POLITECNICO DI TORINO

MASTER's Degree in Electronic Engineering, Electronic Systems

Politecnico di Torino

MASTER's Degree Thesis

TOWARDS COMPUTATIONAL STORAGE

Supervisors

Prof. Guido MASERA

Prof. Alberto DASSATTI

Candidate

Gabriele BORELLO

OCTOBER 2021

Abstract

Today the volume of data generated is constantly increasing. Each year it grows by 27%, reaching 94 Zettabytes in 2021, according to the International Data Group. As this amount increases, the bottleneck due to the transfer of data from where it is stored to where it is processed is increasingly evident. One of the possibilities to overcome this obstacle is Computational Storage, able to modify the old paradigm "Data move to process" in the new "Move process near data".
The following work describes the birth of a Computational Storage (Fixed Computational Storage Service), based on the NVMe protocol, built on FPGA, able to lighten the computational weight of the Host by exploiting the peer-to-peer capabilities of PCI-Express. First of all, the realization of an NVMe Controller is described, based on NVM Express Base Specification version 1.4, capable of managing communication and data transfer between Host and storage device. Once the software was tested via QEMU, it was then transferred to the ARM of an FPGA, and the performance was compared with a real storage device (Samsung's SmartSSD). It was subsequently transformed into computational storage following the draft protocol for such devices presented by SNIA, using a CMB as a buffer for data processing. Through the SPDK software, it was possible to test the correct functioning of the device and evaluate its performance.

Acknowledgements

Before proceeding with the discussion, I would like to dedicate these few and insufficient lines to those who helped me in the realization of this thesis and in my personal and professional growth.
A heartfelt thanks to my supervisor Prof. Guido Masera, for his great support and precious advice.
Thanks to my co-supervisor Prof. Alberto Dassatti, for his immense availability, demonstrated from the beginning, allowing me to carry out a thesis abroad despite the pandemic period, and for showing me the passion and the methods necessary to face practical challenges in electronics.
A special thanks to Enrico Petraglio, Rick Wertenbroek, and Roberto Rigamonti for the fundamental support in the realization of this study and for all the fruitful discussions.
Thanks to all the Reds and Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud (HEIG-VD) team.
Thanks to my Mother, my Father, and my Sister for always supporting me, with infinite patience, even in the most difficult moments. Without them, none of this would have been possible.
Thanks to my Grandparents that I would have liked here next to me.
Thanks to P.C.
Thanks to the friends, those of a lifetime, with whom I grew up and will continue to grow.
Thanks to you Clarice, my accomplice, for the patience and strength with which you always help and support me. With you, the future is less scary.

Table of Contents

List of Tables

List of Figures

Glossary and Acronyms

Namespace

Quantity of non-volatile memory that may be formatted into logical blocks [1].

ACQ

Admin Completion Queue.

ASIC

Application specific integrated circuit.

ASQ

Admin Submission Queue.

BAR

Base Address Registers.

CMB

Controller Memory Buffer.

CSA

Computational Storage Array.

CSD

Computational Storage Drive.

CSP

Computational Storage Processor.

CSS

Computational Storage Service.

CSx

Computational Storage devices.

DW

Double Word, 32 bit.

FCSS

Fixed Computational Storage Service.

HW

Hardware.

IOPS

Input/output Operations per second.

NVM

Non-Volatile Memory.

NVMe-oF

NVMe over Fabrics.

PCSS

Programmable Computational Storage Service.

PM

Persistent Memory.

PMR

Persistent Memory Region.

RDMA

Remote Direct Memory Access.

SW

Software.

SSD

Solid State Disk.

TP

Throughput.

Chapter 1

Smart Storage

In today's world, the amount of data collected increases with each passing day. According to the International Data Group (IDG), the volume of data generated each year grows by 27% [2]. In 2021, the volume of data created, acquired, copied and consumed around the world was estimated at 94 Zettabytes, and it is bound to double by 2024 [3]. Data is constantly transferred between storage and processing units and this inevitably favors the birth of bottlenecks. There is both a cost and a time factor in moving data from where it is stored to where it is processed. The problem becomes more and more consistent as the volume of data increases.

Computational storage presents itself as a possible solution to the problem, modifying the old "Data move to process" paradigm with the new "Move process near data"(Figure 1.1). The creation of a smart storage, capable of processing as well as archiving, would allow to reduce the amount of data sent to the processing unit and to lighten its computational load. By bringing the processing power of processors to traditional storage architectures, it allows an increase in processing speed, thus allowing accelerated analysis and energy savings [4].

Several companies are proceeding towards computational storage with different approaches, from integrating the processing unit into the storage unit, to accelerators that contain no storage space. Most of them take advantage of NVM-Express SSDs [4]. To allow interoperable and vendor-independent implementations, the need for a standard arises, capable of delineating architectural models for computational archiving. In 2018, SNIA (Storage Networking Industry Association), brought together a technical working group (TWG) made up of representatives of large organizations (ARM, DellEMC, Eideticom, IBM, Intel, NetApp, NetINT, NGD Systems, Samsung, ScaleFlux, Seagate) with the purpose of creating the standard, which is still being worked on (at the moment, a draft, version 0.5, has been published)[5]. TWG aims to achieve a full version 1.0 and then pass it to an existing standard organization for future management. One of the most accredited candidates is NVM-Express organization, given its widespread use in computational

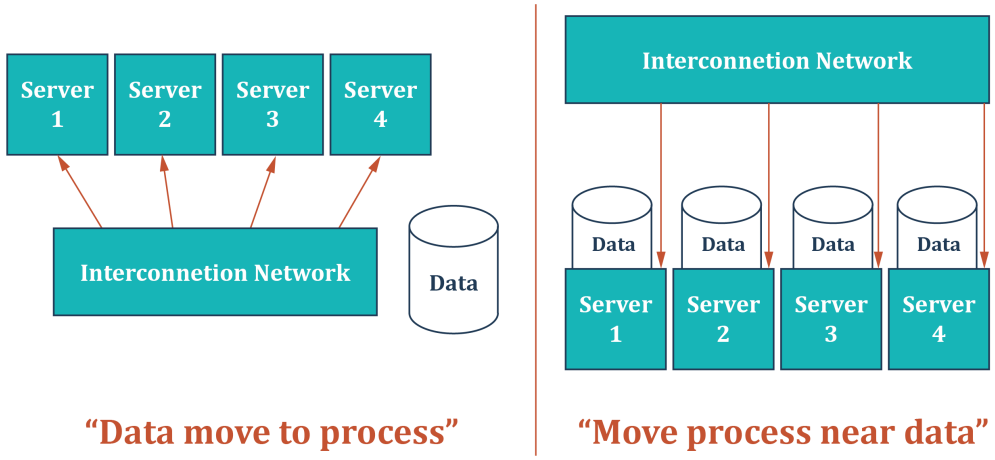

Figure 1.1: Comparison of paradigms: "Data move to process" vs "Move process near data".

storage as the storage interface [4].

An in-depth description and analysis of the NVMe protocol is therefore necessary, as it is a fundamental part of computational storage and of the work described in the following chapters (chapter 2 and chapter 3).

1.1 NVM Express

The NVMe standard is a transfer protocol designed for solid state memories able to allow communication between host and SSD through NVMe controller. A controller is a logical or physical entity capable of handling communication between two devices. SSDs originally used the Serial AT Attachment (SATA) protocol, which was designed primarily for interfacing with mechanical hard disk drives (HDDs). But with the increase in performance, already in the late 2000s, SSDs began to be held back by the transmission speed of SATA, making it necessary to create a new standard [6]. Version 1.0 was released in 2011, the result of the work of the NVM Express Workgroup, made up of more than ninety companies. In 2014 the working group was incorporated into the NVM Express, the consortium responsible for the specification and development of the standard, which today has over one hundred member companies[7]. The benefits of NVMe are numerous:

- The maximum IOPS (Input/output operations per second) for NVMe reaches 1,000,000, against a maximum of 200,000 IOPS for SATA [8].
- Much higher bandwidths are supported thanks to the use of PCI-express links

[8]. Suffice to say that in 2019 PCI-Express Gen5 x4 was introduced capable of reaching 16 GB/s [9].

- End-to-end latency of less than 10 microseconds[8].
- The queues in NVMe technology can have more than 64'000 commands per queue, against 32 of the Sata protocol [8]. As shown in figure 1.2, each core can have its own queues, allowing individual threads to have a dedicated queue. Parallel execution is therefore possible, requiring no I/O block.
- The MSI-X interrupt system is supported. (Appendix A.4).
- NVMe has a simple instruction set, which allows you to use less than half of the instructions sent to the CPU than SATA [8].

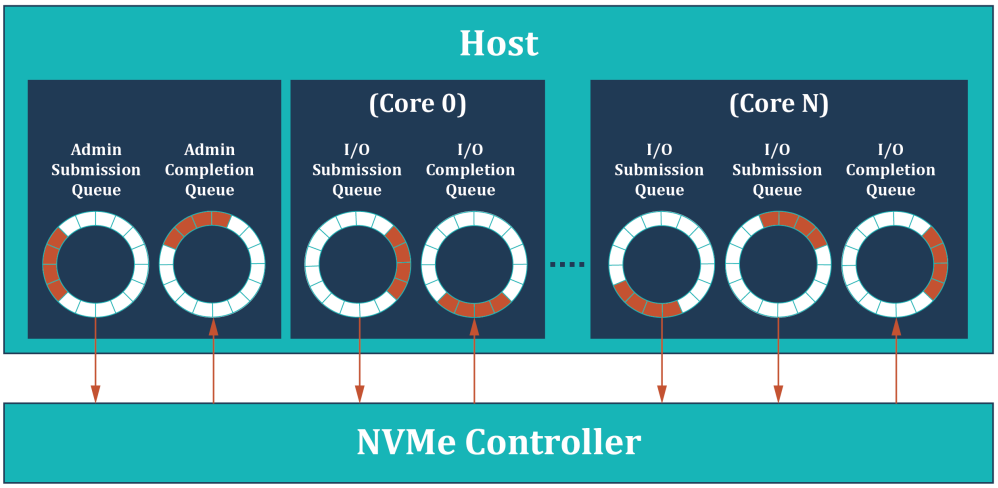

Figure 1.2: Host-Controller Interface. Each core can have one or more I/O queues. Admin queues are unique.

An analysis by a team of researchers from the University of Southern California, San Jose State University and Samsung [10] compared SATA and NVMe. Using a software for the generation of I/O traffic (`fio`), they generated random reads, 1 request per second, to evaluate the performance of the two protocols. Figure 1.3 shows a breakdown of the time elapsed by each request across different sections of the I/O software stack. The software overhead for SSD with SATA represents 28% of the overall access latency. In the case of NVMe it is reduced to only 7.3%.

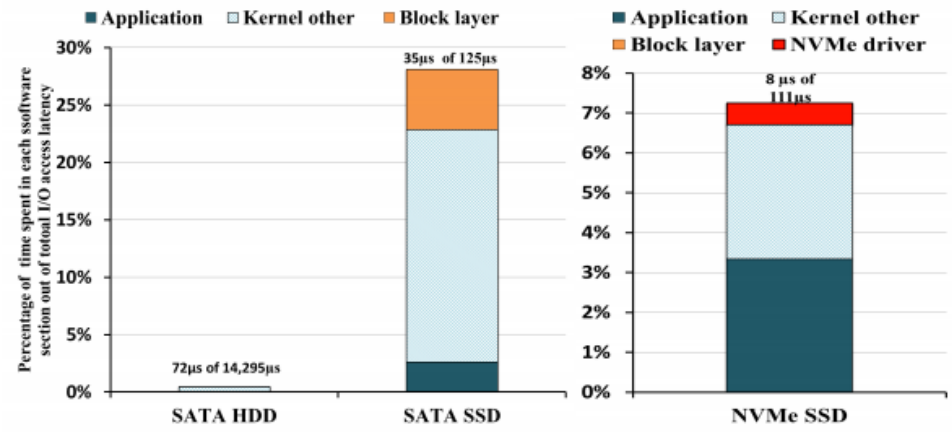

Figure 1.3: "Break-down of the time spent in different sections of the I/O software stack" [10].

1.1.1 NVMe Protocol

The version taken as a reference is the 1.4. [1]. This standard is defined for both over PCI-express and over Fabrics controllers. The first NVMe over Fabrics specification was released in 2016, with the aim of extending NVMe technology to Ethrenet, Fiber Channel, Infiniband and RDMA transports, as well as PCI-Express [11]. NVMe-oF enables the creation of a very high performance storage network by allowing flash devices to be shared between servers [12]. For the rest of the dissertation, the reference will be exclusively to the first case, which is the one that will be used and described in the following chapters (chapter 2 and chapter 3).

The purpose of the protocol is the definition of interface registers between host and NVMe controller, the commands that must be supported and the optional ones. The sending of commands is based on the use of circular queues pairs. Each pair is composed by a submission queue and a completion queue. A queue consists of a set of elements of a fixed size. As shown in the figure 1.4, a queue is made up of a Tail pointing to the next free space, and a Head pointing to the next item to be extracted, unless the queue is empty (Head = Tail). A queue is considered full when Head = Tail + 1. Submission queues are used to send messages from the host to the controller. Conversely, completion queues send a message from the controller to the host once the command has completed. The same completion can be associated with multiple submissions, but a submission can have only one associated completion queue. Usually, the queues are allocated on the host memory.

Communication between host and controller is signaled through a Doorbells system. A doorbell is assigned to each queue. Submission doorbells contain the

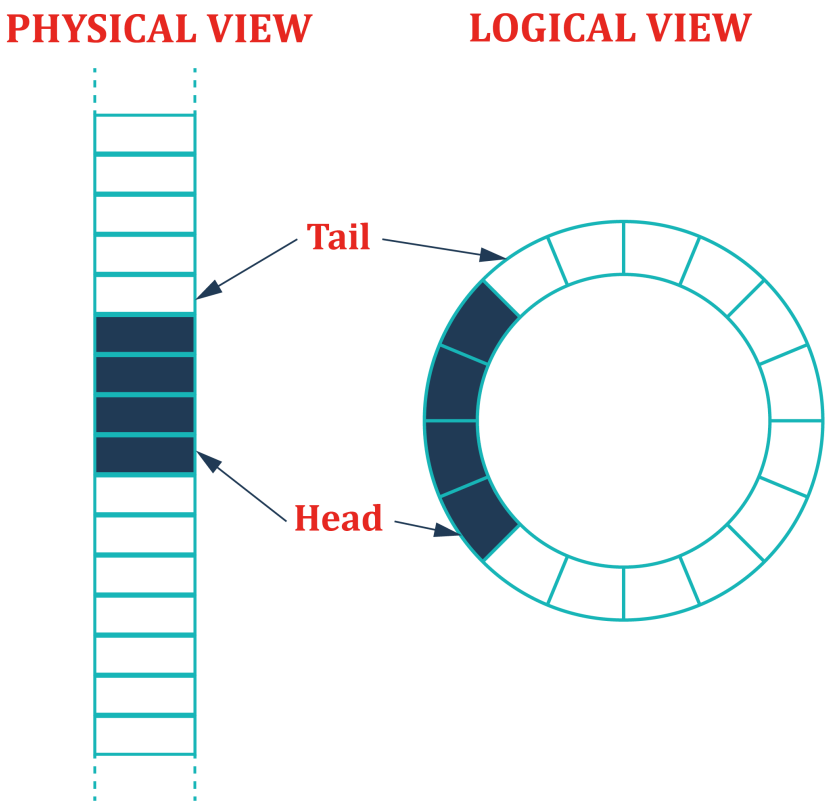

Figure 1.4: Physical view in memory and Logical view for a circular queue.

current Tail value of the corresponding queue. The doorbells of completion, on the other hand, contain the value of the Head of the corresponding queue.

Figure 1.5 shows the command processing cycle:

1. The host inserts the new command, or commands, into the submission queue, in the first available slot, indicated by the Tail;

2. The host updates the Tail value contained within the Doorbell of the corresponding submission queue. This signals to the controller that there are new command(s) in the queue;

3. The controller reads the command(s) in the corresponding queue;

4. The controller executes the command and prepares the future entry for insertion in the completion queue;

5. The controller writes the completion queue entry, or the completion queue entries, starting from the memory location pointed to by the Head indicated in

the doorbell of the corresponding completion queue. In order to distinguish the new advertisements from the previous ones, a field called PhaseTag (Appendix A.1), is complemented;

6. The controller, if specified in the settings, generates an interrupt;

7. The host consumes the new entries inserted in the corresponding completion queue, as long as the PhaseTag remains consistent with the previous one;

8. The host updates the Doorbell Head value of the corresponding completion queue;

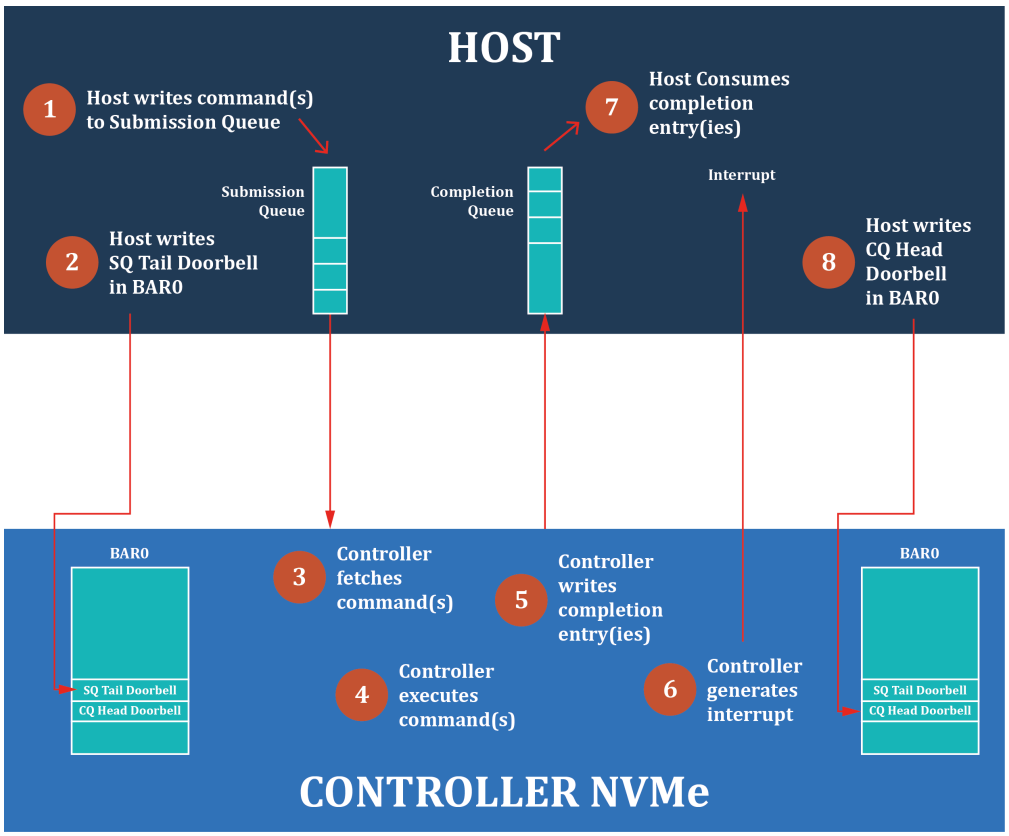

Figure 1.5: Command Processing.

The host can select the arbitration mechanism for selecting the submission queue to be served. All controllers are capable of supporting the Round Robin mechanism. Optionally, the controller may be able to support "Weighted Round Robin with Urgent Priority Class" and/or a vendor specific policy (Appendix A.2).

Submission Queue Entry

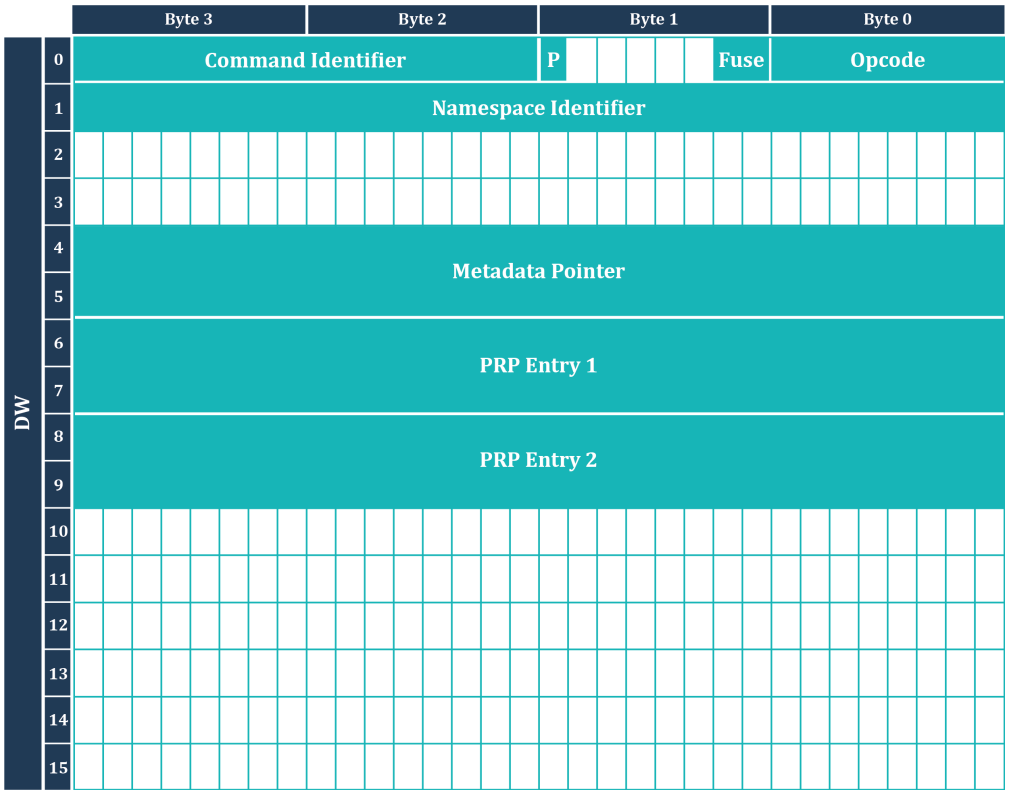

Figure 1.6: Generic Submission Queue Entry.

Each submission queue entry, or submission queue command, is 64 bytes in size, 16 DWs. DWs for a generic submissison queue are defined in Figure 1.6.

- Opcode. Opcode of the command to be executed;
- Fuse operation. Used to merge two commands together. This detail is beyond the scope of this document;
- P. Specifies which method will be used to indicate the addresses for data transfer, whether the Physical Region Page (PRP) (Appendix A.3), or the Scatter Gather List (SGL). In this document, only the first case will be dealt with;
- Command identifier. This field, combined with the submission queue identifier, creates a unique identifier for the command;
- Namespace identifier. Namespace on which the command operates;

- Metadata Pointer. This detail is beyond the scope of this document;
- PRP Entry 1. First PRP entry;
- PRP Entry 2. Second PRP entry, or pointer to a list of PRPs;

Fields not specified are specific fields for individual commands.

Completion Queue Entry

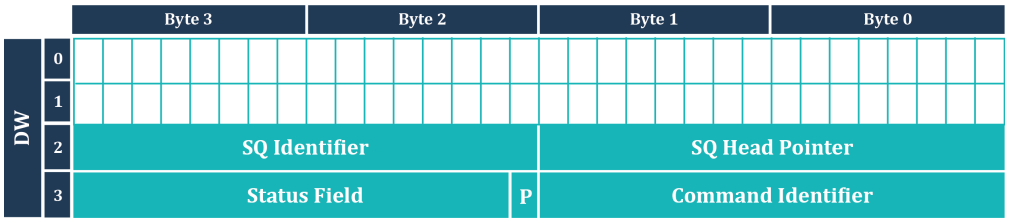

Figure 1.7: Generic Completion Queue Entry.

Each completion queue entry, or completion queue command, is 16 Bytes in size, 4 DWs. DWs for a generic completion queue are defined in Figure 1.6 .

- SQ Identifier. Identifies the submission queue from which the command comes. Fundamental when multiple submission queues refer to the same completion queue;
- SQ Head Pointer. Reports the current value of the submission queue head of the queue specified in SQ Identifier;
- Status field. Indicates the status of the command executed;
- P. Phase Tag to indicate that this completion queue entry is new;
- Command Identifier. Coincides with the command identifier invited by the host in the corresponding submission queue. Together with the SQ Identifier they form a unique identifier;

DW0 and DW1 are specific fields for individual commands.
Two different types of queues are defined:

- Admin Queues;
- I/O Queues;

Admin queues are queues for managing administrative commands. These commands deal with the management of queues and single I/O commands, with the configuration of the controller and with reporting any errors, as well as with the management of the firmware(Optional commands). Each NVMe controller has one and only one admin submission queue (Figure 1.2), to which an admin completion queue is associated. They can contain up to 4K elements. The admin command set is shown in table 1.1.

Command	Required or optional	Category
Create I/O Submission Queue	Required	Queue Management
Delete I/O Submission Queue	Required	Queue Management
Create I/O Completion Queue	Required	Queue Management
Delete I/O Completion Queue	Required	Queue Management
Identify	Required	Configuration
Get Features	Required	Configuration
Set Features	Required	Configuration
Get Log Page	Required	Status Reporting
Asyncronous Event Request	Required	Status Reporting
Abort	Required	Abort Command
Firmware Image Download	Optional	Firmware Management
Firmware Activate	Optional	Firmware Management
I/O Command Set Specific Commands	Optional	I/O Command Set Specific
Vendor Specific Commands	Optional	Vendor Specific

Table 1.1: Admin Command Set.

I/O queues are queues used by the host to send and receive commands for execution. It is possible to have up to 65,535 I/O queues and 65,535 commands for each queue. The I/O command set is shown in table 1.2.

Some commands are optional. The NVMe controller communicates supported commands to the host via the admin identify command.

The control register is located in Bar0 and Bar1 of the PCIe. These interface registers exchange pre-installation and control information between the host and the NVMe controller, which is required for device configuration. Table 1.3 describes the register map for the controller.

Command	Required or optional	Category
Read	Required	Required Data Commands
Write	Required	Required Data Commands
Flush	Required	Required Data Commands
Write Uncorrectable	Optional	Optional Data Commands
Write Zeros	Optional	Optional Data Commands
Compare	Optional	Optional Data Commands
Dataset Management	Optional	Data Hints
Reservation Acquire	Optional	Reservations Commands
Reservation Register	Optional	Reservations Commands
Reservation Release	Optional	Reservations Commands
Reservation Report	Optional	Reservations Commands
Vendor Specific Commands	Optional	Vendor Specific

Table 1.2: I/O Command Set.

Controller initializzation

Now that all the elements have been introduced, it is possible to describe the steps required for the Host to initialize an NVMe controller:

1. Set up the PCI-express registers.

2. Configure the Admin queues, appropriately setting the Admin Submission Queue Base Address (ASQ), the Admin Completion Queue Base Address (ACQ) and the Admin Queue Attributes (AQA) within Bar0. In this way the controller will be able to receive administration commands through the newly created queues.

3. Through information written by the NVMe controller in Controller Capabilities (Cap), the host is aware of the supported arbitration mechanisms and memory page sizes. Therefore, it set the Controller Configuration (CC) register appropriately.

4. Enable the controller via CC.En.

5. Wait for the NVMe controller to finish its internal initialization, signaled by the RDY bit in the CSTS register.

6. Send two admin identify commands on the previously created admin queue, to determine the controller configuration (Identify - Controller data Structure) and the namespace configuration (Identify - Namespace data Structure).

7. Send an admin Set Feature command to determine the number of queues supported by the NVMe controller.

8. Configure the interrupt registers.

9. Create the completion queues first and then the submission queues. It is therefore now possible to send I/O commands via the newly created queues going to modify the doorbell values residing in bar0.

1.1.2 OpenExpress

The work that most allowed us to understand the behavior and structure of an NVMe controller, and inspired the work described in the next chapters was OpenExpress, created by Dr. Myoungsoo Jung (KAIST) who kindly shared the project code with us. In the article "OpenExpress: Fully Automated Hardware-Based Open Search Framework for Fast NVMe Devices of the Future" [13] introduces OpenExpress, an NVMe controller built on FPGAs. It uses a Microblaze CPU as a control unit. The processing path of an NVMe command is completely managed in Hardware. It offers a maximum bandwidth of around 7GB/s without a silicon fabrication. It is suitable for high-speed devices such as magnetoresistive memory (MRAM) and phase change memory (PRAM), in which firmware-based NVMe controllers are not sufficient as they are for Flash memories. The device consists of 3 modules, as shown in figure 1.8:

- Queue dispatching module: it handles the receptions in the submission queues when a value in the doorbells is changed. it fetches submission queue entries from host memory via PCI-Express and decodes the command.

- Data transferring module: it takes care of the actual execution of the command. If required, perform data transfer via DMA between host memory and back-end memory.

- Completion handling module: it is responsible for adding the entry to the completion queue once the command has been executed. It also takes care of sending the interrupt to signal a new queue entry to the Host.

This work was a great inspiration for the realization of the controller described in chapter 2. The division into modules was very useful to make the work more understandable and orderly. The project is open source. It is not supplied entirely, but only the main components. The purpose of the works was not the same, but

the possibility of understanding its structure was of great help. Furthermore, the excellent use of the context table to catalog the characteristics of each queue has been resumed.

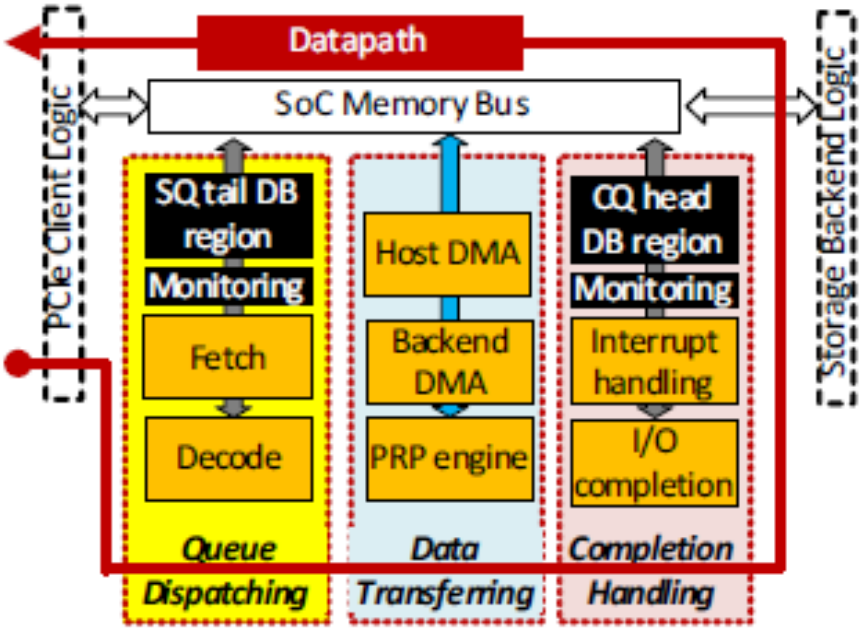

Figure 1.8: "OpenExpress Overview" [13].

1.2 Computational Storage

It is now possible to move on to an analysis of the computational storage protocol. As mentioned, the SNIA document is work-in-progress[5]. However, it provides important definitions and possible architectures that allow the evaluation of the advantages and possible implementations.

1.2.1 Computational Storage Devices

Computational storage devices are architectures that, combined with a storage system, are able to lighten the work done by the CPU and reduce the amount of data exchanged between the processing unit and the storage unit. Figure 1.9 shows different computational storage devices:

a) Computational Storage Processor (CSP). Component that provides the compute service, without providing the storage service. It is associated with a

traditional storage device. Accelerators and storage are on the same PCIe subsystem. In this way the scalability of the two components is independent, Plugs into standard slots and it is possible to exploit the PCIe peer-to-peer to obtain high throughput and low latency [14].

b) Computational Storage Drive (CSD). Devices capable of providing both compute and storage services. Besides the ease of use, it allows to obtain an optimized BW between accelerator and storage [14].

c) Computational Storage Array (CSA). Collection of computational storage devices, which can be both CSP and CSD. In this way it is possible to exploit the benefits of the two architectures [14].

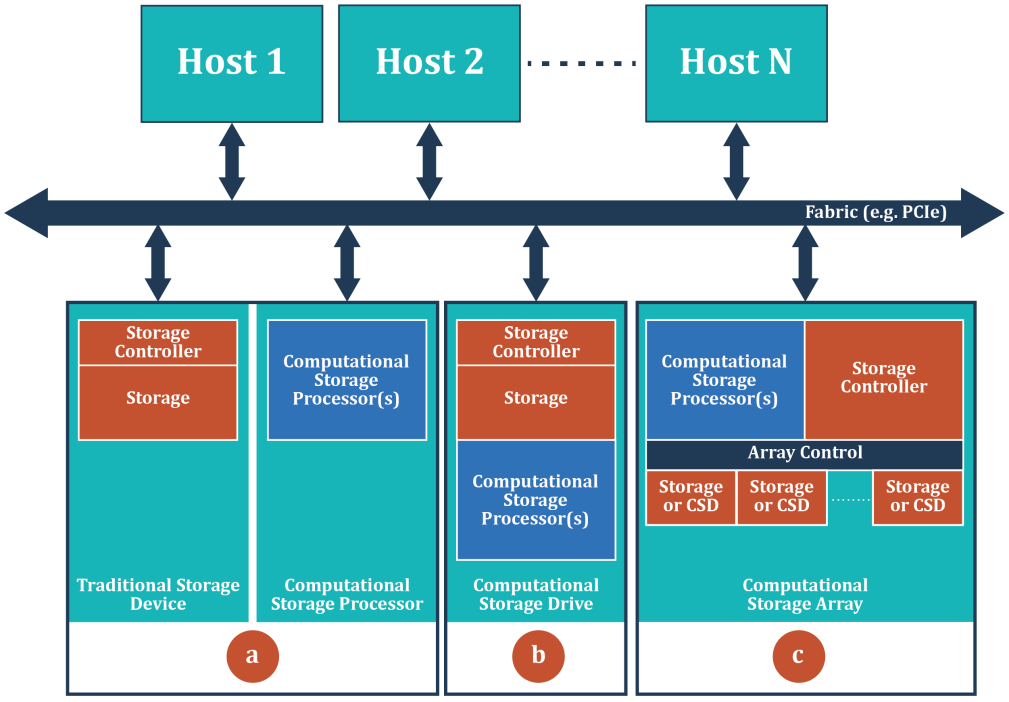

Figure 1.9: Generic Completion Queue Entry.

These devices, regardless of the type, may need to be programmed and therefore called Programmable Computational Storage Service (PCSS). For example, an image of an operating system could be loaded into the device, which can then be programmed according to the user's needs. Alternatively, the device can be a Fixed Computational Storage Service (FCSS). In this second case, no programming is required, the device once configured is ready to be used. Application examples are cryptography, compression or implementation of ordinary expressions.

1.2.2 Theory of Operation

Computational Storage services initialization goes through 3 steps, independent of the type of device, necessary for device identification: discovery, configuration and use.

- Discovery: The device must primarily be discovered. The host must send an acknowledgment command and receive information about the device in response. The controller must specify whether it is a Fixed or programmable device and which functions it can support, thus allowing the Host to adequately prepare the next instructions to be executed.

- Configure. The host now has the information for configuring the device. At this stage the PCSS must be programmed. Instead, FCSSs must receive instructions about the functions to be performed.

- Usage. The device is now configured. The host can continue sending the commands.

1.2.3 Vendors

Several companies have started computational storage projects, both CSD and CSP. Newport Platform is a Computational storage device developed by NGD systems. It is an ASIC that incorporates the functions of an SSD, which can reach sizes up to 64 TB, with a quad-core ARM Cortex-A53 CPU for data processing, making it a PCSS (Programmable Computational Storage Service). The device has the advantage that it can also be used as a simple high-capacity storage device[15]. Other projects are instead based on the use of FPGAs, such as those of ScaleFlux and Samsung. The aim is to exploit hardware reprogrammability to accelerate specific functions. ScaleFlux offers CSD 2000 [16], which uses an FPGA from Xilinx mainly used for in-line data compression and decompression, erasure coding and database analytic functions. Samsung made SmartSSD [17], for video encoding, database acceleration, and machine learning, as well as for compression and decompression. Also in this case the FPGA is Xilinx and the SSD capacity reaches up to 8TB.

The World's first NVMe CSP is from Editicom [18], with NoLoad CSP. It contains an FPGA and DRAM-based processor, but retrieve and process data stored elsewhere on SSD using NVM-Express and the peer-to-peer capability of PCI-Express. The purposes are the same as previously seen for the other devices, with a particular emphasis on cryptography. It can also be used simply for storage management. Nyriad [19] CSP takes a different approach, using GPUs for storage and processing management.

1.3 Document organizzation

In addition to the first introductory chapter just described, the work is made up of three other chapters. The second chapter is dedicated to the presentation of the NVMe Controller created. The first Controller prototype was created in software, using the C language. A program called Test_Host, written ad hoc, and the QEMU machine emulator were used to verify the functionality and actual operation of the NVMe Controller, and to carry out appropriate optimizations. Subsequently, it was adapted for use on FPGAs, orienting itself in a first phase towards a SW implementation, single thread on ARM with the minimum in HW. Later some parts were transformed into hardware to allow an improvement in performance in data transfer. The chapter concludes with the performance evaluation of all the parts previously described. The third chapter discusses the transformation of the device into a Computational Storage Processor (Fixed Computational Storage Service), able to exploit the peer-to-peer capacity of PCI-Express for the implementation of data processing. The last chapter is dedicated to conclusions. Some optimizations and improvements that can be made to the NVMe Controller and Computational Storage are first described, and then the chapter is concluded with the presentation of some possible applications.

Start	End	Symbol	Description
0h	7h	CAP	Controller Capabilities
8h	Bh	VS	Version
Ch	Fh	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
24h	27h	AQA	Admin Queue Attributes
28h	2Fh	ASQ	ASQ base address
30h	37h	ACQ	ACQ base address
38h	3Bh	CMBLOC	CMB Location
3Ch	3Fh	CMBSZ	CMB Size
40h	43h	BPINFO	Boot Partition Information
44h	47h	BPRSEL	Boot Partition Read Select
48h	4Fh	BPMBL	Boot Partition Memory Buf Loc
50h	57h	CMBMSC	CMB Memory Space Control
58h	5Bh	CMBSTS	CMB Status
5Ch	DFFh	Reserved	Reserved
E00h	E03h	PMRCAP	PM Capabilities
E04h	E07h	PMRCTL	PMR Control
E08h	E0B	PMRSTS	PMR Status
E0Ch	E0Fh	PMREBS	PMR Elasticity Buffer Size
E10h	E13h	PMRSWTP	PMR Sustained Write TP
E14h	E1Bh	PMRMSC	PMR Controller Memory Space
E1Ch	FFFh	Reserved	Command Set Specific
1000h	1003h	SQ0TDBL	Submission Queue 0 Tail DB
1000h + (1*(4«DSTRD))	1003h + (1*(4«DSTRD))	CQ0HDBL	Completion Queue 0 Head DB
1000h + (2y*(4«DSTRD))	1003h + (2y*(4«DSTRD))	SQyTDBL	Submission Queue y Tail DB
1000h + (2y*(4«DSTRD))	1003h + (2y*(4«DSTRD))	CQ0HDBL	Completion Queue y Head DB

Table 1.3: Controller Registers definition.

Chapter 2

Controller NVMe

This chapter describes the work done for designing an NVMe controller. The project has been divided in two phases. The first phase consists in the creation of a single thread software version of the controller. The description focuses on the main functions that compose it, as well as the organization of the registers and the Controller Memory. Subsequently, to achieve parallelization of the instructions and an increase in performance, a MultiThread version was created. Initially, the tests to verify the correct functioning of the Controller were entrusted to a firmware (called *Test_Host*). The first program version was written by Enrico Petraglio, R&D engineer at REDS laboratories, who followed the entire project with me. Created with the aim of simulating the behavior of a Host, its flexibility has been exploited to simulate all possible commands supported by the Controller. Subsequently, the QEMU virtual machine was exploited, capable of simulating the behavior of the BIOS and of a real Operating System, and therefore also the initialization and sending of expected commands for an NVMe Controller. In this way, functional tests were much easier than testing them directly on a board. In the second part the controller was adapted for the FPGA. In the beginning it was oriented towards a SW implementation, single thread on ARM with the minimum of HW. Only at a later time some functions were implemented in hardware, to increase performance.

The characteristics of implemented NVMe controller are summarized in table 2.1, which will be used as a reference in the following of the discussion. If not indicated in the table, the value used is the default one described in the protocol [1]. Minimum and Maximum Memory page size are both set to the same value, forcing the host to choose 4096 bytes as the Memory page size. The back-end memory is divided into Logical block (LBA). It is also the minimum amount of data that can be exchanged between controller and host. The LBA format is set to 4096. Having LBAs and pages of the same size made it easier to manage. The maximum size that can be transferred with a single command is set to 131072 (32

LBA). This value could also be higher, it was chosen to keep it low in order not to have a large number of PRP readings in the host memory and not to take up too much space in the controller memory. A possible future optimization could be the variation of this parameter to find an optimal value. The only supported arbitration mechanism is Round Robin.

Vendor ID	10ee
Subsystem Vendor ID	10ee
Serial Number	C0C0
Model Number	NVMeREDS
Max Data Transfer Size	32 LBA
Max Number Of Namespaces	1
Maximum Queue Entries	32
Contiguous Queue Required	yes
Arbitration Mechanism Supported	Round Robin
Memory Page Size Minimum	4096 Bytes
Memory Page Size Maximum	4096 Bytes
Abort Command Limit	4
Async Event Request Limit	4
Submission Queue Entry size	64
LBA Format	4096 Bytes

Table 2.1: Controller NVMe Features.

2.1 First Software implementation

It is possible to divide the NVMe controller into 3 parts. In the first part, the submission queues are managed. In particular, the submission queue to be served is selected. The submission queue entries belonging to that queue are served one at a time. The second part is dedicated to the execution of commands and to the transfer of data, if necessary. The last part deals with the management of completion, in particular the sending of completion queue entries and interrupts. Figure 2.1 shows a behavioral flow chart, useful for explaining how the controller works and will therefore be used as a reference. In the same way, the flow chart of figure 2.2 will be used, in which the main functions performed in a given state are highlighted. In the figures, the "actual states" in which the program can be found are shown in blue, while the "internal states", born from a decision within a state, are represented in orange, in addition to the initial and final states.

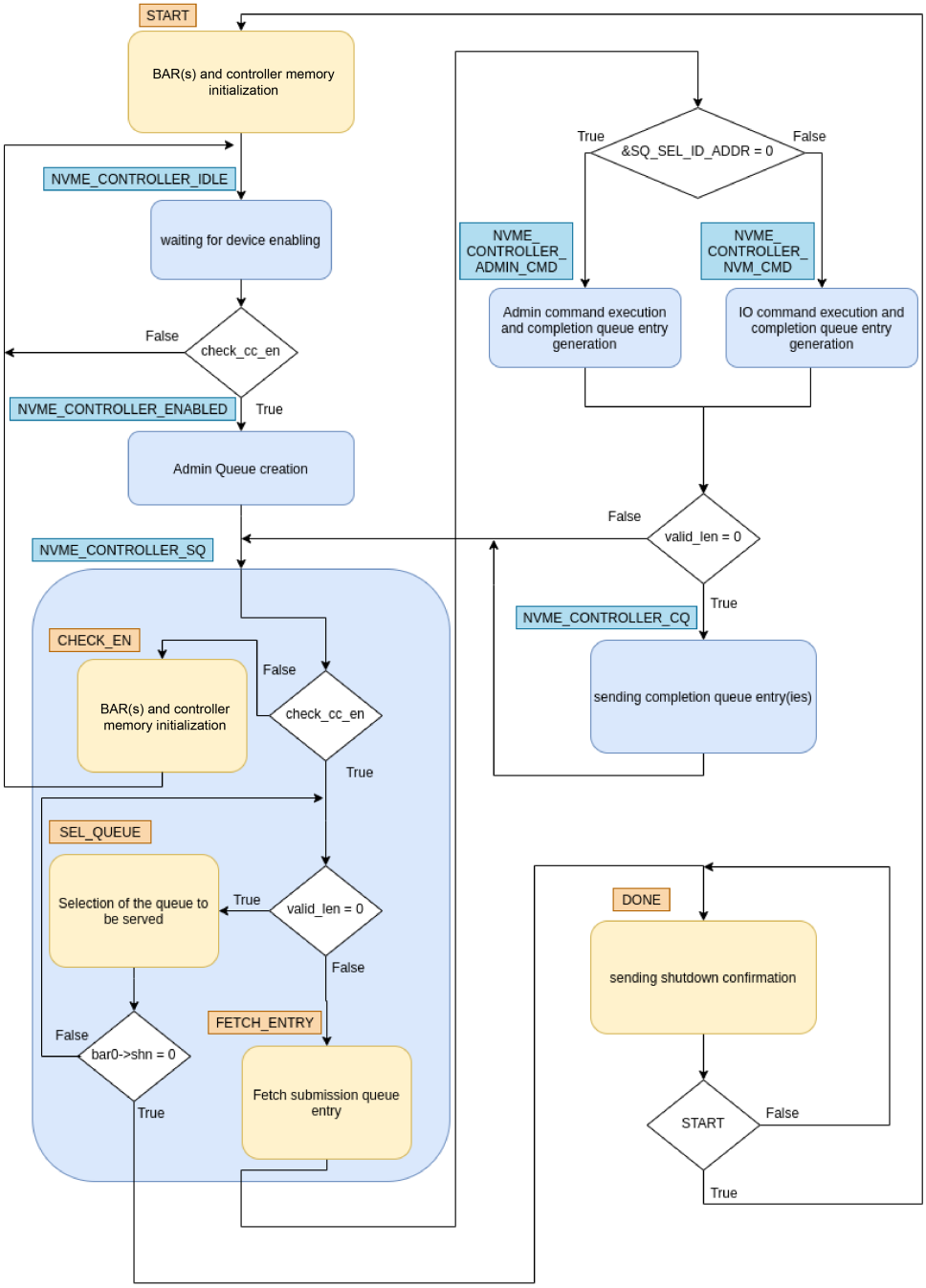

Figure 2.1: NVMe Controller Flow Chart.

First of all, an initialization phase is required. In the *START* state the internal registers, the BAR0 and the controller memory (Figure 2.3) are reset, in order to

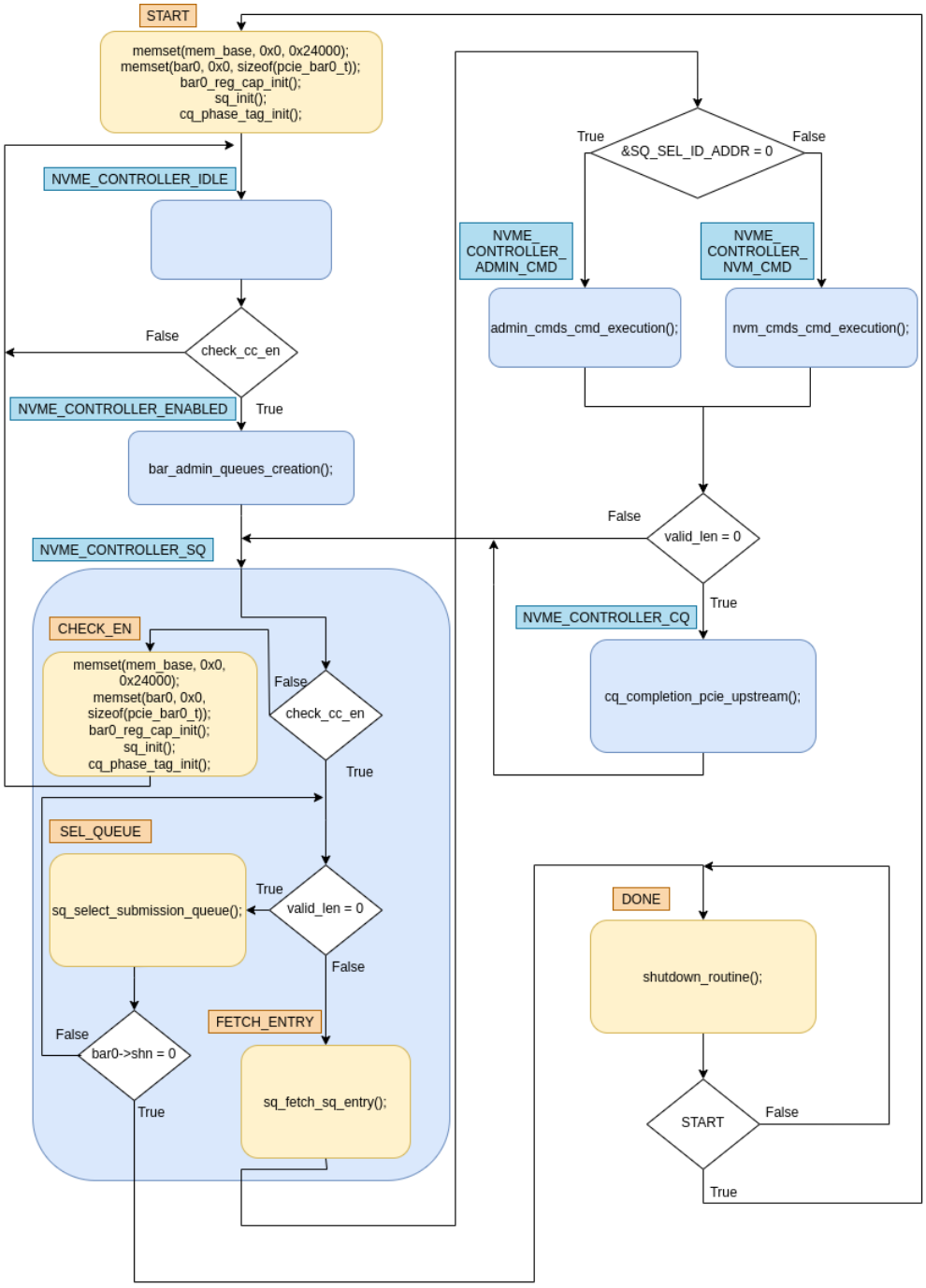

Figure 2.2: NVMe Controller Control Flow chart.

be ready for a new processing. Context tables reside in the Memory controller. These tables are useful for keeping track of all the important characteristics of

the various queues such as the Head, Tail, the queue size, the queue address and which submission/completion queue it is associated with. Each queue, therefore, distinguished by a different ID, has its own context table. Through the function `bar0_reg_cap_init()` the CAP registers in BAR0 are appropriately written. Here the controller communicates to the Host some important characteristics, necessary for the controller initialization. In particular, supported Memory Page Sizes, supported arbitration mechanism and the maximum supported queue size are communicated. In addition, the version of the reference NVMe protocol is specified. `cq_phase_tag_init()` restores a register needed for handling phase tags to the initial value. Each completion queue has a reference bit in this register that represents the phase tag that must be sent for that queue. The correct value of the phase tag to insert in the corresponding completion queue entry is the reference bit contained in the register. Whenever a queue comes to its end and starts from the starting address, the value of the bit corresponding to the queue in the phase tag register is updated. Initially they are all set to the value 1b.

Once all registers are initialized, the *NVME_CONTROLLER_IDLE* status is reached. The controller is waiting for an enable signal from the Host. This signal is located in the BAR0 (`CC.EN`). When it is brought to 1b it means that the host has set all the registers necessary for the BAR for the initialization of the controller, such as the size and address of the admin queues, both submission and completion, as well as the definition of the chosen arbitration mechanism and the size of the pages.

When the enable signal is activated, the controller in free to move to the *NVME_CONTROLLER_ENABLED* state. The controller now has all the information necessary to create the context tables related to the admin queues. Queues are actually created and allocated in Host memory, but the controller, by creating the context table entry for that queue, becomes aware of it and recognizes it as such. This is the task of the `bar_admin_queue_creation()` function, which also takes care of sending a ready signal to the Host via the BAR0 (`CSTS.RDY`) to confirm the recognition of creation of the admin queues.

Now the NVMe controller initializations are finished and the core of the algorithm can begin with the *NVME_CONTROLLER_SQ* status. It represents the first step of the three described above, that is the management of the submission queues. First of all, the enable signal is checked. If it is brought to 0 it means that the host needs to restart the controller which should therefore return to the *NVME_CONTROLLER_IDLE* state, after having reset all the registers and the memory and having reinitialized the CAP entry of the BAR0. The next step depends on the value of `valid_len`. `valid_len` is initialized to 0 and is needed to discriminate whether there are submission queue entries that still need to be served or not. The first time in this state therefore surely `SEL_QUEUE` is reached. Here, the `sq_select_submission_queue()` function selects the queue to be served

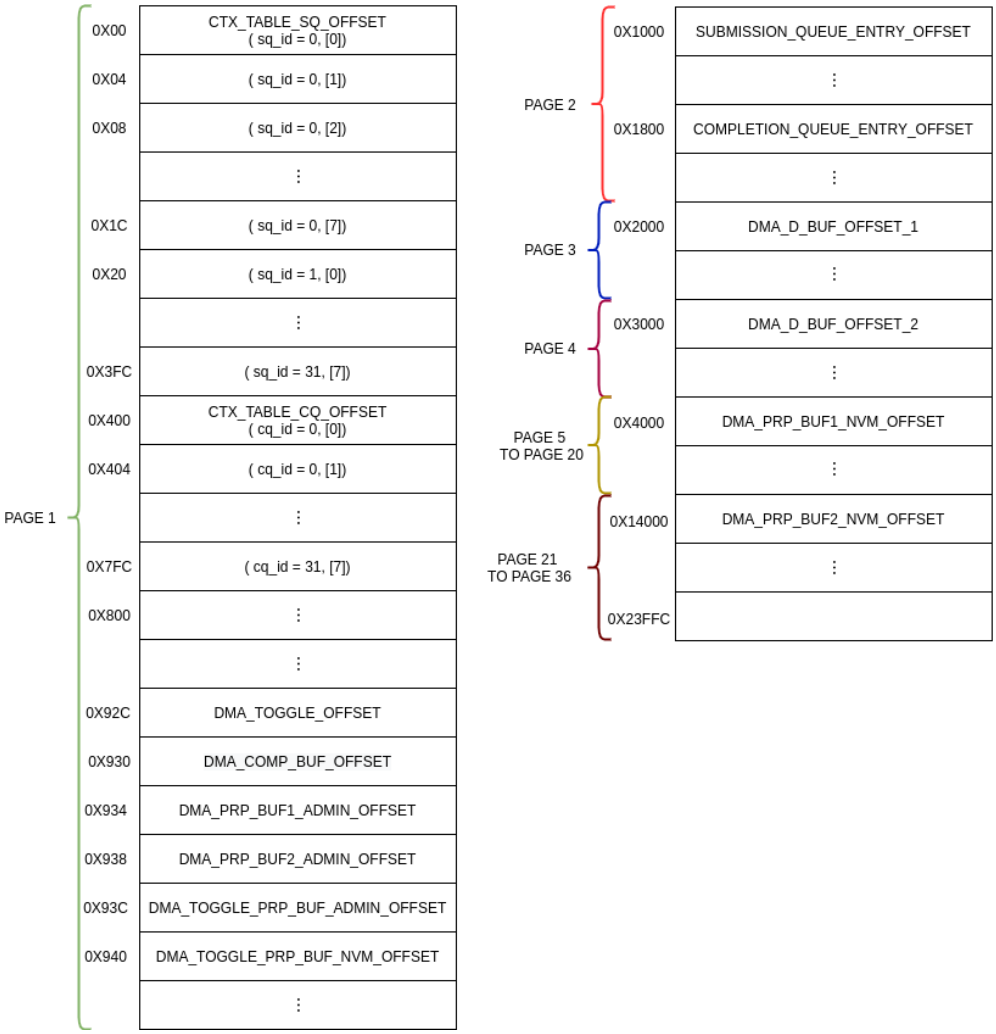

Figure 2.3: NVMe Controller Memory Map.

according to the Round Robin arbitration mechanism. The Host communicates which submission queues need to be served by updating the TAIL value through the doorbell of the corresponding queue in the space provided in the BAR0 (table 1.3). A queue is considered empty if the value of the HEAD is equal to that of the TAIL. The controller knows the value of the HEAD of the queues. It can therefore compare this value with the one communicated through the doorbells to find out if any queue has entries to execute. Once the queue is selected, the controller assigns a value to `valid_len` equal to the difference between TAIL and HEAD. In this way it is possible to know how many entries of the selected queue must be served. Furthermore, `sq_select_submission_queue()` retrieves the entries from the Host memory and copies them to `SUBMISSION_QUEUE_ENTRY_OFFSET` address

of the controller memory, keeping the order unchanged.

If all the queues are empty, a check on the `shn` bit of the BAR0 is performed. This bit is set to 1 when the HOST intends to turn off the NVMe controller. In this case the controller has the duty to conclude all the entries of all the queues that have been inserted in the submission queues and to start the shutdown routine, carried out by the *shutdown_routine()* function. The controller informs the host that the shutdown has occurred without problems by appropriately setting an entry in the BAR0 (`shst`).

At this point, the queue to be served has been selected and the value of `valid_len` is non-zero. *FETCH_ENTRY* is reached. The `sq_fetch_sq_entry()` function selects the command, or entry, to be executed. It has been chosen to continue in order, starting from the entry in the HEAD + 1 position and continuing up to TAIL. However, this is not mandatory. Once selected, the value of `valid_len` can be decremented by 1. The command is ready to be decoded. The `SQ_SEL_ID_ADDR` address contains the ID of the selected queue, useful for selecting the next status. If the content ID is 0 then the server queue is an admin queue. This leads to the *NVME_CONTROLLER_ADMIN_CMD* state. Otherwise, the *NVME_CONTROLLER_NVM_CMD* status is reached.

The `admin_cmds_cmd_execution()` function is the heart of the Admin state. Initially, it decodes the command discriminating among those supported. With reference to table 1.1, the Queue Management commands act directly on the context tables, deleting or creating them. The Configuration commands, on the other hand, have the task of characterizing the controller, providing the Host with information regarding the characteristics of the device. Status Reporting is used by the controller to report Controller health messages and errors. Finally, the Abort command is used to cancel a certain entry of the indicated submission queue. As specified in table 2.2, some commands must write information in the Host memory at a certain specified address. In the Single Thread case, when one of these commands must be executed, it is necessary to consider the PRP1 and PRP2 fields of the command (Figure 1.6). The NVMe controller creates a list of PRPs needed to send requested data. This list is saved in a buffer in the controller memory (`DMA_PRP_BUF_OFFSET`). If the amount of data is able to reside within a single PRP, also taking into account the offset, then the list is made up of a single item. If not, PRP2 completes the list. The maximum amount of data that can be exchanged through an admin command is 4096 Bytes, which is also the size of a Host page (value forced by the controller settings). Consequently, two pages are always sufficient to contain the data coming from an Admin command. A buffer called `DMA_D_BUF_OFFSET_1` is filled with the information to be sent to the Host memory. At this point, the contents of the buffer are copied to the correct host memory address.

In case the ID queue is different from 0, an I/O command must be executed. The

Command	Code	Reading required	Data Size
Delete_io_sq	A_0	N	-
Create_io_sq	A_1	N	-
Get_log_page(lid=0x02)	A_{2_2}	Y	512B
Get_log_page(lid=0x03)	A_{2_3}	Y	512B
Get_log_page(lid=0xC0)	A_{2_C}	Y	512B
Delete_io_cq	A_4	N	-
Create_io_cq	A_5	N	-
Identify(cns=0x00)	A_{6_0}	Y	4096B
Identify(cns=0x01)	A_{6_1}	Y	4096B
Identify(cns=0x02)	A_{6_2}	Y	4096B
Abort	A_8	N	-
Set_feature(fi=0x01)	A_{9_1}	N	-
Set_feature(fi=0x07)	A_{9_7}	N	-
Set_feature(fi=0x08)	A_{9_8}	N	-
Set_feature(fi=0x0B)	A_{9_B}	N	-
Get_feature(fi=0x04)	A_{A_4}	N	-
Get_feature(fi=0x0C)	A_{A_C}	Y	256B
Async_event_Req	A_C	N	-

Table 2.2: Admin Commands supported.

`nvm_cmds_cmd_execution()` function in the *NVME_CONTROLLER_NVM_CMD* state first takes care of decoding the command. A generic I/O command described in figure 2.4 are distinguished by four basic fields:

- PRP Entry1: first PRP in host memory to write/read.
- PRP Entry2: second PRP in host memory where to read/write. It can represent a list of PRPs. (More in appendix A.3).
- Number of Logical Blocks: number of logical blocks to be read/written.
- Starting LBA: it indicates the first LBA of the backend memory from which the operation must start.

The minimum size of the exchangeable data is an LBA. The Number of Logical Blocks establishes the size of the data to be transferred. If a Page in the host

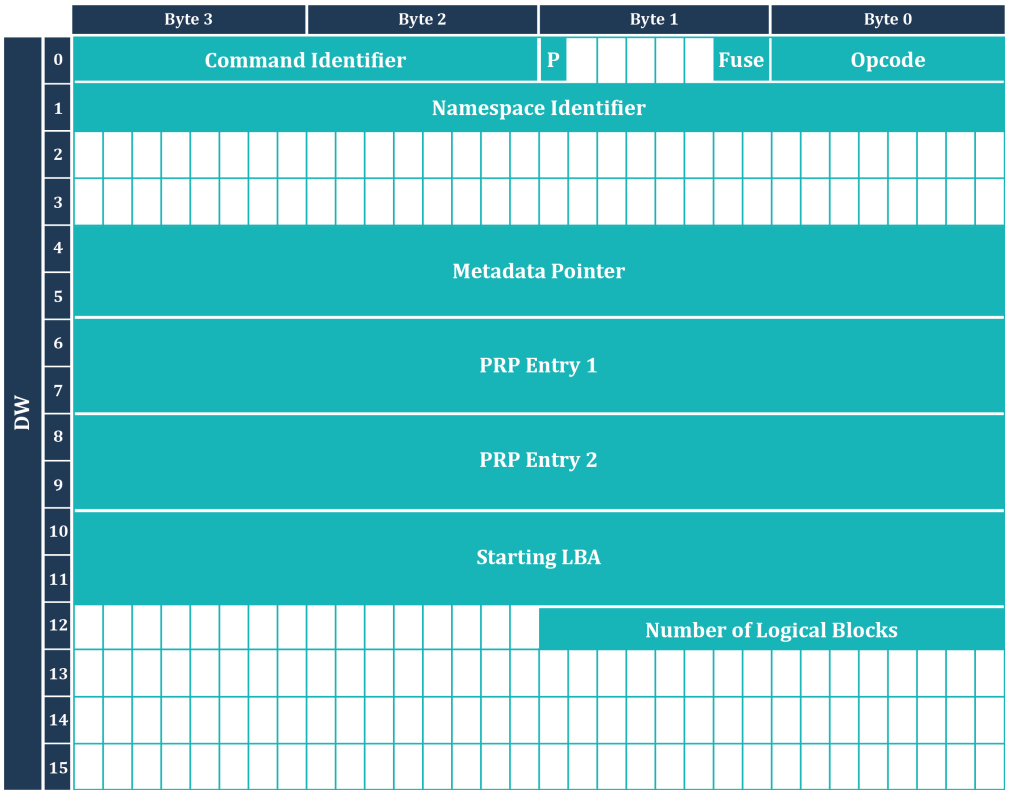

Figure 2.4: Generic I/O Command.

memory is able to contain the amount of data to be exchanged, PRP1 is sufficient for the operation and the PRP2 field is not used. If, on the other hand, the size of data to be transferred is larger than a page but less than 2, the second PRP is used as the address for sending the Bytes exceeding the first page. If not even two PRPs are sufficient to contain the amount of data to be exchanged, then the second PRP is interpreted by the NVMe controller as a pointer to a list of PRPs residing in the host memory. The NVMe controller, once it becomes aware of the amount of data to be exchanged by decoding the I/O command, creates its own list of PRPs, in which all the PRPs necessary for the operation are copied. In case only one PRP is needed, a list with only one element is created. In the Single Thread case, this list is copied into the `DMA_PRP_BUF_ADDR` buffer, the same one used for the admin commands requesting the sending of data. The controller now has all the information it needs to perform a data transfer between host memory and backend memory.

Now that the command has been executed, it is time to create the completion queue entry. In reference to the figure 1.7, the information regarding the corresponding submission queue entry, such as SQ Identifier and command Identifier,

are taken from it. The phase tag is suitably set following the value contained in the phase tag register. If there have been no errors, the Status Field reports the value 0x0, otherwise the appropriate error code is sent. If necessary, the host will send an admin command *Get_Log* to get more information about this error. The newly created completion queue entry is stored in the controller memory. Only once all entries have been served will all created completion queue entries be sent to host memory in the corresponding queue.

At this point, if the value of `valid_len` is different from zero, it means that there are still entries from the same queue to be served. The submission queue HEAD is therefore updated and the new entry is fatched. Otherwise, All entries have been served, the *NVME_CONTROLLER_CQ* status is reached, so that the created completion queue entries are sent to Host memory. This is the task of the `cq_completion_pcie_upstream()` function, which copies the contents of the Controller_memory (`DMA_COMP_BUF_OFFSET`) into the completion queue in Host memory. It also takes care of sending the interrupt message. The interrupt vector, useful for identifying the queue sending it, is a parameter passed by the Host when creating a completion queue. The NVMe controller keeps this information in the Context table of the corresponding queue.

2.2 Second Software Version: adding multithreading

A Multithreaded version was created with the aim of increasing performance, parallelizing operations and simulating the behavior of a DMA engine. Copying data between host memory and device is the bottleneck that needs improvement. The idea is to entrust the task of managing this transmission to a thread that works in parallel to the main thread which will instead keep the task of fetching and decoding new submission queue entries as well as the generation of completion entries. A semaphore enforces the synchronization between the two threads. As shown in Figure 2.5, if the transmission channel between host and controller is occupied by a previous read/write command, the new command that intends to use the same channel will be put on hold.

Some changes to the command path are necessary to prevent resources from being occupied and therefore not available when one of the two threads needs them. This is the case of the buffer for the PRPs (`DMA_PRP_BUF_OFFSET`), and of the buffer `DMA_D_BUF_OFFSET_1` with the Admin data to be sent, which risk being overlapped by the next command before it is completed. To overcome this problem, a double buffer system has been adopted, introducing toggles for buffer arbitrage.

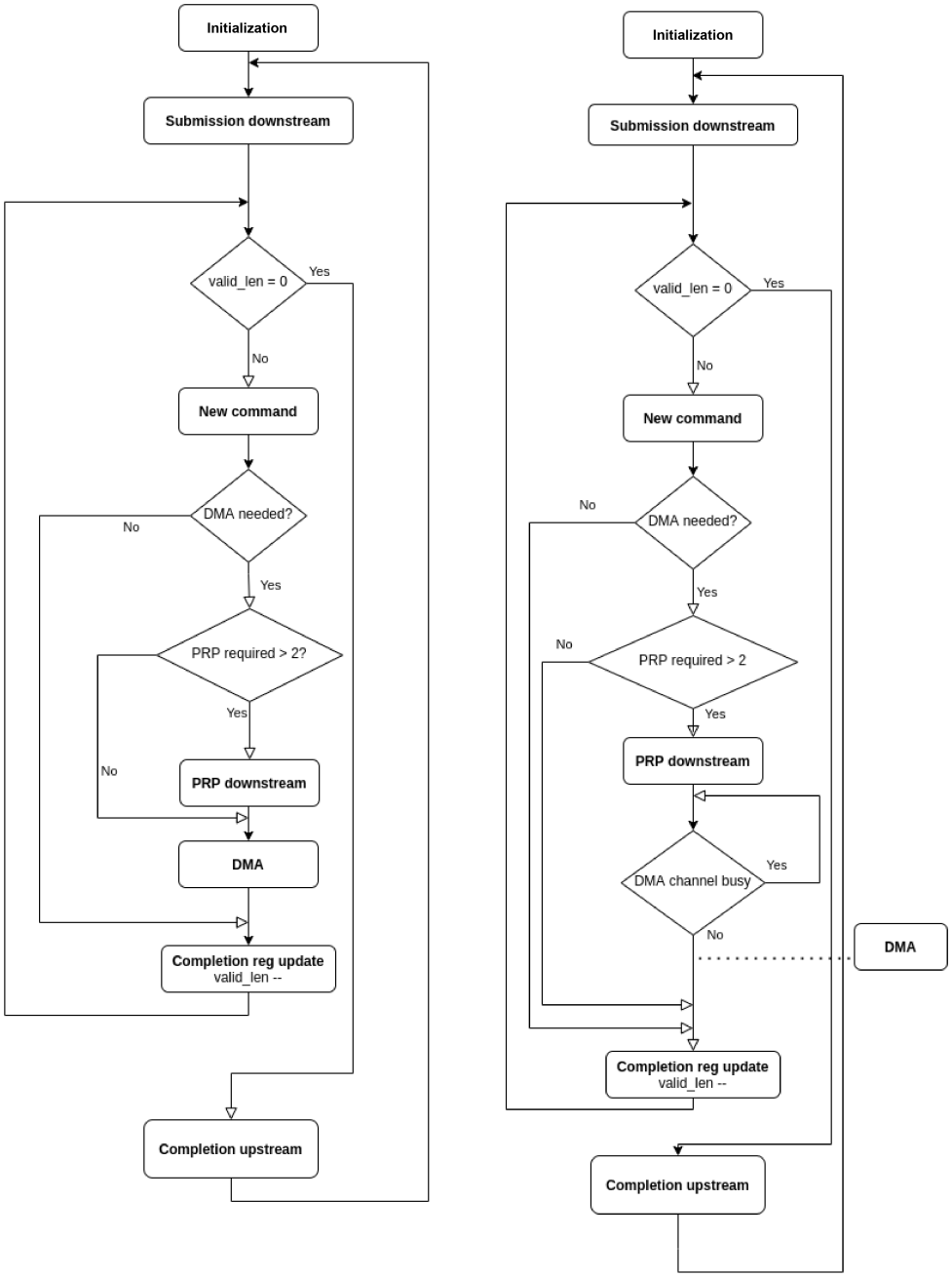

Figure 2.5: On the left, a simplified flow chart of the command cycle in the single Thread case. On the right, a flow chart of the comand cycle in the MultiThread case.

2.3 Test process

It is possible to divide the Test part into two sections. In the first, a simple program called Test_Host was created that was able to simulate as much as possible the behavior of the Host towards the NVMe controller. In the second, the QEMU virtual machine and its features were used to simulate a connection between a real operating system and the NVMe Controller, as described below. For this phase, the help of Rick Wertenbroek, R&D Engineer at REDS laboratories, with whom I had the opportunity to collaborate, was fundamental. Thanks to a system He built (described in Appendix A.5) He provided valuable information about the commands exchanged during the initialization of an NVMe device captured using the WireShark software, which is able to intercept the signals exchanged between two devices(in this particular case they were NVMe-OF commands, which were however very useful to know the expected commands). For the second part, He initialized the QEMU environment, allowing it to expose its RAM to an external process.

2.3.1 Test_Host

Test_Host is a simple program with the purpose of simulating the behavior of a host towards the NVMe Controller. Communication is entrusted to shared memories. In particular, two are used:

- BAR0_shm: shared memory to simulate the BAR0 register of the PCIe.
- DDR_shm: shared memory used to simulate host memory. The addresses of the queues reside in this memory area, as well as the addresses sent in the read and write commands.

The addresses sent by Test_Host do not represent the actual addresses but the offsets with respect to the base addresses of the shared memories. With reference to figure 2.6, first of all, the Test_Host, writes the necessary information in the BAR0_shm, such as size and offset of the admin queues (1). The controller reads this information from shared memory (2). Now, the host fills the admin queues with the desired command (3) and waits for an interrupt signal, passed for convenience to a location not used for these purposes in BAR0. The controller reads the command (4), executes it and creates the completion queue entry (5). It is written to the DDR_shm in the completion queue. An interrupt signal is sent. The Host receives the interrupt and reads the completion queue in the DDR_shm (6) in search of the new entries received, based on the value of the phaseTag, as done by a real Host. Having full control of the commands sent to the controller, it was possible to send them individually so as to evaluate their correct functioning. It was also

easy to observe that the behavior of the queues was as expected, with a look at particular cases, such as the end of queue and the sending of multiple entries at a time. Thanks to the commands obtained from Wireshark, a controller initialization similar to the real case was simulated, checking that the responses were consistent.

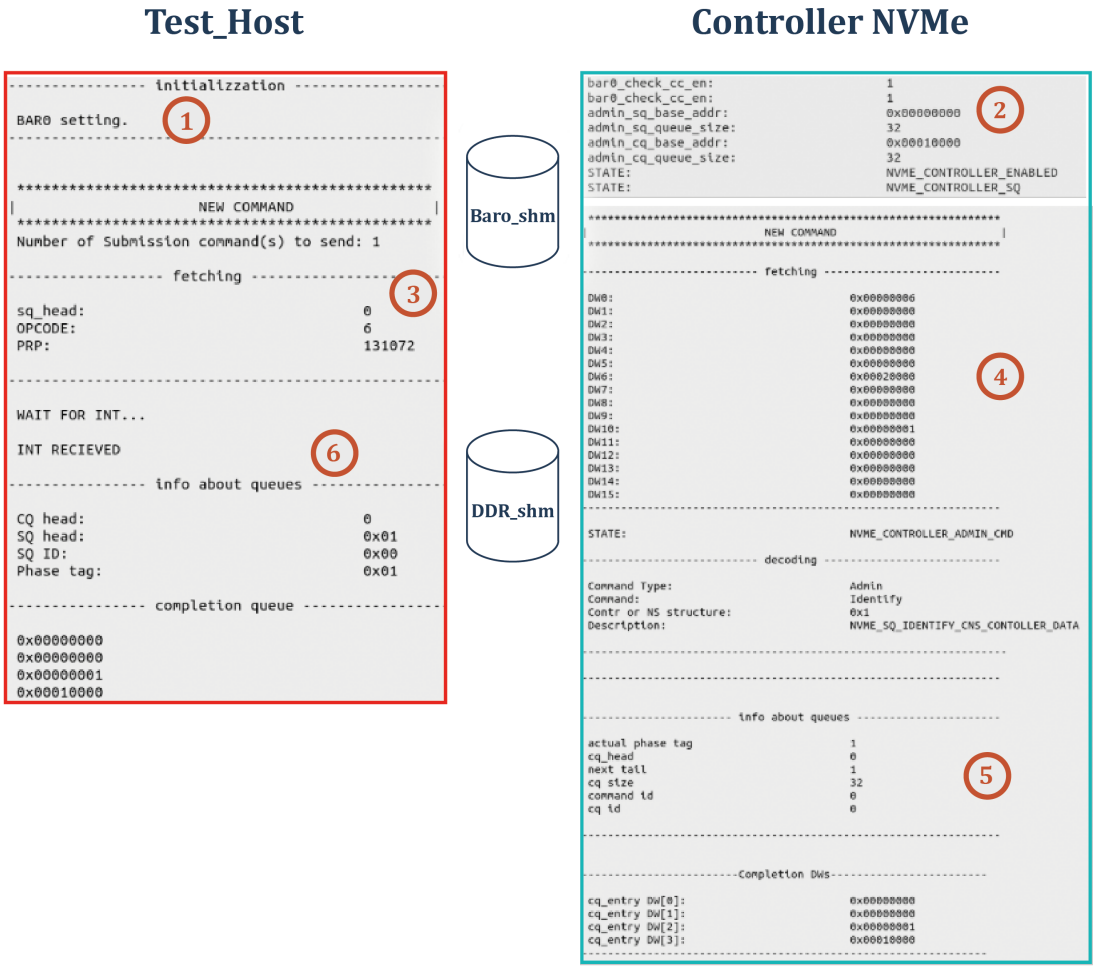

Figure 2.6: Example of communication between Test_Host and Controller_NVMe. In particular, an admin Identify command is sent.

2.3.2 QEMU

QEMU [20] is a free and open source emulator and virtualizer capable of performing hardware virtualization. The structure used, described in an article published on the REDS blog [21], makes it possible to communicate between the RAM of the

hardware platform emulated on QEMU and an external process and, in this specific case, with the NVMe Controller. A shared memory file was used to represent QEMU RAM, using the QEMU memory-backend-file option. A more realistic emulation would have to access host memory via PCIe, but this is the fastest and easiest way to access memory directly. A shared memory file was also used for the BAR0 space. The code was thus minimally modified, maintaining the structure of shared memories. The use of QEMU made it possible to discover and test the initialization commands for an NVMe Controller on PCIe by a BIOS and an Operating System. The BIOS requires 2 Identify commands to find out the general characteristics of the device and the size of the supported queues, and then proceed with the creation of two I/O queues, one completion and one submission. Subsequently, it brings the value of CC.En to 0, resetting the Controller. The routine performed by the operating system (Ubuntu 16.04) provides a first identification of the device through an Identify command and of the health status with a Get Log health information command. The maximum number of queues that the host can create is set through a Set Feature and finally two completion and submission queues are created. At this point, the Host can send I/O commands. The first reading is made at LBA 0 of the NVMe Controller in order to become aware of the Master Boot Record (MBR). The MBR is a 512 Bytes disk space, which contains information necessary for booting an operating system present on the disk and on the existing partitions. For simplicity, existing partitionsit has been copied from a real NVMe disk, and the space dedicated to partitions has been cleared. After discovering the MBR, the Host sends a long series of reads to verify the correct functioning of the queues, also trying to read several LBAs with a single instruction, and the size of the Namespace. Figure 2.7 shows an example of the system implemented with QEMU.

2.4 Tracing

For the Tracing phase, Uftrace [22] was used, a tool capable of analyzing and tracing programs written in C and C++ language. It is inspired by the ftrace framework of the Linux kernel, so the use was intuitive. In particular, two features proved to be very useful:

- uftrace dump –flameGraph. A Flame graph was obtained through this function. It allowed to quickly identify the most frequent paths of the code [23]. The more you climb to the top, the deeper you go into the stack. Each rectangle represents a stack frame. The larger the base of the rectangle, the greater the time spent in that function. During development, this graph was very useful for understanding which functions the time was spent in in order to make the best possible optimizations.

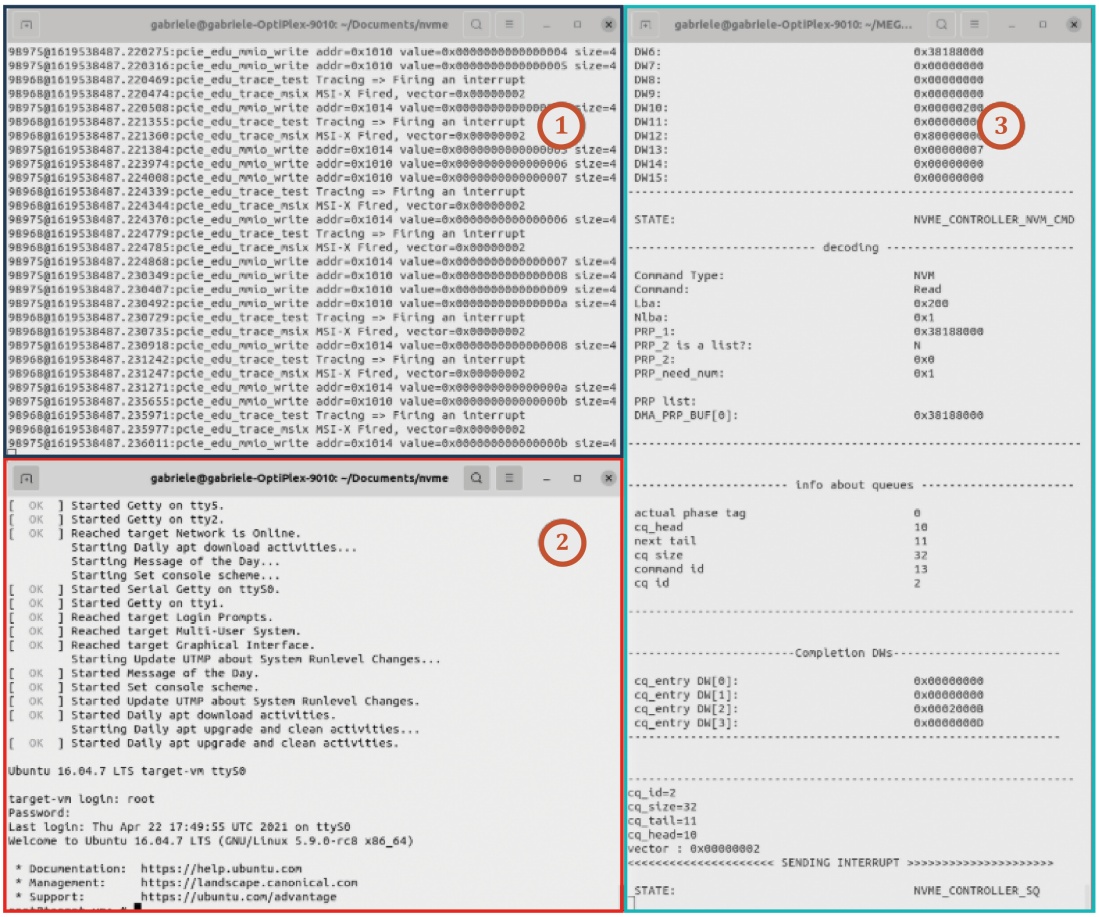

Figure 2.7: Example of the system implemented with QEMU. (1) Tracing of QEMU reads and writes in BAR0 and received interrupts. (2) QEMU outputs. (3) NVMe Controller outputs.

- uftrace dump –chrome. This function for creating a json file that can be read by Chrome Trace Viewer was important in obtaining a complete view of the time progress of the software.

The section is divided into 2 parts. In the first one, a comparison between the Single Thread and Multi Thread version is presented, obtained through the use of Test_Host. In the second part, some important functions in the management of a memory are tested through the QEMU platform presented.

2.4.1 Comparison between Single Thread and Multi Thread versions using Test_Host

In the following paragraph, the tracing of two significant simulations, called *Simulation 1* and *Simulation 2*, is described. The purpose is to emphasize, in the first part, the functions that are used most frequently and, in the second part, the difference in performance between single thread and multi thread cases. Test_Host was used for these simulations, in order to have more freedom on the commands sent. Both simulations have in common 7 admin commands, necessary to initialize the controller, the queues and their characteristics. Generally, these are the first commands that are sent by the operating system at boot time. In particular, the following commands are sent:

- 2 identify commands, to identify controllers and namespaces.
- 1 create_completion_queue, to create a complition queue I/O.
- 1 create_submission_queue, to create a submission queue I/O.
- 1 get_log_page to be notified of any errors.
- 2 set_feature, to set the number of completion and submission queues.

All these commands are loaded into the admin submission queue consecutively, so the queue is selected only once for all commands, which are obviously fetched one at a time.

In *Simulation 1*, after the 7 admin commands, a Read command is sent on the created submission queue. The command has the following characteristics:

- Namespace = 1;
- LBA statring = 0;
- number of LBA = 1;

Then the reading takes place in the Namespace with ID 1, starting from LBA 0 and the size is 4 KBytes.

Simulation 2 is also simply a reading, as well as the initial admin commands. This time the features are:

- Namespace = 1;
- LBA statring = 0;
- number of LBA = 32;

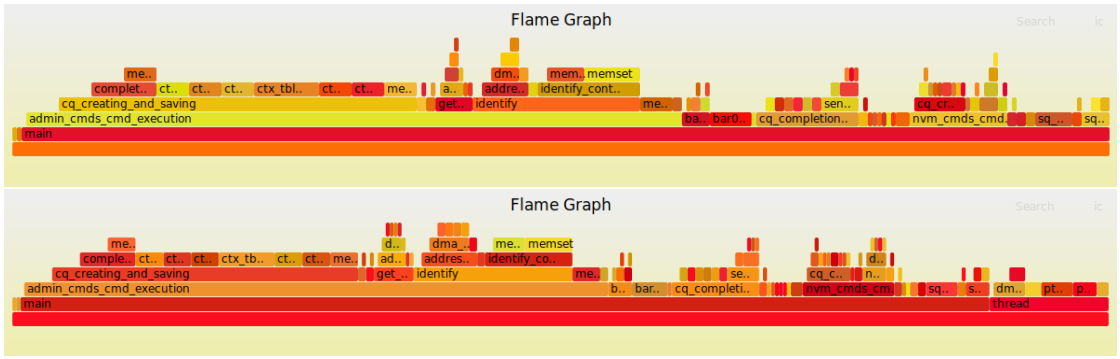

Figure 2.8: *Simulation 1* (7 admin commands, 1 reading 4 KB). Above, Flame graph Single Thread version. Below, Flame graph Multi Thread version.

the size is now 32 LBA (128 KBytes). It is the maximum size for a single I/O command, set for the NVMe controller.

Figure 2.8 shows the *Simulation 1* Flame Graph, single and multi Thread cases. In both cases most of the time is spent inside the admin_cmds_cmd_execution function, where the admin functions are executed, as expected. Some of them require data transfers and, for this reason, they need time to be implemented. This is the case with identify and get_log_page commands. The admin command rectangle in the single Thread case is longer than in the multithreaded case(59.75% and 53.26%, respectively). This is due to the fact that, in the Multithread case, the write operations in the host memory performed by the dma_dma function are managed to a dedicated thread, called dma_Thread. Likewise, the nvm_cmds_cmd function rectangle decreases in width, although more difficult to notice. The difference in size of these rectangles in the two cases is not very marked, 8,90% in SingleThread version against 8.33% in MultiThread one. The reason is that only one I/O command is executed and it requires the smallest possible data transfer (1 LBA).

In *Simulation 2*, figure 2.9, Single Thread case, the increase in size of the `nvm_cmds_cmd_execution` rectangle is evident. Over 19.7% of main is spent in this function. Using the thread allowed the percentage to be lowered to 13%. As a result, the Thread rectangle is increased, inheriting the `dma_dma` function. These graphs have allowed us to see where to apply many optimizations and others are still certainly achievable. The creation of the admin queues and the saving in memory (`cq_creating_and_saving` function) still requires a lot of time in percentage terms, it is evident in all the graphs. Once all completion queue entries have been collected they still need to be submitted by the `cq_completion_pcie_upstream` function, which again takes time. One possibility of optimization could consist in sending the entries to the host memory as soon as possible, as soon as they are created,

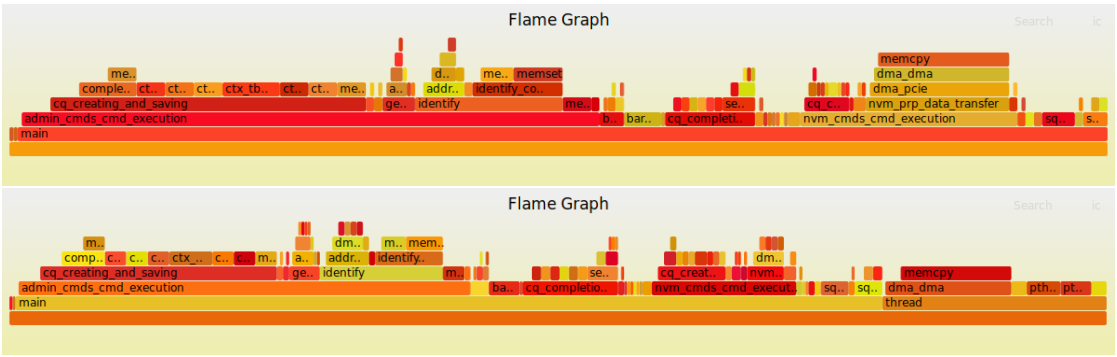

Figure 2.9: *Simulation 2* (7 admin commands, 1 reading 128 KB). Above, Flame graph Single Thread version. Below, Flame graph Multi Thread version.

without saving them in the controller memory, leaving the only task of sending the interrupt to the `cq_completion_pcie_upstream` function, to signal that all the entries have been performed.

The evaluation of the graphs over time has allowed us to appreciate the improvements obtained with the MultiThread version. Particularly interesting is the behavior of the read command. With reference to the *Simulation 1*, Figure 2.10 shows the details of the graphs over time of execution of the I/O command. In the Single Thread case, the time taken to execute the `nvm_cmds_cmd_execution` function is 12,691 us. The internal function that takes care of the transfer is called `dma_dma` and resides in the only available thread, the main one. On the contrary, in the MultiThread case, the task of transferring the data through the `dma_dma` function is entrusted to `dma_thread`. The time savings are minimal, 0.725 us. The `phtread_cond_signal` function takes time and reading a single LBA is insufficient to notice any advantages in the MultiThread version. The results of the *Simulation 2* in figure 2.11 instead underline in a more marked way the time savings obtained thanks to the thread. The time taken in the `nvme_cmds_cmd_execution` function in the Single Thread case is almost 2.5 times compared to the MultiThread case, with 87.922 us compared to only 35.475 us. The `dma_dma` function, which occupied most of the function, is now executed by the thread and therefore all the data transfers, clearly visible in the figure.

Comparing *Simulation 1* and *Simulation 2* of the corresponding versions, it is easy to see that part of the execution of the command is spent on the creation of the PRP list thanks to the `nvm_fetch_prp` function. In *Simulation 1* the time is very short, 0.461 us and 0.357 us. In this case, in fact, the controller has to copy only one LBA and therefore needs only one PRP. The page address is passed directly into the command of the submission queue, the controller simply has to copy this value from the command to the list, which will consist of only one

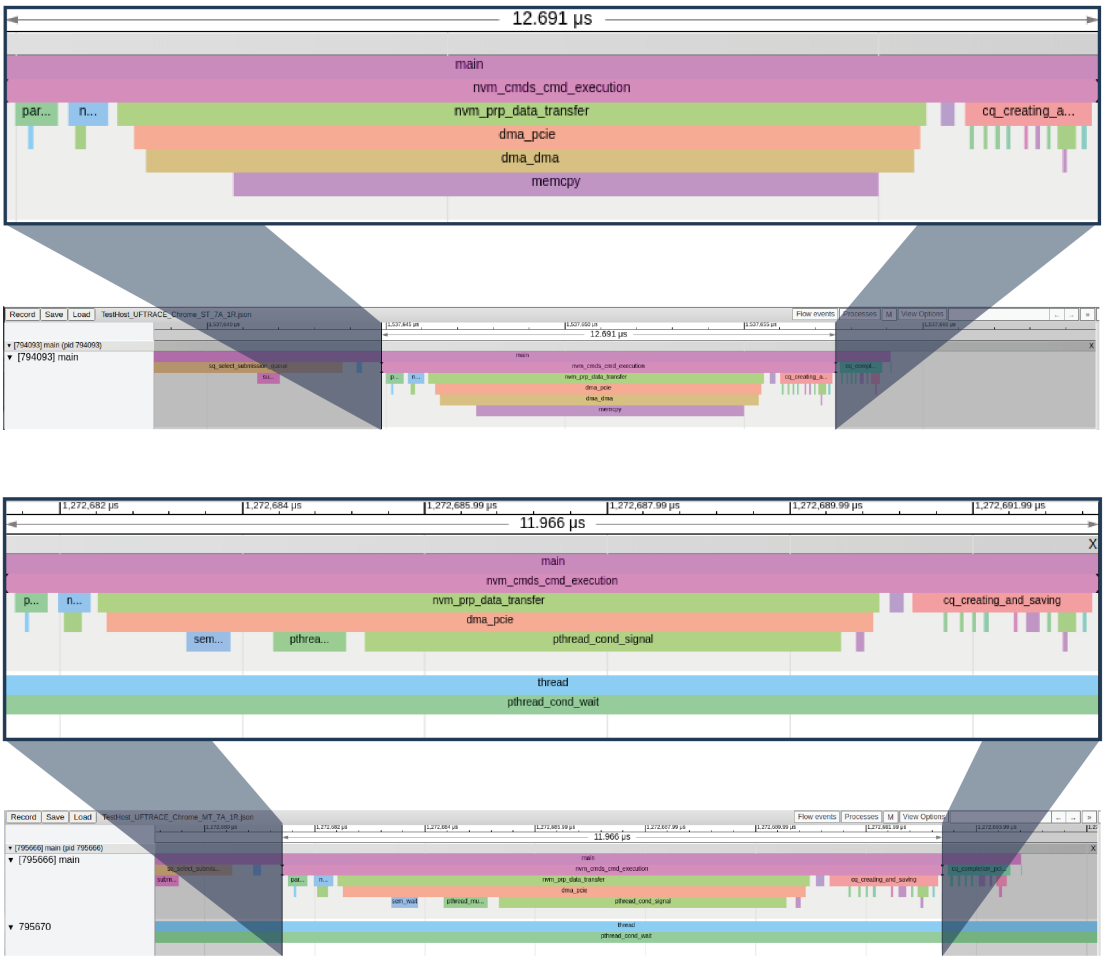

Figure 2.10: Above, reading one LBA (4 KB) in Single Thread. Below, reading one LBA (4 KB) in Multi Thread.

element. Not surprisingly, therefore, the greater importance this function assumes in *Simulation 2*. Not only is a single PRP insufficient, but even two PRPs cannot contain the totality of the data. Hence, the second PRP is to be understood by the controller as an address to a PRP list that is to be copied from host memory to the controller's PRP list. The operation takes time and the function is satisfied after 4,796 us (Single Thread) and 6,405us (MultiThread).

The simulations just presented are intended to underline the advantages of the MultiThread version created. As shown, if a single command requires the exchange of a large amount of data, in this specific case 32 LBA, the time gain is evident.

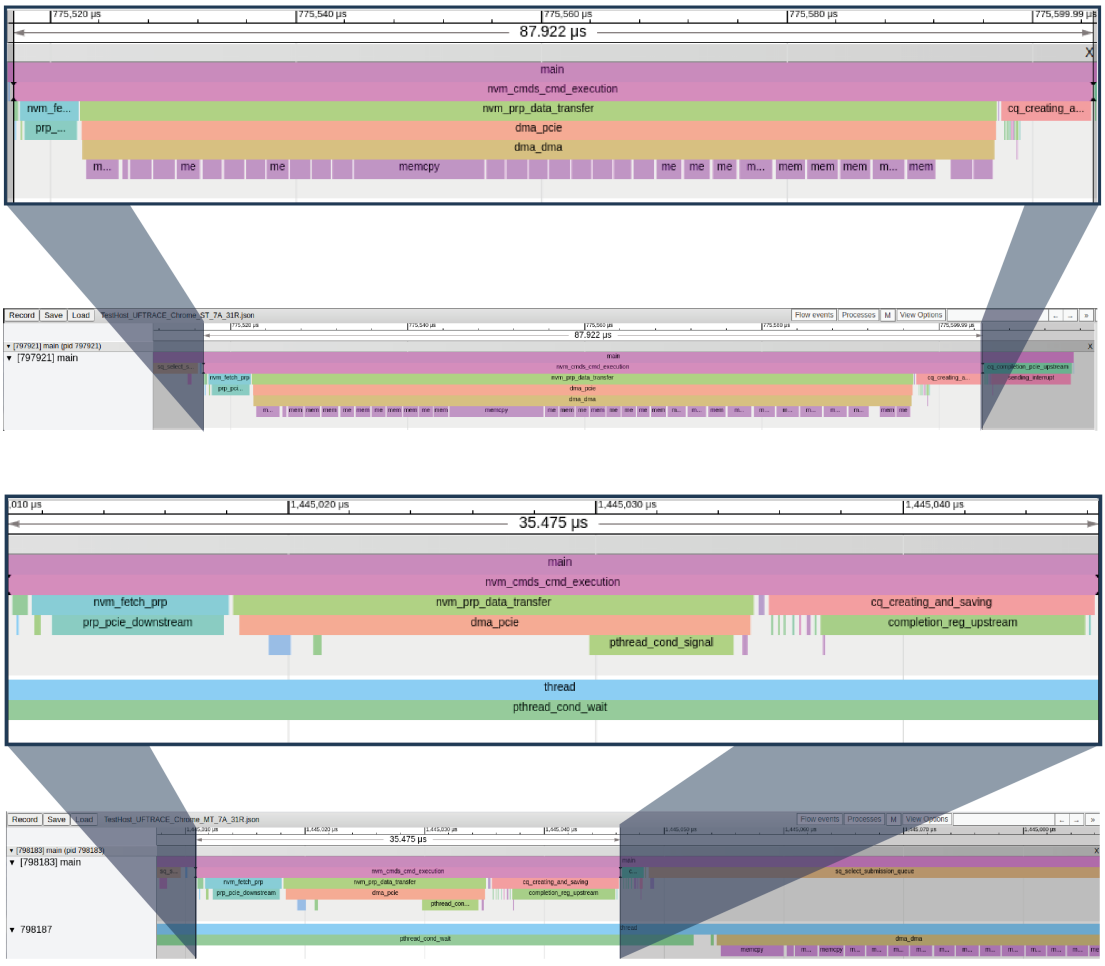

Figure 2.11: Above, reading 32 LBA (128 KB) in Single Thread. Below, reading 32 LBA (128 KB) in Multi Thread.

The same cannot be said for multiple small transfers. If the simulation had included 32 different commands, each with 1 LBA, using one thread would not have been so effective. A reading would in fact have to wait for the end of the previous one to be able to access the `dma_thread`. So the advantages in terms of time would not be significant.

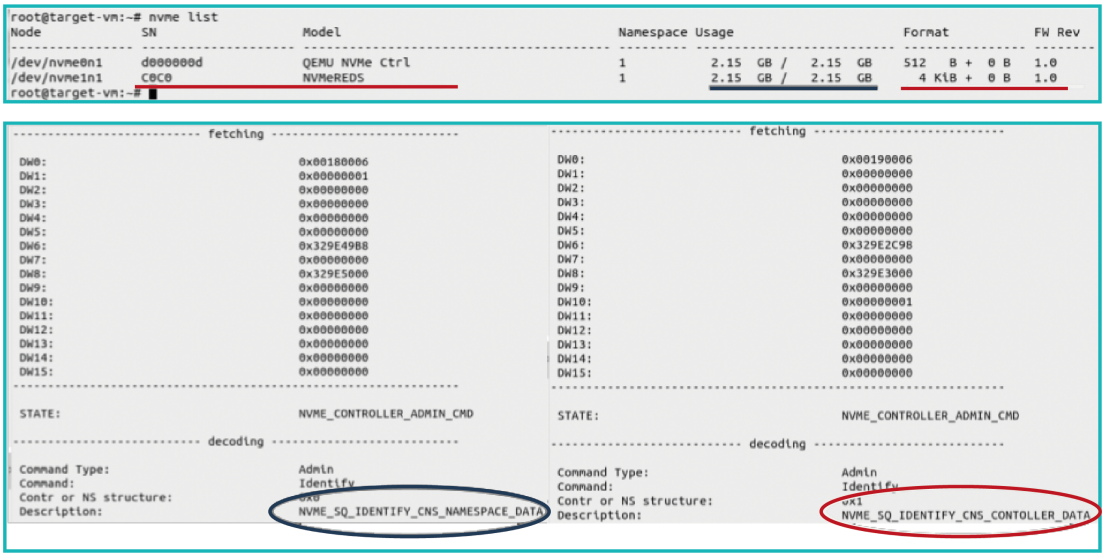

Figure 2.12: Above, the output obtained via the NVMe list command on the host terminal. Below, the admin identify commands received from the NVMe controller.

2.4.2 Function analysis through QEMU

The following section aims to verify the functionality through the QEMU platform. In particular, various utilities, intended as Linux commands, were tested to understand what the requests are and verify the responses of the NVMe controller. Two significant examples are now described, the first for identifying the controller and the second for writing and reading. The example of figure 2.12, describes the response to the `nvme list` command, of the NVMe-cli package, useful for getting to know the main characteristics of the NVMe devices connected to the Host. There are only two commands sent to the controller:

- Identify Namespace data (in blue): used to know the device namespace number and their size. As mentioned, in the case described the name space is 1 of size 2.15 GB.
- Identify Controller data (in red): informs the host of the device name, model, version and LBA format.

This information is written by the controller in the PRPs specified in the submission queue entries to DW6 and DW8. It was the first command tested, it ensured communication between host and NVMe controller.

To test the read and write functions, the utility `dd` was used, capable of copying the specified amount of data from an input file to an output device and vice versa.

```
root@target-vm:~# dd if=MINE_LBA0 of=/dev/nvme1n1 bs=4096 count=1 seek=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.0317539 s, 129 kB/s
root@target-vm:~#
```

```
-------------------------- fetching --------------------------

DW0:                                    0x00000001
DW1:                                    0x00000001
DW2:                                    0x00000000
DW3:                                    0x00000000
DW4:                                    0x00000000
DW5:                                    0x00000000
DW6:                                    0xB95C8000
DW7:                                    0x00000000
DW8:                                    0x00000000
DW9:                                    0x00000000
DW10:                                   0x00000001
DW11:                                   0x00000000
DW12:                                   0x00000000
DW13:                                   0x00000000
DW14:                                   0x00000000
DW15:                                   0x00000000
------------------------------------------------------------

STATE:                                  NVME_CONTROLLER_NVM_CMD

-------------------------- decoding --------------------------

Command Type:                           NVM
Command:                                Write
Lba:                                    0x1
Nlba:                                   0x1
PRP_1:                                  0xB95C8000
PRP_2 is a list?:                       N
PRP_2:                                  0x0
PRP_need_num:                           0x1

PRP list:
DMA_PRP_BUF[0]:                         0xb95c8000
------------------------------------------------------------
```

Figure 2.13: Above, the dd utility sent by the host. Below, the write command received from the Controller.

A 4 KB file named MINE_LBA0 was written entirely in the NVMe controller LBA1. Considering the size of the file, two PRPs are enough to hold the data. In the cases described, no offset is specified in PRP1, so the second PRP is unused. The number of LBAs required is in fact always 1. As shown in Figure 2.13, the controller has successfully received the write request to the desired LBA. DW6 and DW8 represent the PRP in which the controller can find the data to be written in the backend memory. Once the writing has been completed, a reading routine, identical to the one performed during the initialization of the operating system, is executed, useful for verifying once again the correct response of the device.

The test continued with reading the first controller LBA. The controller receives the command to read LBA 1 (Figure 2.14)before starting the usual test read routine. The data is written to a file called MINE_LBA1, so it can be compared with the initial file, MINE_LBA0. As expected, the two files are identical.

```
root@target-vm:~# dd if=/dev/nvme1n1 of=MINE_LBA1 bs=4096 count=1 skip=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB, 4.0 KiB) copied, 0.000725459 s, 5.6 MB/s
root@target-vm:~#
```

```
--------------------------- fetching ---------------------------

DW0:                                    0x000C0002
DW1:                                    0x00000001
DW2:                                    0x00000000
DW3:                                    0x00000000
DW4:                                    0x00000000
DW5:                                    0x00000000
DW6:                                    0xBAF51000
DW7:                                    0x00000000
DW8:                                    0x00000000
DW9:                                    0x00000000
DW10:                                   0x00000001
DW11:                                   0x00000000
DW12:                                   0x80000000
DW13:                                   0x00000007
DW14:                                   0x00000000
DW15:                                   0x00000000
------------------------------------------------------------------

STATE:                                  NVME_CONTROLLER_NVM_CMD

--------------------------- decoding ---------------------------

Command Type:                           NVM
Command:                                Read
Lba:                                    0x1
Nlba:                                   0x1
PRP_1:                                  0xBAF51000
PRP_2 is a list?:                       N
PRP_2:                                  0x0
PRP_need_num:                           0x1

PRP list:
DMA_PRP_BUF[0]:                         0xbaf51000
------------------------------------------------------------------
```

Figure 2.14: Above, the dd utility sent by the host. Below, the read command received from the Controller.

2.5 Performances (QEMU)

As mentioned, the work done up to now with Test_Host and with QEMU has had the purpose of verifying the functioning and understanding the requests received by the Host, without dwelling on performance. Also this section has the same purpose, to exploit a tool for evaluating the performances like `Fio` to be able to verify the functionality in a more in-depth way, looking for errors that had not yet been encountered. `Fio` is a tool capable of generating a series of threads and

processes that perform I/O actions according to specification. Samsung used it in the SmartSSD userGuide [17] to show the performance of their product. The same settings used in the guide for `Fio` will be applied to the NVMe Controller in the chapter dedicated to performance on FPGA, but they have also been used on QEMU and described here to validate the considerations made previously on the SingleThread and MultiThread versions. `Fio` was found to be a very useful tool for verifying the functionality, it allowed to confirm that the controller did not lock in some state due to escaped bugs as well as to obtain a first view of the performance of the device. The following `Fio` features, the same ones used in the SmartSSD UserGuide, were used for all the tests described:

- ioengine = libaio: the way in which the I/O are issued is the native asynchronous way of Linux.
- iodepth = 256: Number of I/O units to keep flying over the file.
- direct = 1: Unbuffered I/O.
- numjobs = 12: number of processes/threads executing the action.
- runtime = 60: execution time.

The bs option, the block size per I/O unit was initially set to 4KB. Four tests for each version of the controller were done, with the intention of sending random reads, random writes, sequential reads and sequential writes. Table 2.3 shows the results for the single Thread version, while table 2.4 shows those of the MultiThread case. RW specifies which type of read/write is performed. IO indicates the amount of I/O done in total, BW shows the observed bandwidth.

RW	IO	BW
Random read	675868 KB	11257 KB/s
Random write	701120 KB	11681 KB/s
Sequential read	699776 KB	11659 KB/s
Sequential write	705336 KB	11741 KB/s

Table 2.3: Results obtained through `Fio` in the Single Thread version, bs = 4KB.

As expected, there are no substantial differences between the random and sequential cases. There are no advantages or disadvantages of choosing one type of transfer over the other. Comparing the two tables it is clear that the Multi Thread case does not bring advantages in terms of performance, but rather in some cases it worsens them slightly. Random reads go from a bandwidth of 11257 KB/s

RW	IO	BW
Random read	560076 KB	9331 KB/s
Random write	716464 KB	11936 KB/s
Sequential read	569460 KB	9488 KB/s
Sequential write	757988 KB	12623 KB/s

Table 2.4: Results obtained through `Fio` in the Multi Thread version, bs = 4KB.

in the SingleThread case to just 9331 KB/s in the MultiThread case. The same behavior occurs with sequential reads. Writings, on the other hand, have a slight improvement both in the random case (from 11681 KB/s to 11936 KB/s) and in the sequential case (from 11741 KB/s to 12623 KB/s). This is due to the fact that bs is set to 4 KB so reads/writes are done on only 1 LBA at a time. Each command the controller receives requires a transfer of 1 LBA at most. The main thread gets to request the use of the communication channel managed by the `dma_thread` before it has even finished. The improvements are not obvious, quite the contrary.

RW	IO	BW
Random read	20328 MB	346835 KB/s
Random write	18224 MB	310681 KB/s
Sequential read	19334 MB	320995 KB/s
Sequential write	18345 MB	310998 KB/s

Table 2.5: Results obtained through `Fio` in the Single Thread version, bs = 128KB.

Subsequently bs was therefore raised to 128 KB, with the intention of forcing the host to request larger readings of the single LBA, which needed PRP lists up to the maximum limit of 32 LBA. Tables 2.5 and 2.6 show the results. The performance in the case of Multi-Thread improved significantly in all four tests carried out, precisely because the host started sending 128 KB read and write requests which therefore require transfers of more LBAs at a time and not just one. The random reads go from 346835 KB/s to 373871 KB/s, as well as the sequential reads which register a +51716 KB/s. Both writes also follow the same trend going from just over 310000 KB/s to over 360000 KB/s. This confirms the hypothesis made in the previous sections: The Multi-thread version involves appreciable improvements only if the single read and write command requires a sufficiently large amount of data.

For comparison, the same `Fio` commands with bs = 4KB were used on a QEMU

RW	IO	BW
Random read	21916 MB	373871 KB/s
Random write	21258 MB	362555 KB/s
Sequential read	21888 MB	372711 KB/s
Sequential write	21228 MB	360451 KB/s

Table 2.6: Results obtained through `Fio` in the Multi Thread version, bs = 128KB.

native NVMe simulated device. This was useful for verifying that performance was consistent with existing simulated NVMe devices. The results reported in table 2.7 show worse performances than those obtained with the NVMe Controller reported in table 2.3.

RW	IO	BW
Random read	465784 KB	7758 KB/s
Random write	437388 KB	7280 KB/s
Sequential read	422452 KB	7034 KB/s
Sequential write	493024 KB	8213 KB/s

Table 2.7: Results obtained through `Fio` in a QEMU native NVMe simulated device, bs = 4 KB.

2.6 FPGA implementation

The last phase of the NVMe Controller project was the implementation on FPGA. The Xilinx Vivado [24] software suite was used for synthesis and analysis. The Vivado IP integrator made it possible to quickly integrate and configure IP, Intellectual Property, from the Xilinx IP library. The Advanced eXtensible Interface (AXI), multi-master and multi-slave communication interface, is used. AXI uses well defined master and slave interfaces that communicate via five different channels:

- Read address
- Read data
- Write address
- Write data

- Write response

The channels are shown in figure 2.15. The address channel contain address and control information when performing a basic handshake between master and slave. A master reads and writes data to a slave. The read response information is inserted into the read data channel, while the write response information has a dedicated channel. In this way the master can verify that a write transaction has been completed. Each data exchange is called a transaction. A transaction includes address and control information, data sent, and any response information. Actual data is sent in bursts that contain multiple transfers [25].

Initially the intent was to bring the NVMe Controller design to a Zynq processor, test its functionality and evaluate its performance, and then gradually adding hardware components to obtain optimizations. Xilinx FPGA ZC706 was used. The first version made used the following main IPs:

- Axi memory mapped to pci express: for communication with the pci express.
- Block Memory Generator to generate a memory space (BRAM) used as BAR0 for communication between host and NVMe controller.
- Zynq-7000 processing system: the heart of design. Via the Vivis platform, the NVMe controller script is loaded onto the processor.
- Axi Interconnerct: for the connections of the various blocks.
- Axi Uart lite: to be able to control the operation of the Controller via the terminal.

The code did not need many changes, other than an adaptation for the translation of the addresses between the AXI domain and PCIe domain and vice versa. A 250 MB BAR Axi has been opened to manage the addresses referring to the host memory. In this window the address of the PCIe space is mapped, while two Axi Base Address Translation Configuration Register are used to contain the most significant part of this address. Each time a PRP is supplied, it must therefore be transformed into a Bar Axi address and the Translation configuration addresses appropriately set.

Figures 2.16 and 2.17 show the Design Layout with the blocks used.

2.7 Performances (FPGA)

As already mentioned above, the performance of Samsung's SmartSSD, presented in the userGuide [17], has been used as a reference and shown in table 2.8.

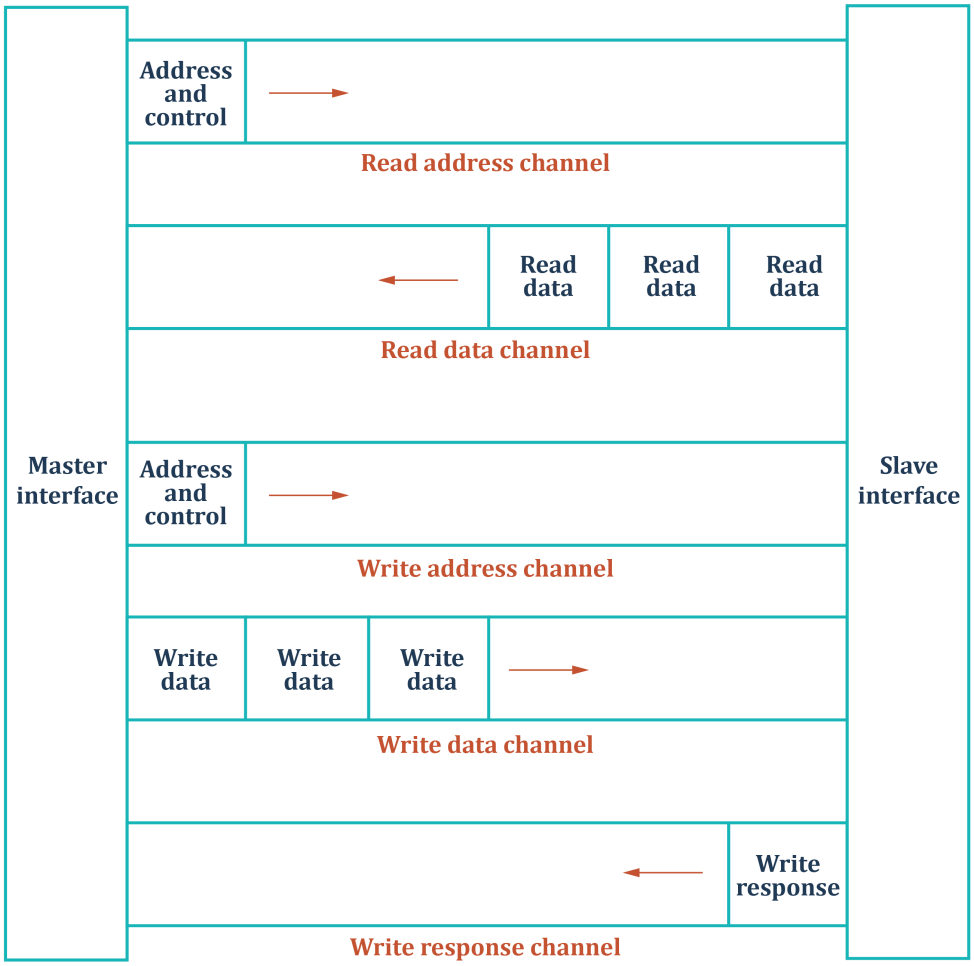

Figure 2.15: Axi Interface example: A read and write burst consisting of 4 beats or data transfers.

Through `Fio`, the performance of the device subject to repeated reads and writes for 60s both random and sequential was evaluated as previously done in the simulations with QEMU. The following `Fio` features were used for all the tests described:

- ioengine = libaio.
- iodepth = 256.
- direct = 1.

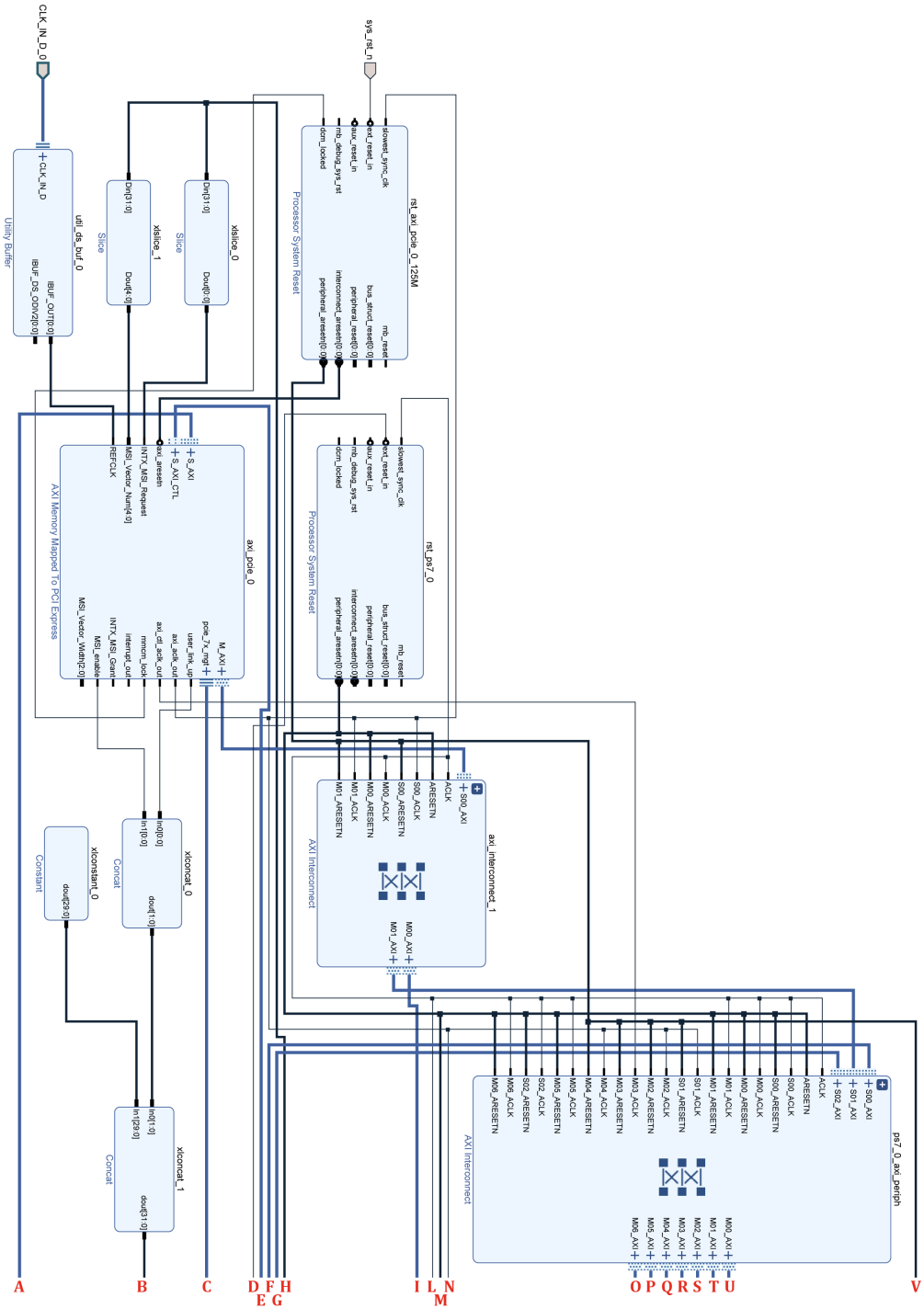

Figure 2.16: NVMe Controller Design Layout for FPGAs - part 1.

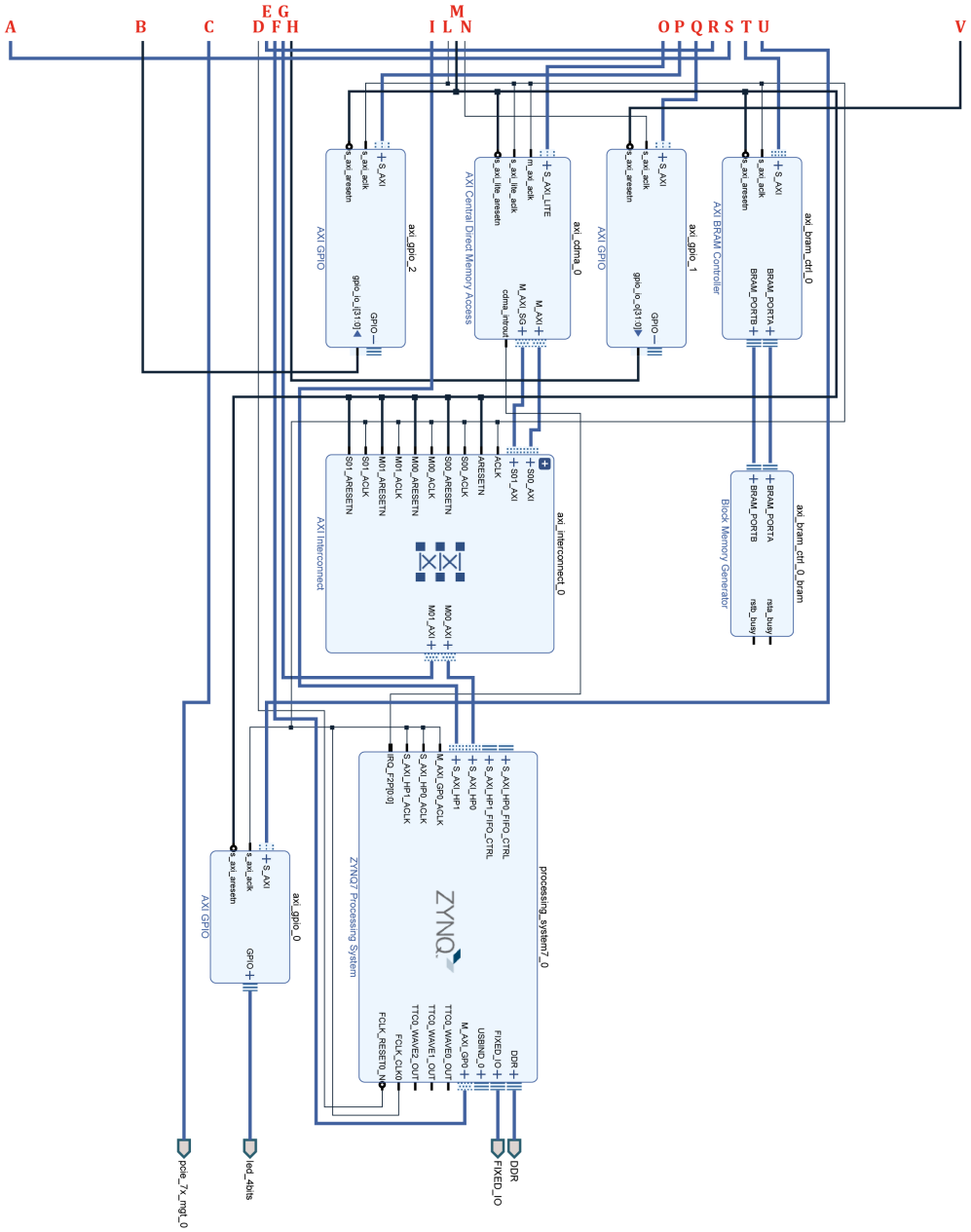

Figure 2.17: NVMe Controller Design Layout for FPGAs - part 2.

RW	IO	BW
Random read	174809 MB	2913.2 MB/s
Random write	149439 MB	2490.5 MB/s
Sequential read	196587 MB	3253.1 MB/s
Sequential write	191090 MB	3171.2 MB/s

Table 2.8: Performance of Samsung's SmartSSD under the stimulus of `Fio` for the four types of transfer.

- numjobs = 12.
- runtime = 60.
- bs = 4 KB:

The first version of the NVMe controller on FPGA used a Zynq clocked at 50 MHz. The results are shown in Table 2.9.

RW	IO	BW
Random read	724 MB	12.100 MB/s
Random write	382 MB	6.362 MB/s
Sequential read	787 MB	12.900 MB/s
Sequential write	411 MB	6.602 MB/s

Table 2.9: Performance of the NVMe Controller clocked at 50 MHz under the stimulus of `Fio` for the four types of transfer.

Comparing the results of table 2.8 with table 2.9 , a very substantial difference is evident, of two orders of magnitude. The NVMe controller does not exceed 12.9 MB/s, compared to 3253.1 MB/s for the SmartSSD. It is therefore necessary to understand where the optimizations can be made to approach the case taken as a reference. Embedded CPU is moving data, no DMA is being used. With PCIe DMA IP or XDMA IP you could get much higher numbers, having the built-in CPU doing `memcpy()` instead of proper DMA explains the numbers we see. In the Zynq-7000 technical reference manual ([26], table 228) it is compared the bandwidths of all interfaces (CPU I/O vs PS DMA vs PL DMA on GP, HP, ACP ports etc ...). The maximum CPU I/O value is approximately 25 MB/s. Another factor that makes this `memcpy()` even slower is because Zynq's AXI interface is clocked at 50MHz at 32bit wide (compared to PCIe IP it is 128bit at 125MHz). Therefore the results obtained are not surprising, as the maximum bandwidth is

about 25 MB/s and in addition, you have to take into account the overhead to read the host queues, schedule the translation from AXI to DMA for each transfer etc.

The first optimization was to increase the clock speed for the Zynq AXI interface to 125 MHz. The performance is shown in table 2.10.

RW	IO	BW
Random read	1116 MB	18.600 MB/s
Random write	471 MB	7.845 MB/s
Sequential read	1229 MB	19.300 MB/s
Sequential write	503 MB	8.124 MB/s

Table 2.10: Performance of the NVMe Controller clocked at 125 MHz under the stimulus of `Fio` for the four types of transfer.

Performance has improved although not yet significantly. Random read went from 12.1 MB/s to 18.6 MB/s. The least noticeable improvements are found in the writes with an increase of only 1483 KB/s for the random and 1522 KB/s for the sequential. The sequential read reached a maximum bandwidth of 19.3 MB/s, the highest recorded value. The result is still not surprising, the gap with the limit of 25 MB/s has been reduced, but it is still unsurpassable. It was necessary to bypass the barrier by inserting an IP CDMA (AXI Central Direct Memory Access) and inserting a new function (`DoSimpleTransfer()`) to replace `memcpy()`. The results are shown in the table 2.11.

RW	IO	BW
Random read	6442 MB	131 MB/s
Random write	6442 MB	120 MB/s
Sequential read	6451 MB	131.3 MB/s
Sequential write	6449 MB	120.4 MB/s

Table 2.11: Performance of the NVMe Controller clocked at 125, with the addition of the DMA block, MHz under the stimulus of `Fio` for the four types of transfer.

The 25 MB/s limit now no longer exists, a maximum bandwidth of 131.3 MB/s could be reached for the Sequential Read case reducing the gap with the SmartSSD reference case to only one order of magnitude. At the moment it is the DoSimpleTransfer function that takes care of releasing DMA, but only 4 KB at a time. Furthermore, bs is set to 4KB which forces the Host to send 1 LBA read or write commands which generates the worst case in terms of performance, as seen in

the previous sections. The performances achieved by the Samsung Device are still far away, but further optimizations and possible improvements will be described in the 4 chapter, in the section 4.1, "Optimizations and Improvements".

Chapter 3

Computational Storage

As anticipated in the introductory chapters, Computational storage does not currently have a defined standard. SNIA is currently working on it, but at the moment only one draft is available [5]. In the document drawn up by SNIA there are some illustrative examples that have made it possible to obtain a trace to reach a first prototype of Computational Storage. Of particular inspiration was the example B.4.1 of the document in which a PCIe OpenCL-based Programmable Computational Storage Drive (CSD) is presented (figure 3.1). OpenCL [27], a framework based on the ANSI C and C++ language with a host-device structure that can be run on a variety of platforms, takes care of data processing, while an NVMe Controller is entrusted with the task of managing the transfers. The two devices communicate via a PCIe bar called, in figure 3.1, Bar Peer To Peer.

In short, following the theory of operations presented in the first chapter:

- Discovery: Both NVM Express and OpenCl have a robust Discovery routine. In the case of NVMe, the Identify command deals with the identification of both the Namespace and the Controller. Furthermore, Bar0 contains spaces for device identification information.
- Configuration: OpenCL provides a rich set of APIs to configure the CSP.
- Usage: This phase consists of 3 operations:
    1. The host sends a read request to the NVMe controller. This reading will be written in the P2P bar. As it always has, the controller will send a completion queue entry and an interrupt.
    2. OpenCL must process the data by taking it in the P2P bar and entering the result in the same bar.
    3. A write request must be sent to the controller to store the result in a location on the disk.

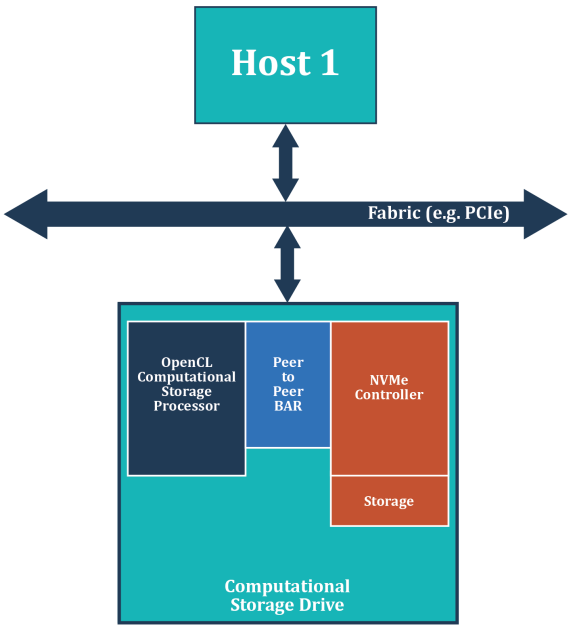

Figure 3.1: SNIA illustrative example: PCIe OpenCL-based Programmable Computational Storage Drive (CSD).

The idea of Computational storage described in the example revolves around the concept of Peer To Peer and the use of a support bar to be used as a buffer register for processing. The Peer to Peer therefore needs a deepening.

3.1 PCie Peer-to-Peer Communication

PCIe peer-to-peer (P2P) communication is a PCIe feature that allows two PCIe devices to directly transfer data between each other without using host RAM as temporary storage. Swapping between two devices over PCIe with DMA normally involves two steps. First, the reading device copies the data from its back-end memory to the Host DRAM. Subsequently the writing device takes the data just written in the DRAM and writes them in its memory. This takes time and host DRAM occupation. As shown in Figure 3.2, P2P uses a memory space called CMB as a transfer buffer for DMA between two devices on PCI Express, avoiding the use of host DRAM. The CMB (Controller Memory Buffer) is a PCIe bar, first introduced in the NVMe standard in 2014 in version 1.2, which can be used for certain specific data types. It can be set up to place queues that usually reside in host memory, or it can contain PRP lists, for example. Also it can be used as a DMA buffer for offloaded NVMe copies. This can improve performance and

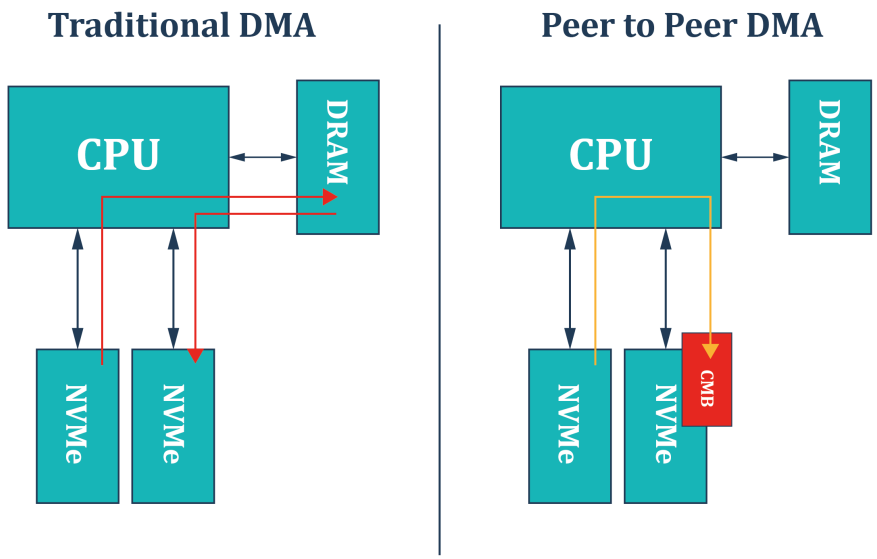

Figure 3.2: Contrast between traditional DMAs and Peer-2-Peer DMA.

offloads the host CPU.

Changes had to be made to the NVMe Controller in order to include the Controller Memory Buffer. As described in the [1] protocol in the chapter dedicated to the CMB, in BAR0 two registers must be set for the description of the CMB properties:

- CMBLOC (Controller memory Buffer Location): specifies which BAR of the PCIe has been chosen as CMB and, if necessary, the offset.
- CMBSZ (Controller Memory Buffer Size): Here is defined the size of the CMB and what abilities it has, if it is possible to save the completion and submission queues, if it is possible to save the PRP lists and if possible use it to read and write data.

The CAP.CMBS field is used to communicate to the Host the ability of the device to support the CMB. When the host intends to use the CMB it must appropriately set the CMBMSC register of the BAR0 to communicate the Controller Base Address. If the controller receives an I/O command with PRP between the address indicated in the Controller Base address and this address added to the size of the CMB bar, then the Controller must write or read in the CMB BAR and not from the Host memory.

The adaptation of the NVMe controller started with the addition of a new BAR, as well as with the setting of the registers described above. In the version used on

QEMU it was enough to add a new shared memory called shm_CMB. A Check on the PRP received at each transaction to compare it with the address contained in the Controller Base Address has been included, so as to divert the command to the BAR CMB if necessary. In the version for FPGA it was necessary to add a second Bar, called Bar2, in the PCIe and the consequent memory to be used as a bar. Once the device is made capable of supporting the use of the support buffer for P2P, it is possible to use this space as the support buffer for the computational storage. The data to be processed can be taken from a second device using the P2P mode or from the same device. In particular:

- The controller receives from the Host a read request with PRP corresponding to an address in the BAR CMB, used as a buffer.
- A data writing in the BAR CMB triggers a data processing.
- the controller receives a write request with PRP corresponding to the same address in the BAR CMB sent for reading. Now the data transcribed in the Back-end memory will be the ones that have been processed, without the need to involve the CPU.

It is therefore essential to handle P2P, a real key tool for computational storage. Once implemented and tested, it will be sufficient to add the desired processing functions of the data contained in the CMB bar.

3.2 Validation

For validation, Storage Performance Development Kit (SPDK) [28] was used. SPDK provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications. It provides some tools for Peer-to-Peer, although they are currently marked as experimental. In particular, SPDK's identify example application shows whether the selected device is capable of supporting the CMB and which options can be used. This is the command to use, by properly entering the PCI ID of the device:

Before running an SPDK application, some huge pages need to be allocated and any NVMe devices need to be disconnected from the native kernel drivers. This is possible by running the script:

```
sudo scripts/setup.sh
```

SPDK's identify example application shows whether the selected device is capable of supporting the CMB and which options can be used. This is the command to use, by properly entering the PCI ID of the device:

```
Controller Memory Buffer Support
================================
Supported:                              Yes
Total Size:                             4194304 bytes
Submission Queues in CMB:               Not Supported
Completion Queues in CMB:               Not Supported
Read data and metadata in CMB           Supported
Write data and metadata in CMB:         Supported
```

Figure 3.3: SPDK identify command output.

```
./build/examples/identify -r traddr:<pci id of ssd>
```

`cmb_copy` example application copies NLBAr starting from the SLBAr LBA from the Nr namespace on the read NVMe SSD to the NLBAw starting from the SLBAw LBA from the Nw namespace on the write NVMe SSD using the CMB of the device specified in -c as a DMA buffer.

```
./build/examples/cmb_copy -r <pci id of write ssd>-Nr-SLBAr-NLBAr -w
<pci id of write ssd>-Nw-SLBAw-NLBAw -c <pci id of the ssd with cmb>
```

3.2.1 QEMU

The validation of the operation and of the necessary settings for P2P and computational storage are initially entrusted to QEMU, to exploit once again the great flexibility. As mentioned, a new shared memory, called CMB_shm has been added to the design. Through the CAP.CMBS register the characteristics of the P2P bar have been set. Bar2 was chosen as the PCIe Bar to host the CMB space. A 4MB space has been reserved for the CMB Bar. Data reads and writes to the CMB are obviously supported, while Submission Queue and Completion Queue writes to that space are not. The SPDK Identify command in figure 3.3 confirms that the desired settings have been correctly interpreted by the Host.

For the P2P validation test, a native QEMU device, called NVMeQEMU, was used as support, in addition to the NVMe controller, to simulate the passage of data from one device to another. Note that the NVMeQEMU has 512B LBAs.

The test consists of two SPDK cmb_cpy commands. The first is as follows:

- r: ControllerNVME-1-0-1
- w: NVMeQEMU-1-2-8

```
 ************************************************************
|                          NEW COMMAND                       |
 ************************************************************

--------------------------- fetching ---------------------------

 DW0:                                    0x00170002
 DW1:                                    0x00000001
 DW2:                                    0x00000000
 DW3:                                    0x00000000
 DW4:                                    0x00000000
 DW5:                                    0x00000000
 DW6:                                    0xF91FB000
 DW7:                                    0x00000000
 DW8:                                    0x00000000
 DW9:                                    0x00000000
 DW10:                                   0x00000000
 DW11:                                   0x00000000
 DW12:                                   0x00000000
 DW13:                                   0x00000000
 DW14:                                   0x00000000
 DW15:                                   0x00000000
------------------------------------------------------------

 STATE:                                  NVME_CONTROLLER_NVM_CMD

--------------------------- decoding ---------------------------

 Command Type:                           NVM
 Command:                                Read
 Lba:                                    0x0
 Nlba:                                   0x1
 PRP_1:                                  0xF91FB000
 PRP_2 is a list?:                       N
 PRP_2:                                  0x0
 PRP_need_num:                           0x1

 PRP list:
 DMA_PRP_BUF[0]:                         0xf91fb000
------------------------------------------------------------
```

Figure 3.4: Read command received from the NVMe Controller from SPDK.

- c: ControllerNVME

This means that a single LBA, LBA 0, of the NVMe controller must be written to the disk at the LBA2 of NVMeQEMU through the CMB Bar made available by the ControllerNVME device. NVMeQEMU requires that the last field of the command be set to 8 because the LBA is at 512B and not at 4096B. Figure 3.4 shows the read command sent to the NVMe Controller. PRP1 in DW6 contains the address of bar2 of the NVMe Controller, verifiable through the use of the `lspci -vvv` command, added to a certain offset. Specifically, the address is 0xf9000000 with an offset of 0x001FB000. Unexpectedly, however, this address is not indicated

```
*************************************************************
|                          NEW COMMAND                      |
*************************************************************

--------------------------- fetching --------------------------

DW0:                                    0x00170001
DW1:                                    0x00000001
DW2:                                    0x00000000
DW3:                                    0x00000000
DW4:                                    0x00000000
DW5:                                    0x00000000
DW6:                                    0xF91FB000
DW7:                                    0x00000000
DW8:                                    0x00000000
DW9:                                    0x00000000
DW10:                                   0x00000002
DW11:                                   0x00000000
DW12:                                   0x00000000
DW13:                                   0x00000000
DW14:                                   0x00000000
DW15:                                   0x00000000
-------------------------------------------------------------

STATE:                                  NVME_CONTROLLER_NVM_CMD

--------------------------- decoding --------------------------

Command Type:                           NVM
Command:                                Write
Lba:                                    0x2
Nlba:                                   0x1
PRP_1:                                  0xF91FB000
PRP_2 is a list?:                       N
PRP_2:                                  0x0
PRP_need_num:                           0x1

PRP list:
DMA_PRP_BUF[0]:                         0xf91fb000
-------------------------------------------------------------
```

Figure 3.5: Write command received from the NVMe Controller from SPDK.

in the CMBMSC register of Bar0, as specified in the standard. However, this does not result in an error in the execution of the command.

The second SPDK command sent is:

- r: NVMeQEMU-1-2-8
- w: ControllerNVME-1-2-1
- c: ControllerNVME

This means that the previously written NVMeQEMU LBA2 must now be written to the disk at the NVMe Controller LBA2 via the CMB Bar made available by the NVME Controller device. Figure 3.5 shows the write command sent to the NVMe Controller. The PRP is again the same used in the previous command. The

```
root@target-vm:~# hexdump MINE_LBA0
0000000 63eb 1090 d08e 00bc b8b0 0000 d88e c08e
0000010 befb 7c00 00bf b906 0200 a4f3 21ea 0006
0000020 be00 07be 0438 0b75 c683 8110 fefe 7507
0000030 ebf3 b416 b002 bb01 7c00 80b2 748a 8b01
0000040 024c 13cd 00ea 007c eb00 00fe 0000 0000
0000050 0000 0000 0000 0000 0000 8000 0001 0000
0000060 0000 0000 faff 9090 c2f6 7480 f605 70c2
0000070 0274 80b2 79ea 007c 3100 8ec0 8ed8 bcd0
0000080 2000 a0fb 7c64 ff3c 0274 c288 bb52 0417
0000090 07f6 7403 be06 7d88 17e8 be01 7c05 41b4
00000a0 aabb cd55 5a13 7252 813d 55fb 75aa 8337
00000b0 01e1 3274 c031 4489 4004 4488 89ff 0244
00000c0 04c7 0010 8b66 5c1e 667c 5c89 6608 1e8b
00000d0 7c60 8966 0c5c 44c7 0006 b470 cd42 7213
00000e0 bb05 7000 76eb 08b4 13cd 0d73 845a 0fd2
00000f0 d083 be00 7d93 82e9 6600 b60f 88c6 ff64
0000100 6640 4489 0f04 d1b6 e2c1 8802 88e8 40f4
0000110 4489 0f08 c2b6 e8c0 6602 0489 a166 7c60
0000120 0966 75c0 664e 5ca1 667c d231 f766 8834
0000130 31d1 66d2 74f7 3b04 0844 377d c1fe c588
0000140 c030 e8c1 0802 88c1 5ad0 c688 00bb 8e70
0000150 31c3 b8db 0201 13cd 1e72 c38c 1e60 00b9
0000160 8e01 31db bff6 8000 c68e f3fc 1fa5 ff61
0000170 5a26 be7c 7d8e 03eb 9dbe e87d 0034 a2be
0000180 e87d 002e 18cd feeb 5247 4255 0020 6547
0000190 6d6f 4800 7261 2064 6944 6b73 5200 6165
00001a0 0064 4520 7272 726f 0a0d bb00 0001 0eb4
00001b0 10cd 3cac 7500 c3f4 b411 db2f 0000 0000
00001c0 0000 0000 0000 0000 0000 0000 0000 0000
*
00001f0 0000 0000 0000 0000 0000 0000 0000 aa55
0000200 0000 0000 0000 0000 0000 0000 0000 0000
*
0001000
```

Figure 3.6: NVMe Controller LBA0 content.

```
root@target-vm:/home/root/spdk# hexdump LBA2_Result
0000000 63eb 1090 d08e 00bc b8b0 0000 d88e c08e
0000010 befb 7c00 00bf b906 0200 a4f3 21ea 0006
0000020 be00 07be 0438 0b75 c683 8110 fefe 7507
0000030 ebf3 b416 b002 bb01 7c00 80b2 748a 8b01
0000040 024c 13cd 00ea 007c eb00 00fe 0000 0000
0000050 0000 0000 0000 0000 0000 8000 0001 0000
0000060 0000 0000 faff 9090 c2f6 7480 f605 70c2
0000070 0274 80b2 79ea 007c 3100 8ec0 8ed8 bcd0
0000080 2000 a0fb 7c64 ff3c 0274 c288 bb52 0417
0000090 07f6 7403 be06 7d88 17e8 be01 7c05 41b4
00000a0 aabb cd55 5a13 7252 813d 55fb 75aa 8337
00000b0 01e1 3274 c031 4489 4004 4488 89ff 0244
00000c0 04c7 0010 8b66 5c1e 667c 5c89 6608 1e8b
00000d0 7c60 8966 0c5c 44c7 0006 b470 cd42 7213
00000e0 bb05 7000 76eb 08b4 13cd 0d73 845a 0fd2
00000f0 d083 be00 7d93 82e9 6600 b60f 88c6 ff64
0000100 6640 4489 0f04 d1b6 e2c1 8802 88e8 40f4
0000110 4489 0f08 c2b6 e8c0 6602 0489 a166 7c60
0000120 0966 75c0 664e 5ca1 667c d231 f766 8834
0000130 31d1 66d2 74f7 3b04 0844 377d c1fe c588
0000140 c030 e8c1 0802 88c1 5ad0 c688 00bb 8e70
0000150 31c3 b8db 0201 13cd 1e72 c38c 1e60 00b9
0000160 8e01 31db bff6 8000 c68e f3fc 1fa5 ff61
0000170 5a26 be7c 7d8e 03eb 9dbe e87d 0034 a2be
0000180 e87d 002e 18cd feeb 5247 4255 0020 6547
0000190 6d6f 4800 7261 2064 6944 6b73 5200 6165
00001a0 0064 4520 7272 726f 0a0d bb00 0001 0eb4
00001b0 10cd 3cac 7500 c3f4 b411 db2f 0000 0000
00001c0 0000 0000 0000 0000 0000 0000 0000 0000
*
00001f0 0000 0000 0000 0000 0000 0000 0000 aa55
0000200 0000 0000 0000 0000 0000 0000 0000 0000
*
0001000
```

Figure 3.7: NVMe Controller LBA2 content.

result of these two serial operations should result in the NVMe controller LBA0 and LBA2 being identical. The result shown in figure 3.7 confirms this hypothesis.

Once the P2P mechanism was validated it was easy to add data processing. A simple function, called Computational_Sum(), is triggered when a read command with corresponding PRP from the P2P Bar, with any offset, is received by the NVMe Controller. The function simply adds 1 to all received 32-bit data. The previous simulation was repeated, this time activating the Computational_Sum() function. Figure 3.8 shows the obtained results.

3.2.2 FPGA

Once verified the correct setting and functioning of the peer-to-peer through QEMU, the test was extended to the implementation on FPGA. The same CMB settings presented in figure 3.3 have been maintained. To obtain a 4MB Bar to be used as a CMB it was not possible to use a BRAM as done for the BAR0 due to lack of space. It was necessary to use the DDR of the ARM.

It is not currently allowed to use `cmb_copy` with a single device. It was therefore necessary to create a new setup, which included two devices, loaded on two FPGAs, with identification C0C0 and C1C1. In both, the use of DMA was initially disabled.

LTTng [29], an open source tracing framework for Linux, was used for the tracing.

```
root@target-vm:/home/root/spdk# hexdump LBA2_SUM_Result
0000000 63ec 1090 d08f 00bc b8b1 0000 d88f c08e
0000010 befc 7c00 00c0 b906 0201 a4f3 21eb 0006
0000020 be01 07be 0439 0b75 c684 8110 feff 7507
0000030 ebf4 b416 b003 bb01 7c01 80b2 748b 8b01
0000040 024d 13cd 00eb 007c eb01 00fe 0001 0000
0000050 0001 0000 0001 0000 0001 8000 0002 0000
0000060 0001 0000 fb00 9090 c2f7 7480 f606 70c2
0000070 0275 80b2 79eb 007c 3101 8ec0 8ed9 bcd0
0000080 2001 a0fb 7c65 ff3c 0275 c288 bb53 0417
0000090 07f7 7403 be07 7d88 17e9 be01 7c06 41b4
00000a0 aabc cd55 5a14 7252 813e 55fb 75ab 8337
00000b0 01e2 3274 c032 4489 4005 4488 8a00 0244
00000c0 04c8 0010 8b67 5c1e 667d 5c89 6609 1e8b
00000d0 7c61 8966 0c5d 44c7 0007 b470 cd43 7213
00000e0 bb06 7000 76ec 08b4 13ce 0d73 845b 0fd2
00000f0 d084 be00 7d94 82e9 6601 b60f 88c7 ff64
0000100 6641 4489 0f05 d1b6 e2c2 8802 88e9 40f4
0000110 448a 0f08 c2b7 e8c0 6603 0489 a167 7c60
0000120 0967 75c0 664f 5ca1 667d d231 f767 8834
0000130 31d2 66d2 74f8 3b04 0845 377d c1ff c588
0000140 c031 e8c1 0803 88c1 5ad1 c688 00bc 8e70
0000150 31c4 b8db 0202 13cd 1e73 c38c 1e61 00b9
0000160 8e02 31db bff7 8000 c68f f3fc 1fa6 ff61
0000170 5a27 be7c 7d8f 03eb 9dbf e87d 0035 a2be
0000180 e87e 002e 18ce feeb 5248 4255 0021 6547
0000190 6d70 4800 7262 2064 6945 6b73 5201 6165
00001a0 0065 4520 7273 726f 0a0e bb00 0002 0eb4
00001b0 10ce 3cac 7501 c3f4 b412 db2f 0001 0000
00001c0 0001 0000 0001 0000 0001 0000 0001 0000
*
00001f0 0001 0000 0001 0000 0001 0000 0001 aa55
0000200 0001 0000 0001 0000 0001 0000 0001 0000
*
0001000
```

Figure 3.8: Contents of LBA2 after using Computational storage.

The first test involved sending and processing 1MB (256 LBA) between the C0C0 device and the C1C1 device using the C1C1 CMB as a buffer. The parameters set for `cmb_copy` are the following:

- r: ControllerNVME_C0C0-1-0-256
- w: ControllerNVME_C1C1-1-2-256
- c: ControllerNVME_C1C1

To allow a comparison between the use of the CMB and the use of host memory as a buffer for exchanging information between two NVMe devices, some changes to the `cmb_copy` code have been made. Through the use of `spdk_dma_malloc` which allows to allocate memory space in the Host memory, it was possible to replace the CMB address with a Host memory address, creating an easy way of comparing the two implementations. The MBR is 4MB, the data rate taken as a reference is 1MB, not very high but sufficient to understand the features and compare the performances. The times have been taken from the sending of the reading until

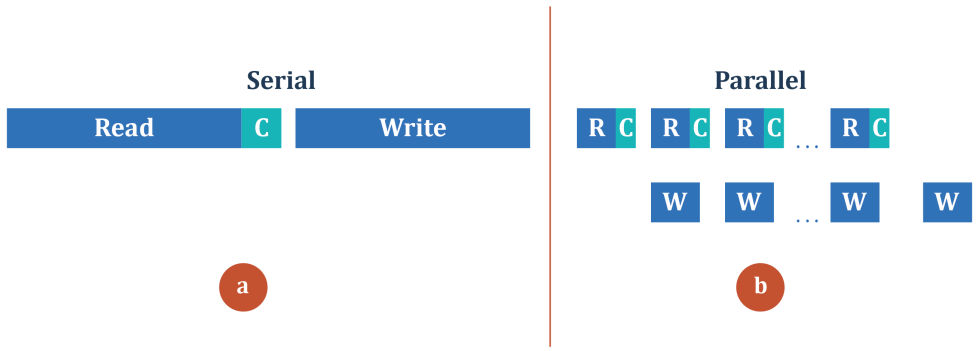

Figure 3.9: Comparison between serial and parallel implementation of `cmb_copy.c`. Read (R) reading phase. Write (W) writing phase. (C) computational phase.

the reception of the DONE sign of the writing, thus excluding the generation part of the address. The results are shown in table 3.1.

Buffer	Number of LBA	Time [ms]
CMB	256	28,050687
Host	256	729,351561

Table 3.1: 1MB transfer and processing times from C0C0 device to C1C1 device using CMB and Host memory as buffer.

With the use of the CMB the time taken for reading, processing and writing is about 28 ms, compared to more than 729 ms using the host memory, more than 25 times faster.

Parallel implementation

Some changes to the `cmb_copy` code have been made to allow for an improvement in performance. `cmb_copy` uses a serial approach (figure 3.9 (a)).

After the initialization phase and after selecting the buffer address, the `spdk_nvme_ns_cmd_read` function is executed to read the desired number of LBAs. This function manages the sending of reading commands to the C0C0 device. Once the `read.done` signal has been obtained, the writing phase on the C1C1 device begins with the `spdk_nvme_ns_cmd_write` function until completion signaled by the `write.done` signal. The implementation can therefore be defined as "serial". To speed up the execution, a parallel version was created as shown in figure 3.9 (b). The `spdk_nvme_ns_cmd_read` function is first executed on the first LBA to be read.

Then `cmb_copy` enters a loop where the `spdk_nvme_ns_cmd_write` functions are performed on the first LBA previously read and `spdk_nvme_ns_cmd_read` on the second LBA. The loop restarts by updating the LBAs value to be taken into consideration once the `write.done` and `read.done` signals have been obtained. Once the loop is completed, the write function of the last LBA is performed. Figure 3.10 shows two examples of tracing through Flame chart of the serial (above) and parallel (below) implementation. The first executes function `spdk_nvme_ns_cmd_read` and `spdk_nvme_ns_cmd_write`, and its sub-functions, at the beginning of the `cmb_copy` function and approximately in the middle of it. In the second, however, such functions from only one LBA at a time are performed continuously in a loop until the desired number of LBAs are consumed.

Table 3.2 shows the times for parallel implementation using the CMB and host memory.

Buffer	Number of LBA	Time [ms]
CMB	256	92,300754
Host	256	601,038201

Table 3.2: 1MB transfer and processing times from C0C0 device to C1C1 device using CMB and Host memory as buffer with `cmb_copy` parallel implementation.

By comparing table 3.1 and table 3.2, in the case of buffering in host memory there is an improvement in performance, from 729,35ms to 601,03ms. In the case of CMB, on the other hand, there is a clear worsening (from 28,05ms to 92.30ms), due to the wait times for read.done and write.done. The problem of the slowness of the parallel implementation is due to the number of NVMe I/O commands that have to be sent to the device. In the parallel implementation, the `spdk_nvme_ns_cmd_write` and `spdk_nvme_ns_cmd_read` functions are executed at each cycle, requiring a transaction of 4KB each. The host is therefore obliged to send to the controller a read/write request of only one LBA (4KB) at a time. For the serial implementation (original `cmb_copy`) instead, it is true that read and write do not work in parallel, but the host can send read and write commands larger than 4KB, using the PRP lists. Having to wait less interrupts the serial implementation is faster.
To make the most of the NVMe I/O commands, the number of LBAs to read/write at each cycle has been changed. In this way the commands sent from the Host to the device will not have the maximum limit of 4KB each command but will be able to exploit the entire PRP list (Up to a maximum of 32 LBAs per command, set as the maximum limit). The graph 3.11 shows the time taken for a transaction (+ processing) of 1MB as the number of pages per cycle varies. The red line is the serial reference (28.05ms). With only one LBA per cycle it is well over the line,

Figure 3.10: Above, `cmb_copy` flame chart with serial implementation. Below, `cmb_copy` flame chart with parallel implementation.

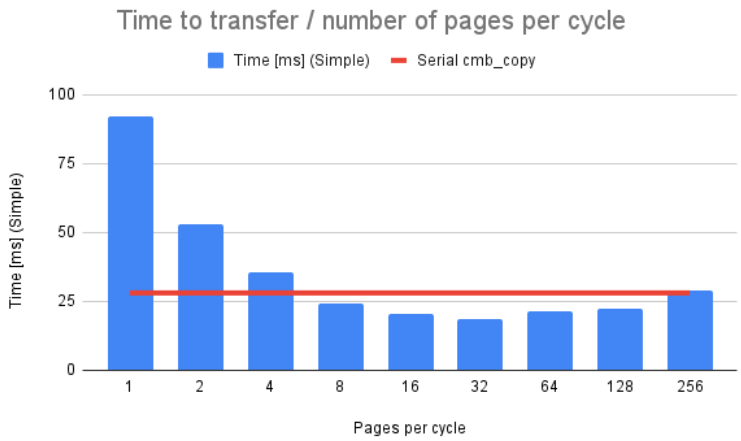

Figure 3.11: Transfer + processing time of 1MB as the number of LBAs per cycle changes in the parallel implementation of `cmb_copy` (NO DMA).

92,30 as seen. By increasing the number of LBAs per cycle, the time is reduced until a minimum of 32 LBAs is reached as expected, with 18,56ms, well below the serial threshold. It then grows again until it corresponds to the reference value when each cycle corresponds to 1MB, therefore identical to the serial case. Even if the host memory is used as a buffer, with a higher number of pages per cycle the time for the transaction decreases, passing from 601.04ms to 576.11ms. Table 3.3 shows the numerical results just described. For the case with the CMB as buffer, the times are shown as the number of LBAs per cycle varies, while for the buffer in the Host memory only the minimum value with 32 LBAs per cycle, in addition to the initial case with only one LBA per cycle.

Insertion of the DMA

In the measurements just described, the DMA block was deactivated, as mentioned, so that all transactions, both the CMB writes and the host memory writes, were performed via memcpy. Table 3.4 shows the same measurements previously seen but this time with the DMA block activated.

The major improvements are present when working with host memory, in the serial case it becomes 20 times faster, going from over 700 ms to just 36.02 ms, while in the parallel one with one page per cycle 6 times, reaching 22.53 ms . When using the CMB instead there is a minimal change of a few milliseconds. The minimum, at 32 LBAs per cycle, goes from 18.56 ms to 17.57 ms, saving just 1 ms. This is due to the fact that only the read command received from the C0C0 device enjoys the benefits of DMA, while C1C1 copies the data from the CMB

Buffer	Number of LBA	pages per cycles	Time [ms]
CMB	256	1	92,30
CMB	256	2	53,01
CMB	256	4	35,71
CMB	256	8	24,11
CMB	256	16	20,56
CMB	256	32	18,56
CMB	256	64	21,48
CMB	256	128	22,17
CMB	256	256	28,77
Host	256	1	601,04
Host	256	32	576,11

Table 3.3: Transfer + processing time of 1MB as the number of LBAs per cycle changes in the parallel implementation of `cmb_copy`, with both CMB and host memory as a buffer.

Type	Buffer	Number of LBA	pages per cycles	Time [ms]
serial	CMB	256	-	27,24
serial	Host	256	-	36,02
paral	CMB	256	1	93,55
paral	CMB	256	2	42,13
paral	CMB	256	4	33,01
paral	CMB	256	8	21,62
paral	CMB	256	16	18,57
paral	CMB	256	32	17,57
paral	CMB	256	64	19,03
paral	CMB	256	128	21,47
paral	CMB	256	256	27,61
paral	Host	256	1	100,83
paral	Host	256	32	22,53

Table 3.4: Transfer + processing time of 1MB as the number of LBAs per cycle changes in the parallel and serial implementation of `cmb_copy`, with both CMB and host memory as a buffer, using DMA.

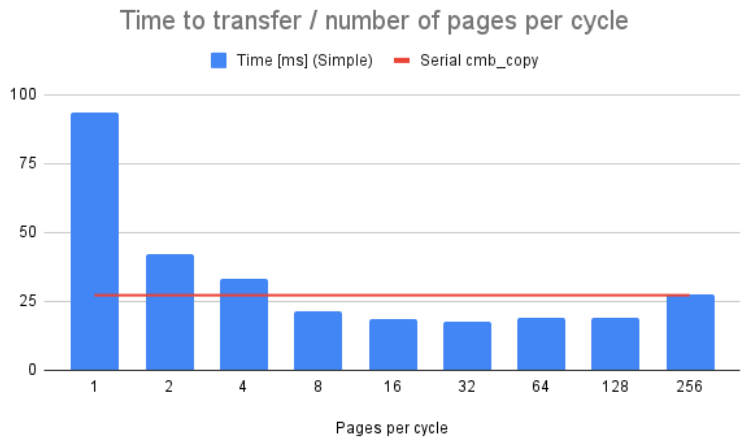

Figure 3.12: Transfer + processing time of 1MB as the number of LBAs per cycle changes in the parallel implementation of `cmb_copy` (DMA).

buffer to the back-end memory via memcpy. The cycle still has to wait for both operations to be concluded so it does not gain much time. The graph of the time as the number of LBAs per cycle changes, shown in figure 3.12, however, remains consistent with expectations.

Chapter 4

Future steps and Conclusion

The main purposes of the following chapter are to describe some optimizations and improvements to be applied to the described devices and to draw the final conclusions, with particular emphasis on the possible applications.

4.1 Optimizations and Improvements

The section retraces the previous chapters to highlight possible optimizations that may lead to improvements in terms of costs and speed.

Management of completion queue entries

In chapter 2, it has been underlined how the completion queue entries, once created, are saved in the memory of the controller and kept until all the submission queue entries of the selected queue have not been all served. Then they are written into the host memory and, once the writing is complete, the interrupt is sent. In both Flame graph 2.8 and 2.9 it is evident how these operations managed by the functions `cq_creating_and_saving` and `cq_completion_pcie_upstream` require a large amount of time, as well as cost for the storage space of completion queue entries in the controller memory. A first possible solution could be to write the completion queue entry directly into the memory once it is generated. Once all the Submission queue entries have been served, the interrupt is sent. This method would save both time, since it is no longer necessary to copy the completion queue entry into the memory of the controller and read it again during the sending phase, as well as cost saving, freeing up space in the controller memory.

Another possibility to improve costs and speed of execution is to use the CMB to store the Completion Queues and access the content directly. In the CMBSZ field of BAR0 it is possible to set Completion Queue Support (CQS). The controller

thus signals to the host that it can support completion queues within the Controller Memory Buffer.
The same path could be followed for the Submission Queues. By appropriately setting Submission Queue Support (SQS) in the BAR0 it would be possible to use this space to store the queues, allowing you to access them when necessary, without having to save them in the controller memory as is currently done, saving additional space.

DMA on FPGA

In chapter two, in the section 2.6 dedicated to the performance analysis of the NVMe Controller on FPGA, it was seen how the addition of an IP CDMA allowed to improve the performance of the device, reaching only one degree of magnitude from the Samsung SMART Storage device taken as a reference. Currently, however, as specified in the chapter, the DoSimpleTransfer function, which takes care of sending the transfers and which is memcopy, is limited to only 4KB at a time. The DMA block will therefore be able to transfer a maximum of 4KB at a time, thus requiring several cycles for transfers of large sizes. One way to get even closer to the reference could be to create a DMA block capable of accepting a list of PRPs and managing it internally, so as to be able to perform writes and reads on the host memory of more than 4KB, limited only by the maximum number of LBAs. transferable with a single NVMe I/O command, currently set to 32, as seen.

Maximum number of LBAs per I/O command

The table 2.1 shows the characteristics set for the NVMe controller and for the Computational Storage accordingly. Max Data Transfer Size is the maximum number of Bytes that can be transferred with a single I/O command. It is set to 131072, which is 32 LBA. This value has been selected keeping the settings used in the OpenExpress [13] project, considered at first suitable for the purpose. Having a maximum value of LBAs per transfer made it possible to save the list of PRPs in the controller memory, making it easy to handle. However, during the evolution of the project it emerged that this setting has important consequences on performance. As seen in Chapter 3, Table 3.11 and 3.12 table, the shortest transfer and processing time for parallel implementation is achieved precisely at transfers of 32 LBAs at a time. By varying the maximum number of transferable LBAs at a time, this minimum could be further lowered, until the optimal value is found. As seen in the previous section 4.1, by creating a DMA block capable of handling PRP lists, the maximum number of PRPs that can be handled by a single transfer would currently be 32. Also in this case, by varying the Max Data Transfer Size it would be possible to improve performance for large data transfers. By appropriately setting the `mdts` field in the Admin Identify Controller Data

command, it is possible not to impose limitations on the maximum number of Transfers per I/O command. This would not make it possible to save the contents of the entire list into controller memory as it is currently done. A possible solution could be to manage each PRP individually by extracting it from the list in the host memory one at a time. Or you could re-exploit it in CMB. As already seen previously for submission queues and completion queues, the Controller Memory Buffer is able to host also the PRP lists, by appropriately setting the LISTS field inside the BAR0.

SPDK problems and p2pmem

SPDK was very useful for testing the functioning of the CMB and Peer-To-Peer. As seen in chapter 3, cmb_copy was used between two Controller_NVMe, using as Buffer the CMB of the device that receives the write command, as described in figure 3.2. On the other hand, difficulties have been encountered in carrying out the same operation but using the CMB of the device receiving the read command as a buffer. Taking the example of chapter 3 with C0C0 reading and C1C1 writing, the C0C0 device correctly receives the read command and fills its own CMB. However, the C1C1 device does not seem to be able to read in the BAR2 of C0C0 dedicated to the CMB. Also, as described in the specification [1] and mentioned in the chapter 3, the register 0x50 in the BAR0 called `Controller Base Address` should contain the address of the BAR2 where the CMB assigned by the Host resides. Using SPDK, this register is never set forcing to enter the address manually. SPDK is not standard and uses its own drivers and not the native Linux ones. A further step will surely be to move towards using p2pmem [30]. P2pmem is a Linux kernel framework to allow PCIe devices to exchange DMA while under the control of the host CPU. The use of p2pmem was tested on QEMU after recompiling the kernel by adding the appropriate settings in Kernel Configuration/Device Drivers, but not all intel processors allow p2p, only a few are whitelisted in the kernel. Unfortunately the one available is not among them.

4.2 Applications and Conclusions

4.2.1 Applications

The document "Computational Storage Architecture and Programming Model" [5] written by SNIA, as already seen in the previous chapters, specifies the definitions of the world of Computational Storage dictates some guidelines on the possible structures that can be used for the realization through different application examples. It also provides some possible uses in the chapter called "Computation Storage Services". Among the possible Fixed Computational Storage Services applications,

in addition to the application of a simple expression on the data as seen in the case described in the previous chapters, the following are highlighted:

- Compression: Read data from a certain location, compress/decompress them and write the result to a destination location. The compression algorithm could be set during the "Configuration" phase allowing greater flexibility. Likewise it can be used for both single stream and multiple stream video compression to allow parallel compressions.

- Encryption: in the same way it could be used for encrypts or decrypts the data, varying both algorithm and keying information during the "Configuration" phase.

- Encoding and Decoding.

- Data Deduplication: also for deduplication or, on the contrary, for Duplication, the computational storage can be easily adapted, specifying the starting location, the target location and the selected algorithm.

- Pipeline: combine several commands previously described together, performing a series of operations on the data according to a specification of the data flow.

The application possibilities are therefore varied and find space in different fields. Another example could be the applications in the field of mining, or rather farming, of emerging cryptocurrencies such as Chia [31] that promise to solve the problem of the enormous energy consumption required by blockchains such as Bitcoin, replacing the "proof of work" paradigm, which requires the sharing of computational power, with the "proofs of space and time" paradigm, where the storage space is shared. In short, this operation consists of two moments:

- Plotting: consists of 4 phases. The first stage generates all your free space tests by creating seven cryptographic hash tables and saving them in your temporary directory. Stage 2 propagates back through the hashes, stage 3 algorithmically sorts and compresses these hashes into the temporary directory as you begin building the final file, and stage 4 completes the file and moves it to the final texture file.

- Farming: in which the created plots are checked against each new Chia network block.

This seems to be the path taken by NGD Systems with Chia-AutoPlot CSD [32], a computational storage device that automatically tracks Chia and moves it to the hard disk, thus relieving the CPU of the burden of managing the plotting.

4.2.2 Conclusions

The world of computational storage is constantly evolving and will probably be widely implemented in the years to come. The latest NVM Express version Base Specification 2.0 [33] released on 23 July 2021, incorporates a completely new Key Value Command set, which is added to the I/O command set presented in the chapter 2, which winks at the world of Computational Storage as underlined by SNIA in the presentation entitled "Key Value Standardized" [34].

As described in "Key Value Command Set Specification, version 1.0" [35], the NVMe-KV command set provides the key to store a corresponding value on an NVMe memory, then retrieves that value from the media by specifying the key corresponding. Access is therefore not done by address but by key. In this mode, the data will no longer be addressed by a Logical Block Address, but by a key, which will also allow not to have blocks of fixed size, but variable. The main operations available are:

- Storing: Data is stored as a single value associated with a key.
- Retrieving: Data is received as a single value associated with a key.
- Deleting: key-value pair may be deleted.
- Listing: List all the keys present on the device.

SNIA has standardized a Key Value API [34] that works with NVMe Key Value, allowing access to data on a storage device using a key instead of a block address. The use cases are varied, from storing photos or videos as a single addressable object to storing records associated with a unique identifier, but also for computational storage with filtering by key or compression and decompression.

The computational storage device presented is the first step for the creation of a performing device that allows in the future to achieve the performance of devices on the market. There are several optimization possibilities as presented in this last chapter. This work has allowed us to enter the world of NVMe and computational storage which, as we have seen, is currently in its infancy, awaiting the "Computational Storage Architecture and Programming Model 1.0" promised by SNIA that officially draws the standard.

Appendix A

Insights NVMe protocol

A.1 Phase Tag

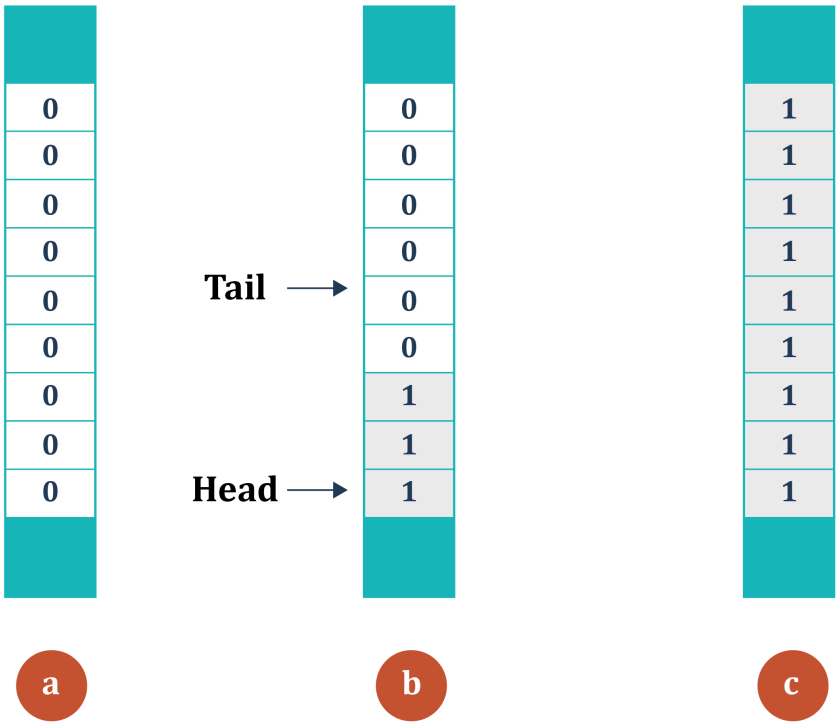

Figure A.1: Phase tag example.

The phase tag (P) is a bit defined in the completion queue entry to indicate if

that entry has just been entered. In this way the host can know if it is new without having to consult any register.
Initially the reference completion queue is completely empty. The bit corresponding to the PhaseTags are therefore all 0 (fig.A.1, (a)).
Completion queue entries begin to be inserted into the queue. These entries have a PhaseTag value of 1. In the example in figure fig.A.1, (b) 3 completion queue entries have been added. The host, by checking the phase tags, will be able to determine how many new entries have been inserted.
Once the end of the queue is reached, all the positions of the PhaseTag bit of the queue will be at a value equal to 1. The controller from now on must send completion queue entries with phase tag equal to 0 to allow the host to distinguish it from the previous entries (fig.A.1, (c)). When the queue is full again, the controller will return to sending 1 as the phase tag, and so on.

A.2 Round Robin and Weighted Round Robin with urgent priority class

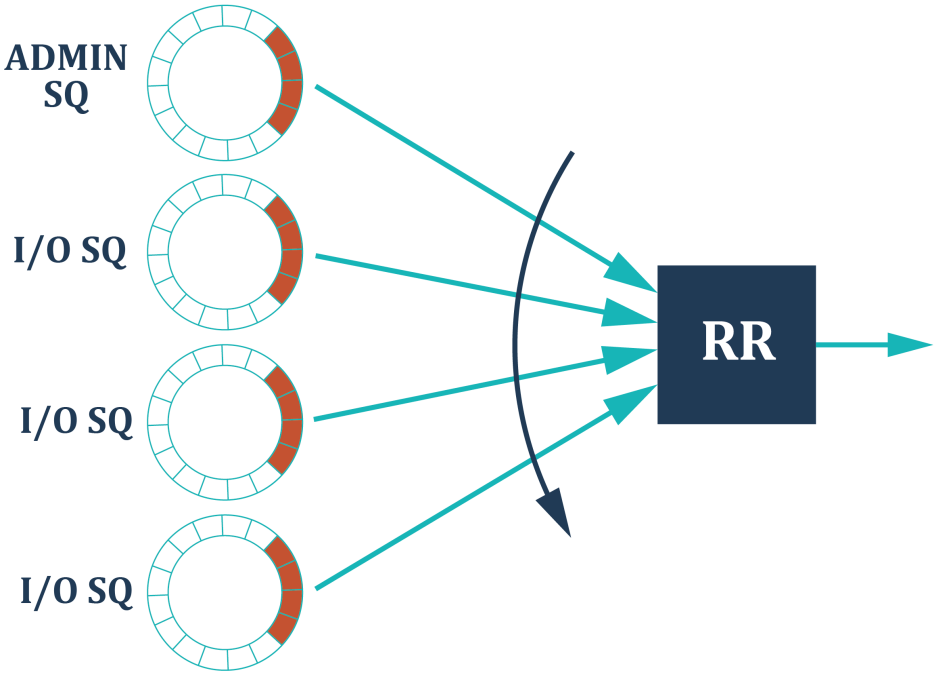

Figure A.2: Round Robin.

In the nvme protocol it is possible to have more than 64000 queues. An arbitration mechanism is needed to determine which queue is to be served.
The simplest method is called Round Robin (figure A.2). All queues are assigned the same priority, the order of execution is based on the order of arrival in the Ready list in a circular manner. It is also the easiest method to implement, not having to manage any priorities. All NMVe controllers must be able to handle this method.

An optional method for NVMe controllers is the Weighted Round Robin with urgent priority class. In this case each queue is associated with a certain priority which establishes which queue has the right to be served before the others (figure A.3). Normally, the admin queue is associated with a higher priority. Queues with the same priority are served following Round Robin arbitration. Priority is assigned when creating the submission queues through the QPRIO field.

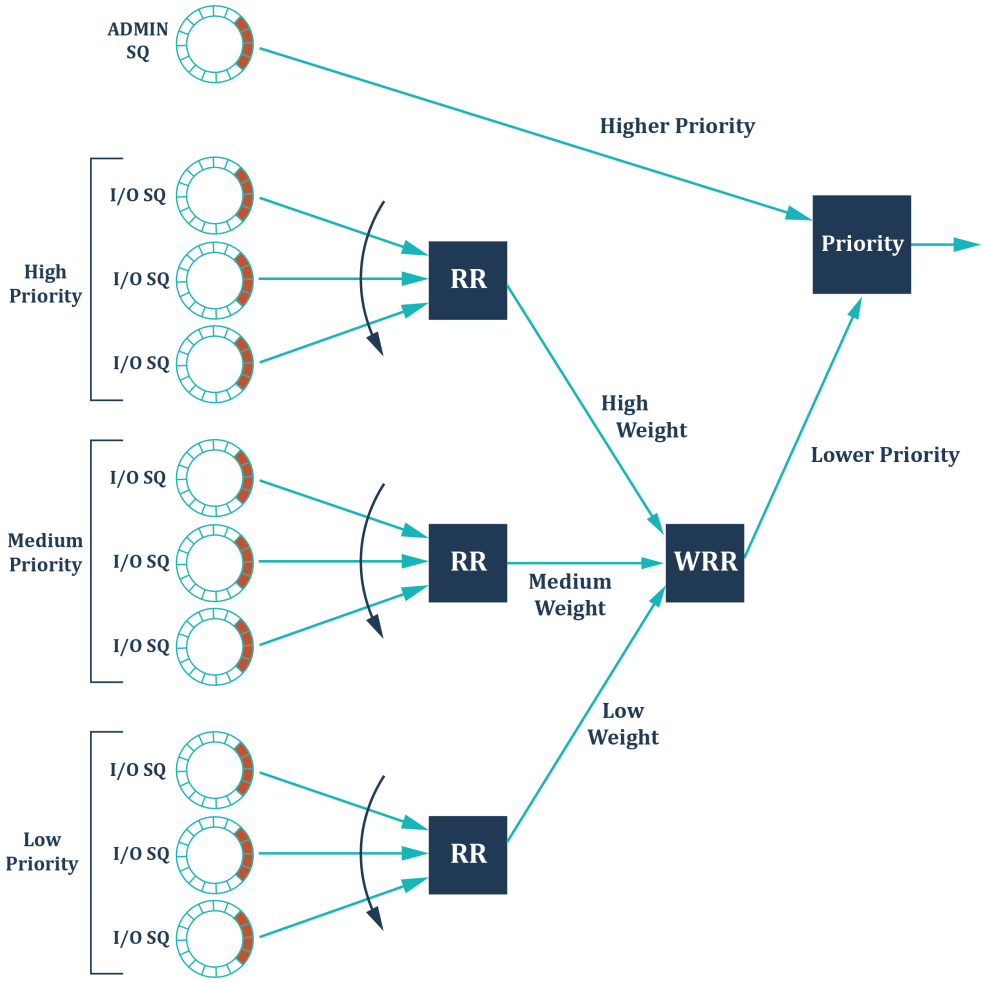

Figure A.3: Weighted Round Robin with urgent priority class.

A.3 Physical Region Pages

The Physical region pages entry is a pointer to a 64-bit physical memory address. It is used as a mechanism for data transfer between the controller's back-end memory and host memory. As shown in figure A.4 The PRP is made up of two parts: Page Base Address and Offset. The value of n is determined by the physical memory page size, which is specified in the CC.MPS field of the BAR0 (Table 1.3). For example, if the memory pages are 4KB then bits 11: 0 represent the offset, while the remaining 52 bits are the Page Base Address. The offset is DW aligned, so the first 2 bits should be 00b.

Figure A.4: PRP field.

In case the data exchange between controller and host exceeds the size of a physical page, the NVMe commands have a second field for the PRP. It is possible to divide it into two cases: adding a second PRP is sufficient or it is necessary to add more PRPs.
If it is sufficient to add a second PRP (figure A.5) it will simply be indicated in the appropriate field as for the first PRP with null offset.

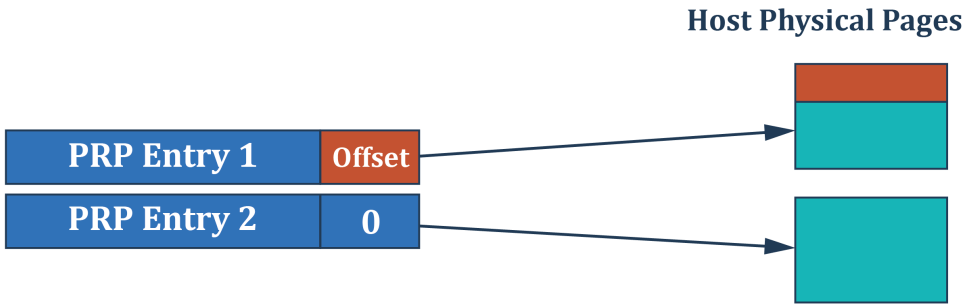

Figure A.5: Case 1: two PRPs are sufficient for the amount of data to be transferred.

In the second case, the second field of the PRP must not be interpreted by the controller as a simple PRP, but as a pointer to a list of PRPs (figure A.6). Each PRP in the list will have a null offset.

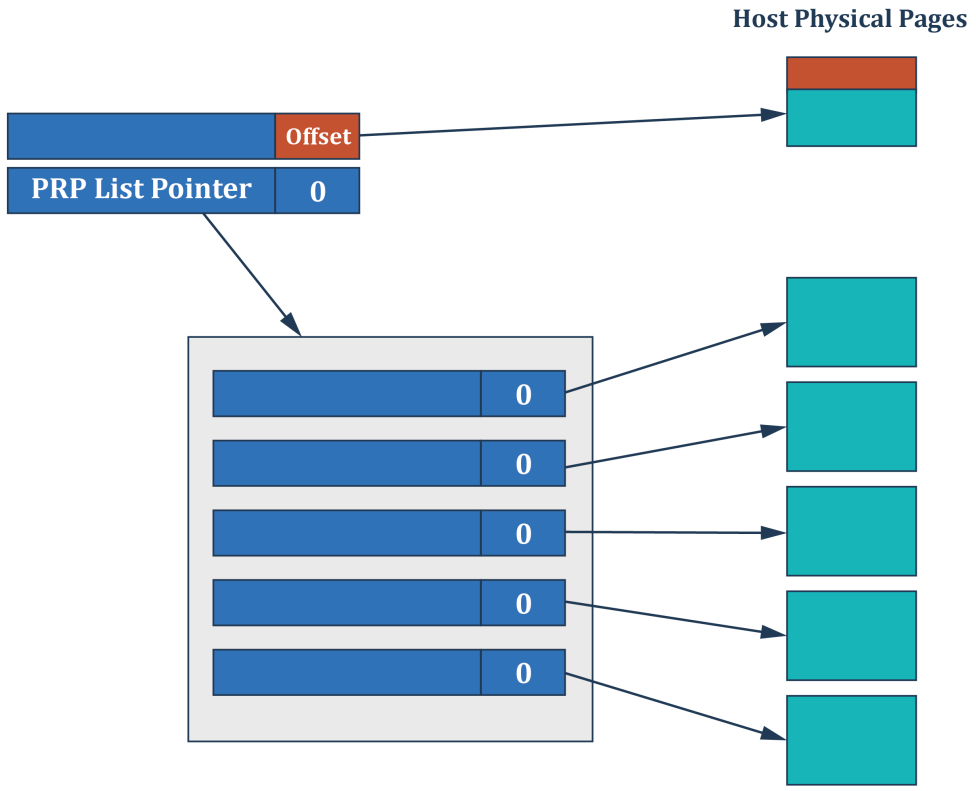

Figure A.6: Case 2: two PRPs are not enough. The second PRP therefore represents a list of PRPs.

A.4 MSI-X Advantages

Message Signaled Interrupts (MSI) are an alternative in-band method to signal an interrupt, they use messages to send the interrupt and therefore do not need dedicated pins, which makes the device simpler, cheaper and more reliable. They are supported by version 2.2 of PCI-Express.
PCI defines two extensions for message signaled interrupts: MSI and MSI-X. The former can send up to 32 interrupts, while the latter up to 2048 to facilitate communication with more processors.

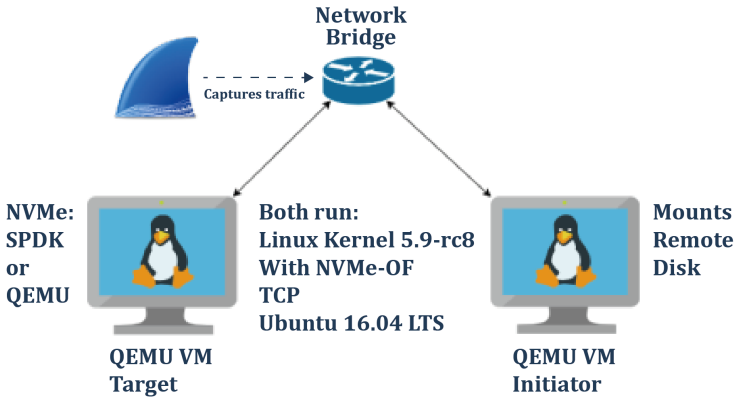

Figure A.7: System architecture to intercept NVMe commands sent using WireShark.

A.5 Wireshark

- Target:
  - Lauches SPDK_OF Target.
  - Configures the SPDK target.
  - Waits...
- WireShark:
  - Captures packets on bridge.
- Host
  - Connects to target.
  - Check if NVMe drive is found.
  - Check if partitions exists (if not creates it and formats the disk).
  - Mounts partition.
  - Write "Hello REDS!" to file.
  - Unmounts partition.
  - Disconnects drive.

Bibliography

[1] *NVM Express Base Specification*. Version 1.4. NVM Express, Mar. 2020. URL: https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.5R1.pdf (cit. on pp. xiii, 4, 17, 53, 69).

[2] N. Werdmuller. *What is computational storage? Everything you need to know*. Oct. 2020. URL: https://www.techradar.com/news/what-is-computational-storage-everything-you-need-to-know (cit. on p. 1).

[3] Statista. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2024*. Oct. 2020. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (cit. on p. 1).

[4] D. Robinson. *Computational storage winds its way towards the mainstream*. Feb. 2020. URL: https://www.nextplatform.com/2020/02/25/computational-storage-winds-its-way-towards-the-mainstream/ (cit. on pp. 1, 2).

[5] *Computational Storage Architecture and Programming Model*. Version 0.5. SNIA, 2020. URL: https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.5R1.pdf (cit. on pp. 1, 12, 51, 69).

[6] the free encyclopedia Wikipedia. *NVM Express*. URL: https://en.wikipedia.org/wiki/NVM_Express (cit. on p. 2).

[7] NVMe Express. *About*. URL: https://nvmexpress.org/about/ (cit. on p. 2).

[8] NVM-Express. *NVM-Express Overview*. 2016. URL: https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf (cit. on pp. 2, 3).

[9] the free encyclopedia Wikipedia. *PCI Express*. 2016. URL: https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf (cit. on p. 3).

[10] Xu Q. Siyamwala H. Ghosh M. Suri T. Awasthi M. Guz Z. Shayesteh A. Balakrishnan V. «Performance Analysis of NVMe SSDs and their Implication on Real World Databases». In: (2015). URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=2CA58A08BFB2821F59A7996DAC8742F1?doi=10.1.1.697.1493&rep=rep1&type=pdf (cit. on pp. 3, 4).

[11] *NVM Express over Fabrics*. Version 1.0. NVM Express, June 2016. URL: https://nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605-1.pdf (cit. on p. 4).

[12] Western Digital Blog. *NVMe over Fabrics (NVMe-oF™) Explained*. URL: https://blog.westerndigital.com/nvme-of-explained/ (cit. on p. 4).

[13] Myoungsoo Jung. «OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices». In: (2020). URL: https://www.usenix.org/conference/atc20/presentation/jung (cit. on pp. 11, 12, 68).

[14] Bowen J. *Computational Storage*. 2020. URL: https://www.snia.org/sites/default/files/SDCEMEA/2020/4%20-%20Jamon%20Bowen%20Xilinx%20-%20Computational%20storage%20.pdf (cit. on p. 13).

[15] NGD Systems. *NGD Systems Newport Platform: The World's First NVMe Computational Storage Drive (CSD) for Scalable Compute in Storage*. URL: https://www.ngdsystems.com/technology/computational-storage (cit. on p. 14).

[16] ScaleFlux. *What is CSD 2000*. URL: http://scaleflux.com/ (cit. on p. 14).

[17] Xilinx. *SmartSSD Computational Storage Drive*. URL: https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html (cit. on pp. 14, 40, 43).

[18] Editicom. *NoLoad CSP*. URL: https://www.eideticom.com/ (cit. on p. 14).

[19] Nyriad. *Making Storage Fast and Bulletproof*. URL: https://www.nyriadtechnology.com/ (cit. on p. 14).

[20] QEMU. *QEMU Documentation*. URL: https://qemu-project.gitlab.io/qemu/ (cit. on p. 29).

[21] Wertenbroek R. *Accessing the RAM of a QEMU Emulated System from another Process*. URL: https://blog.reds.ch/?p=1379 (cit. on p. 29).

[22] Uftrace. *Uftrace userManual*. URL: https://github.com/namhyung/uftrace (cit. on p. 30).

[23] Brendan G. *Flame Graphs*. URL: http://www.brendangregg.com/flamegraphs.html (cit. on p. 30).

[24] Xilinx. *Vivado Design Suite*. URL: https://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 42).

[25] the free encyclopedia Wikipedia. *Advanced eXtensible Interface*. URL: https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface (cit. on p. 43).

[26] Xilinx. *Zynq-7000 SoC Technical Reference Manual*. URL: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (cit. on p. 47).

[27] the free encyclopedia Wikipedia. *OpenCL*. URL: https://en.wikipedia.org/wiki/OpenCL (cit. on p. 51).

[28] SPDK. *What is SPDK*. URL: https://spdk.io/doc/about.html (cit. on p. 54).

[29] LTTng. *What is LTTng?* URL: https://lttng.org/docs/v2.13/#doc-nuts-and-bolts (cit. on p. 58).

[30] Linus Torvalds. *linux-p2pmem*. URL: https://github.com/sbates130272/linux-p2pmem#readme (cit. on p. 69).

[31] Bram Cohen and Krzysztof Pietrzak. *The Chia Network Blockchain*. URL: https://www.chia.net/assets/ChiaGreenPaper.pdf (cit. on p. 70).

[32] Scott Shadley. *CHIA-Auto-Plotting-Farming-Blockchain-Crypto Done Right*. URL: https://www.ngdsystems.com/page/CHIA-Auto-Plotting-Farming-Blockchain-Crypto-Done-Right (cit. on p. 70).

[33] *NVM Express Base Specification*. Version 2.0. NVM Express, July 2021. URL: https://nvmexpress.org/wp-content/uploads/NVMe-NVM-Express-2.0a-2021.07.26-Ratified.pdf (cit. on p. 71).

[34] *Key Value Standardized*. SNIA, Sept. 2020. URL: https://www.snia.org/educational-library/key-value-standardized-2020 (cit. on p. 71).

[35] *Key Value Command Set Specification*. Version 1.0. NVM Express, July 2021. URL: https://nvmexpress.org/wp-content/uploads/NVMe-Key-Value-Command-Set-Specification-1.0a-2021.07.26-Ratified.pdf (cit. on p. 71).