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6. IEEE, 2018.

[26] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[27] Introduction to nand memories. https://developer.ridgerun.com/wiki/index.php/Introduction_to_Nand_Memories.

[28] Cnex: Lightnvm: The open channel ssd subsystem. https://static.googleusercontent.com/media/research.google.com/en//archive/bigtableosdi06.pdf.

[29] Javier González and Matias Bjørling. Multi-tenant i/o isolation with open-channel ssds. In *Nonvolatile Memory Workshop (NVMW)*, 2017.

[30] Steve Byan, James Lentini, Anshul Madan, Luis Pabon, Michael Condict, Jeff Kimmel, Steve Kleiman, Christopher Small, and Mark Storer. Mercury: Host-side flash caching for the data center. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[31] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40. IEEE, 2013.

[32] Western digital's ultrastar dc zn540 is the world's first zns ssd. `https://www.tomshardware.com/news/western-digitals-ultrastar-dc-zn540-is-the-worlds-first-zns-ssd`. Accessed on 19.2.2021.

[33] Radian's zns ssd first to pass zoned namespaces compliance testing. `https://apnews.com/press-release/pr-newswire/66a7470397043ffcc345c02da42a7380`. Accessed on 19.2.2021.

[34] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating host aware {SMR} drives. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[35] Zonefs - zone filesystem for zoned block devices. `https://www.youtube.com/watch?v=3U2YUQhgB80`. Accessed on 4.1.2021.

[36] Javier González. Zoned namespaces use cases, standard and linux ecosystem. `https://www.snia.org/sites/default/files/SDCEMEA/2020/3%20-%20Javier%20Gonzalez%20Zoned%20namespacese.PDF`, 2020.

[37] Zoned namespaces: Standardization and linux ecosystem. `https://www.youtube.com/watch?v=5hdUHSILkzA`, 2020. Accessed on 28.12.2020.

[38] Cold data storage. `https://www.komprise.com/glossary_terms/cold-data/`.

[39] Case study: Integration of salsa, a unified storagesoftware stack developed by ibm research, withradian's zoned flash ssd. `https://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases/`.

[40] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new lsm-style garbage collection scheme for {ZNS} ssds. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[41] Adnan Asad Vohra and Dr. Srividya P. Design and implementation of sequential read ahead in zoned namespaces for solid state drives. `https://ijisrt.com/assets/upload/files/IJISRT20MAY889.pdf`.

[42] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[43] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The case of femu: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, CA, February 2018.

[44] Tim Chan. A peek into the future of storage. `https://www.bhphotovideo.com/explora/computers/buying-guide/a-peek-into-the-future-of-storage2`.

[45] Disks for data centers. `https://storage.googleapis.com/pub-tools-public-publication-data/pdf/44830.pdf`, 2016.

[46] Anton Shilov. Seagate: Hard disk drives set to stay relevant for 20 years. `https://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases/`.

[47] Bernard Marr. How much data do we create every day? `https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/`.