



Towards lowering the overhead of Open-Channel SSD

Freysteinn Alfreðsson

Thesis of 60 ECTS credits
Master of Science (M.Sc.) in Computer Science

January 2019



Towards lowering the overhead of Open-Channel SSD

by

Freysteinn Alfreðsson

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

January 2019

Supervisors:

Dr. Gylfi Þór Guðmundsson, Supervisor
Adjunct Professor, Reykjavík University, Iceland

Dr. Philippe Bonnet, Co-Supervisor
Full Professor, IT University, Denmark

Committee Members:

Dr. Björn Þór Jónsson, Committee Member
Associate Professor, ITU, Denmark

Dr. Marcel Kyas, Committee Member
Assistant Professor, Reykjavík University, Iceland

Copyright
Freysteinn Alfreðsson
January 2019

Towards lowering the overhead of Open-Channel SSD

Freysteinn Alfreðsson

January 2019

Abstract

The revolutionary change that Solid State Disks (SSDs) have introduced to the storage space industry has added many new challenges for data-driven application developers when developing execution planners. Traditionally the application, operating system, and storage controller have all separately handled the storage access pattern optimization and scheduling. Despite successful use for decades, due to the predictable behavior of the spinning disks, finding similar guidelines for optimizing access to SSDs has proven much harder. The problem is in part due to their increased performance but mostly due to their unpredictable behavior and black box nature. The key to addressing this issue is to open up the black-box, exposing the internal complexity of maintaining a healthy SSD, such that the schedulers can take the actual cost of all operations into account in their planning. The first contribution of this thesis is a user-space library called LightNVM-Direct that uses the Open-Channel SSD protocol to expose the internals of a compliant SSD device. The direct communication allows LightNVM-Direct to bypass the kernel entirely and eliminate all of its overhead. This bypass leaves the optimization and scheduling solely in the hands of the developer of the data-driven user-space application. We used the uFLIP-OC benchmark to evaluate LightNVM-Direct using an SSD device called Dragon Fire Card. The results were inconclusive, however, as little gain was observed in overall throughput despite the lower kernel overhead. The second contribution is a proposal of a new protocol we called RNVMe, that adds near-data processing capabilities by allowing part of the application to run on the SSD in the form of Remote Procedure Calls (RPC). This RPC mechanism would enable applications to run simple functions, such as filters and data pre-processing, before delivering the final result to the host computer.

Umstangs lágmörkun á Open-Channel SSD

Freysteinn Alfreðsson

janúar 2019

Útdráttur

Solid State Diskar (SSD) hafa umbylt gagnageymsluíðnaðinum en hafa einnig kynnt til sögunar nýjar áskoranir fyrir forritara sem búa til aðgangsstýringar. Sögulega hafa forritin, stýrikerfið og diskarnir séð um að hámarka afköst gagnaaflesturs af diskum óháð hvert öðru. Síðustu áratugi hefur þetta fyrirkomulag fyrst og fremst gengið upp vegna þess hve reglubundin og fyrirsjáanleg hegðun hefðbundinna mekanískra diska er. Ekki hefur gengið jafn vel að finna sambærilega hlutverkaskiptingu fyrir nýju SSD diskana. Vandamálið er í raun þríþætt: Fyrst þurfti að uppfæra mikið af hugbúnaðinum þar sem að afkastageta diskanna varð fljótt mun meiri en nokkur hafði gert ráð fyrir; Mun alvarlegri vandi liggur hinsvegar í því að erfiðara er að spá fyrir um hegðun diskanna og einnig því að diskarnir eru í raun lokað kerfi sem enginn nema framleiðandinn veit nákvæmlega hvernig virkar. Þetta gerir það að verkum að aðgangsstýringarnar á efri lögum eiga erfitt með að spá fyrir um kostnað aðgerða og því er erfitt um vik að skipuleggja aðgangsstýringar. Open-Channel SSD er viðbót við NVMe staðalinn sem opnar á möguleika fyrir SSD framleiðendur að svipta hulunni af hegðun diska þeirra. Að opna á bein samskipti við diskana er ekki nóg til að leysa vandann því aðgangsstýringarnar á efri lögunum taka á sig bæði aukið flækjustig og aukna ábyrgð. Í þessari ritgerð eru lögð drög að því að lágmarka umstangið og kostnaðinn við að útfæra aðgangsstýringu í notendaforriti. Fyrsta framlag okkar er að klippa út stýrikerfið (kjarnann) með því að útfæra og prófa forritasafn sem við köllum LightNVM-Direct. Með beinum samskiptum milli notendaforrits og SSD disks getur LightNVM-Direct farið framhjá stýrikerfiskjarnanum og losað þannig við allan kostnað og umstang sem því fylgir. Þetta fyrirkomulag færir aðgangsstýringuna alfarið í hendur notendaforritsins, án afskipti hinna lagana. Við notuðum uFLIP-OC til að bera saman frammistöðu LightNVM-Direct og hefðbundari útfærslu sem kallast LightNVM. Seinna framlag ritgerðarinnar er tillaga að nýjum samskiptastaðli sem við köllum RNVMe, sem veitir örgagnavinnslu eiginleika á gagnageymslunni. Þetta er gert með því að leyfa hluta af forritinu að keyra á SSD gagnageymslunni sem Remote Procedure Call (RPC). Þetta gerir forritum kleift að kalla á einfaldar aðgerðir sem geta síað og forunnið gögnin áður en þeim er skilað til tölvunnar.

Towards lowering the overhead of Open-Channel SSD

Freysteinn Alfreðsson

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

January 2019

Student:

.....
Freysteinn Alfreðsson

Supervisors:

.....
Dr. Gylfi Þór Guðmundsson

.....
Dr. Philippe Bonnet

Committee Members:

.....
Dr. Björn Þór Jónsson

.....
Dr. Marcel Kyas

The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this Thesis entitled **Towards lowering the overhead of Open-Channel SSD** and to lend or sell such copies for private, scholarly or scientific research purposes only. The author reserves all other publication and other rights in association with the copyright in the Thesis, and except as herein before provided, neither the Thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

.....
date

.....
Freysteinn Alfreðsson
Master of Science

Contents

Contents	xi
List of Figures	xiii
List of Tables	xv
List of Abbreviations	xvii
1 Introduction	1
2 Background	3
2.1 Non-Volatile Memory Express (NVMe)	3
2.1.1 DMA memory	4
2.1.2 NVMe queues	4
2.1.3 The queue structure and notification doorbells	5
2.1.4 NVMe commands	6
2.2 User-Space NVMe	6
2.2.1 Requirements	7
2.2.2 Available user-space NVMe libraries	10
2.2.3 The NVMeDirect framework	11
2.2.4 Drawbacks of the Kernel bypass	12
2.3 Solid State Disk (SSD)	13
2.3.1 Flash Translation Layer (FTL)	14
2.3.2 Open-Channel SSD	15
2.3.3 liblightnvm	15
2.3.4 Dragon Fire Card	16
2.3.5 OX Controller	17
2.4 Benchmarking	19
2.4.1 uFLIP	19
2.4.2 uFLIP-OC / Fox	19
3 Debugging Framework	21
3.1 OX run-time command line	21
3.2 Discussion	24
3.2.1 Future Work	24
4 LightNVM-Direct: A liblightnvm Kernel Bypass	27
4.1 Architecture	27
4.1.1 Changes to liblightnvm	27
4.1.2 Changes to NVMeDirect	28

4.1.3	Control Path	29
4.1.4	Data Path	29
4.2	Experiments with LightNVM-Direct	29
4.2.1	Experimental Setup	30
4.2.2	Evaluation 1: Context-switching	30
4.2.3	Evaluation 2: Read Throughput	31
4.2.4	Evaluation 3: Write Throughput	32
4.2.5	Evaluation 4: Thread Throughput	32
4.2.6	Evaluation 5: Latency	34
4.3	Discussion	35
4.3.1	Sources of Complexity	36
4.3.2	Future Work	36
5	AppNVM	39
5.1	Definition and Categorization of AppNVM	40
5.1.1	NVMe Rule Engine (NVMe RE)	40
5.1.2	Thin-AppNVM	40
5.1.3	NVMe Remote Procedure Calls (NVMe RPC)	41
5.1.4	NVMe Inter-Process Communication (NVMe IPC)	41
5.1.5	Full AppNVM	42
5.2	RNVMe: Our proposal for a NVMe RPC protocol	43
5.2.1	The RNVMe command	43
5.3	OX RNVMe Architecture	45
5.3.1	The OX RNVMe Command Handler	45
5.3.2	The OX RNVMe Modular Backend	47
5.4	The AppNVM Upload Protocol	48
5.5	Security	50
5.5.1	Preventing malicious DMA transfers from the host	50
5.5.2	Preventing malicious uploads of AppNVM	50
5.6	Discussion	51
5.6.1	Future Work	51
6	Conclusion	53
	Bibliography	55

List of Figures

1.1	Overview of the communication pathway of the I/O traffic from user-space applications down to the storage device(s). In figure (a) we see the standard I/O traffic passing through the operating system's API. In (b) we see how the Kernel can be bypassed to reduce overhead. Finally, in (c) we depict how we can delegate part of the application logic to the storage device for near data processing.	2
2.1	NVMe queues are DMA buffers used by the host to submit commands to the NVMe device. The queues are circular buffers, where the submission queue is used by the host to submit commands, and the completion queue is used to notify the host of completed commands. The host keeps track of the tail pointer of the submission queue, and the NVMe device keeps track of the head pointer of the completion queue.	5
2.2	NVMe devices have one admin queue which the host uses to create submission and completion queues. The queues are DMA memory buffers which the host uses to send I/O commands to the NVMe device. This Figure is based on a Figure by Picoli [5]	5
2.3	The NVMe command entries written to the submission queue by the host are 64-bytes, or 16 double words (DWORDs), where all NVMe commands have the same format for the first two DWORDs. However, the opcode decides the format of the rest of the DWORDs.	8
2.4	NVMe completion entries are written by NVMe device to the completion queue. These entries are 32-bytes where all DWORDs follow the same standard format.	8
2.5	NVMeDirect <i>proc</i> file-system hierarchy is used by the user-space application to create new NVMe queues and to map them into its virtual memory address space using <i>ioctl</i> function calls.	11
2.6	Open-Channel SSD NVMe Command structure has the same structure for read, write, and erase commands. The PRP entries are pointers to either data to be read or written to the NVMe device depending on the type command submitted. The PPA list contains the list of NVMe physical addresses which the command is going to operate.	16
2.7	All of our work and experiments were conducted on the Dragon Fire Card. . . .	17
2.8	This Figure shows the internals of the OX controller as depicted by Picoli [5]. The OX Controller is the first open source Open-Channel SSD firmware controller. OX supports multiple types of FTLs implementations and different types of media managers for different types of storage technologies.	18
3.1	OX command line showing the help system	23

4.1	We separated the control and data paths in our implementation of NVMeDirect. This separation enabled us to use the <i>nvme</i> driver for admin commands and to use a kernel bypass for I/O commands.	28
4.2	Both figures show context switches through all the experiments in logarithmic scale, where (a) is the context switches using the standard <i>liblightnvm</i> and (b) uses our kernel bypass, <i>LightNVM-Direct</i>	30
4.3	The bar charts show combined experiments using 100% read throughput of pattern engines 1, 2, and 3, for both Standard and Direct experiments.	32
4.4	The bar charts show write throughput using pattern engine 1 using 100% write operations.	33
4.5	The bar charts show combined experiments using 100% write throughput using pattern engines 2 and 3.	33
4.6	The area chart shows the throughput of 8 threads during 100% write using pattern engine 2 with and without a kernel bypass.	34
4.7	The area chart shows the throughput of 16 threads during 100% write using pattern engine 2 with and without a kernel bypass. The chart is more erratic in (b) because the thread count is higher than the number of CPU cores.	34
4.8	The scatter plot shows the latency of writes during experiments using pattern engine 2 running with eight threads.	35
4.9	The scatter plot shows the latency of writes during experiments using pattern engine 2 running with sixteen threads. Writes in (b) show multiple outliers that have up to an order of magnitude worse latency than in (a).	35
5.1	RNVMe is an NVMe RPC protocol that allows the application programmer to call subroutines that are capable of NDP on the NVMe device. By moving part of the application specific functionality onto the NVMe device, we are not only bypassing the kernel but the overhead of the PCIe layer as well.	39
5.2	The proposed RNVMe command identifies the RPC function using the combined values of the vendor and the function fields. The rest of the fields are used for parameters to the function and return values. The parameters and return values can either be a 64-bit value or a pointer to a DMA buffer.	45
5.3	This Figure shows the internals of the OX controller as depicted by Picoli [5]. It has been modified to show the design change needed to OX to add a new RPC backend that is capable of submitting multiple I/O requests to the internal queues which are not visible to the host. Each RPC command can result in this backend submitting and completing multiple commands to complete an RPC NVMe and sending a completion entry back to the host.	46

List of Tables

2.1	This table lists the most common NVMe admin commands. It is only possible to send these commands to the admin queue.	6
2.2	The NVMe BAR is the main interface for interacting with the NVMe device. The ASQ and ACQ are used to map the Admin submission and completion queues to DMA buffers, and the doorbell registers from addresses 1000h and upwards are used to notify the NVMe device of new NVMe commands and handled completed commands by the host.	7
2.3	This table lists the most common NVMe commands. It is only possible to submit these commands to I/O submission and completion queues.	8
2.4	The user-space application uses these NVMeDirect <i>ioctl</i> attributes when it interacts with the <i>/proc/nvmed/nvmeon1/admin</i> file.	12
2.5	Whenever the user-space application creates a new queue using the NVMeDirect kernel module, it creates a directory in the <i>proc</i> file-system with the queues identification number using the <i>sq</i> , <i>cq</i> , and <i>db</i> files. These files are then used by the user-space application to map the PCIe registers and submission and completion queue DMA buffers into the applications virtual address space. . . .	12
2.6	New <i>liblightnvm</i> backends need to implement the interface listed in this table. It is the core interface used by the <i>liblightnvm</i> framework to interact with its backends and therefore the NVMe device.	16
3.1	The table lists the commands in the main category of the debugging command line.	22
4.1	The table lists all parameters and values related to our experiments that we ran with Fox. We skipped all the combinations where the threads were less than channels multiplied by LUNs. These settings resulted in 270 different experiments in total.	29
4.2	Experiments are labeled Direct if they use LightNVM-Direct and Standard if they use the traditional <i>liblightnvm</i>	29
4.3	The table shows the latency statistics for the experiments in Figures 4.8 and 4.9. The experiments used pattern Engine 2 with 100% writes using eight and sixteen threads.	36
5.1	The RNVMe command is to support calling a function on the NVMe device with multiple parameters. These parameters can either be DMA buffers or 64-bit numbers.	44

List of Abbreviations

AMD-Vi	AMD's I/O Virtualization Technology
ASIC	Application-Specific Integrated Circuit
AppNVM	Application NVMe
BAR (PCIe)	Base Address Register
CQ	Completion Queue
DFC	Dragon Fire Card
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
ECC	Error-correcting code
FPGA	Field-Programmable Gate Array
FTL	Flash Translation Layer
Intel VT-d	Intel's Virtualization Technology for Directed I/O
LBA	Logical Block Address
LUN	Logical Unit
MLC	Multi-Level Cell (NAND flash)
MTRR	Memory Type Range Register
NDP	Near-Data Processing
NVM	Non-Volatile Memory
NVMe	NVMe Express
NVMf	NVMe Over Fabrics
OC-NVMe	Open-Channel SSD NVMe
OOB	Out-Of-Band
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
PPA	Physical Page Address
PRP	Physical Region Page
RPC	Remote Procedure Call
SAS	Serial Attached SCSI
SATA	Serial ATA
SCSI	Small Computer System Interface
SPDK	Storage Performance Development Kit
SQ	Submission Queue
SSD	Solid State Disk
SoC	System on a Chip
UIO	User-space I/O
VFIO	Virtual Function I/O
XFI	(10 Gigabit Small) Form Factor Pluggable
uFLIP	Understanding Flash I/O Patterns

Chapter 1

Introduction

Data-driven application developers have been facing new challenges with today's revolutionary changes in storage technologies and the growing need for a storage of vast quantities of data. Traditionally, three separate layers have been responsible for storage access scheduling. The application, the operating system, and the on-storage controller of the storage device. Data-driven applications are designed to predict ahead of time their extensive access patterns with carefully designed execution planners. Operating systems and storage controllers, on the other hand, lack such foresight and are designed to deal with general access patterns and to make ephemeral plans. With the robust performance of new storage technologies, it has become apparent to data-driven application developers that these shortsighted schedulers have both become a bottleneck and have started to counteract the applications execution planners. Traditional database systems have been dealing with such issues for decades but what has changed in recent years is the sophistication of the devices and their storage controllers.

The most prevalent storage technology is the Solid State Disk (SSD) which has been phasing out the traditional spinning magnetic disks that dominated the market for decades. SSDs have introduced orders of magnitude better performance and have virtually eradicated the penalty of random access patterns. Despite these salient features, it has proven to be hard for data application developers to optimize for the SSDs, due mostly to the apparent differences in both physical characteristics and on-device scheduler implementations. These differences vary wildly between both vendors and SSDs models and therefore no common optimization techniques have been established. The significant shift from the previous technology is not only due to the SSD having more complex storage logic and more advanced capabilities, but also in the fact that the advancement in cheap chipset technologies has allowed the enterprise SSDs themselves to take on the characteristics of a fully functional computer.

Various solutions to these performance bottleneck problems have been explored [1], such as eliminating operating system overhead by bypassing the operating system kernel and communicating directly with the SSD from the application [2] [3]. Another solution was to expose the internal characteristics of the SSD by creating a new type of SSD called Open-Channel SSD [2] [4]. This new SSD allowed the application developer to directly control the on-device scheduling, effectively making the application's execution planner able to optimize fully down to the device level.

In our research, we have explored two avenues of optimizing the I/O access. First, we implemented and evaluated bypassing the operating system, allowing a user-space application to directly access the storage device or device controller via the Open-Channel SSD standard. Second, we propose a new way of doing near data processing (NDP) by moving part of the application logic to the storage medium.

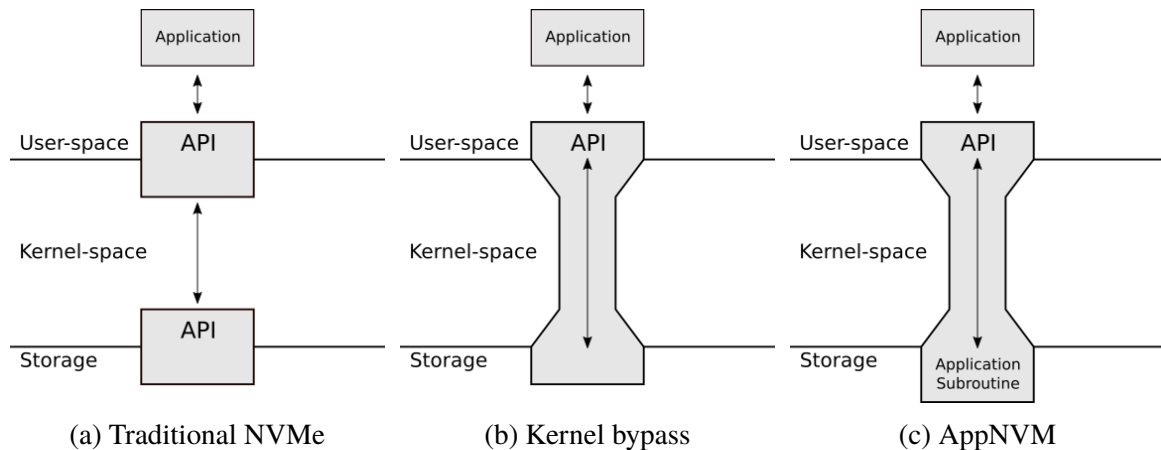


Figure 1.1: Overview of the communication pathway of the I/O traffic from user-space applications down to the storage device(s). In figure (a) we see the standard I/O traffic passing through the operating system’s API. In (b) we see how the Kernel can be bypassed to reduce overhead. Finally, in (c) we depict how we can delegate part of the application logic to the storage device for near data processing.

The three parts of Figure 1.1 present an overview of the topics addressed in this thesis. The traditional enterprise SSDs use the NVMe communication protocol which exposes the SSD as a block device and conceals its physical characteristics, where it is the vendors’ responsibility to make sure that the flash chips wear evenly and to utilize the internal parallelism of the SSDs according to their access pattern scheduler.

In our work, we used an Open-Channel SSD which extends the NVMe standard and exposes the internal characteristics of the device and moves the burden of wear leveling and access pattern scheduling to either the operating system or the application. Figure 1.1a shows the traditional way of communicating with an SSD where all communication from the application goes through the operating system kernel. Our contributions in this thesis as follows: First is the LightNVM-Direct, a kernel bypass for open-channel SSD. We also wrote a debugging framework to make this work easier. The second contribution is the AppNVM is a remote procedure call (RPC) system that will allow near data processing (NDP) to be done on the storage device.

The rest of this thesis is structured as follows: In Chapter 2 we cover the background material that we build our work upon, such as the storage technology (SSD), the communication protocols (NVMe) and our evaluation tools. This is followed by a short description of the debugging framework we developed in Chapter 3. In Chapter 4 we describe our first major contribution, the LightNVM-Direct bypass of the Kernel. In Chapter 5 we define our second contribution, the AppNVM. AppNVM is a remote procedure call (RPC) system that will allow near data processing (NDP) to be done on the storage device. Finally we end this thesis by drawing conclusions in Chapter 6.

Chapter 2

Background

In this chapter, we will go through the background needed for our contributions of bypassing the kernel for Open-Channel SSD based NVMe devices, and for our proposal of a new NVMe based protocol to minimize data movement by moving part of the application logic to the NVMe device. We have structured our background into four different parts.

1. NVMe: We start with the NVMe protocol, the main SSD protocol used today. NVMe is designed to incorporate the high parallelism that SSDs are capable of, as well as their high throughput, and thus, NVMe resides directly on the CPU's main bus for maximum performance.
2. User-space NVMe: After introducing the internals of NVMe, we examine what is needed to communicate with an NVMe device directly from a user-space application. We also review the limitations of bypassing the operating system kernel, and the solutions that already exist to provide a bypass for NVMe devices.
3. SSD: Next we review the internals of the SSD, highlighting the challenges imposed on storage developers and how Open-Channel SSD was designed to help deal with those challenges. We describe the Dragon Fire Card (DFC), a fully programmable NVMe device that we used for our implementation of the kernel bypass and our benchmark experiments, as well as its companion firmware OX, the first open source Open-Channel SSD firmware, which we ran on the DFC NVMe.
4. Benchmarking: Lastly we introduce the uFLIP-OC benchmark which is a benchmark for to identify the I/O patterns best suited for a given Open-Channel SSD. In our work we used a tool called *fox* which is capable of implementing the uFLIP-OC benchmark to evaluate our results. We compared our implementation to the traditional one by compiling two versions of the *fox* program, one using our kernel bypass, and another using the traditional backend. We ran the same benchmark on both of versions of *fox* and to evaluate our results.

2.1 Non-Volatile Memory Express (NVMe)

The NVMe communication protocol is the main SSD protocol used today. It was designed to replace the traditional spinning disks protocols SATA and SAS. Unlike the previous protocols, the NVMe protocol communicates directly through the Peripheral Component Interconnect Express (PCIe) bus, which is a high-speed serial computer expansion bus standard used to communicate with devices attached to the host computer.

From a system programmer's perspective, the programmer can read and write directly to the PCIe device registers using regular memory addresses. This memory representation is made possible by the I/O memory management unit (IOMMU) which is responsible for mapping the PCIe device registers to physical memory addresses. The PCIe registers of a device mapped to physical memory are together called the PCI Configuration Space. This configuration space is standardized to make it easy for Operating Systems to identify the device and for drivers to find their driver specific registers, called the Base Address Registers (BARs). Each BAR points to a region of device-specific registers which represent the main interface to the device's functionality used by driver programmers.

For our contribution, we only need to focus on the most relevant BAR register. The NVMe standard defines a mandatory memory layout, pointed to by the main BAR that all NVMe devices must implement. This layout contains special registers that manage queues that are used to send administration and I/O commands.

2.1.1 DMA memory

Direct Memory Access (DMA) is a mechanism to offload work from the CPU by allowing the device to access the physical memory of the host independently from the CPU. In modern high-speed devices, it is used to transfer data to and from the main physical memory and the device. When either the device or the CPU has prepared the data into a DMA buffer, they can notify each other that the data is ready for consumption. The CPU sends notifications by writing into a notification register on the device, while the device signals the CPU by signaling an interrupt.

2.1.2 NVMe queues

The NVMe standard's primary means of communicating with the NVMe device is through two queue types called submission and completion queues. The host uses the submission queue to send NVMe commands to the NVMe device, and the NVMe device uses the completion queue to notify the host of completed NVMe commands. Both of these queues are DMA buffers and are circular lists for NVMe commands, as seen in Figure 2.1.

Since the queues are a communication abstraction that does not present any physical characteristics of the device, the driver programmer can create as many queues as needed. This decision could be one queue per CPU core or one queue per thread, depending on the requirements and design. In our design, we decided to create one submission and completion queue per pthread.

2.1.2.1 Admin queue

The admin queue is a particular queue which is only used to send administrative commands to the NVMe device and is the only queue that is always available. This queue is identical to every other queue, apart from having the lowest identity number, zero.

One of the primary functions of the admin queue is to create a new submission and completion queues. These queues are created by submitting the create submission queue and the create completion queue commands. Table 2.1 lists most of the standard administration commands and Figure 2.2 shows the communication flow through those queues.

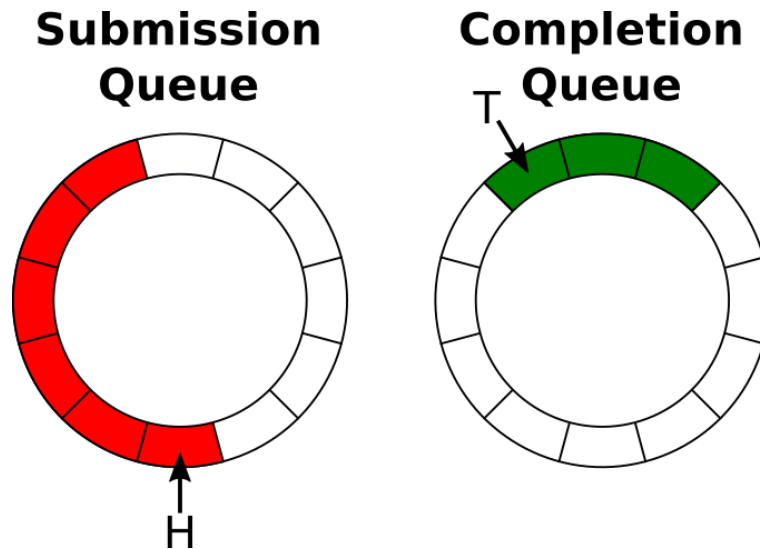


Figure 2.1: NVMe queues are DMA buffers used by the host to submit commands to the NVMe device. The queues are circular buffers, where the submission queue is used by the host to submit commands, and the completion queue is used to notify the host of completed commands. The host keeps track of the tail pointer of the submission queue, and the NVMe device keeps track of the head pointer of the completion queue.

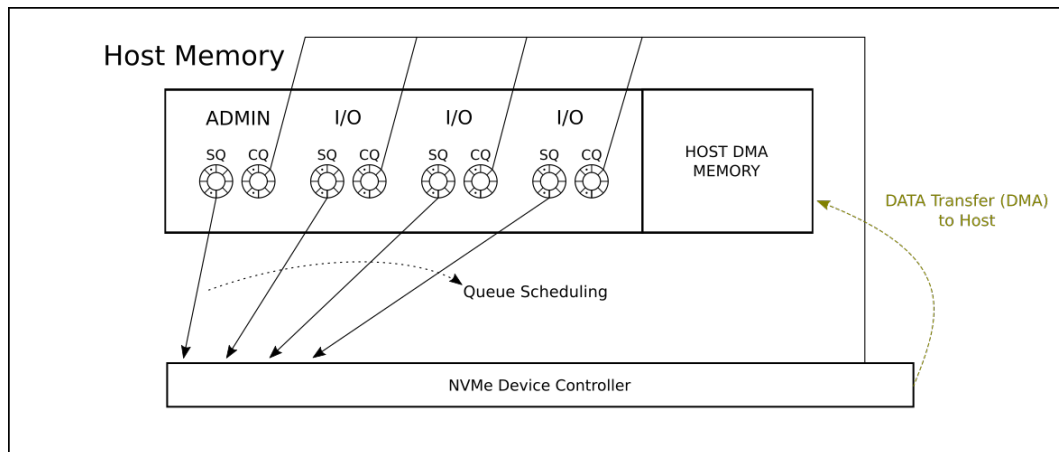


Figure 2.2: NVMe devices have one admin queue which the host uses to create submission and completion queues. The queues are DMA memory buffers which the host uses to send I/O commands to the NVMe device. This Figure is based on a Figure by Picoli [5]

2.1.3 The queue structure and notification doorbells

The submission and completion queues are made up of two distinct parts: the DMA buffer and its associated register in the NVMe BAR. Table 2.2 shows the two registers, the submission queue tail doorbell, and the completion queue head doorbell, which are owned by each queue in the BAR. These doorbells are used to notify the SSD when the host has either inserted new entries into the submission queue DMA buffer or processed one or more completion entries from the completion queue DMA buffer. The host notifies the NVMe device about changes by writing the location of the head and tail pointers to their respective registers as seen in Figure 2.1.

Opcode	Command
00h	Delete I/O Submission Queue
01h	Create I/O Submission Queue
04h	Delete I/O Completion Queue
05h	Create I/O Completion Queue
06h	Identify
09h	Set Feature
0Ah	Get Feature

Table 2.1: This table lists the most common NVMe admin commands. It is only possible to send these commands to the admin queue.

The SSD shares the DMA queues with the host when the host sends the size and the physical address pointer to the queues using the create submission queue and create completion queue commands. These commands make sure that both the host and the NVMe device agree on the DMA buffers.

2.1.4 NVMe commands

The NVMe standard defines 64-byte command entries for the submission queue and 16-byte completion queue entries. The standard further breaks down those bytes into 16-bit words; however, most written work and code refers to double words or DW for short.

Figure 2.3 shows the structure of a typical NVMe submission command. All commands share the same format for the first two DW. The opcode field tells the NVMe device which command is being submitted and in turn what fields to expect in DW four to thirteen. Table 2.3 lists the three most common NVMe commands.

The completion entries are 16-bytes or four DW and contain five fields, as shown in Figure 2.4. The status field holds both the return value of a completed command and a phase bit. This phase bit is used to tell the driver if the completed command is an old command from the last circulation, and is flipped by the NVMe device every time it needs to move the head pointer to the beginning of the circular DMA buffer.

2.2 User-Space NVMe

A popular method to improve the performance of high-speed devices has been to bypass the kernel and write directly to the device from user-space, as was depicted in Figure 1.1b. This method starts to become beneficial when the round-trip time of an NVMe command is less than a context switch quantum. Bypassing the kernel was first studied on network devices when their throughput exceeded the capacity of the CPU and has now been explored by researchers on SSDs for the same reasons. An excellent example of this is the NVMeDirect library developed by Kim, Lee, and Kim, which we decided to base our bypass on, and the open source solutions developed outside of academia, such as Intel’s SPDK [4] [6] and Micron’s unvme [7].

Start	End	Symbol	Description
00h	07h	CAP	Controller Capabilities
08h	0Bh	VS	Version
0Ch	0Fh	INTMS	Interrupt Mask Set
10h	13h	INTMC	Interrupt Mask Clear
14h	17h	CC	Controller Configuration
18h	1Bh	Reserved	Reserved
1Ch	1Fh	CSTS	Controller Status
20h	23h	NSSR	NVM Subsystem Reset (Optional)
24h	27h	AQA	Admin Queue Attributes
28h	2Fh	ASQ	Admin Submission Queue Base Address
30h	37h	ACQ	Admin Completion Queue Base Address
...	Optional and reserved fields
1000h	1003h	SQoTDBL	Submission Queue o Tail Doorbell (Admin)
$1000h + 1 \cdot (4 \ll CAP.DSTRD)$	$1003h + 1 \cdot (4 \ll CAP.DSTRD)$	CQoHDBL	Completion Queue o Head Doorbell (Admin)
$1000h + 2 \cdot (4 \ll CAP.DSTRD)$	$1003h + 2 \cdot (4 \ll CAP.DSTRD)$	SQ ₁ TDBL	Submission Queue 1 Tail Doorbell
$1000h + 3 \cdot (4 \ll CAP.DSTRD)$	$1003h + 3 \cdot (4 \ll CAP.DSTRD)$	CQ ₁ HDBL	Completion Queue 1 Head Doorbell
$1000h + 4 \cdot (4 \ll CAP.DSTRD)$	$1003h + 4 \cdot (4 \ll CAP.DSTRD)$	SQ ₂ TDBL	Submission Queue 2 Tail Doorbell
$1000h + 5 \cdot (4 \ll CAP.DSTRD)$	$1003h + 5 \cdot (4 \ll CAP.DSTRD)$	CQ ₂ HDBL	Completion Queue 2 Head Doorbell
...
$1000h + 2y \cdot (4 \ll CAP.DSTRD)$	$1003h + 2y \cdot (4 \ll CAP.DSTRD)$	SQ _y TDBL	Submission Queue y Tail Doorbell
$1000h + (2y + 1) \cdot (4 \ll CAP.DSTRD)$	$1003h + (2y + 1) \cdot (4 \ll CAP.DSTRD)$	CQ _y HDBL	Completion Queue y Head Doorbell

Table 2.2: The NVMe BAR is the main interface for interacting with the NVMe device. The ASQ and ACQ are used to map the Admin submission and completion queues to DMA buffers, and the doorbell registers from addresses 1000h and upwards are used to notify the NVMe device of new NVMe commands and handled completed commands by the host.

2.2.1 Requirements

For the driver to be considered as fully implemented in user-space the following requirements must be satisfied:

Opcode	Command
00h	Flush
01h	Write
02h	Read

Table 2.3: This table lists the most common NVMe commands. It is only possible to submit these commands to I/O submission and completion queues.

Offset	DWORD0			DWORD1
0	Opcode	Flags	Command ID	Namespace ID (NSID)
2	Reserved			
4	Command specific DWORD4			Command specific DWORD5
6	Command specific DWORD6			Command specific DWORD7
8	Command specific DWORD8			Command specific DWORD9
10	Command specific DWORD10			Command specific DWORD11
12	Command specific DWORD12			Command specific DWORD13
14	Reserved			

Figure 2.3: The NVMe command entries written to the submission queue by the host are 64-bytes, or 16 double words (DWORDs), where all NVMe commands have the same format for the first two DWORDs. However, the opcode decides the format of the rest of the DWORDs.

Offset	DWORD0	
0	Command Specific	
1	Reserved	
2	SQ Identifier	SQ Head Pointer
3	Status Field	P Command Identifier

Figure 2.4: NVMe completion entries are written by NVMe device to the completion queue. These entries are 32-bytes where all DWORDs follow the same standard format.

- **Map the PCIe BAR of the device into the process virtual memory in user-space:**
The PCIe BAR is the interface the host CPU uses to communicate with the PCIe device and is mapped in physical memory on CPUs that support memory mapped I/O. This mapping means that all I/O operations on those physical memory addresses will read and write directly into the PCIe device's registers. In order for a user-space program to communicate with a PCIe device, it must have full access to these memory mapped regions of memory.

This mapping can be done using a custom kernel module, or by using the libraries `uio` [8] and `VFIO` [9] (Virtual Function I/O). In our implementation, we used a custom kernel.

- **DMA memory:** A user-space program must be able to read and write to DMA buffers to be able to operate as a driver for a device. For this to work, the DMA physical memory pages must be locked in RAM for communication between the host and the device to make sure that the pages are not swapped to disk by the operating system. The locking can be accomplished using one of the following methods: a) the `mlock` POSIX function; b) `mmap` with the `MAP_LOCKED` flag; c) the use of huge-pages¹; or d) by marking the pages in the page tables as locked using a kernel module, which is the method we decided to use in our implementation.
- **Virtual to Physical address mapping:** PCIe devices use physical memory addresses for DMA buffers and therefore user-space processes, which only see virtual addresses, must have the capability to look up the physical addresses of these DMA buffers. Otherwise, neither the host nor the device would agree on where the DMA buffers are in memory.

Currently, the standardized ways to access the mappings are through the *proc* file-system or the VFIO² library. However, we decided to use the same method as NVMeDirect and use our custom kernel module to do the lookups.

- **Cache coherency:** CPU caches are paramount for all modern programs, but it is imperative to disable caching on top of PCIe memory mapped registers and DMA buffers. Otherwise, the host and the device will experience inconsistent behavior in their communication due to the CPU caching of old register values.

Disabling the cache can be done through user-space using the */proc* file-system by either setting special Memory Type Range Registers (MTRR) or in recent x86 CPUs by marking the pages in the page table as non-cached using the Page Attribute Table (PAT). Our solution disables the cache by marking the pages in the PAT in our kernel module.

- **Memory barriers:** Both modern compilers and CPUs reorder memory accesses to improve performance. While this is usually what we want, it can be problematic when the reads and writes are operating directly on the device's registers where the order of those read and write operations matter. It is critical for the user-space driver programmer to make sure that the order of the operations on the device registers happen in the intended order by adding memory barrier statements to the code. These statements tell the compiler or CPU to finish all memory operations before executing the next command. The programmer should also use the `volatile` keyword in C for register pointers to guarantee that the compiler does not optimize away the memory accesses.

To clarify, if the programmer omits the `volatile` keyword the compiler might deem writing to a memory location, that the code then never reads from, as unnecessary and would therefore not generate any machine instructions for it. Lacking these commands would leave us with a non-functioning program, even though the C code is correct. If on the other hand, we leave out the memory barrier commands, the instructions sent to the CPU might end up in a different order than defined by the C code or even assembly code. An example of this would be triggering some functionality of a device by writing to a particular register, before writing the parameters for that functionality in another register.

¹As of this writing, huge-pages are locked in physical memory and do not swap to the secondary storage.

²The `uio` library is not listed with VFIO because it does not support DMA memory management.

- **New API:** All user-space programs that offer a kernel bypass must provide a new API. This new API is both because the POSIX system call API is reserved and designed for traditional block devices, and because exposing the internals of the device gives the library designer the option of creating a robust API that is more fitting to the device-specific functionality. In our case, we decided to reuse the API of the *liblightnvm* library to make sure that our code would be compatible with existing software.
- **Polling or Interrupt-driven:** The driver may check whether the command has been successfully read or written using interrupts or by polling the device. User-space programs need help from the kernel to be able to receive interrupts. Currently, both *uio* and *VFIO* library offer standardized ways of handling interrupts in user-space. Our implementation only uses polling; however, the newest version of *NVMeDirect* does include support for interrupts.

2.2.2 Available user-space NVMe libraries

There are currently three different full user-space NVMe kernel bypass libraries: *SPDK*, *UNVMe*, and *NVMeDirect*. All of these implementations are limited to standard NVMe commands and do not support Open-Channel SSD. We decided to base our implementation on *NVMeDirect* and add support for Open-Channel SSD. We chose *NVMeDirect* because it can coexist with the *nvme* kernel driver and therefore makes it possible to use the *nvme* driver to process admin commands.

2.2.2.1 SPDK

The *SPDK* driver [4] [6] is a C library developed by Intel which provides a direct mapping to the NVMe device BARs and DMA buffers in user-space. It uses part of the Data Plane Development Kit (DPDK) [10] package to handle the internal plumbing and relies on huge-pages to map the PCIe BARs and DMA memory. The *SPDK* framework initially used the *uio* library to handle PCIe communication, but today it supports both *uio* and *VFIO* libraries at compile time.

The main selling point of *SPDK* is that it maximizes speed by not having any threads and that it only uses non-blocking calls. The drawback is however that the application has to poll to check if the I/O has completed.

2.2.2.2 UNVMe

UNVMe [7] is a user-space NVMe driver developed by Micron Technology and implements an interface to bypass the kernel. It is a library, and like the other user-space NVMe drivers, it provides an alternative API from the standard POSIX interface. The NVMe library uses the *VFIO* library to communicate with the PCIe device from user-space.

2.2.2.3 NVMeDirect

NVMeDirect [11] is a user-space NVMe that provides a direct user-space to NVMe device communication. Unlike *SPDK* and *UNVMe*, it uses a custom kernel module to expose the PCIe BARs and DMA buffers to the user-space library using the *proc* file-system.

One of the most significant selling points of *NVMeDirect* is that it piggybacks itself on top of the standard *nvme* kernel driver, making it possible to use both *NVMeDirect* and the standard *nvme* kernel driver concurrently on separate SSD partitions.

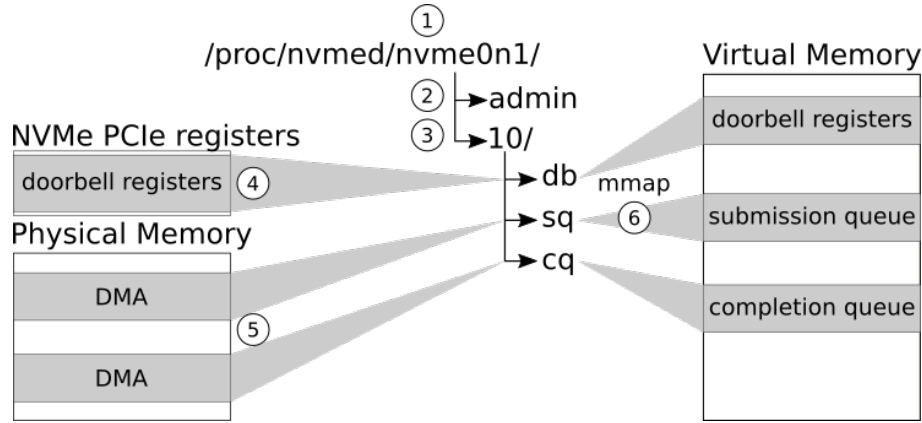


Figure 2.5: NVMeDirect *proc* file-system hierarchy is used by the user-space application to create new NVMe queues and to map them into its virtual memory address space using *ioctl* function calls.

Currently, the NVMeDirect is a polling solution, but the project has been working on adding interrupt support. However, we did not use the interrupt solution because it did not exist when we started working on our code.

The NVMeDirect user-space API designed exclusively with an NVMeDirect file descriptor interface in mind, and does not allow the programmer to write direct NVMe commands to the NVMe device. This design allows them to implement specific user-space I/O schedulers and plumbing for direct buffer handling. This high-level abstraction was, however, unnecessary for our implementation and therefore had to be removed entirely.

2.2.3 The NVMeDirect framework

The NVMeDirect framework relies on a kernel module to hand over control over the NVMe device into user-space. Figure 2.5 shows the steps that the user-space driver developer needs to be able to map and interact with the NVMe device. These steps are as follows:

1. When the NVMeDirect kernel module loads into memory, it creates the directory `/proc/nvmed` which has directories named after the NVMe devices that the kernel module finds. In the example given in Figure 2.5, it finds the first NVMe device, `nvmeon1`, and creates `/proc/nvmed/nvmeon1`.
2. When the user-space library interacts with the SSD, it does so by calling *ioctl* calls to the `/proc/nvmed/nvmeon1/admin` file. Table 2.4 lists the available *ioctl* calls to the `admin` file, where most are used to manage NVMe queues and to get physical addresses from allocated memory.
3. On creation of new NVMe queues using the *ioctl* function, the NVMeDirect kernel module creates a directory with the queue's identification number in the *proc* file-system.
4. This new directory contains the three files listed in Table 2.5, which allow the library to map the NVMe queue's PCIe BAR doorbell registers and queue DMA buffers to the process's memory.

ioctl	Description
NVMED_INFO	Get SSD information such as size.
QUEUE_CREATE	Create a submission and completion queue pair.
QUEUE_DELETE	Delete a specified submission and completion queue pair.
QUEUE_GET_BUFFER_ADDR	Returns the physical address of the given virtual address.
QUEUE_GET_USER	Returns the user's quota.
QUEUE_SET_USER	Sets the user's quota.

Table 2.4: The user-space application uses these NVMeDirect *ioctl* attributes when it interacts with the `/proc/nvmed/nvmeon1/admin` file.

ioctl	Description
sq	Used to mmap the submission queue DMA buffer to user-space.
cq	Used to mmap the completion queue DMA buffer to user-space.
db	Used to mmap the PCIe registers of the queue to user-space.

Table 2.5: Whenever the user-space application creates a new queue using the NVMeDirect kernel module, it creates a directory in the *proc* file-system with the queues identification number using the *sq*, *cq*, and *db* files. These files are then used by the user-space application to map the PCIe registers and submission and completion queue DMA buffers into the applications virtual address space.

5. The user-space application is responsible for DMA buffers by allocating memory using locked memory pages. NVMeDirect does this by using the *mmap* function with the *MAP_LOCKED* parameter.
6. The user-space application maps the doorbell registers, the submission queue, and completion queue into memory using the *mmap* function on the *db*, *sq*, and *cq* *proc* files inside the queue's directory. When the user-space program maps the queues, the kernel module allocates the DMA buffers for the queues and marks them as non-cached before mapping the addresses into the user-space programs virtual memory.

After these steps, the user-space application has a fully functional user-space bypass for the queues and does not need any further interaction with the kernel module to function.

2.2.4 Drawbacks of the Kernel bypass

While the direct user-space to PCIe device communication can improve performance, it does come with its list of drawbacks:

- **No kernel file-system:** Without the support from the kernel, we are not able to create any kernel file-systems on top of the user-space device.
- **No shared counters in the kernel:** Again, without the support from the kernel, we lose all shared counters and therefore all support from the tools that rely on those counters.
- **Harder to control access to multiple applications:** The user-space NVMe library has to control the access to the NVMe device of multiple applications. This access can be hard to control, both because the applications could be running as separate users

and because other software, such as kernel code, does not follow these access controls mechanisms.

- **Kernel-mode and user-space interference:** The Linux kernel is continuously in development, and new changes might interfere with the user-space drivers, both when it comes to how they behave on devices, and how they introduce new features in other areas, such as memory management.
- **Shared code between user-space and kernel-mode is hard:** Some code bases, such as NVMeDirect, share code with the *nvme* kernel module. While convenient, this means that with each release of a new Linux kernel the code must be updated, which is problematic when the direction of the *nvme* code is not following the other code bases. We experienced this and had to add functions that had been removed from the *nvme* driver to be able to use our implementation of NVMeDirect with newer kernels.
- **VFIO requires the CPU to support AMD-Vi or Intel VT-d:** VFIO needs direct I/O support from the CPU, this means that it can be hard to support it within virtual machines.
- **DMA virtual to physical mappings might break in the future:** Apart from VFIO, none of the other methods of mapping virtual memory addresses to physical memory addresses guarantee that the mappings will work in the future. New features that merge similar physical pages might change DMA physical addresses, and huge pages might at some point support swapping.
- **Pinned memory is locked memory:** The user requires a sufficiently high limit for locked memory and that the pages used will be prevented from page merging and swapping.
- **DMA control becomes insecure:** By giving the user-space application full control of the DMA buffers, a compromised application can read and write to anywhere in physical memory, allowing for DMA malware [12]. The infected application could instruct the NVMe command to write a page of physical memory which the application does not have access to down to the NVMe device, and then read it back into its memory.

2.3 Solid State Disk (SSD)

The advent of SSD devices has completely changed the rules for optimizing I/O access patterns and secondary storage performance. Unlike its predecessor, the spinning magnetic disk, an SSD uses flash chips as its storage medium, which is both much more tolerant of hostile environments and carries virtually no random access penalty. Despite the many advantages of SSDs they also have some drawbacks. One of the main challenges with SSDs is that write operations do not have predictable latency. The reason for this is that the flash chips have a certain number of times they can be erased and written to before they become permanently unusable. To mitigate this problem, traditional SSD storage controllers have a function called the Flash Translation Layer (FTL), which is responsible for wearing out the chips evenly through a process called wear-leveling. The implementation of the FTL is neither disclosed by the vendors nor is its behavior predictable.

Today's state of the art SSDs are connected directly to the PCIe bus and comply with the NVMe specification which defines a standardized way for SSDs to communicate over

PCIe. The design of the NVMe standard was for devices that are capable of hundreds of thousands of I/O operations per second. This immense change in I/O throughput and its effect on the Linux kernel was addressed by [13] where they showed that the kernel was only capable of handling a million I/O operations per second. To address this limitation, they introduced a new subsystem called “multi-queue” which improved the performance by an order of magnitude. While the new subsystem is an essential contribution to the Linux kernel, their research also exposed another limitation in the traditional user-space libraries and showed that it also caps in around one million I/O operations per second. This critical finding showed that the SSD is not just an evolutionary change from the traditional storage medium, but a revolutionary change where every part of the software stack is affected and needs a reevaluation of both algorithms and API conventions to optimize against this technology.

2.3.1 Flash Translation Layer (FTL)

The FTL is a module that exposes the SSD as a traditional block device that uses logical addresses to hide the internal physical details. This encapsulation makes it much easier for storage driver developers to interact with the SSD and keeps the responsibility of wear-leveling and internal performance optimization in the vendors hands.

2.3.1.1 Wear-Leveling

The process of wear-leveling is to wear out the flash chips evenly to prolong the life of the SSD. The FTL keeps track of all the logical and physical mappings and tries to write new pages to different locations every time the FTL writes to the chips. This rotation of writes means that there are multiple older versions of the same data that the FTL needs to periodically garbage collect by marking the old pages as free. Another common operation of the FTL is to swap less worn pages with worn pages. This operation is called write-amplification because it adds a steep overhead where one write operation takes considerably longer due to the FTL having to read and write the old page to the new location.

The FTL also needs to keep track of defunct flash chips both due to damages during production and due to chips slowly wearing out. For this reason, the SSDs always have more spare storage than accessible by the storage developer to aid with garbage collection and for spare storage for bookkeeping.

2.3.1.2 Scheduling and Performance Optimization

The SSD has multiple parallel units that the FTL needs to schedule read, write, and erase operations to. Channels are the main parallel units of the SSD, where each channel is a controller that is responsible for multiple chips. These chips have internal parallel units which are called planes, which the FTL is also able to utilize. When the FTL issues commands to the channels, it does so by writing the commands to the channel controllers registers. The main concern of the FTL is that write commands take much longer to complete than read commands, and therefore it will need to issue the commands in a strategic manner to maximize performance, while at the same time keeping track of wear-leveling.

2.3.1.3 Drawbacks

Initial research into how to optimize I/O patterns using the SSDs was performed by [14], where they created a benchmark called uFLIP, which they used to evaluate experiments of

different I/O workloads and how they affect I/O performance. The ultimate goal of this kind of research is to find optimization techniques for system and application programmers. Their findings were that SSDs have vastly different behavior characteristics, not only as a new technology but also between different SSDs. This vast difference included SSDs from the same vendor and even the same SSD after firmware updates. The main reason for this significant difference in behavior is the black box FTL unit which is responsible for the logical block interface.

2.3.2 Open-Channel SSD

A solution to the black box nature of FTL was proposed by Bjørling et al. with the creation of Open-Channel SSD [15] [16] which introduced new NVMe commands to the NVMe specification. The goal of the new commands was to allow direct physical placement of blocks by exposing the internals of the SSD using a new hierarchical physical page address space. Another contribution by Bjørling was a new subsystem in the Linux kernel called *lightnvm* [3] which leverages the new command set and offers new predictable white-box FTL implementations which are controlled by the host operating system. It also paved the way for application developers to take direct control of the physical placement of data blocks on the SSD and to tailor their I/O patterns to their application-specific needs [17].

The Open-Channel SSD command set defines a new hierarchical physical page address (PPA) space for the SSD which consists of channel, LUN, block, page, and sector, and offers commands that extract the dimensions of each part of the PPA. Another difference between the traditional logical address space is that the new command set also supports vector I/Os with up to 8 PPAs.

Figure 2.6 shows the structure of the Open-Channel SSD read, write, and erase commands. The read and write commands have four fields which represent pointers to DMA buffers, as can be seen in the figure. These pointers are defined as follows:

- **Metadata:** Used to store Out-of-Band (OOB) data on each flash page. The OOB is traditionally used by the FTL to keep track of wear-leveling and garbage-collection. This field, however, varies in size between hardware vendors and can be used as pleased by the application programmer.
- **PRP1:** Points to the first 4-KiB Physical Region Page (PRP) of the data that is to be read or written by the NVMe device.
- **PRP2:** Points to an continuous area of up to seven additional 4-KiB PRPs that are to be read or written. The amount of 4 KiB pages is controlled by the length field in double word 12.
- **PPA list:** Points to a buffer that contains a list of Physical Page Addresses (PPA) that are to be read, written, or erased. Where the number of PPAs match the number of PRPs.

2.3.3 liblightnvm

The Open-Channel SSD subsystem in the Linux kernel is called *lightnvm* and its companion user-space library is called *liblightnvm*. The *liblightnvm* framework relies on all the NVMe device communication to happen in the kernel by submitting low level Open-Channel SSD

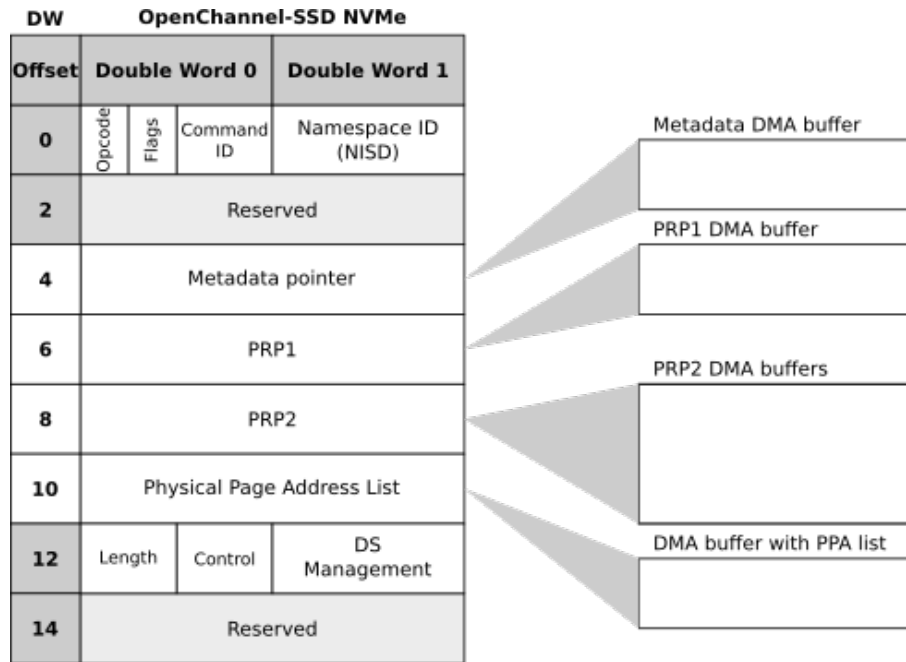


Figure 2.6: Open-Channel SSD NVMe Command structure has the same structure for read, write, and erase commands. The PRP entries are pointers to either data to be read or written to the NVMe device depending on the type command submitted. The PPA list contains the list of NVMe physical addresses which the command is going to operate.

Function	Description
open	Open a device
close	Close a device
user	Send a NVMe user command to device
admin	Send a NVMe admin command to device
vuser	Send a vectored NVMe user command to device
vadmin	Send a vectored NVMe admin command to device

Table 2.6: New *liblightnvm* backends need to implement the interface listed in this table. It is the core interface used by the *liblightnvm* framework to interact with its backends and therefore the NVMe device.

NVMe commands through an *ioctl* system call. While these *ioctl* NVMe commands are structured in a similar way as the Open-Channel SSD commands, they are placeholders that are remapped and copied to the actual DMA buffers by the kernel. Internally the NVMe commands are sent via the nvme kernel module, which uses both interrupts and polling.

2.3.4 Dragon Fire Card

The Dragon Fire Card (DFC) is a fully programmable storage device that we used to run our experiments. By flashing the controller on the DFC it can function as a NVMe device; however, we flashed it with an Open-Channel SSD controller for our experiments.

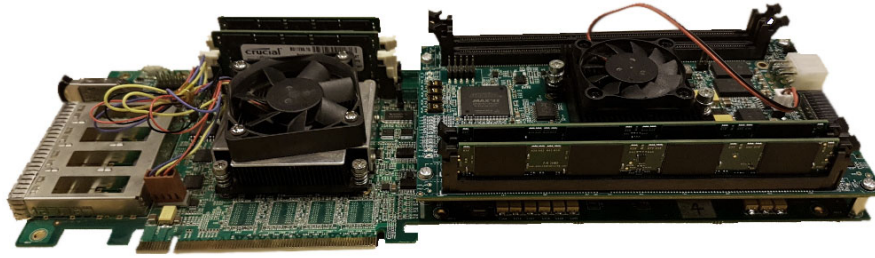


Figure 2.7: All of our work and experiments were conducted on the Dragon Fire Card.

The DFC consists of two different cards. The main card is an ARM based System-on-a-Chip (SoC) with 8 cores and 16-GiB of memory. The SoC is a PCIe device that can be directly plugged into the host computer.

The second card is a Field-Programmable Gate Array (FPGA) that contains the flash storage chips and is connected on top of the SoC board using PCIe. The FPGA is a standalone card which is removable from the primary card and can be replaced by any compatible PCIe card. This, however, has a severe limitation, as the DFC is only able to communicate with the host's PCIe bus using the FPGA. Therefore, it will not function with any other PCIe storage solution unless it also implements this type of custom bridge to the host computer.

2.3.4.1 Specifications

VVDN Technologies created the Dragon Fire Card (DFC) for FreeScale, which was later acquired by NXP. It is a SoC PCIe that is split into two Printed Circuit Boards (PCBs) cards, as can be seen in Figure 2.7. The primary PCB is the LS2085A SoC, which contains an eight-core 2 GHz ARM Cortex[®]-A57 64-bit processor and 16 GiB of ECC memory. The LS2085A has a 4xPCIe interface to the host computer but also contains its own isolated PCIe ecosystem which contains two 4xPCIe to the second PCB. The LS2085A also has four 10 Gbit XFI interfaces which provide the option of connecting either 10 Gbit Ethernet connectors with RDMA over Converged Ethernet (RoCE) or for adding 40 Gbit Infiniband support.

The secondary PCB is the primary storage card containing an FPGA and 4x DDR3/NAND DIMM connectors. The storage card contains 512 GiB of storage from two Micron MLC NAND that consist of two flash cards modules with four chips each, organized in 16 KiB pages, 512 pages per block, 2048 blocks per LUN spread in 2 planes, and 4 LUNs per chip.

The onboard operating system on the DFC card can be updated via an SD card reader on the LS2085A card. The DFC community has provided a Linux distribution using the Yocto Project, which is a Linux firmware generation tool-chain. It uses a build engine called bitbake which is based on the same design principles as *portage* in the Gentoo Linux distribution, where all the source files are downloaded and compiled each time the system is upgraded. This design gives us the flexibility to add our code and libraries to the distribution.

2.3.5 OX Controller

The OX Controller is an Open-Channel SSD firmware for the Dragon Fire Card written by Picoli [5], and it is the first open source LightNVM-enabled NVMe controller and is designed to execute I/O commands in parallel.

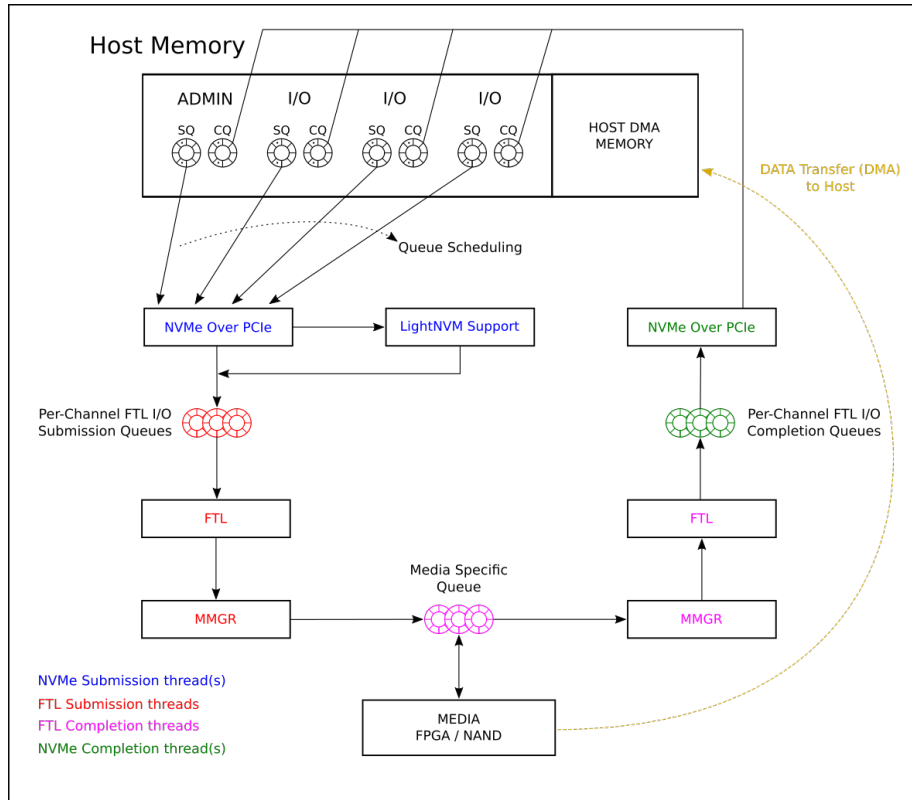


Figure 2.8: This Figure shows the internals of the OX controller as depicted by Picoli [5]. The OX Controller is the first open source Open-Channel SSD firmware controller. OX supports multiple types of FTLs implementations and different types of media managers for different types of storage technologies.

OX runs as an user-space program on the SoC’s on-board firmware Linux system. It interacts with the host machine by mapping part the PCIe Configuration Space and BARs that are exposed to the host into the OX user-space virtual memory.

OX also maps the PCIe BAR registers of the FPGA storage card into the OX virtual memory using a modified version of uio. This way OX acts as the gatekeeper between the host and the FPGA’s flash chips, similarly to how we bypass the kernel from the host. The OX controller is written in C and has a full kernel bypass for both the host facing PCIe interaction and to the FPGA’s PCIe BAR registers. It is able to get interrupts from the FPGA using device files presented by the uio³ support in the firmware Linux kernel. Another salient design of the OX controller is that it allows adding multiple types of controllers and even support for multiple types of storage backends. This means that it is simple for us to extend its functionality without sacrificing the old functionality. An overview of the design can be seen in Figure 2.8.

Internally the NVMe interconnect handler creates one pthread per submission and completion queue. It also has one pthread that handles the completed I/Os from the FPGA and puts them on the correct completion queue.

³The uio driver has been sloppily modified to support this particular FPGA and needs special care if the kernel is updated.

2.4 Benchmarking

To compare our Open-Channel SSD bypass with the traditional *liblightnvm* implementation we decided to use the uFLIP-OC benchmarking tool, called *fox*. This tool uses the *liblightnvm* library to communicate with the Open-Channel SSD and needed no changes to the code to work with our implementation of the *liblightnvm* library.

2.4.1 uFLIP

Initial research into how to optimize I/O patterns using the SSDs was performed by [14], where they created a benchmark called uFLIP, which they used to evaluate experiments of different I/O workloads and how they affect I/O performance. The ultimate goal of this kind of research is to find optimization techniques for system and application programmers. Their findings were that SSDs have vastly different behavior characteristics, not only as a new technology but also between different SSDs. This vast difference included SSDs from the same vendor and even the same SSD after firmware updates. The main reason for this significant difference in behavior is the black box FTL unit which is responsible for the logical block interface.

2.4.2 uFLIP-OC / Fox

A significant contribution from Picoli [5] was uFLIP-OC, a benchmark designed to identify the I/O patterns best suited for a given Open-Channel SSD. In his work he implemented the benchmark using a program he designed called *fox*, which is capable of dispatching Open-Channel SSD commands using the *liblightnvm* library. This program is a multi-threaded I/O generation tool where you can create benchmark(s) for different individual parallel units within the SSD, such as channels and LUNs. It comes with three different I/O engines which allow us to specify what type of I/O patterns we would like to perform on our selected regions.

Currently *fox* supports the following three I/O pattern engines:

- **Engine 1:** In this pattern engine each thread is assigned a geometry of channels and LUNs at start-up. Each thread then submits its I/Os to pages in sequence within a block, starting from page 0.
- **Engine 2:** Like in pattern engine 1, the channels and LUNs are assigned to threads at start-up, but in engine 2 the I/Os are submitted in round-robin over each parallel unit. In [5] it is shown that this pattern engine gives optimal performance.
- **Engine 3:** Pattern engine 3 uses the same I/O submission pattern as engine 2, but with the difference that each thread is now only responsible for either write I/O operations or read I/O operations.

Chapter 3

Debugging Framework

One repeated obstacle we encountered was the complexity of handling errors that came up either on the OX controller or in the interaction with it from the host. Some problems only occurred when the OX controller's debugging mode was disabled, which in turn made them considerably harder to spot. Another obstacle was that the default debugging output showed more detail than was relevant to our problem. This often made obvious problems unnecessarily hard to detect and diagnose.

A minor contribution we did was a new run-time command line for the OX controller to address the issue. This new addition was tailored after the command line of router equipment and significantly simplified the debugging process. Before our addition, a large fraction of debugging time went into running recompiled OX firmwares with `printf` statements with extensive C macros. These temporary solutions with the addition of using a debugger to see run-time status information used to end up in wasted work on the debugging code was never reused.

With the addition of the command line, we can now pick and choose the debugging options that we want at run-time, and we are able to create status commands which can easily be executed at any point in time. New commands can be added and safely published with future versions of OX, this can make them useful for all future users and developers. For convenience, the command line also includes both auto-completion on commands and an interactive help system which users are familiar with from other systems.

The OX run-time command line is written using the GNU readline library which offers built-in support for auto-completion. A simple help system was written using a hierarchy of C structs that contain and reflect the command structure. This includes both the help strings and a function pointer to the functionality behind the command. This hierarchy makes it simple to add commands that are automatically part of the help and auto-completion functionality without code duplication.

3.1 OX run-time command line

The new run-time command line was designed to be simple and accessible, especially to people that have had experience with a command line of networking equipment. It is designed with a command hierarchy in mind, where the user is able to choose a category and is able to run a command under that category.

The new run-time command line was published with OX version 1.4 and contained the commands listed in Table 3.1.

Command	Description
help	Used to print out help on categories and commands
admin	Category: Used for testing
admin create-bbt	An interactive command that creates or updates the bad block table
admin erase-blk	An interactive command that allows you to erase any specified block
debug	Category: Used to enable and disable debugging options
debug on	Enables all debugging output of admin and I/O commands
debug off	Disables all debugging output of admin and I/O commands
show	Category: Used to display run-time information
show debug	Shows if debugging information output is enables or disabled
show mq	Category: Displays run-time and statues information for multi-queue
show mq status	Shows the status of the internal queue groups of multi-queue
exit	Used to terminate the OX controller and get into a command line shell

Table 3.1: The table lists the commands in the main category of the debugging command line.

```

ox> help
help: Usage: help [command]
      Displays information about builtin commands.

      Displays brief summaries of builtin commands. If a command is
      specified, it will give a listing of all its sub-commands.

      List of sub-commands:
        admin:  Used for testing
        debug:  Enables or disables debugging output
        show:   Displays run-time information
        exit:   Exit the OX application
ox> help show
show: Usage: show [sub-command]
      Displays run-time information and status information.

      List of sub-commands:
        debug:  Shows if debugging mode is enabled
        mq:     Displays run-time information for multi-queue
ox> help show mq status
status: Shows the status of all the internal mq queues

      Displays run-time status of the internal queue groups of mq. This includes
      their name and status of each queue.
      Key  Description
      Q:   Queue number
      SF:  Submission Free queue - Available for new submission entries
      SU:  Submission Used queue - Ready to be processed
      SW:  Submission Wait queue - In process
      CF:  Completion Free queue - Available for new completion entries
      CU:  Completion Used queue - Processed, but waiting for completion
ox> show mq status
ox-mq: FTL_LNVM
      Q00: SF: 62, SU: 0, SW: 2, CF: 64, CU: 0
      Q01: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q02: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q03: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q04: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q05: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q06: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q07: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      EXT: 0, TO: 0, TO_BACK: 0
ox-mq: DFCNAND_MMGR
      Q00: SF: 60, SU: 0, SW: 4, CF: 64, CU: 0
      Q01: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q02: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q03: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q04: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q05: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q06: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      Q07: SF: 64, SU: 0, SW: 0, CF: 64, CU: 0
      EXT: 0, TO: 0, TO_BACK: 0
ox>

```

Figure 3.1: An example of how the help system in the OX command line would be used to show the status of the internal multi-queues.

Listing 3.1: File: include/ox_cmdline.h

```

typedef struct ox_cmd ox_cmd;
typedef int (*ox_cmdline_func_t)(char *line, ox_cmd *cmd);

typedef struct ox_cmd {
    char *command; /* Name of command */
    struct ox_cmd *next; /* Sub-commands */
    ox_cmdline_func_t func; /* Function to run */
    void *value; /* Default parameter to run */
    char *short_help; /* A short description */
    char *help; /* A long description */
} ox_cmd;

```

In Figure 3.1, we show both the run-time help in the command line, and run the “show mq status” command. This command prints out the status of the different queues that OX has internally.

All of the run-time commands are defined in a null terminated array of structs of type *ox_cmd*, as defined in listing 3.1. Each struct can be a category, a command, or both. In the case of a category, the next pointer points to another array of *ox_cmd* structs. For all commands the *func* function pointer points to a function that implements that particular command. For convenience, it also contains a void pointer to a default value used for the function pointer. This is helpful when you want to reuse the same function for multiple commands with different default values.

When the run-time command line runs the function pointer provided by the *ox_cmd* struct, it passes that same *ox_cmd* instance to the function, including the full command line string that was entered by the user. This makes it possible for the programmer to do extra parsing on the command if needed.

An example of how a category array of commands would be constructed in C is listed in listing 3.2. For clarity, the long help string is put together using multiple concatenated strings.

3.2 Discussion

The new OX command line makes it much easier to debug interaction with the OX controller and the host. It also adds the possibility to add more features that could be enabled at runtime. Some features that could be useful in the future are, to be able to set write protection on certain areas of the SSD, enabling a safe mode, disabling writes, and slowing down I/O commands for easier debugging.

3.2.1 Future Work

A limitation in the current implementation is that all commands have to have exact matches to the command strings in the *ox_cmd* struct. A better approach would be to change the command and category strings to regular expressions to allow for more flexibility for the current architecture. An alternative would be to introduce a full parser.

Listing 3.2: File: ox_cmdline.c

```
ox_cmd show_cmd[] = {
    { "debug",
      NULL,
      cmdline_show_debug,
      NULL,
      "Shows if debugging mode is enabled",
      "Shows if debugging mode is enabled\n"
      "\n"
      "Displays if reporting of live debugging information of NVMe commands\n"
      "is enabled."
    },
    { "mq",
      mq_cmd,
      NULL,
      NULL,
      "Displays run-time information for multi-queue",
      "Usage: show mq [sub-command]\n"
      "Displays run-time information and status information for multi-queue."
    },
    /* End with null termination */
    { NULL, NULL, NULL, NULL, NULL, NULL }
};
```

Chapter 4

LightNVM-Direct: A liblightnvm Kernel Bypass

Our motivation was to introduce direct control of the device in user-space and to increase the throughput of an Open-Channel SSD by introducing a kernel bypass that supports the Open-Channel SSD NVMe command set, as was depicted in Figure 1.1b. We wanted to be able to benchmark our bypass using the *fox* tool provided by Picoli [5]. The *fox* tool relies on the *liblightnvm* library, therefore it was easiest for us to add the new bypass as a backend to the *liblightnvm* library. This way we would not have to make any changes to the *fox* application and we could keep using the *liblightnvm* API.

4.1 Architecture

We chose to base our implementation on NVMeDirect for two reasons. First, because it was simple for us to modify, and second because it was capable of running along side the kernel *nvme* driver. This allowed us to separate the control and data paths by using the Linux kernel modules for admin commands and to use NVMeDirect for I/O commands. This design can be seen in Figure 4.1.

The *liblightnvm* library expects all new backends to implement the API functions listed in Table 2.6. One limitation that we encountered was that there was no initialization function and therefore no convenient way of passing parameters to our NVMeDirect backend. One parameter that would have been convenient to have is to notify the backend of how many submission and completion queues will be needed to initialize DMA buffers in advance. Instead, we decided to initialize the DMA buffers of each thread in the first submission of an I/O command and then reuse them throughout the duration of the process lifetime. The drawback of this approach is that the first I/O call will always be more expensive than the subsequent calls.

4.1.1 Changes to liblightnvm

The *liblightnvm* library that we used for our implementation had support for three different backends, *ioctl*, *sysfs*, and Logical Block Addresses (LBA). The *ioctl* and *sysfs* backends add the core support functionality for the *liblightnvm* where the *ioctl* part is used to send user-space defined NVMe commands, and the *sysfs* is to query the physical address dimensions. The LBA backend was not required for our implementation.

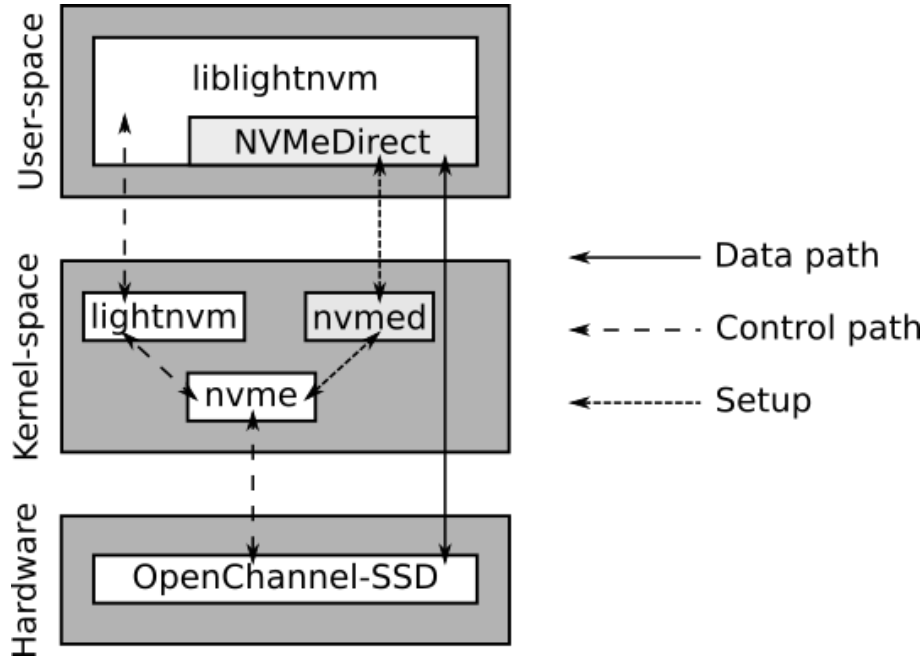


Figure 4.1: We separated the control and data paths in our implementation of NVMeDirect. This separation enabled us to use the `nvme` driver for admin commands and to use a kernel bypass for I/O commands.

A limitation of our version of `liblightnvm` was that all backends were loaded in sequence. This meant that if we added our backend, then it would have had to load all the other backends as well. In our version we had to manually remove the support for the older backends for ours to work. This was inconvenient, because this meant that we had to keep separate binaries of `fox` that were linked with different backends, instead of being able to choose the correct backend when `fox` was started. This limitation has been removed from the latest version of `liblightnvm`.

4.1.2 Changes to NVMeDirect

The original NVMeDirect is split into a kernel module and a user-space library. The user-space library was designed top-down with a new file descriptor API in mind. It therefore provided a new type of file handle API and a buffering interface designed around that API. Because our aim was to make this code a backend for the `liblightnvm`, we had to remove all of the buffering API code and rewrite most of the user-space code to adapt it to the `liblightnvm`. The reason why we were not able to use much of the existing code was because every part of the code relied heavily on the file-handle API without a clear segregation of the NVMe plumbing.

The NVMeDirect kernel module was designed to use the `nvme` kernel module to execute admin commands and therefore also to provide partitioning information of the SSD disk. Because the DFC card is only designed as an Open-Channel SSD, neither the `nvme` nor the NVMeDirect kernel module were able to see the DFC card. We therefore needed to change the code to get the dimensions of the disk through the `lightnvm` module's kernel data structures, and to make minor changes so that the rest of the code would check for the existence of the `lightnvm` information before proceeding.

Unit	List
Pattern engines	1, 2, 3
Threads	1, 2, 4, 8, 12, 16, 32
Channels	1, 2, 4, 8
LUNs	1, 2, 4
I/O	100% reads, 100% writes
Blocks	64
Pages	512

Table 4.1: The table lists all parameters and values related to our experiments that we ran with Fox. We skipped all the combinations where the threads were less than channels multiplied by LUNs. These settings resulted in 270 different experiments in total.

Label	Backend	Comment
Standard	<i>liblightnvm</i>	Uses the standard backend from [5]
Direct	<i>LightNVM-Direct</i>	Uses the LightNVM-Direct version of <i>liblightnvm</i>

Table 4.2: Experiments are labeled Direct if they use LightNVM-Direct and Standard if they use the traditional *liblightnvm*.

4.1.3 Control Path

We decided to keep the original control path that goes through the kernel using the *ioctl* call. This was done to simplify our design and because the kernel’s *nvme* driver would still be responsible for the admin queue of the SSD. Therefore, adding the admin queue to user-space as well would have resulted in incoherent behavior.

The way this was implemented was to retain the same code for the *liblightnvm admin* and *vadmin* functions that relied on the *ioctl* interface.

4.1.4 Data Path

The data path in our NVMeDirect backend completely bypasses the kernel, as seen in Figure 4.1. This was done by implementing our *user* and *vuser* functions to create a real NVMe command from the intermediate NVMe command C struct, and then writing it into the submission queue DMA buffer and then submitting it by writing directly into the NVMe BAR doorbell register of the DFC.

4.2 Experiments with LightNVM-Direct

We aimed to compare the performance between our kernel bypass provided by LightNVM-Direct to the traditional LightNVM implementation using the uFLIP-OC benchmark. We wanted to recreate the experiments conducted by Picoli [5] and to compare their latency and throughput to see if there was a noticeable performance difference between them.

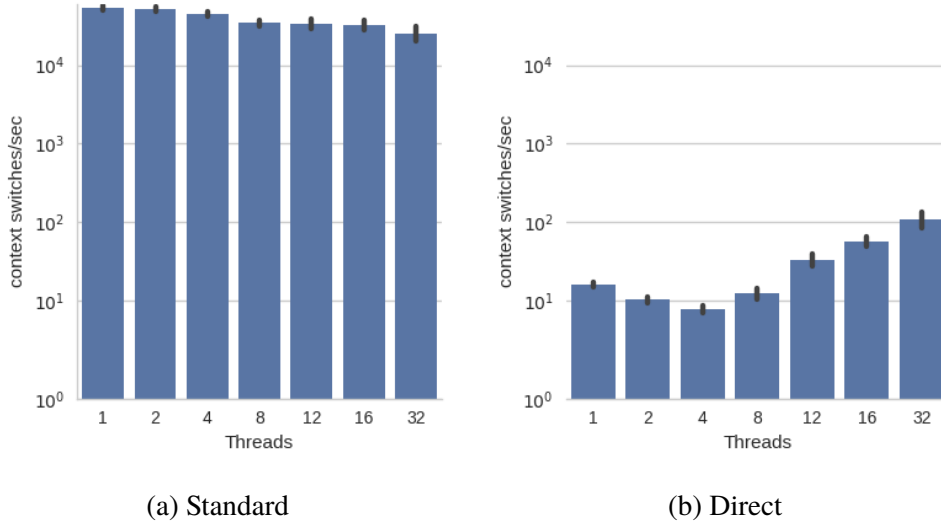


Figure 4.2: Both figures show context switches through all the experiments in logarithmic scale, where (a) is the context switches using the standard *liblightnvm* and (b) uses our kernel bypass, *LightNVM-Direct*.

4.2.1 Experimental Setup

The experiments were all conducted on a DFC card as described in section 2.3.4, using a *HP Z230 Tower Workstation* with a 4 core Intel Core i7-4790 3.60GHz CPU with 8 hyper-threading threads, and 32 GiB 1600MHz DDR3 RAM from Micron.

We conducted our experiments using the *OC-uFLIP* benchmark by compiling a traditional version of *fox* and another with our *LightNVM-Direct* backend. We have labeled our backends in our experiments as described in Table 4.2, which will be used throughout the rest of this chapter.

We chose the parameters listed in Table 4.1 for our experiments with the *uFLIP-OC* benchmark and chose the configurations where all combinations where the number of threads were less then the number of channels multiplied LUNs. We chose these combinations to get comparable results to the experiments conducted in [5]. The reasoning for these combinations is that channels and LUNs make up the parallel units of the SSD and it is therefore unnecessary and counterproductive to use more threads than there is parallelism available in the card.

The final result were 270 different experiments which we ran both using the *Standard* and *Direct* backends. We ran these experiments twice, however, our evaluation of the results showed that there was no difference between the backends when the channels were set to one or two, and when LUNs where set to one. Therefore, we decided to remove those charts from our results to make the presentation more compact.

4.2.2 Evaluation 1: Context-switching

First on the agenda was to evaluate the amount of context switches between the backends. For this task we used the Linux *perf* tool to monitor our experiments. The *perf* tool samples counters from various probe points from the hardware and the kernel and allowed us to get a deeper insight into the behavior of our experiments in regards to context switching and CPU behavior.

Each of the 270 experiments consisted of 32768 I/O operations issued to the DFC card. Because the *Standard* backend uses a system-call¹ to dispatch the I/O commands, we therefore expect at least the same amount of context switches. In the case of the *Direct* backend, that bypasses the kernel, we expect only involuntary context switches.

The results are presented in the two bar charts in Figure 4.2. In Figure 4.2a we can observe that the *Standard* backend has around 32 thousand context switches for all experiments, except when we exceeded 16 threads. We believe the reason for this anomaly is due to sampling errors, caused by short duration of those experiments.

The longer experiments on the other hand were observed to exceed 32 thousand context-switches. This, we believe, is a consequence of the longer running time and the number of involuntary context-switches longer running processes inevitably experience.

Figure 4.2b shows that the number of context switches never exceeded 150 in our experiments. While these results show that there is less kernel involvement in our communication with the DFC, we observed that all of our experiments with the *Direct* backend used 100% CPU utilization per thread. This usage is a result of our backend using polling to wait for I/O commands to complete and because our experiments with *fox* only queue one command at the time.

Using the *perf* tool we also were able to observe that the experiments only execute 0.02 instructions per cycle. We believe this is because the hardware threads are dominantly waiting for read operations to complete from the DFC's registers.

4.2.3 Evaluation 2: Read Throughput

Our next evaluation was to look at the read throughput of our experiments using the uFLIP-OC benchmark. We observed that there was no significant difference in throughput between pattern engines. We therefore combined the results from all three pattern engines in our bar-chart in Figure 4.3.

Our hypothesis was that our *Direct* backend would get better throughput than the *Standard* backend, due to the removal of the kernel's overhead with our bypass.

The results in Figure 4.3 show the throughput difference of four configurations of the DFC's parallel units of channels and LUNs, with different amount of threads. Contrary to our expectation the *Direct* backend gained no throughput improvement with eight or fewer threads, and lost throughput when the thread count exceeded eight. We believe that the throughput loss that occurs when we exceed eight threads is due to the wasted CPU cycles when the backend is polling for the completion of I/O commands. Our host computer has eight hardware threads and when more threads are used these hardware threads need to be shared. This sharing means that the operating system is not able to distinguish between a CPU bound thread and an I/O bound thread. The operating system therefore gives the threads a longer context switch quantum than it needs which causes an increase in overall overhead.

Thus, despite our expectations, the bottom line is that the results indicate that the *Direct* backend has not benefited from our kernel bypass when it comes to increased actual throughput for read operations.

¹Some system-calls only need to switch from user-mode to kernel-mode, and therefore do not need to execute a context switch.

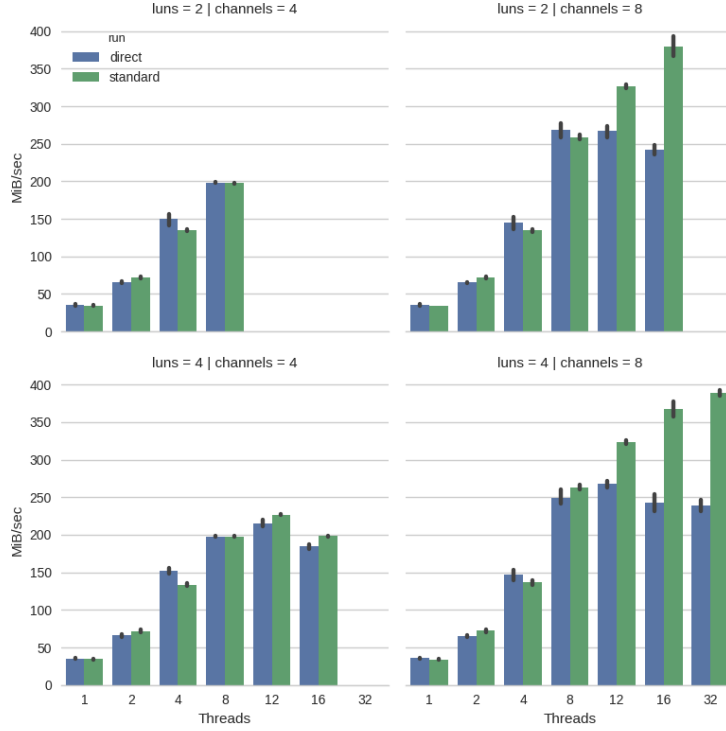


Figure 4.3: The bar charts show combined experiments using 100% read throughput of pattern engines 1, 2, and 3, for both Standard and Direct experiments.

4.2.4 Evaluation 3: Write Throughput

In our experiments we observed that the write throughput capacity of pattern engines 2 and 3 were not significantly different. Because of this, and to make the results more apparent, we have chosen to combine their results in the graphs depicted in Figure 4.5.

After evaluating the read throughput in 4.2.3, we expected the write throughput of our *Direct* backend to at most match the *Standard* backend in throughput.

Figure 4.4 shows our results for the write throughput of pattern engine 1. There we observe that the write throughput follows the same trend as was observed in the read throughput, where the *Direct* backend has write throughput that is similar to the *Standard* backend for eight or fewer threads.

The write throughput of pattern engines 2 and 3 on the other hand showed a different trend than we expected when eight threads were exceeded. Figure 4.5 shows the write throughput of pattern engines 2 and 3. There we see that the write throughput starts to degrade instead of flatlining when the threads are more than eight. We believe that the reason for the degraded throughput is related to the fact that pattern engines 2 and 3 are able to utilize more parallelism, and that when we exceed eight threads, our write throughput converges to pattern engine 1's throughput as we add more threads.

4.2.5 Evaluation 4: Thread Throughput

Our results show that the *Direct* backend has not benefited from our kernel bypass. We therefore shift our evaluation from comparing the two backends to analyzing what is affecting our write throughput when the threads exceed eight threads.

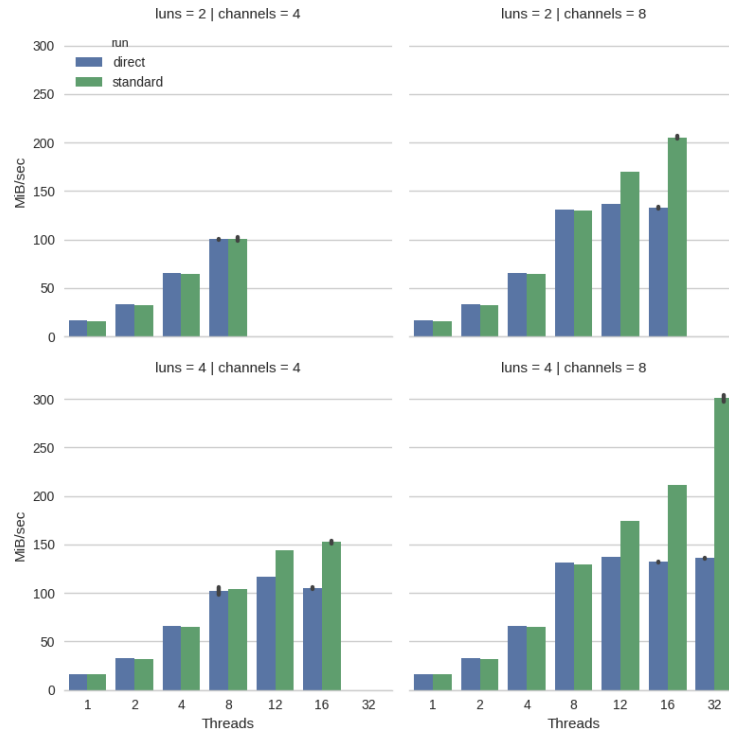


Figure 4.4: The bar charts show write throughput using pattern engine 1 using 100% write operations.

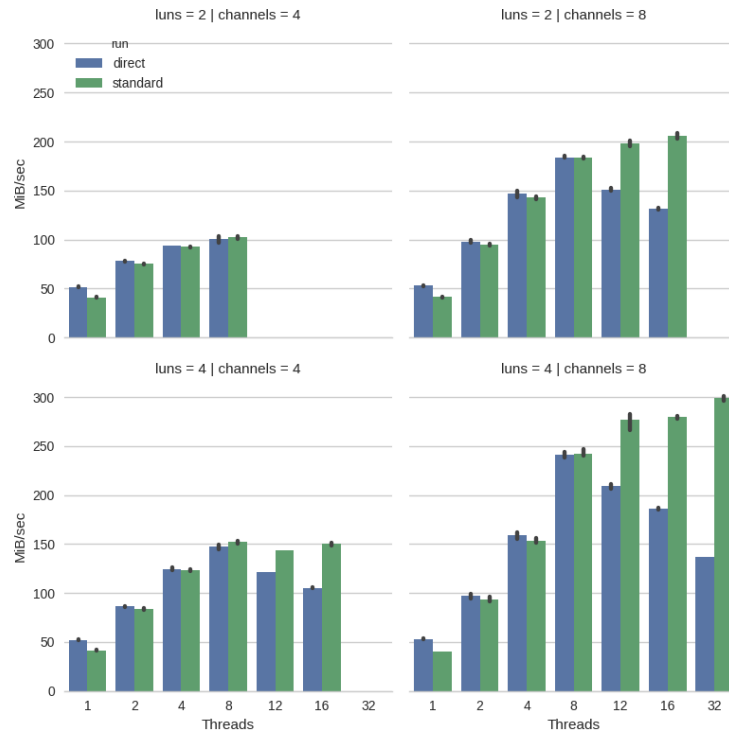


Figure 4.5: The bar charts show combined experiments using 100% write throughput using pattern engines 2 and 3.

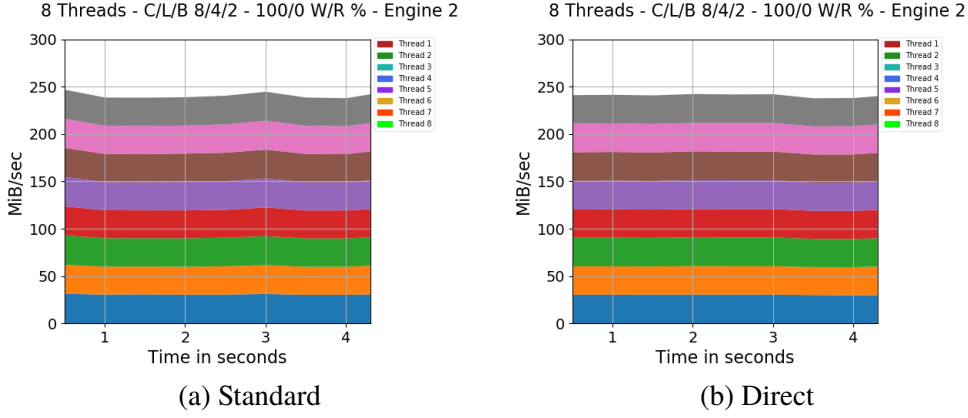


Figure 4.6: The area chart shows the throughput of 8 threads during 100% write using pattern engine 2 with and without a kernel bypass.

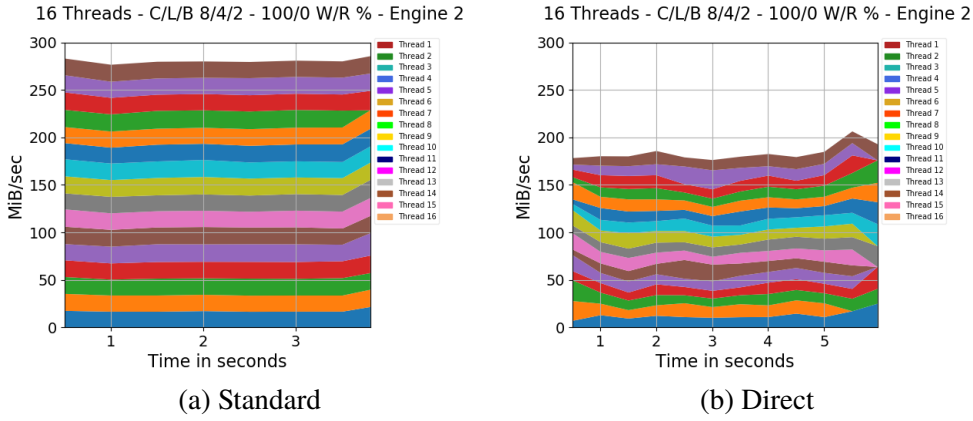


Figure 4.7: The area chart shows the throughput of 16 threads during 100% write using pattern engine 2 with and without a kernel bypass. The chart is more erratic in (b) because the thread count is higher than the number of CPU cores.

We decided to analyze and compare two independent I/O writing experiments that both used four channels and two LUNs. These two experiments varied only by their number of threads and were run on both backends.

Figures 4.6a and 4.6b show area charts where the write throughput of each thread is depicted as a differently colored area. These area charts show comparable write throughput to our results in evaluation 3.

However, when we analyze the area chart for the sixteen threads in Figure 4.7 we can see how the write throughput of each thread in the *Direct* backend is erratic. What is interesting in this area chart is how the overall write throughput increases when some of the threads finish running. Nonetheless, while the behavior of the threads in Figure 4.7b are interesting, they do not explain why the overall throughput of the *Direct* backend was this low.

4.2.6 Evaluation 5: Latency

To find an explanation for the low write throughput, we decided to compare the latency of the same eight and sixteen thread experiments from the Figures in 4.6 and 4.7. The results are presented as scatter plots in Figures 4.8 and 4.9.

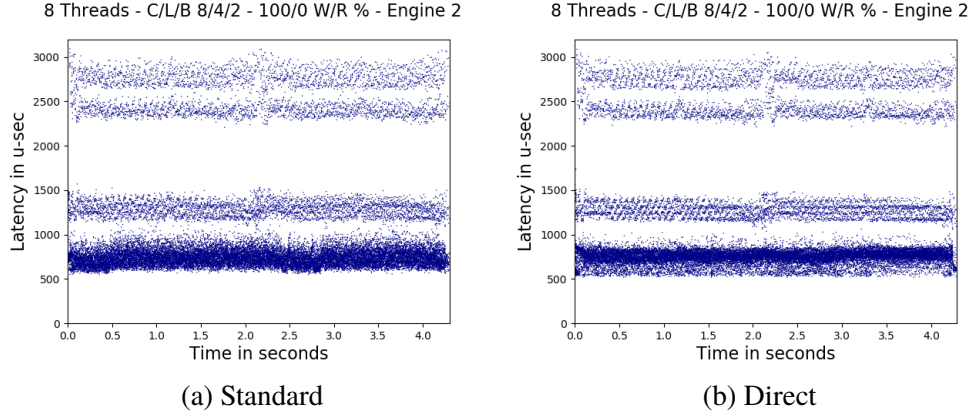


Figure 4.8: The scatter plot shows the latency of writes during experiments using pattern engine 2 running with eight threads.

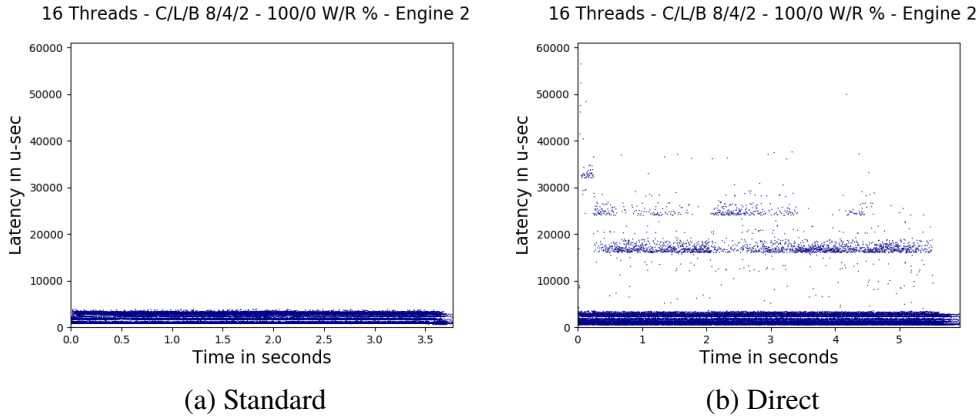


Figure 4.9: The scatter plot shows the latency of writes during experiments using pattern engine 2 running with sixteen threads. Writes in (b) show multiple outliers that have up to an order of magnitude worse latency than in (a).

Our finding, seen in Figure 4.9b, is that when the threads exceed the number of hardware threads, our performance is degraded to high latency outliers.

Table 4.3 compares the statistics of all four experiments from the Figures in 4.8 and 4.9. It shows that our performance degradation in our *Direct* backend is due to 10% of our submitted I/O commands having up to an order of magnitude higher latency than the *Standard* backend.

4.3 Discussion

In our work we set out to combine *liblightnvm* with LightNVM-Direct and to evaluate our work using the DFC card as the storage media and using Picoli’s *fox* program to run the uFLIP Open-Channel SSD benchmark.

Our key result is that a data path, from a user-space applications to a Open-Channel SSDs, that bypasses the kernel using LightNVM-Direct does not perform as well as a data path that traverses the kernel with LightNVM. There are two reasons for that:

1. LightNVM overhead is minimal as most of the work related to NVMe command construction is already done in user-space before an I/O command is submitted.

	8 Threads		16 Threads	
	Standard	Direct	Standard	Direct
Mean	1039 μs	1039 μs	1780 μs	2676 μs
Stdev	619 μs	611 μs	910 μs	4716 μs
Median	773 μs	783 μs	1582 μs	1150 μs
Max	3094 μs	3086 μs	3911 μs	61002 μs
95th percentile	2692 μs	2682 μs	3258 μs	16531 μs
90th percentile	2373 μs	2354 μs	3161 μs	3180 μs
Cohen's d	0		0.26	

Table 4.3: The table shows the latency statistics for the experiments in Figures 4.8 and 4.9. The experiments used pattern Engine 2 with 100% writes using eight and sixteen threads.

2. In our experiments, polling has a negative impact on performance as soon as the number of host cores is lower than the number of submitting I/O threads in user-space. We believe that this is due to the kernel scheduler not being able to distinguish between CPU bound threads and a I/O bound thread that is waiting for I/O using polling, which results in the kernel giving the thread a longer context switch quantum than it needs.

4.3.1 Sources of Complexity

In this section we would like to discuss complexities which we encountered which affected our work.

- One of our goals was to compare a new implementation of a kernel bypass based on *SPDK* that has been in development in *liblightnvm*. Unfortunately, we were unable to get it working with the DFC. The main problem was that the OX controller only supports the Open-Channel SSD 1.2 specification, however, the *liblightnvm* team are now prioritizing the 2.0 specification which is not compatible with the DFC.
- Originally we ran our LightNVM-Direct experiments on a 12 core computer. Those experiments also showed the degraded performance when the threads exceeded the CPU hardware threads. This computer unfortunately had a faulty motherboard with a broken South Bridge which affected our results.

4.3.2 Future Work

Our core insight is that kernel bypass does not improve performance. In fact, the solution does not scale beyond a given number of concurrent I/O threads corresponding to the number of CPU hardware threads. We see two complementary directions for future work:

1. Fixing the polling problem. LightNVM-Direct only supports polling-based interactions to Open-Channel SSD devices. The same design decision was taken in *SPDK*. The goal is to minimize overhead on the data path and thus I/O latency. The drawback is that throughput suffers when the number of threads submitting I/Os is larger than the number of CPU hardware threads. A solution to this problem is to introduce a similar method used by UNVMe and *SPDK* for traditional NVMe. There they submit NVMe multiple commands asynchronously and leave it to the application to decide when it needs to poll for the results.

2. An alternative is to introduce interrupts in either the LightNVM-Direct or the SPDK backends. This could be accomplished using the user-space I/O library UIO and its companion support from the kernel.

Chapter 5

AppNVM

The proposal of AppNVM came from [18], where they based their AppNVM on ideas of the OpenFlow software-defined network protocol. Their proposal was a new rule engine for NVMe devices that the application programmer could interact with using LightNVM. This rule language would enable the application programmers to express their Quality of Service (QoS) requirements and to implement rules that would tailor their FTL to the application-specific needs.

Another type of AppNVM was implemented in the *OX 2.0* firmware [19], by Picoli. There they created a version of AppNVM that does not use a rule engine. Instead, the programmer can write their traditional FTL using the firmware specific API. This architecture forces the programmer to follow the traditional design of an FTL and only allows the programmer to create new types of schedulers and wear-leveling algorithms. Another limitation of this type of AppNVM is the need to deploy the FTL onto the Dragon Fire Card manually.

Since [18], no work has been done on AppNVM in general literature nor has anyone tried to define further what AppNVM should entail. Therefore, due to the limited work and broad and unclear definitions of AppNVM, our first contribution in this chapter is a definition of what we believe AppNVM should be, which we then will further categorize into five different categories. Our second contribution is a new AppNVM protocol that we have

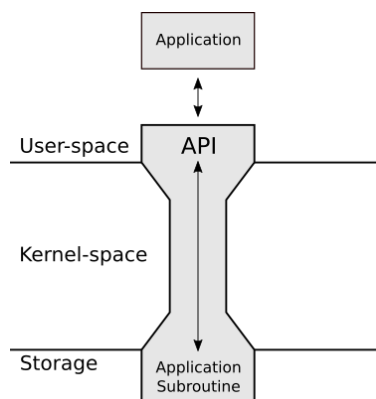


Figure 5.1: RNVMe is an NVMe RPC protocol that allows the application programmer to call subroutines that are capable of NDP on the NVMe device. By moving part of the application specific functionality onto the NVMe device, we are not only bypassing the kernel but the overhead of the PCIe layer as well.

named RNVMe, which is depicted in Figure 1.1c, and our last contribution is a discussion of an upload protocol would be used to add new functionality to an AppNVM enabled device.

5.1 Definition and Categorization of AppNVM

In our exploration of this topic, we found that the definition of AppNVM had grown beyond the rule language proposal of [18] and had become any interaction with an NVMe device. This lack of clarity lead us to create our definition of AppNVM:

- AppNVM is an application-driven protocol which the host application can use to optimize or accelerate its interaction with the NVMe device.
- AppNVM allows the application programmer to invoke or possibly define application-specific functionality on the NVMe device. This functionality could be anything from setting scheduling rules to creating and calling functions that reside on the NVMe device.
- Using AppNVM must be dynamic and usable at runtime without rebooting the NVMe device.

Using our proposed definition of AppNVM, we have further defined five different categories of AppNVM.

5.1.1 NVMe Rule Engine (NVMe RE)

NVMe RE is the AppNVM proposed by [18]. It allows the application programmer to specify their application's I/O behavior using a specialized rule language. This rule language would allow the application programmer to either define his FTL originally proposed originally proposed behavior, or to give the FTL hints on what type of I/O patterns to expect so that it can optimize the scheduler accordingly. This AppNVM is the least invasive solution and only affects the performance and possibly the wear of the device.

- **Pros:** This addition can be added without any changes to existing applications, and while the traditional NVMe commands would be used, it would only need new commands to add support for the rule language. This addition would also be simple for SSD vendors to add as it does not necessarily need to expose the underlying hardware implementation.
- **Cons:** While this rule language would be an optimization for the current SSD environment, it would limit the potential of AppNVM. The NVMe RE does not add any NDP capabilities and might, therefore, leave devices with massive resources under-utilized.

5.1.2 Thin-AppNVM

Thin-AppNVM allows the application programmer to create their FTL logic and upload it to the NVMe device. This type of AppNVM confines the programmer to create a traditional FTL and does not allow the programmer to deviate from traditional NVMe standard.

This definition would be the closest to the implementation of OX 2.0 in [5]. However, due to the lack of an upload protocol for new FTLs and the lack of a dynamic way to enable those FTLs, we have to say that it violates our definition of a true AppNVM.

The main difference between the NVMe RE and the Thin-AppNVM is that the application programmer can write the FTL in a programming language instead of a less flexible rule language.

- **Pros:** Thin-AppNVM can give the application programmer more flexibility than the rule language and possibly access to a rich API for developing FTLs. As with the NVMe RE, it would be easy to add this type of AppNVM to applications without needing to rewrite them.
- **Cons:** The main limitation of this type of AppNVM is that the method of implementing new FTLs would most likely be vendor specific, due to the myriad of architectures and APIs that vendors currently use to implement their NVMe devices. Therefore it would most likely be hard to standardize the programming environment between vendors.

5.1.3 NVMe Remote Procedure Calls (NVMe RPC)

The idea behind NVMe RPC is to give the application programmer the ability to invoke NDP capable functions on the NVMe controller and preferably to be able to define new ones. We believe that this version of AppNVM gives a right balance between flexibility and reusability, and therefore our proposed RNVMe protocol that we introduce in chapter 5.2 is an NVMe RPC.

For NVMe RPC to be useful, we would encourage the application programmer to create small and well-defined functions that can be reused by multiple applications. This encouragement is more likely to form an ecosystem of useful and standardized functions that would make this model more sustainable and productive for future applications.

Another criterion is that the functions should give the application programmers the capability of doing NDP, such as map-reduce operations or functions that hide the details of the storage data structures, which can be beneficial to large environments where the controller connects to multiple disks and multiple hosts at the same time.

For new NVMe RPC functions to be available to the NVMe device, the functions would either need to be uploaded to the NVMe controller or be available on there beforehand. In chapter 5.4 we will discuss which design principles we believe an AppNVM upload protocol should have.

- **Pros:** This type of AppNVM is flexible and is both capable of NDP and utilizing the resources available on the NVMe device. It also pushes for a reusable ecosystem of functions.
- **Cons:** For this type of AppNVM to be useful, the applications would either need to be partially rewritten or rewritten from the ground up to harness the potential of NVMe RPC.

5.1.4 NVMe Inter-Process Communication (NVMe IPC)

NVMe IPC goes a step further than the NVMe RPC and gives the application programmer the option of running a stand-alone process on the NVMe controller. An extreme example of this would be a database implementing and running its backend completely on the NVMe Controller, and communicating with it using either a stream or datagram interface. This AppNVM implementation treats the NVMe device as an accelerator similarly to Graphical Processing Units (GPUs) and allows both NDP and full utilization of the NVMe's resources.

The main differences between NVMe IPC and NVMe RPC are that NVMe IPC is application specific and mandates a large number of resources, while the NVMe RPC pushes for lightweight NDP functions that can be used by multiple applications.

Our vision of how the application programmer would perceive an NVMe IPC device would be as follows:

1. The application programmer would write an NVMe IPC application using a predefined API that gives access to internal functions of the NVMe device, such as submitting read, write, and erase I/O commands. This NVMe IPC application could either be a compiled application or written in a programming language that is compiled by the vendor's driver or NVMe device, all depending on the implementation.
2. When the host application starts it can upload the NVMe IPC application to the NVMe device and interact with it using either an identifier it receives when the NVMe IPC application is submitted to the NVMe device or by using a unique identifier that was chosen at compile time.
3. The host application can open a two-way stream using an NVMe IPC library to the NVMe device and interact with the running NVMe IPC application on the device. This type of communication could provide the application with the capability to redirect input and output similarly to console application, where debugging output could be directly printed out to the terminal, and console commands could be issued directly to an NVMe IPC.

This AppNVM implementation could either give the application programmer the option of writing backends that only interact with host applications, or it could also open up the option of creating advanced solutions, such as NVMe IPC applications that can interact with other NVMe IPC applications, NVMe firewalls, NVMe routing solutions, and more.

- **Pros:** The application programmer has exclusive access to the resources on the NVMe controller and can use those resources to accelerate the backend of the host application, offer NDP, and to create large hierarchical NVMe solutions in large environments.
- **Cons:** While NVMe IPC shares the same cons with NVMe RPC, it also has the problem of encouraging application programmers to create large applications on the SSD or controller which are not reusable by other applications. This AppNVM also pushes the availability of resources on the SSD controller, as it becomes a more generalized resource.

5.1.5 Full AppNVM

We define a full AppNVM as an implementation that gives the ultimate flexibility where the application programmer has complete control and can implement any functionality on the NVMe controller. This definition would include the power to implement any of the other AppNVM types and to be able to define new kinds of NVMe commands. We believe a full AppNVM least favorable approach in the real world, due to how hard it is to standardize and adapt to future applications.

- **Pros:** The application programmer has full control over the NVMe device and is, therefore, able to create any functionality, even the other four types of AppNVM.

- **Cons:** The main problem with a full AppNVM is that it allows too much flexibility and becomes impossible to standardize the usage and deployment procedure for both application programmers and vendors.

5.2 RNVMe: Our proposal for a NVMe RPC protocol

We propose a new AppNVM protocol that we have named RNVMe. This AppNVM is an NVMe RPC as was described in section 5.1.3, and allows applications to call subroutines located on the NVMe device. Our primary motivation is to design a protocol that adds NDP capabilities and encourages reusable functions. We propose RNVMe because we think it strikes a balance between both flexible and reusable code, compared to the other AppNVM solutions.

An example of how the RNVMe protocol could be useful is in a distributed database system such as Spark. Spark could call RPC functions that would do NDP such as accumulating results or storing and loading data with a higher storage abstraction. In large environments with a hierarchy of controllers connected to multiple hosts, the RNVMe protocol could be used to abstract the physical characteristics of the environment away using RPC functions.

In this chapter, we will introduce a new NVMe command that forms the RNVMe protocol. We will also discuss how we recommend changing the *OX* firmware to implement this new protocol and to how to make it easy to add new RPC functions dynamically at runtime.

5.2.1 The RNVMe command

Our proposal extends the NVMe standard by adding a new command with the opcode 0x99. This new NVMe command only provides the functionality to call RPC functions that are already present in the SSD controller. To upload new functions, we would need a separate NVMe based upload protocol. A discussion on what is needed for an upload protocol that could support any AppNVM is discussed in section 5.4.

We believe that this protocol should work alongside either the NVMe standard or Open-Channel SSDs. This combination would allow the application programmer to both support direct control over the data and to use NDP when applicable. In large environments, it could be preferable only to use RNVMe if the underlying physical structure is complicated.

The structure of the new RNVMe command can be seen in Figure 5.2 and Table 5.1 describes the purpose of each field. We designed the command with the goal to be simple and flexible. It allows the application programmer to call globally unique functions that are identified by a 64-bit number, which we further split into vendor and function identifiers. Our view is that complex function names add unnecessary complexity to the protocol and that the function definitions should not be part of the protocol either. Therefore we think that the function definitions should either be available online or could be retrieved using a standardized RNVMe function call. However, we think that the design of such a function call is outside the scope of our contribution.

The parameters and return values of an RNVMe function only support two data-types. These data-types are pointers to DMA buffers and 64-bit binary numbers. The protocol supports up to 255 DMA buffers and 255 binary numbers as parameters, and it also supports 255 DMA buffers and 255 binary numbers of return values. Nevertheless, we do not encourage implementations to support this excessive upper limit and recommend limiting the DMA buffers to 15 and keep the upper bits of the “*DMA in n*” and “*DMA out n*” fields as reserved for future changes to the protocol.

Field	size	Description
Vendor	32-bits	Represents a unique id that stands for the vendor or standard committee that created the functions. We believe that this field should be globally unique and that vendors should apply for their private vendor ID, but this would require an organizational body such as the Organizationally Unique Identifier which is managed by the Electrical and Electronics Engineers, Incorporated (IEEE).
Function	32-bits	Every function added by the vendor must have a unique function ID, and should always represent the same function. New versions of the function should get a new ID. This uniqueness would ensure backward compatibility between different applications.
Input Parameters	64-bits	If there are more than one parameters to the RPC function, then this field contains a pointer to a DMA page that contains the rest of the 64-bit parameters in sequence. Otherwise, if there is only one parameter to the RPC function, then this field contains that parameter. The first parameters in the sequence are DMA buffers where the number of DMA buffers is determined by the “DMA in n” field. The following number of 64-bit parameters in the sequence are determined by the “In n” field.
Output Parameters	64-bits	Similar to the “Input Parameters” field, if there is more than one result from the function, then this field contains a pointer to a DMA page that contains the rest of the 64-bit parameters in sequence. Otherwise, if there is only one result from the RPC function, then this field contains that result. The number of result values is determined by the “DMA out n” and the “Out n” fields.
DMA in n	8-bits	This field is used to specify how many of the first parameters are pointers to DMA backed pages. Only the first parameters can be DMA pointers. This field is used by both the host and the SSD controller to know which parameters need DMA backing, and which are used for other purposes.
DMA out n	8-bits	Similarly to the “DMA in n”, this field is used to specify how many of the first results are DMA backed pages. Only the first results can be DMA pointers.
In n	8-bits	This field shows how many 64-bit parameters there are after the list of DMA backed pointers. The application programmer is free to give any meaning to these 64-bits values. This means that if we include the value from “DMA out n”, then the total number of parameters possible to an RPC function is 510 parameters.
Out n	8 bits	This field shows how many 64-bit results there are from the RPC function after the DMA packed pointers.

Table 5.1: The RNVMe command is to support calling a function on the NVMe device with multiple parameters. These parameters can either be DMA buffers or 64-bit numbers.

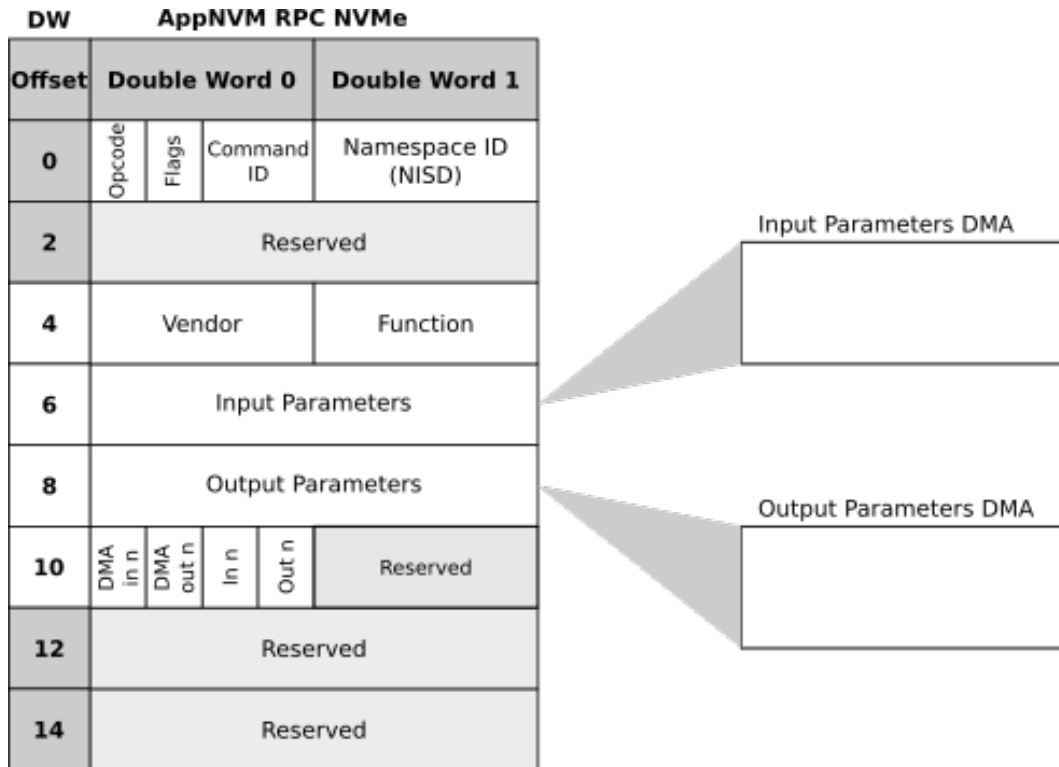


Figure 5.2: The proposed RNVMe command identifies the RPC function using the combined values of the vendor and the function fields. The rest of the fields are used for parameters to the function and return values. The parameters and return values can either be a 64-bit value or a pointer to a DMA buffer.

5.3 OX RNVMe Architecture

Since our equipment consisted of the DFC and the OX firmware, we have decided to include a description of how we would have implemented RNVMe in OX. The implementation of RNVMe in OX would consist of two parts. The first part is an internal backend that can dispatch multiple reads, writes, and erase commands to the flash chips. This backend is crucial because each RNVMe command can result in multiple commands to the flash chips. The second part is a modular framework that allows for the dynamic addition of new RPC functions at runtime. We do not include a description of an upload protocol. However, the RNVMe framework we propose would make it easy for such a protocol to add RNVMe functions.

5.3.1 The OX RNVMe Command Handler

Figure 5.3 shows how the data flow in OX changes with the inclusion of the new RNVMe backend. The NVMe command parser would need to be changed to support RNVMe commands. By design, the OX controller uses one thread per NVMe submission queue to dispatch I/O requests to the internal FTL and media manager logic. We would recommend using the same NVMe submission queue threads to run the new RNVMe commands. We would however also recommend that the application programmer or operating system driver would keep RNVMe commands on a dedicated submission queue, due to some of the RNVMe commands taking longer to execute than traditional I/O commands.

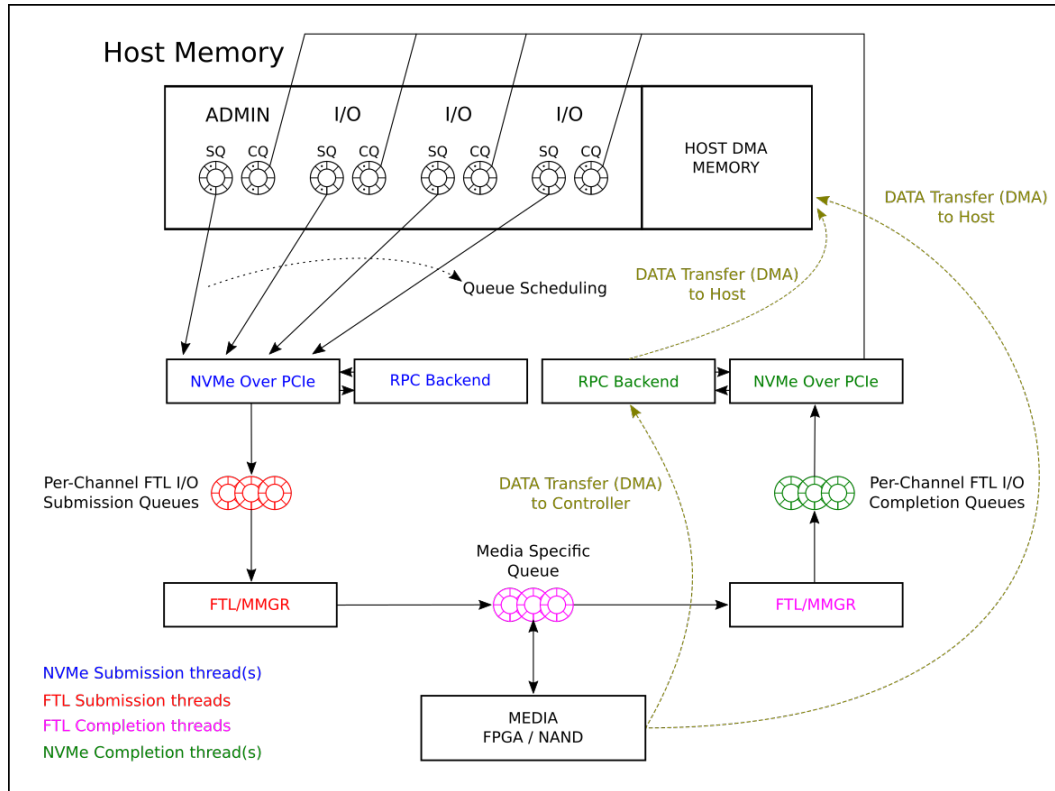


Figure 5.3: This Figure shows the internals of the OX controller as depicted by Picoli [5]. It has been modified to show the design change needed to OX to add a new RPC backend that is capable of submitting multiple I/O requests to the internal queues which are not visible to the host. Each RPC command can result in this backend submitting and completing multiple commands to complete an RPC NVMe and sending a completion entry back to the host.

Internally two separate threads are assigned to each channel of the FPGA’s flash chips. One thread is responsible for submitting commands to the FPGA, and another is responsible for delivering completion records to the NVMe completion queue threads. As a result of this thread delivery system, we would need to add the following new facilities to *OX* to be able to allow the RNVMe functions to dispatch multiple I/O commands internally:

- We would need to add new internal I/O submission, and completion queues for each RNVMe enabled submission queue. These queues would not be visible to the host and are only used to dispatch I/O related commands from RNVMe functions to the FPGA’s flash chips. The internal channel command threads would need to be able to consume I/O commands from the internal submission queue and to notify the NVMe completion queue thread when all of the dispatched I/O related commands from the RNVMe function have completed.
- All NDP related DMA transfers from the internally submitted I/O commands would need to be delivered to the DFC’s physical memory for the RNVMe function to be able to process them. This functionality is already available in the FPGA and only needs a flag set in the internal flash commands to the FPGA to function and is therefore easy to implement.

- The *OX* controller needs to be able to submit DMA transfers from the DFC card over to the host. While the DFC supports this, however, it is only able to do so through the FPGA.

The only real change to the *OX* architecture would be the introduction of the internal RNVMe I/O queues. All other requirements are either available or need minimal additions for them to function.

5.3.2 The OX RNVMe Modular Backend

We propose that the new RNVMe backend would use the shared library support of the Executable and Linkable Format (ELF) provided by Linux. This library support gives the application programmer the option of compiling new RNVMe commands as *.so* library files and then directly copy them into a directory on the DFC, where *OX* can then dynamically initialize them at runtime. This deployment method allows us to quickly deploy new modules without the need for an upload protocol.

The steps that we propose for loading new *.so* modules into *OX* to enable new RNVMe commands are as follows:

1. **Initializing the environment:** Before loading any RNVMe modules, the backend must load a standard RNVMe library *.so* file that contains all the functions that the RNVMe modules can use. This library would include functions to register new RNVMe commands, submit I/O commands, and execute DMA transfers.

The *dlopen* function would be used with the *RTLD_NOW* and *RTLD_GLOBAL* flags to load the library. Doing so will link the functions so that future modules will be able to use them.

2. **Monitoring for new RNVMe modules:** The backend should use the *fcntl* system call with the *F_NOTIFY* flag to monitor for newly uploaded *.so* files to the RNVMe library folder. This notification system will make sure that the RNVMe backend can insert new RNVMe modules every time a new *.so* file is available, without the need to periodically polling the directory.
3. **Adding new RNVMe modules:** New *.so* files could either be uploaded during runtime or be stored on the DFC card. To make sure that the backend does not load a partially uploaded *.so* file, we recommend that the ending of the file being uploaded to be *.part* and then changed to *.so* when the upload is complete.
4. **Initializing the NVMe modules:** When the *.so* file has been successfully uploaded, the backend should use the *dlopen* call on the uploaded *.so* file with the *RTLD_LAZY* flag. This will initialize the new library without functions from different RNVMe modules classing with each other.

The backend then creates a *Module* struct from Listing 5.1 and uses the *dlsym* function to load the *rnvme_init* and *rnvme_exit* functions from the NVMe module into the *init* and *exit* function pointers in the struct. These two functions are then used to initialize the NVMe module and to deconstruct the module on shutdown.

Listing 5.1: Internally the RNVMe backend creates one *Module* struct per loaded *.so* file and initializes the *init* and *exit* function pointers to their internal *.so init* and *exit* functions. These function pointers are then called when the module is loaded and when the module is removed.

```

typedef struct Module
{
    char* name; // A human readable name for the module.
    void* handle; // A dlopen file handle.
    int (*init)(void*); // The modules loading function.
    int (*exit)(void*); // The modules unloading function.
    int status; // Was the module loaded successfully.
} module;

```

5. **The RNVMe module's init function:** The `init` function pointer in the *Module* struct is responsible for calling a registration function provided by the RNVMe standard library. This function creates an instance of the *RpcFunction* struct from Listing 5.2, which is then called for each dispatched RNVMe command from the host.

Listing 5.2: Each RNVMe function is kept in a *RpcFunction* struct, and *OX* calls the *func* function pointer for each RNVMe command dispatched from the host.

```

typedef struct RpcFunction
{
    Module* module; // Owning module.
    unsigned int vendor_id; // Vendor ID.
    unsigned int function_id; // Function ID.
    int (*func)(nvm_command* cmd); // The actual RNVMe function.
} RpcFunction;

```

6. **Submitting RNVMe functions from the host:** The RNVMe must be submitted from a library to make it convenient for the application programmer. We would recommend changing the *LightNVM-Direct* to support the new RNVMe commands, or changing *liblightnvm* and the *LightNVM* framework in the Linux kernel to support it. This would mean that a new *rnmvme_command* struct would need to be added to the *LightNVM* headers and the appropriate mapping of DMA buffers would be done by the kernel before the command is dispatched.

5.4 The AppNVM Upload Protocol

In this chapter, we contribute a discussion on what principles we think an AppNVM upload command should strive to follow. Currently, the NVMe specification contains two commands to upload new firmware and enabling them. These commands are *Firmware Commit* and *Firmware Image Download* and force the design of having a few firmware slots which the NVMe device can update and boot from. These firmware commands are too limiting for AppNVM uploading, and therefore, we will have to introduce a new upload protocol.

All commands in the NVMe specification are 64-byte long and many commands reserve bits for future use. We believe that an upload protocol that limits a fixed number of bits to differentiate between AppNVM implementations and hardware configurations is too limiting. We, therefore, propose that the new AppNVM upload protocol to not be limited to the 64-byte NVMe command structure and should use a higher level capabilities discretion that would be communicated between the host and the NVMe using DMA. We propose that the high-level description would fulfill the following design principles:

- **Push for a common AppNVM upload standard:** Should support all types of AppNVM architectures and should not contain any AppNVM specific features to prevent the formation of multiple upload standards. This requirement would allow protocols other than RNVMe to use the same description and upload protocol.
- **Use unique identifiers for capabilities:** The protocol should use globally unique identifiers for capabilities, such as GUID, to make the distinction between capabilities clear and to allow for diversity. This requirement allows vendors to create vendor-specific backends for RNVMe without requiring all implementations to be compatible.
- **Support multiple backends:** The protocol should be able to support multiple types of hardware and software architectures. An AppNVM architecture might support multiple programming languages and even FPGA uploading at the same time. It also might support multiple versions of the same architecture over time.

An example of what an AppNVM capabilities description could look like is depicted in Listing 5.3. There we designed a description that uses GUID to distinguish between capabilities and upload methods. The host application uses the GUID values to see which capabilities are available on the NVMe device before communicating with it. The description is split into three different regions, the first being the upload capabilities. The upload capabilities list what type of AppNVM backends are supported. This could be different types of RNVMe backends or even other types of AppNVM. The second region contains the interfaces that the NVMe supports which would list the supported protocols, such as RNVMe. The third category lists the hardware capabilities which would influence what functionality could be uploaded to the NVMe device.

Listing 5.3: This shows an example of what the AppNVM high-level get features description could be. The listing shows a capabilities description that shows hardware capabilities, upload capabilities, and the protocol supported. It has no fixed size binary limitations and allows multiple protocols to coexist at the same time.

```

appnvm: {
  upload: {
    be6f3344-f106-11e8-8a4c-0021ccdb24ac: "OX RNVMe Vo.9"
    ae6bdaf6-ffff-11e8-ba69-0021ccdb24ac: "OX RNVMe V1.0"
    dd1c5824-ffff-11e8-a50d-0021ccdb24ac: "OX RNVMe Java v8"
  },
  interface: {
    f84d0846-ffff-11e8-940c-0021ccdb24ac: "RNVMe v1.0"
  },
  hardware {
    23708d34-ffff-11e8-8bae-0021ccdb24ac: "ARMv8-A",
    fec9a848-ffff-11e8-9655-0021ccdb24ac: "16-GiB",
    1285062a-ffe-11e8-a16c-0021ccdb24ac: "VVDN FLASH FPGA",
  }
}

```

It would be ideal if AppNVM implementations would be standardized to use known architectures, such as the RISC-V instruction set, or if they used a driver compiled programming language as is done in graphical accelerator drivers. However, we believe that even if such a standard would exist, it would eventually need a replacement.

5.5 Security

The main security problem related to AppNVM is the DMA buffer handling and the broad access it has to the host's physical memory, which opens up the door for DMA malware [12]. This full access stems from the fact that PCIe devices have direct access to the physical memory through the IOMMU, and therefore allow the attacker to transfer data to and from any memory address. An NVMe bypass allows an attacker to write memory pages the attacker does not have access to onto the NVMe device, and then read them back into accessible memory. AppNVM however, opens up a new attack vector where the attacker could upload AppNVM code that could read and modify any memory at will using the DMA facilities available on the device itself.

5.5.1 Preventing malicious DMA transfers from the host

We would argue that if an application is using a user-space NVMe to try to boost performance, then the application is most likely running on a dedicated server. This exclusiveness would mean that if the application becomes compromised, then everything of real value on the host is already compromised. Therefore, one could argue that without proper hardware support, a kernel space bypass should never be considered safe.

We believe that to make RNVMe safe from misused DMA addresses, a kernel bypass should not be used, and the kernel should be in charge of the RNVMe DMA buffers. For this to work, the NVMe device would need to implement an RNVMe function that the kernel can use to query how many of the parameters and results require DMA transfers for the dispatched RNVMe command. This requirement would be necessary to enforce the proper use of the DMA memory and would require caching for the subsequent RNVMe commands.

5.5.2 Preventing malicious uploads of AppNVM

A harder problem is to secure the upload of new AppNVM functionality that can execute DMA transfers to the host. We believe that the only correct solution is to limit who can upload AppNVM functionality to the NVMe device; however, we also believe that DMA transfers can be made more secure on the NVMe device as well by sandboxing the AppNVM application and limiting the DMA transfers to the physical addresses handed to the AppNVM application.

In large environments where the NVMe device is connected to multiple hosts, it would be necessary to protect the AppNVM upload functionality. We propose that one of the following methods to be used to protect the AppNVM upload.

- **Password protection:** The upload protocol could force a password to be included in the payload for the NVMe device to accept the AppNVM application. This password could either be set manually or be kept by the Trusted Platform Module (TPM) on the master host.
- **Digital signature:** The NVMe device could require the AppNVM application to be digitally signed using a public-key cryptosystem for the AppNVM application to be accepted by the device. This security system would either need the vendor to sign all sanctioned modules or that the company that is using the NVMe device would deploy and sign all AppNVM applications.

- **Host limiting:** In a large environment, the NVMe device or switch to the NVMe device could be instructed to only accept AppNVMe uploads from specific hosts, while still allowing other hosts to call AppNVMe functionality.

One method that could be used to set the password and settings of the NVMe device would be to upload the settings through the AppNVMe upload protocol.

5.6 Discussion

We set out to design an AppNVMe protocol that would add NDP to NVMe and Open-Channel SSDs. We realized that there had been limited work done on AppNVMe and that no attempt on classifying what types of AppNVMe were possible. Therefore, we defined five types of AppNVMe and listed the pros and cons of each, in an effort to clarify the options. We suspect that our list will need amendments and even that more types of AppNVMe might be added to this list in the future.

We designed a new AppNVMe protocol, that we named RNVMe, and described our vision of how it could be put into practice. This protocol was of the type AppNVMe RPC from our definition of AppNVMe types. We chose this type of AppNVMe because we think it strikes a balance between flexibility and reusability, while still being simple enough for vendors to standardize their programming interface. In contrast the AppNVMe IPC pushes the application programmers to create less reusable solutions and a larger programming environment which is harder to standardize between vendors.

5.6.1 Future Work

In our ongoing work on RNVMe, we have created a partial proof-of-concept implementation of the OX backend that handles adding new RPC commands. We hope to explore the implications and possibilities that the RNVMe protocol would entail. Some of the questions that would need to be explored are as follows.

- Should RNVMe commands be put on a dedicated NVMe queue with a longer timeout value? Each RNVMe command might result in multiple I/O commands on the SSD, therefore, mixing RNVMe commands with regular I/O could make the latency of the NVMe queue less predictable.
- Should the RNVMe commands include an estimated timeout limit? This limit could be used both for priority-scheduling of RNVMe commands and to limit the upper bound on commands that are allowed to fail if they take too long.
- Should RNVMe commands support returning substantial answers, or should the design try to force the programmers to make light answers? Large RNVMe commands add both time and space complexity to the RPC functionality. Commands that return substantial results might either need to return pointers to the controller's memory for later extraction or even to a location on the SSD storage itself.

For RNVMe to be fully functional on OX, a new upload protocol for RNVMe would need to be completed. We believe the upload protocol needs to be designed to work with all future AppNVMe solutions. It is our concern that each new protocol added to the AppNVMe space would add a different type of upload protocol.

The final question that will need to be addressed for RNVMe to be a feasible AppNVM solution is if the programming environment could be standardized. This standardization would either require a standardized instruction set architecture on the SSDs, or a standard programming language that would either be compiled by the SSD firmware or by a vendor specific compiler provided with the SSD's host driver. To answer this question, we would need to define what should be possible in an RNVMe function, and what implications that would have on the RNVMe programming environment.

Chapter 6

Conclusion

The project described in this thesis had two major contributions. First was the creation of the LightNVM-Direct backend for *liblightnvm* to allow direct communication between user-space and Open-Channel SSD compliant devices. We compared the performance of our solution to the native LightNVM implementation and found that our solution is comparable to LightNVM as long as the number of threads submitting I/Os does not exceed the number of CPU hardware threads. If the number of threads exceeds this boundary, the overhead, grows due to the spin-lock thrashing of the more numerous threads on the host side. In other words, our experiments indicate that the overhead of LightNVM itself is very low regarding latency and that it scales well. Therefore, it seems to be little need to introduce user-space I/Os that can bypass LightNVM when accessing Open-Channel SSDs for performance concerning latency and throughput.

However, there are incentives to keeping the Open-Channel SSD driver in user-space other than performance. In highly data-driven applications, where developers are creating sophisticated execution planners, predictability can be of even greater value than max performance. Having full control over the storage device's queues and settings is the only viable option in the cases where the kernel driver scheduling decisions are not in line with the applications execution planner. Therefore, we believe that our work is still of value, despite the lack of expected performance improvements from our user-space bypass.

Another reason why there has been an interest in moving the drivers into user-space has been due to license issues, due to companies developing FTLs not beeing keen on sharing their work under the GPL license in the Linux kernel. For this reason, a full user-space driver for Open-Channel SSD is still desirable for some.

The second contribution of this thesis was a new NVMe RPC protocol that we called RNVMe. It allows simple I/O routines to be run on the NVMe device to do near-data processing on the SSD disk itself. We think this is made feasible because of the sophistication of the SoC chips used on today's state-of-the-art NVMe disks. These SoC have large quantities of RAM and sometimes decent CPUs which tend to be underutilized. This underutilization is especially true if the responsibility of managing the device has been taken over by the software stack, as in the case of Open-Channel SSDs. We had already started work on proof-of-concept implementation of RNVMe for OX, and recently there is general interest emerging in the field of computational storage [20]. The underutilization of the SSD resources may however affect the design choices of Open-Channel SSDs when it comes to mass production. With the removal of the FTL in Open-Channel SSDs, there is a motive to remove the excess resources to make the production cheaper. We would consider such a reduction of resources a lost opportunity, both because these resources are fairly cheap and because it would significantly limit the potential future development of NDP on the devices.

We find the challenges of on disk NDP to be an exciting avenue for future work as the option of AppNVM shows great potential and many of the challenges are yet unsolved.

Bibliography

- [1] S. Kim and J.-S. Yang, “Optimized I/O determinism for emerging NVM-based NVMe SSD in an enterprise system”, in *Proceedings of the 55th Annual Design Automation Conference*, ACM, 2018, p. 56.
- [2] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, “NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs”, in *HotStorage*, 2016.
- [3] M. Bjørling, J. González, and P. Bonnet, “LightNVM: The Linux Open-Channel SSD Subsystem”, in *FAST*, 2017, pp. 359–374.
- [4] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, “SPDK: A development kit to build high performance storage applications”, in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2017, pp. 154–161.
- [5] I. L. Picoli, C. V. Pasco, B. Þ. Jónsson, L. Bouganim, and P. Bonnet, “uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives”, in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ACM, 2017, p. 20.
- [6] Intel, *SPDK*, <http://www.spdk.io>, Accessed: 2017-12-14.
- [7] Micron, *unvme*, <https://github.com/MicronSSD/unvme>, Accessed: 2017-12-14.
- [8] *uio*, <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/driver-api/uio-howto.rst?h=v4.14.81>, Accessed: 2018-11-18.
- [9] *VFIO*, <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/vfio.txt?h=v4.14.81>, Accessed: 2018-11-18.
- [10] D. Scholz, “A look at Intel’s dataplane development kit”, *Network*, vol. 115, 2014.
- [11] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, *NVNeDirect*, <https://github.com/nvmedirect/nvmedirect>, Accessed: 2017-12-14.
- [12] P. Stewin and I. Bystrov, “Understanding DMA malware”, in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2012, pp. 21–41.
- [13] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, “Linux block IO: introducing multi-queue SSD access on multi-core systems”, in *Proceedings of the 6th international systems and storage conference*, ACM, 2013, p. 22.
- [14] L. Bouganim, B. Jónsson, and P. Bonnet, “uFLIP: Understanding flash IO patterns”, *arXiv preprint arXiv:0909.1780*, 2009.
- [15] M. Bjørling, J. Madsen, J. González, and P. Bonnet, “Linux kernel abstractions for open-channel solid state drives”, in *Non-Volatile Memories Workshop*, 2015.

- [16] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan, “The necessary death of the block device interface”, IT-Universitetet i København, Tech. Rep., 2012.
- [17] J. González, M. Bjørling, S. Lee, C. Dong, and Y. R. Huang, “Application-driven flash translation layers on open-channel ssds”, in *Proceedings of the 7th Non Volatile Memory Workshop (NVMW)*, 2016, pp. 1–2.
- [18] M. Bjørling, M. Wei, J. Madsen, J. Gonzalez, S. Swanson, and P. Bonnet, “AppNVM: A software-defined, application-driven SSD”, in *Non-Volatile Memories Workshop*, 2015.
- [19] I. Picoli, *OX AppNVM*, <https://github.com/DFC-OpenSource/ox-ctrl/wiki/7.--AppNVM:-A-Framework-for-Application-specific-FTLs>, Accessed: 2018-11-12.
- [20] *New SNIA Technical Work Group to Focus on Computational Storage*, https://www.snia.org/news_events/newsroom/new-technical-work-group-focus-computational-storage, Accessed: 2019-01-17.



School of Computer Science
Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is