LOW OVERHEAD & ENERGY EFFICIENT STORAGE PATH FOR NEXT GENERATION COMPUTER SYSTEMS

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2019

By
Athanasios Stratikopoulos
School of Computer Science
Contents

Abstract 13
Declaration 15
Copyright 16
Acknowledgements 17
List of Acronyms 19

1 Introduction 22
  1.1 Evolution of Storage Systems . . . . . . . . . . . . . . . . . . . 23
  1.2 Overview of Computer Systems . . . . . . . . . . . . . . . . . . . 25
    1.2.1 Contemporary Computer Systems . . . . . . . . . . . . . . 25
    1.2.2 Next Generation Computer Systems . . . . . . . . . . . . . 27
  1.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
    1.3.1 Current Performance Limitations . . . . . . . . . . . . . . 28
    1.3.2 How to convert a Computer Component to Active Element? 29
    1.3.3 How to Enable Storage Elements? . . . . . . . . . . . . . . 30
  1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
  1.5 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
  1.6 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

2 NVMe System Architecture: A Software Perspective 34
  2.1 The Advent of NVMe SSDs . . . . . . . . . . . . . . . . . . . . . . 34
  2.2 The NVMe Software Layers . . . . . . . . . . . . . . . . . . . . . 36
    2.2.1 The File System & Direct I/O Layer . . . . . . . . . . . . . 37
    2.2.2 The Block I/O Layer . . . . . . . . . . . . . . . . . . . . . . 38
    2.2.3 The NVMe Driver . . . . . . . . . . . . . . . . . . . . . . . . 40
3 NVMe System Architecture: A Hardware Perspective

3.1 Introduction

3.2 NVMe Controller

3.3 Flash Memory

3.4 PCI Express Subsystem

3.5 Summary

4 Memory & Storage Systems: State-of-the-Art

4.1 Introduction

4.1.1 Key-Value Store Database
6.2 Experimental Methodology ........................................... 102
6.2.1 Hardware Platform ............................................. 103
6.2.2 Benchmark Applications ......................................... 105
6.2.3 Performance Metrics ............................................. 107
6.3 FIO - Results on Zynq 7000 SoC ................................. 107
6.3.1 Performance Analysis ........................................... 108
6.3.2 Power Analysis ................................................... 116
6.3.3 Area Analysis .................................................... 118
6.3.4 Discussion ....................................................... 119
6.4 Portability to Modern SoC and Evaluation ....................... 120
6.4.1 Performance Analysis ........................................... 121
6.4.2 Power Analysis ................................................... 124
6.4.3 Area Analysis .................................................... 125
6.5 Redis I/O - Results on Zynq UltraScale+ MPSoC .................. 125
6.5.1 Performance Analysis ........................................... 126
6.5.2 Latency ......................................................... 126
6.6 Summary ......................................................... 130

7 Conclusion and Future Work ............................................. 131
7.1 Summary ........................................................... 131
7.2 Future Directions .................................................. 132
7.2.1 FastPath Optimisations ......................................... 133
7.2.2 FastPath: An FPGA-based alternative to NVMeoF ........... 134
7.2.3 Near-Data Processing ........................................... 134

Bibliography ............................................................. 136

A NVMe Driver Modifications ............................................ 152
A.1 Allocate NVMe Queue Pairs for FastPath ......................... 152
A.2 Allocate DMA Regions for FastPath Blocks ..................... 153
A.3 Configure FastPath_Submit Module ................................ 154
A.4 Configure FastPath_Complete Module ............................ 155
A.5 The nvme_mmap function ......................................... 155

B Example of Using the FastPath_API ................................ 157
B.1 Allocate a FastPath Block ......................................... 157
B.2 Use the DMA regions for Direct Access .......................... 158
B.3 Submission of an I/O request ................................. 158
B.4 Release the FastPath Block ................................. 159

Word Count: 29420
# List of Tables

2.1 Comparison between AHCI and NVMe. ........................................ 35  
2.2 The NVMe command set. ....................................................... 44  
3.1 The I/O bandwidth of different PCIe generations. ...................... 62  
4.1 A classification of related work based on the used platform. ......... 77  
4.2 A classification of related work based on the memory hierarchy. .... 78  
5.1 The FastPath API. ............................................................. 85  
6.1 The experimental systems. .................................................... 103  
6.2 The Xilinx Zynq-7000 SoC characteristics. ............................... 103  
6.3 The Xilinx Zynq UltraScale+ MPSoC characteristics. .................... 104  
6.4 The YCSB workloads, as presented in [42]. .............................. 106  
6.5 The overall latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds ($\mu$s). ........................................ 109  
6.6 The average submission block latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds ($\mu$s). ........................................ 110  
6.7 The average completion block latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds ($\mu$s). ........................................ 110  
6.8 The experimental conditions between the FastPath evaluation board and the specifications of the Samsung SM953 NVMe SSD. ............. 112
6.9 The total on-chip power of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) on Zynq 7000 SoC. The power is reported in Watt (W).

6.10 The utilisation of three main resources (i.e. the LUTs as logic, the LUTs as memory, the CLB registers) for five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) on Zynq 7000 SoC.

6.11 The normalised report of throughput, energy efficiency and the percentage of the unutilized LUTs for logic and memory. The comparison is between four systems (Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) against the Baseline on Zynq 7000 SoC. The FastPath_v3 is configured with one, two and four fast paths (FastPath_v3_1, FastPath_v3_2, FastPath_v3_4).

6.12 The overall latency of FastPath_v3 against the Baseline for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).

6.13 The total on-chip power FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC. The power is reported in Watt (W).

6.14 The utilisation of three main resources (i.e. the LUTs as logic, the LUTs as memory, the CLB registers) for FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.
# List of Figures

1.1 The high demands of storage applications in data centres. Source: [www.statista.com](http://www.statista.com) .......................................................... 23  
1.2 The hierarchical latency in a computer system. ............................... 24  
1.3 The block diagram of two contemporary computers communicating via network. ................................................................. 26  
1.4 The block diagram with the network interface card (NIC) as a network element (NE), implemented either by an FPGA or a SoC. 27  
1.5 The average Linux I/O latency for single 4KB block size I/O request, on the Xilinx Zynq-7000 SoC. The latency is reported in microseconds ($\mu$s). ...................................................... 29  
1.6 The block diagram with the storage device (SSD) as an active element, implemented with an FPGA. ................................. 31  

2.1 Shipments of hard and solid state disk (HDD/SSD) drives worldwide from 2015 to 2021. The number of shipments is reported in millions. Source: [www.statista.com](http://www.statista.com) .................................................. 35  
2.2 The I/O request path on NVMe SSDs. ................................................. 36  
2.3 The block I/O multiple queue scheme. .................................................... 39  
2.4 The data flow in the NVMe driver. ........................................................... 41  
2.5 The namespace configuration for an NVMe SSD with one NVMe controller. ............................................................ 43  
2.6 The namespace configuration for an NVMe SSD with two NVMe controllers. .............................................................. 44  
2.7 The NVMe command. ........................................................................ 45  
2.8 The NVMe completion queue entry. ........................................................ 47  
2.9 The RDMA-based NVMe-over-Fabrics system architecture. ............... 50  

3.1 The block diagram of internals of an NVMe SSD device [49]. ............... 54
3.2 Block management in flash memory. ....................... 55
3.3 Garbage collection process in flash memory-based SSDs. ... 56
3.4 Flash memory organisation: from a NAND device (left) to a Single Level memory Cell (right). ...................... 59
3.5 Schematic representation of a floating gate memory cell (left) and the corresponding capacitive model (right). Diagram taken from [49]. ... 59
3.6 NAND string (a) and NAND array (b). Diagram taken from [49]. ... 60
3.7 PCIe bus composed of two lanes with a bidirectional pair of signals. ... 62
3.8 Three PCIe slots (from top to bottom: PCIe x4, PCIe x16, PCIe x1). ... 62
3.9 PCIe tree topology (The main components are: Root Complex, PCIe switches and PCIe endpoints). .................. 63
3.10 The structure of PCIe layers. ...................... 64
3.11 The transaction layer packet (TLP) structure. ........... 65
3.12 The conventional computer system layout in x86 architecture. ... 66
3.13 The layout of a modern Intel-based system. ................ 67
3.14 The layout of a Processor-FPGA SoC. ................... 67
4.1 The structure of the internals in a key-value store (KVS) database. 70
5.1 The overview of two systems that comprise FPGAs and NVMe SSDs. Figure 5.1a shows the FPGA connectivity on traditional systems via PCIe, while Figure 5.1b shows the connectivity on Processor-FPGA SoC via a Network-on-Chip interconnect. ... 82
5.2 The memory mapping of FPGA-based modules into user level applications and NVMe driver. .................. 84
5.3 The namespace configuration for FastPath and Linux I/O paths on an NVMe SSD. ...................... 88
5.4 The I/O request path on the Software_raw NVMe system. ... 90
5.5 The I/O request path on the FastPath_v1 NVMe system. ... 93
5.6 Design of axi_nvme_submit module. ...................... 93
5.7 Design of axi_nvme_complete module. ...................... 94
5.8 The I/O request path on the FastPath_v2 NVMe system. ... 95
5.9 Design of FastPath_Control module. ...................... 96
5.10 Design of FastPath_Submit module. ...................... 97
5.11 Design of FastPath_Complete module. ...................... 98
5.12 The I/O request path on the FastPath_v3 NVMe system. ... 99
5.13 Design of FastPath_Control_v3 module. ............................... 100

6.1 The layout of a Processor-FPGA SoC. ................................. 104

6.2 The overall latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for single 4KB block size I/O request, on Zynq 7000 SoC. The latency is reported in microseconds (µs). ................................. 108

6.3 The average block latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds (µs). ................................. 109

6.4 The average I/O bandwidth on five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. ................................. 112

6.5 The average I/O bandwidth on FastPath_v2 system for various depth sizes, ranging from 1 to 32, on Zynq 7000 SoC. ................. 113

6.6 The average I/O bandwidth on FastPath_v3 system for multiple fast paths, ranging from 1 to 4, on Zynq 7000 SoC. ................. 114

6.7 The normalised IOPS of four systems (Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) against the baseline, on Zynq 7000 SoC. ................................. 114

6.8 The I/O operations per request of four systems (Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) against the baseline, on Zynq 7000 SoC. ................................. 115

6.9 The normalised IOPS of FastPath_v3 system against the baseline for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4) on Zynq 7000 SoC. ................................. 116

6.10 The energy efficiency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3 with 1, 2, 4 fast paths) against the baseline on Zynq 7000 SoC. ................................. 117

6.11 The average block latency of FastPath_v3 system against the Baseline for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4). ................................. 122
6.12 The average I/O bandwidth on FastPath_v3 and Baseline systems for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).  

6.13 The I/O operations per second of FastPath_v3 and Baseline systems for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).  

6.14 The energy efficiency of FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).  

6.15 The throughput of YCSB benchmark, when Redis server uses the FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.  

6.16 The read latency of YCSB benchmark workloads (A-D) when Redis server uses the FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.  

6.17 The insert latency of YCSB benchmark workloads (D,E) when Redis server uses the FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.  

6.18 The update latency of YCSB benchmark workloads (A,B) when Redis server uses the FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.  

6.19 The scan latency of YCSB benchmark workloads (A,B) when Redis server uses the FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.  

7.1 The block diagram of future computer systems, which comprise multiple active elements for processors (PE), storage (SE), and network (NE).
Abstract

LOW OVERHEAD & ENERGY EFFICIENT STORAGE PATH FOR NEXT GENERATION COMPUTER SYSTEMS
Athanasios Stratikopoulos
A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy, 2019

The constant growth of data is pushing storage systems towards ever-increasing I/O bandwidth and lower latency requirements. In recent years, the Non-Volatile Memory Express (NVMe) standard has enabled SSD drives to deliver high I/O rates by allowing storage to be connected directly to the processing chip via the fastest available interconnect (i.e. PCIe). Although SSDs have become ubiquitous in data centres, reducing the latency gap with main memory is still a first-order challenge. Additionally, the adoption of FPGAs in data centres is creating opportunities to accelerate various applications and/or OS operations. While FPGAs in data centres have been connected via PCIe to mostly x86 servers, there are now heterogeneous System on Chips (SoCs) with multi-cores and FPGAs integrated on the same die and connected by an on-chip interconnect.

This thesis analyses the source of performance overhead on existing state-of-the-art storage devices and proposes a novel low overhead and energy efficient storage path called FastPath, that operates transparently to the main processor. Experimental results show that FastPath can achieve up to 82% lower latency, up to 12x higher throughput, and up to 10x more energy efficiency for a standard microbenchmark on an Xilinx Zynq 7000 SoC. Further experiments have been conducted on a state-of-the-art SoC (e.g. Xilinx Zynq UltraScale+ MPSoC), using a real application, such as the Redis in-memory database. The Redis database is configured to deliver requests issued by the Yahoo! Cloud Serving Benchmark.
(YCSB) into the storage device via FastPath. The experimental evaluation shows that FastPath achieves up to 60% lower tail latency and 15% higher throughput than the baseline storage path in the Linux kernel.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

First of all, I would like to express my gratitude to my principal supervisor Prof. Mikel Luján for his guidance during the last four years and for always keeping me up-to-date with ongoing research. At the same time, I would like to express my appreciation to my second supervisor Prof. John Goodacre for discussing novel trends in computer architecture and introducing me to the industrial world. Both of them contributed the most to the outcome of this research work.

I would like to appreciate Dr. Suhaib A. Fahmy and Dr. Javier Navaridas for agreeing to review and examine this thesis and providing me with fruitful comments. Special thanks to Arm Ltd. and EPSRC for funding my research work via an iCASE PhD Scholarship. Also, I would like to thank the School of Computer Science at the University of Manchester, and in particular the Student Support Office and the Academic Support Office, for the assistance and willingness to address any of my issues on a daily basis. In addition I would like to acknowledge Carl Duffy, Hyeong-Jun Kim and Jin-Soo Kim for the useful discussion about the NVMeDirect framework.

Furthermore, I am grateful to all the APT members for creating a friendly and productive environment that enables new students to achieve their goals. I want to thank Dr. Dirk Koch and Dr. Andrew Attwood for the valuable discussions around my PhD thesis. Moreover, I would like to thank Dimitris Poulis for his assistance during the initial stage of my work. Special thanks to John Mawer for his tolerance and technical support for every day in the last four years. It was also a great privilege to be surrounded by colleagues such as Andy, Guillermo, Konstantinos, Khoa, Swapnil, Raul, Seckin, Merve, Yaman, Gengting, Jim, Jamie, Cosmin, Bernard, Josh, Anuj, Carol, Tim, Babis, Orion, Andreas, Michalis, Eleni, Kyriakos, Harry, Ioannis, Nuno, Will, Foivos, and Juan.

I also feel fortunate for making new friends from all over the world and sharing some amazing moments with them during the last four years. Special thanks to
my close friends from the APT group: Ioanna, Serhat, Sebastian, Andrey, Anna, Garibaldi, Maria, Crefeda, Qian, Dennis, Valentina, and Mireya. I would like to thank Paris, Laura, Iris, Nikos, Kostas, Idoia, Kostas, Sarah, Dimitra, Laura, Magda and Sandra for their friendship and support. I would like to express my appreciation to the APT football members and the Vietnamese friends with whom I had a great time during the football and badminton matches accordingly. Also, a big thank to my friends Panagiotis, Aris, Isidoros, and Eleni who gave me generous support since the first year of our studies at the University of Manchester and they never stopped supporting me.

Of course, I cannot forget my best friends from my undergraduate study including Kostas, Nikos, Michalis, Vasilis, Dimitris, Dimitris, George and Staviani. Special thanks also to my good friends Gregory Chrysos, Foteini Simantiraki, and Christos Rousopoulos who encouraged me to follow the PhD path and never hesitated for me. My deepest appreciation goes to my good friends Nikos Foutris and Christos Kotselidis who gave me constructive comments and warm encouragement during the most stressful periods of the PhD.

Finally, a big thank to my whole family and especially my cousins who were very supportive all this time. This work is dedicated to my parents Dimitris and Sofia and my sister Stela, to whom I owe my deepest gratitude for always encouraging me to pursue my dreams.
List of Acronyms

**AGP**  Accelerated Graphics Port

**AHCI**  Advanced Host Controller Interface

**AOF**  append-only file

**API**  Application Programming Interfaces

**ASIC**  Application Specific Integrated Circuits

**BBM**  Bad Block Management

**BIO**  Block I/O

**BIOS**  Basic Input/Output System

**DMA**  Direct Memory Access

**DWD**  Double-Word

**ECC**  Error Correction Code

**EMIB**  Embedded Multi-die Interconnect Bridge

**EP**  endpoint device

**FFS**  Flash File System

**FPGAs**  Field Programmable Gate Arrays

**GPUs**  Graphics Processing Units

**HBA**  Host Bus Adapter

**HARP**  Heterogeneous Architecture Research Platform
HDD  Hard Disk Drives
I/O  Input/Output
IOH  Southbridge-I/O Hub
KVS  key-value store
LUTs  Look-Up Tables
MCH  Northbridge-Memory Controller Hub
MOSfet  Metal-Oxide Semiconductor field-effect transistor
MSI/MSI-X  message signaled interrupt
NDP  Near-Data Processing
NE  network element
NIC  Network Interface Card
NVMe  Non-Volatile Memory Express
NVMeoF  NVMe-over-Fabrics
OS  Operating System
PCIe  Peripheral Component Interconnect Express
PE  processing element
PRPs  physical region pages
RC  Root Complex
RDB  redis database file
RDMA  Remote Direct Memory Access
SAS  Serial Attached SCSI
SATA  Serial Advanced Technology Attachment
SCSI  Small Computer System Interface
SE  storage element

SoC  Systems on Chip

SAN  Storage Area Network

SR-IOV  Single Root I/O Virtualisation

SSD  Solid-State Drives

TLPs  Transaction Layer Packets

YCSB  Yahoo! Cloud Serving Benchmark
Chapter 1

Introduction

The emergence of social networks along with the growth of big data [122, 131] and its importance in machine learning [89], have triggered the need for storage systems to cope with large scale of data volume. Figure 1.1 shows the ever increasing demand on storage capacity worldwide for various applications, such as search engines [120], deep neural networks [17], social networks [138], and cloud service providers [19, 81]. For example, Facebook, the largest social network worldwide, stores and processes petabytes of data from their customers every week [138]. Thus, storage research has pushed much effort towards low latency and high performance storage systems.

In recent years, computer systems have followed a trend of enabling traditional Input/Output (I/O) operations, such as networking and storage, to operate independently from the processors, thereby increasing performance and energy efficiency [31, 52, 58, 59, 108, 120]. This thesis elaborates an investigation to identify the source of performance overhead on existing state-of-the-art storage devices and propose a novel low overhead and energy efficient solution that facilitates the concept of independent storage devices.

This chapter begins with a discussion on the evolution of storage advancements; from legacy magnetic disks to contemporary storage technologies (Section 1.1). Section 1.2 presents an overview of computer systems and discusses the trends of new generation computer systems. Section 1.3 describes the motivation for enabling storage devices to operate independently from the processor. Section 1.4 outlines the contributions of this thesis, while Section 1.5 summarises the ensued publications. Finally, Section 1.6 presents the structure of this thesis.
1.1. EVOLUTION OF STORAGE SYSTEMS

Legacy storage devices, such as Hard Disk Drives (HDD), are connected to the host computer via various protocol-based interfaces, such as Small Computer System Interface (SCSI) [7], Serial Advanced Technology Attachment (SATA) [12] and Serial Attached SCSI (SAS) [8]. These interfaces along with the Operating System (OS) have been designed and optimised incrementally, following the traditional assumption that processors deliver higher performance than any peripheral I/O device [115].

The advent of faster storage technologies, such as Solid-State Drives (SSD), enabled storage devices to perform at higher I/O rates and indicated the established interfaces as insufficient [144]. As a result, storage vendors standardised a new interface specification, the Non-Volatile Memory Express (NVMe) [3], that leverages high throughput interconnect (i.e. Peripheral Component Interconnect Express (PCIe)) providing significant I/O performance improvements against conventional SSD devices [144]. In addition, researchers pushed effort in new technology advancements such as the three dimensional stacking flash memory (e.g. 3D-Xpoint [44, 110]), that comprises a stackable cross-gridded data access...
array. Although this technology reduced the latency significantly, it is not widely deployed, due to the high cost. Hence, SSDs have become ubiquitous in data centres but there remains a performance overhead to be tackled. The primary reason is that computer systems still need to traverse multiple software layers in the OS, requiring the processor at first to perform the context switch and then start executing instructions on behalf of the OS. This process can significantly impact the storage performance [144, 146, 147], which is expressed with three metrics: the latency, the bandwidth and the I/O operations per second (IOPS). The latency is the time spent to execute an I/O transaction from the application memory address space to the storage device or backwards. The bandwidth is the rate that data move from storage device to main memory and backwards, while the IOPS metric is used to evaluate the throughput of data across various block sizes (e.g. 4KB, 128KB).

Figure 1.2 presents the access latency from the fastest to the slowest hardware component within a computer system, in a top-down order. The presented layers...
are grouped into four categories based on the level in which the data reside: a) the processor, b) the memory hierarchy, c) the non-volatile memory, and d) the magnetic disk drive. At the top, the processor uses registers to store temporary data with latency less than $1 \text{ ns}$ (1 clock cycle). The second group includes the multiple levels of cache memory with latency up to $10 \text{ ns}$ and the main memory with latency up to $100 \text{ ns}$. The third group refers to non-volatile memory advancements with latency ranging from $10 \text{ µs}$ to $115 \text{ µs}$, while the final group presents the magnetic disks with latency in the scale of milliseconds. Conventional computer systems require data to move from storage devices to the main memory to be processed by processor’s cores and then return to storage. As shown in Figure 1.2, there is a significant gap between the latency of storage devices and main memory, thereby reducing this gap is of paramount importance to deliver higher I/O performance to applications.

### 1.2 Overview of Computer Systems

For almost thirty years, the performance of processors had doubled every two years, as the number of transistors in a dense integrated circuit doubled [29, 111]. The transistor technology determines the efficiency of processors, as new technologies integrate shorter channel length and demand lower gate voltage to saturate. The aftermath of Dennard scaling [47] indicated that the operating frequency of the processors reached an upper bound, driving computer architects to discover alternative solutions to boost performance. Consequently, computer systems have become initially multi-cores and lately heterogeneous.

The following sections outline the primary components of contemporary computer systems and discuss new trends in computer architecture for the next generation of computer systems.

#### 1.2.1 Contemporary Computer Systems

Contemporary computers have been designed to exploit the diversity of different components in order to achieve high performance and energy savings. The emergence of heterogeneous computer systems introduced the concept of combining simpler processing cores, which can run in parallel with hardware accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) [66]. While accelerators have been traditionally connected via
PCle to mostly x86 computer systems, there is also a trend of heterogeneous Systems on Chip (SoC) with multi-cores and accelerators integrated on the same die and connected by an on-chip interconnect (e.g. Xilinx Zynq family and Intel/Altera Stratix 10 SoC; including future products prototyped by Intel-Altera Heterogeneous Architecture Research Platform (HARP) using Xeon class multi-cores and facilitated by the Embedded Multi-die Interconnect Bridge (EMIB) [105]). The SoCs provide low latency, fast data movement within the chip and energy efficiency because accelerators are physically connected via a physical on-chip interconnect, instead of using an external bus.

Figure 1.3 presents various components in conventional computer systems, such as the processor, the main memory, the storage media (e.g. HDD, SSD), the Network Interface Card (NIC), and accelerators (e.g. GPUs, FPGAs). The processor cores execute instructions that perform various arithmetic or logical operations between values stored in registers; and load/store operations of data from/to a memory location. The main memory is used for storing data near the processor via the memory bus, while the storage devices are used for storing data permanently in the legacy magnetic disks or in modern non-volatile memory, from which the data can be restored after powering down the computer. The NIC establishes a network link via a protocol (e.g. Ethernet), enabling multiple computers attached to the same network to communicate. Finally, the accelerators perform specialised tasks to offload the application code from the processor and increase performance and energy efficiency.

![Figure 1.3: The block diagram of two contemporary computers communicating via network.](image-url)
1.2.2 Next Generation Computer Systems

The components of computer systems can be identified as active elements, based on their ability to operate independently within a system. For example, the first active element in traditional computers is the processing element (PE), referring to the main processor which has the administrative role in the system for performing various operations, such as the networking, storage, and acceleration. In recent years, large-scale supercomputers in data centres show a trend of converting traditional computer components into active elements, that can operate independently from the processing element [52, 58, 59, 88]. The primary goal is to reduce the latency and achieve significant performance improvement, by eliminating any redundant load from the PE.

In recent years, research effort has focused on enhancing the NIC with computational power, driving the composition of the network element (NE) as presented in Figure 1.4. In this case a computer component (e.g. FPGA, SoC) is employed as the NE next to NIC to perform operations, offloaded by the main processor. The NE has been implemented either by using an FPGA (e.g. Catapult architecture with QSFP connectivity [31]) or a SoC (e.g. Bluefield SmartNIC with a NoC interconnect [108]), to enable the NIC component to operate independently.

The Catapult architecture has been implemented by Microsoft; extending the Microsoft data centre facilities with FPGA devices. The FPGAs have been used both for accelerating particular tasks at the application level (e.g. Bing search engine, deep neural network applications) and increasing the network bandwidth, by having an FPGA-based network capable of establishing a communication link between all the servers [120]. The second version of Catapult architecture [31]

![Figure 1.4: The block diagram with the network interface card (NIC) as a network element (NE), implemented either by an FPGA or a SoC.](image-url)
created the springboard for individual communication in the data centre. This was enabled by enhancing the network interface with FPGA devices - one device per server - responsible for performing the computation required to create packets and transmitting them to other servers, without requiring the processing elements of the system to interfere for every packet transaction in the system. On the other hand, the Mellanox Bluefield SmartNIC network adapter has employed a multi-core SoC that can be connected over PCIe to a server. This network adapter card has comprised multiple Arm cores, a network controller (i.e. Connect X5) and specialised accelerators, thereby enabling the NEs to communicate with low latency without the need of PE supervision [108].

1.3 Motivation

Following the composition of the network elements by adopting computational power near the NIC, research has focused on addressing the performance overheads in the PE related to storage devices [78, 130]. This is rational, considering that latest storage technologies, such as PCIe-based SSDs (i.e. NVMe SSDs) deliver up to three orders of magnitude higher performance than legacy magnetic disks [115]. Nevertheless, there are still performance bottlenecks that sustain the latency gap with main memory and need to be tackled. The majority of the bottlenecks derive from the OS software layers [118, 147].

The following sections discuss the key aspects that motivated the research contribution of this thesis. Section 1.3.1 presents an analysis of the performance limitations in current storage systems. Section 1.3.2 describes the technologies required for converting a computer component (e.g. NIC, SSD) into an active element. Finally, Section 1.3.3 presents a high-level view of the proposed solution.

1.3.1 Current Performance Limitations

The current software stack in the OS (e.g. Linux OS) adds extra overhead when accessing block devices (e.g. NVMe drives) due to the numerous software layers that submission and completion requests must traverse (including a number of expensive system calls). Figure 1.5 provides an insight of this overhead as measured on the Xilinx Zynq-7000 SoC, that consists of two Arm Cortex A9 processors and an FPGA, running the standard Flexible I/O (fio) benchmark [71] (typically used in assessing I/O performance). As shown in Figure 1.5, for all configurations of
1.3. MOTIVATION

Figure 1.5: The average Linux I/O latency for single 4KB block size I/O request, on the Xilinx Zynq-7000 SoC. The latency is reported in microseconds (µs).

read/write and sequential/random operations, the actual time of the device latency accounts only for 9-10% of the total latency. This means that up to 91% of the total execution time of a single I/O request is spent in submitting and completing the I/O request. Excluding the time spent in the device, the remaining time is distributed amongst the SYSCALL (SUBMIT I/O) to the BIO layer, the processing taking place at the BIO layer in the kernel (KERNEL), and the device driver (DRIVER). Despite the overhead is further exacerbated due to the low IPC in the Xilinx Zynq SoC (32-bit Arm ISA), even in high performing x86 systems this overhead can be up to 40% of the total execution time [147].

1.3.2 How to convert a Computer Component to Active Element?

A question that arises is how a computer component can be converted to an active element. Previous approaches for implementing a network element showed that further functionality is required to omit the PE. The main hardware components capable of delivering such functionality in digital computer systems are:

- General purpose processors perform tasks written in a high programming language (e.g. C) by decomposing them into small and simple subtasks performed by one or more cores. Their cost is lower than FPGAs and ASICs, but obtain lower performance and higher power consumption, as
they are designed for general purpose applications. Changes can be applied very easily by recompiling the new software program.

- **Application Specific Integrated Circuits (ASIC)** are special purpose hardware chips, requiring long development time by using electronic design automation (EDA) software tools. ASIC are costly due the fabrication but can achieve higher performance than general purpose cores and FPGAs, while consuming lower power [64]. Once an ASIC is manufactured, it can be used only for specific applications for which they are designed.

- **Field Programmable Gate Array (FPGA)** are digital circuits that can change hardware behaviour “on the fly”, due to their ability to support full and partial reconfiguration. Thus they offer higher flexibility than ASIC. Additionally, FPGAs are programmable by using a hardware description language (e.g. Verilog, VHDL) or high-level synthesis tools to compile high level languages such as C/C++ and OpenCL. FPGAs cost more and provide higher performance and energy efficiency than general purpose cores [65, 141].

### 1.3.3 How to Enable Storage Elements?

This thesis aims to address how computer systems can mitigate the existing software overhead (Section 1.3.1) and provide fast access to storage devices. The proposed solution shows that the establishment of a storage element (SE) can tackle the software overhead and facilitate storage devices to operate independently from the PE. To enable the transformation of storage devices into active elements, an Arm-FPGA SoC is used because FPGAs offer high flexibility and hardware benefits such as high performance and energy efficiency [65, 66]. In addition, an NVMe SSD is employed to exhibit the SE, as NVMe is a state-of-the-art interface for contemporary SSD devices. Figure 1.6 illustrates such a system, in which the SE is a SoC with integrated FPGA, physically connected to the NVMe SSD via PCIe. The SE can be connected to the PE via the PCIe bus or a chip-to-chip interconnect. However the scope of this thesis is to investigate a direct FPGA-based path from the SE to the NVMe SSD, as depicted in green colour in Figure 1.6; rather than the connectivity between PE and SE.
1.4 Contributions

In summary, this thesis makes the following contributions:

- A novel system architecture is proposed to demonstrate how storage resources can operate as active elements with no dependency on the processor of the system. The proposed architecture is called FastPath, and is evaluated in terms of performance and energy efficiency versus the baseline Linux OS on a Zynq 7000 SoC, against the fio [71] standard I/O benchmark. The experimental analysis shows that FastPath achieves up to 82% lower latency, up to 12x higher throughput, and up to 10x more energy efficiency than the baseline storage path in the Linux kernel.

- The FPGA device is used as the viable technology to enable the independent storage path operating orthogonally to the legacy storage path in the Linux kernel.

- A new classification scheme of the related work is presented in Chapter 4. The state-of-the-art of memory and storage systems are classified, based on the level at which the proposed technique is applied.

- A new application programming interface (FastPath_API) is presented as a thread-safe standard C library “libfnvme”, that interfaces with the FPGA-based FastPath architecture. Applications calling FastPath_API functions are able to perform direct access to the storage device, with no intermediate copy of data, as data are mapped directly in DMA regions.
• A new methodology is presented, showing how FPGAs can be used to dele-
gate functionality of a Linux device driver and allow FPGA-based systems
to access devices with no processor interference.

• Portability of FastPath architecture on a modern Arm-FPGA SoC (i.e.
Zynq UltraScale+ MPSoC) is ensured, allowing evaluation of the proposed
system in an in-memory database scenario with workloads from the Yahoo
Cloud Serving Benchmark (YCSB) [42].

1.5 Publications

A part of the material presented in this thesis has appeared in the following
publications:

• Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, Mikel
Luján, FastPath: Towards Wire-speed NVMe SSDs. In the 28th Interna-
tional Conference on Field Programmable Logic and Applications (FPL
’18), Dublin, Ireland, 2018.

• Athanasios Stratikopoulos, John Goodacre, Mikel Luján, Low Overhead
Poster Presentation in the PhD Forum in Design Automation And Test in
Europe Conference (DATE ’19), Florence, Italy, 2019.

• Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, Mikel
Luján, Enabling Low Overhead & Energy Efficient FPGA-based Storage

1.6 Thesis Structure

The remainder of the thesis is organised as follows:

• Chapter 2 provides a description of the software layers in the Linux op-
erating system responsible for operating I/O operations in block devices,
such as NVMe SSDs.

• Chapter 3 presents a detailed view of the internals of SSD drives, along
with an analysis of the PCIe standard interconnect.
• **Chapter 4** reviews related work on state-of-the-art memory and storage technologies. In particular, it studies the different approaches for tackling performance overheads and classifies that amount of work in five separate groups based on the level at which the proposed techniques are applied.

• **Chapter 5** presents the implemented systems including a software-based system (Software_raw) that bypasses part of the OS storage software stack and interfaces directly with the device driver; and the FastPath architecture, which evolved through three design versions. The first version (FastPath_v1) is responsible for implementing both submission and completion paths similar to the driver functionality. The second version introduces a control module in the FastPath architecture (FastPath_v2) which receives asynchronously I/O requests, removing any locking mechanism required by the previous implementation. The final version includes the integration of four fast paths (FastPath_v3), capable of submitting and completing four parallel I/O operations concurrently, via four independent memory ports.

• **Chapter 6** evaluates the performance and energy efficiency of the four systems, as presented in Chapter 5, against the baseline Linux kernel I/O paths for the fio I/O benchmark on an ARM-FPGA Zynq 7000 SoC. In addition, the performance of the latest FastPath system is evaluated against the baseline system for a real in-memory database application on an ARM-FPGA Zynq UltraScale+ MPSoC.

• **Chapter 7** summarises the thesis contributions and presents opportunities for future work.

• **Appendix A** presents the undertaken modifications in the NVMe driver source code, to enable FastPath to operate independently and coexist with the legacy Linux kernel I/O path.

• **Appendix B** presents examples of application code that uses the FastPath_API functions to access the NVMe SSD.
Chapter 2

NVMe System Architecture: A Software Perspective

2.1 The Advent of NVMe SSDs

Legacy hard disk drives (HDD) used to be the dominant technology in the storage market, since the advent of the first computers. However, with the emergence of multi-core and heterogeneous computer systems which delivered significantly higher performance than conventional computers, applications required storage systems to deliver higher I/O performance. The latest storage advancements, such as solid-state drives (SSD) increased storage performance by replacing the mechanical components of HDDs with non-volatile memory technology, capable of delivering faster access time and lower latency [144]. Consequently, SSDs are anticipated to obtain part of the HDD dominance in the storage market, as shown in Figure 2.1.

In recent years, the performance of SSDs was limited by the insufficient interface protocols (e.g. SATA [12]) designed initially for legacy HDDs. This motivated the emergence of a new interface specification, Non-Volatile Memory Express (NVMe) [3] which has been designed to mitigate the inefficiency of legacy storage interfaces, by leveraging the PCIe interconnect [125]. For example, a legacy SATA SSD device communicates with the system through the Advanced Host Controller Interface (AHCI) and the Host Bus Adapter (HBA). AHCI is an interface protocol while HBA is a chipset used to communicate with the storage controller via SATA bus. With the adoption of PCIe interconnect as the primary SSD interface, the AHCI protocol had to adjust and tackle several challenges to
2.1. THE ADVENT OF NVME SSDS

Figure 2.1: Shipments of hard and solid state disk (HDD/SSD) drives worldwide from 2015 to 2021. The number of shipments is reported in millions.
Source: www.statista.com

sustain the PCIe I/O bandwidth. These challenges emerged because the AHCI was designed for legacy HDDs; therefore it supports a single command queue, allows up to one interrupt and requires a synchronization mechanism for atomic access of multiple processors on the command queue. Table 2.1 presents the main differences between a legacy protocol (AHCI) and a state-of-the-art protocol (NVMe). The NVMe specification is discussed in more detail in Section 2.3.

This chapter aims to provide the reader with a better understanding of the NVMe system software. Section 2.2 presents a breakdown analysis of the NVMe software layers within the Linux OS storage stack, while Section 2.3 discusses the NVMe v1.1 specification, which is the version used in the undertaken experiments. Finally, Section 2.4 describes NVMe-over-Fabrics, which is a concept that has emerged to enable remote NVMe access via a network protocol.

<table>
<thead>
<tr>
<th></th>
<th>AHCI</th>
<th>NVMe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Command Queues</td>
<td>1</td>
<td>64k</td>
</tr>
<tr>
<td>Maximum Queue Depth</td>
<td>32</td>
<td>64k</td>
</tr>
<tr>
<td>Number of Interrupts</td>
<td>Single Interrupt</td>
<td>2048 MSI-X Interrupts</td>
</tr>
</tbody>
</table>
2.2 The NVMe Software Layers

The NVMe system architecture is a two-layer architecture in which a software stack processes user requests and submits them to the storage device, while the underlying hardware executes the software stack operations. Figure 2.2 presents the NVMe architecture highlighting the most critical parts of each layer. The control flow of the software layer is depicted in black arrows and includes the system call and the software operations from every layer within the stack. At the beginning user applications, through the File System or Direct I/O, submit requests (Step 1) to the Block I/O which end up in the NVMe driver (Step 2). The driver, in turn, retrieves those requests, creates corresponding NVMe commands, and writes them to memory (Step 3). The NVMe controller reads the commands via PCIe Root Complex and serves the associated I/O requests (Step 4).

The Linux OS software layers are executed on the main processor of the system and they communicate indirectly with the NVMe controller via the PCIe Root Complex. The PCIe Root Complex is a hardware unit used as a bridge between the main processor/memory and the PCIe devices. The processor is connected
to the PCIe Root Complex via the Direct Media Interface (DMI) while the SSD device uses the PCIe interconnect. The hardware layer of the NVMe architecture, including the PCIe subsystem, is discussed in detail in the next chapter.

2.2.1 The File System & Direct I/O Layer

The Linux OS manages the NVMe SSDs as any other block device in the system. Therefore, the associated Linux device driver is developed with respect to the I/O software stack. Any application (e.g. dump disk -dd, or a user program) can read/write data from/to the disk by using either a file system or directly through a system call (e.g. ioctl, read, write).

Typically, applications run in user mode, which is a system mode with limited permissions. On the contrary, OS kernels are built in a privileged mode that enables them to access and administrate all system resources. Therefore, every OS, including Linux, offers various system call functions that allow them to perform device operations on behalf of the requesting applications. Moreover, several specifications of data structures, variables, and routine calls have emerged, widely used in software libraries to compose Application Programming Interfaces (APIs) (e.g. POSIX, Windows API). The APIs have been used as abstraction layers to pass user level requests to the OS kernel.

The file system is such a layer in the storage system, allowing any application to access data within a block device without interacting directly with the physical elements in the storage device, such as sectors and logical block addresses. A sector is a group of low-level storage units, while a logical block address is an indirect storage address used by drivers and storage controllers to point to particular chunks of data in storage. More information about the organisation of flash memory is given in Section 3.3.1. Data are packed in groups called files, and stored in a specific logical block address, from which it can be easy to retrieve. The file system includes data structures (e.g. inode), that store the logical block addresses along with some information (i.e. access permission, file descriptor, file ownership, and file type), associated with any file in the file system. Thus, the file system is aware of the disk location where a file resides and allows multiple applications to use the APIs and share files, in case they are permitted to.

Linux OS uses the technique of “buffering”, which stores data temporarily within buffers (in kernel memory address space) to increase I/O performance and then copies buffer contents from/to the block device. Due to this technique, the
successful execution of a \texttt{write} system call does not necessarily mean that the data have been written to the block device. If the application requires the data to reach the block device, then it uses the \texttt{fsync} system call to flush the buffers into the block device.

Applications, that require to no copy from user to the kernel buffers and require the read operations to access the block device rather than hit the memory cache, can use the Direct I/O feature of a file system. The Direct I/O feature is enabled through the \texttt{O\_DIRECT} flag, once a file is opened, and drives the data into the block device directly through a Direct Memory Access (DMA) request. DMA is a feature of contemporary computer systems, that allows software and hardware subsystems to perform memory operations (read/write) without dependency on the processor [43].

Linux OS supports numerous file systems, however the most widely used for block devices are the \textit{ext* family} (e.g. \texttt{ext3} [140], \texttt{ext4} [107]), \texttt{XFS} [136], \texttt{JFS} [124], \texttt{ReiserFS} [50] and \texttt{btrfs} [123]. The latest Linux kernels (since version 3.8) have included the \textit{Flash-Friendly File System (F2FS)} [91].

2.2.2 The Block I/O Layer

The Block I/O (\texttt{BIO}) subsystem is a software layer in the Linux OS that performs I/O requests on block devices [72]. In essence, it is responsible for communicating and manipulating I/O requests from applications to the storage devices, thereby providing a single entry point from the upper levels to the lower levels of the software stack.

Following the trend in modern multi-core systems, the \texttt{BIO} subsystem has evolved to support multiple request queues [26]. Figure 2.3 depicts the multiple queue block I/O scheme, including two level of queues (in the kernel address space): the software staging queues, and the hardware dispatch queues. The software staging queues can be configured to be one per socket or one per core, whereas the number of hardware dispatch queues matches the number of storage device queues.

The following \texttt{BIO} features are described based on the NVMe paradigm, in which the number of storage device queues is equal to the number of physical cores. In particular, \texttt{BIO} provides staging, statistics, error handling, and fair scheduling of requests to improve performance. \texttt{BIO} receives I/O requests from processor’s cores on behalf of a user level application and adds these requests
The block I/O multiple queue scheme.

Figure 2.3: The block I/O multiple queue scheme.

on the software dispatch queues (staging). After staging the requests, BIO may perform I/O scheduling [70], including merging I/O requests and forwarding I/O requests with high priority levels to be executed quickly. Afterwards, the requests are dispatched in a hardware dispatch queue based on the queue mapping stage. The mapping between the two level of queues allows BIO to operate even when the number of software staging queues and hardware dispatch queues differ. A hardware dispatch queue collects multiple I/O requests targetting a specific storage queue, and forwards these requests one by one to the NVMe driver for execution. After the fulfilment of a request, the NVMe driver notifies BIO, which consequently returns the completion status (success/error) to the user applications.

Finally, the multiple queue block I/O scheme increases the potential for emerging storage systems that leverage the PCIe interface; previously limited by the legacy kernel I/O stack [26].
2.2.3 The NVMe Driver

The NVMe block device is a “hot plug” component in the Linux OS, as it can be inserted and removed in a running computer system without requiring explicit interruption. Thus, the Linux OS probes the device, and invokes the right device driver (i.e. NVMe driver) to perform the initialisation phase. During the initialisation phase, the NVMe driver allocates specific data structures to establish the communication between the processor and the NVMe controller. In the default configuration, the allocated data structures have the form of in-memory circular queue pairs (i.e. submission, completion) with 64 KB size, as shown in Figure 2.4. The number of queue pairs is equal to the number of physical cores of the processor, to discard synchronisation issues and ensure that queue entries are not dispersed among different core caches [14]. For example, a system with four cores allocates one pair of administrative queues and four pairs of I/O queues. Nevertheless, the submission/completion scheme of the NVMe queues is configurable and can allow multiple cores to share one completion queue, thereby leading to more efficient memory utilisation [80].

After the completion of the initialisation phase, the administrative queues are used to perform the administrative operations, such as the creation/deletion of an I/O queue pair, while the remaining queues are used to perform I/O transactions on behalf of the processor’s cores. Figure 2.4 illustrates the hardware flow of the NVMe driver functionality in five steps. At first, the driver dispatches an I/O request, synthesises an NVMe command for the request, and submits the command in the submission queue (Step 1). Then the new tail is written in a memory mapped PCIe register called “doorbell” (Step 2), which notifies the NVMe controller of new submission entries (Section 2.3.3). Thereafter the NVMe controller starts fetching the submitted commands (Step 3) and processing them (Step 4). After the execution of each command, the NVMe controller issues a completion entry in the completion queue of the same pair of queues from which it fetched the command, and triggers an interrupt to notify the associated processing core of the completion (Step 5). Finally, the interrupt is handled by the interrupt handler in the NVMe driver which analyses the completion entry and delivers information about the outcome back to BIO. The interrupt driven completion method in the NVMe driver can be replaced by a polling method, which is discussed in more detail in Section 2.3.4.
2.3 The NVMe Specification

NVMe is a host interface specification designed and optimised to perform efficient data transfers over PCIe for non-volatile storage media. During the last seven years since the first release, the NVMe specification has been revised three times introducing various improvements [3]. Nevertheless, the high performing and scalable I/O access on the underlying SSDs remains the primary objective in the design of NVMe.

The main advantages of the NVMe specification compared with other storage predecessors (i.e. SATA HDD, SATA SSD) are:

- Optimised command submission and completion paths.
• Standardisation of an efficient set of commands. Each command is 64B in size and is capable of performing 4KB or 128KB I/O requests, resulting in low latency.

• Parallel operations by supporting up to 65,535 I/O queues with up to 64K commands per I/O queue.

• Support for data protection (compatible with SCSI Protection Information, commonly known as T10 DIF, and SNIA DIX standards).

• Supports two methods of completing NVMe commands: the message signalled interrupt (MSI/MSI-X) [5] and polling scheme. Section 2.3.4 analyses both methods in detail.

• Support for I/O virtualised architectures, such as Single Root I/O Virtualisation (SR-IOV) [119].

• Support for multiple non-volatile memory regions called namespaces. Namespaces are groups of logical memory blocks.

• Support for multi-path I/O and namespace sharing.

• Enhanced and documented error reporting.

The following sections analyse the internals of the NVMe specification. Section 2.3.1 describes the namespace entity and its importance for enabling multiple I/O paths and virtualisation support. Section 2.3.2 describes the NVMe command set, while Section 2.3.3 presents the basics of the “doorbell” register. Finally, Section 2.3.4 details two methods of command fulfillment from the NVMe controller back to the NVMe driver.

2.3.1 Namespace and Multiple I/O paths

The NVMe system architecture allows multiple I/O paths to share a physical storage device, by partitioning the non-volatile memory volume into namespaces [14, 16]. The namespaces are groups of logical memory blocks, configured by the NVMe device driver to arbitrate access to a particular storage partition. Figure 2.5 presents multiple I/O paths that access the NVMe SSD via a port in the PCIe Root Complex. For simplicity, this port is mentioned as PCIe Root Complex for the rest of the thesis. The N number of the I/O paths, shown in Figure
2.3. THE NVME SPECIFICATION

2.5, is bounded by the maximum number of the NVMe queues in the NVMe system architecture. In addition, the NVMe controller communicates physically via one PCIe Root Port; and is connected to multiple host computers by using an external PCIe switch. More details about the PCIe subsystem are given in Section 3.4. The NVMe driver allocates the namespaces and assigns a unique identifier for each namespace, thereby allowing the NVMe controller to forward every incoming request from the I/O paths to the corresponding and authorised partition.

NVMe SSDs can also be configured to allow communication via multiple PCIe Root Ports [16]. Figure 2.6 presents the block diagram of an NVMe device with two PCIe Root Ports and two NVMe controllers. The configuration of this device allows both private (NSID 1, NSID 3) and shared (NSID 2) namespaces. Consequently, the namespace is the foundation for hosting multiple I/O paths transparently. Finally, the NVMe system architecture can be extended with PCIe-based virtualisation support (i.e. SR-IOV) to allow multiple I/O paths from separate computers to share the same PCIe interface [119], but this is beyond the scope of this thesis. More details are provided in the NVMe specification [16].
2.3.2 The NVMe Command Set

The NVMe command set is composed of multiple commands divided into two groups: Admin commands and I/O commands. Both groups of commands are used to instruct the NVMe controller to perform a specific task. Table 2.2 presents the mandatory commands that are widely used by the NVMe driver, both for configuring the queues and executing I/O requests. The following paragraphs describe the fields of the NVMe commands, and the NVMe completion queue entry.

Table 2.2: The NVMe command set.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Admin Commands</th>
<th>I/O Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Delete I/O Submission Queue</td>
<td>Flush</td>
</tr>
<tr>
<td>01h</td>
<td>Create I/O Submission Queue</td>
<td>Write</td>
</tr>
<tr>
<td>02h</td>
<td>Get Log Page</td>
<td>Read</td>
</tr>
<tr>
<td>04h</td>
<td>Delete I/O Completion Queue</td>
<td></td>
</tr>
<tr>
<td>05h</td>
<td>Create I/O Completion Queue</td>
<td></td>
</tr>
<tr>
<td>06h</td>
<td>Identify</td>
<td></td>
</tr>
<tr>
<td>08h</td>
<td>Abort</td>
<td></td>
</tr>
<tr>
<td>09h</td>
<td>Set Features</td>
<td></td>
</tr>
<tr>
<td>0Ah</td>
<td>Get Features</td>
<td></td>
</tr>
<tr>
<td>0Ch</td>
<td>Asynchronous Event Requests</td>
<td></td>
</tr>
</tbody>
</table>
The NVMe Command Fields  Every NVMe command is a set of sixteen Double-Word (DWD) (32-bits) fields, as shown in Figure 2.7. The first ten DWDs are used by every command to provide information about the requested operation, while the last six store command-specific information. In more detail every NVMe command includes:

- **DWD 0**: This field includes the opcode, the flags and the command_id.

- **DWD 1**: This field has the namespace identifier (nsid), in case the NVMe system hosts multiple namespaces.

- **DWD 2-3**: This is reserved.

- **DWD 4-5**: This field is a metadata pointer (mptr), which points to a continuous physical buffer of metadata. In case the metadata are not interleaved with the logical block data, the DWD 2-3 hold information about the alignment of the metadata buffer.

- **DWD 6-9**: The NVMe driver uses physical memory regions called physical region pages (PRPs), either as contiguous memory pages or as scatter-gather lists, to write or read data to and from the disk, respectively. The prp1 field contains the address of the first physical page, which holds the

\[
\begin{array}{|c|c|c|}
\hline
\text{DWD (0 - 1)} & \text{Opcode, Flag, Command_Id} & \text{Namespace_Id (nsid)} \\
\hline
\text{DWD (2 - 3)} & \text{Reserved} & \\
\hline
\text{DWD (4 - 5)} & \text{Metadata} & \\
\hline
\text{DWD (6 - 7)} & \text{Prp1} & \\
\hline
\text{DWD (8 - 9)} & \text{Prp2} & \\
\hline
\text{DWD (10 - 11)} & \text{Slba} & \\
\hline
\text{DWD (12 - 13)} & \text{Control} & \text{Dsmgt} \\
\hline
\text{DWD (14 - 15)} & \text{Reftag} & \text{Apptag} & \text{Appmask} \\
\hline
\end{array}
\]

Figure 2.7: The NVMe command.
data to be transferred. Whereas, prp2 field contains the address of the second physical page or a list of pages, in case the data transfer length exceeds one memory page. In case the top 2 bytes of DWD 2-3 have a non-zero value, the DWD 6-9 fields hold a scatter-gather list entry (sgl). For more details, please refer to [15].

- DWD 10-11: This field holds the 64-bit starting logical block address (slba), where the data should be read/written from/to.

- DWD 12: The control field contains information about the configuration of the requested operation. For example, it indicates the controller for limited retry efforts in case of failure, and mentions that the data should be written in the NVM media before notifying command completion. Also, it holds the number of logical blocks (length) to be read/written.

- DWD 13: This field is reserved but the lowest 8 bits show information about the data associated with logical block addresses [15].

- DWD 14-15: The last two fields hold three tags: a) the Initial Logical Block Reference Tag (ilbrt), b) the Logical Block Application Tag Mask (lbatm) and c) the Logical Block Application Tag (lbat). All these tags are used for cross-checking, in case the namespace is formatted for end-to-end protection.

The NVMe Completion Queue Entry  The NVMe completion process starts when the NVMe controller has fulfilled the execution of an NVMe command, and prepares a response back to the NVMe driver. The response is a group of information about the command execution, and it is written in the completion queue, that is paired with the associated submission queue of the NVMe command. Consequently, the NVMe controller sends an interrupt to the NVMe driver to notify of a new completion entry. The NVMe completion queue entry holds information about the execution of a particular submitted command. The NVMe controller issues completion entries back to the driver, to indicate successful or failing operations. The completion queue entry is composed of four DWDs, as shown in Figure 2.8; and each entry has the size of 16 Bytes.

The first two double-words contain command specific information, while the third (DWD 2) stores the submission queue identifier (sq_id) and its current
2.3. THE NVME SPECIFICATION

head pointer (`sq_head_ptr`). The head pointer is updated by the controller and can be used to inform the driver about the available space in the submission queue. Finally, the fourth double-word (DWD 3) has information about the currently completed command identifier (`command_id`), along with the status field and the phase. The status field returns ‘0’ in case of success, otherwise, it reports any command-specific error codes (e.g. an invalid namespace, an invalid field in command, a data transfer error, etc.) [16]. The phase is a single bit, used to indicate the current state of the completion queues to the NVMe driver, as NVMe queues are circular queues. The phase changes its value similar to a flipped coin, every time the completion queue is full, and the head returns to the first place. The initial value of phase is ‘0’.

2.3.3 The NVMe Doorbell Registers

Every PCIe device allocates a range in main memory during device enumeration to store device specific registers [43]. The base address pointing to that memory range is stored in a hardware register within the PCIe Root Port, named the PCIe Base Address Register (BAR).

Being a PCIe-based device, the NVMe SSDs stores device specific registers (e.g. “doorbell” registers) into the address range pointed to by the BAR. The “doorbell” register is used in the NVMe system architecture to inform the NVMe controller for new submission entries in the NVMe submission queues [14]. Thus, the number of allocated doorbell registers in the host system is equal to the number of the NVMe submission queues, and are stored sequentially within the allocated memory, forming a doorbell vector.


2.3.4 MSI/MSI-X Interrupts & Polling

The completion of I/O requests can be broadcasted to the NVMe driver via two methods. The first method is interrupt driven, in which different interrupt identifiers are assigned to each completion queue. Whereas, the second is the method of polling the completion queues.

The NVMe system supports Message Signaled Interrupts (MSI) and its extension MSI-X [5]. Both MSI and MSI-X replace the conventional dedicated interrupt lines, with special memory-based messages capable of delivering multiple interrupts in a PCIe-based system. In particular, the OS assigns some memory-mapped I/O addresses in every peripheral MSI/MSI-X device. The device can write some interrupt-specific data into one of these addresses, which will trigger an interrupt into the processor linked to the I/O completion queue [13]. The MSI method allows a device to use up to 32 interrupts, while the MSI-X supports up to 2048 interrupts. Considering, that NVMe aims to be a scalable standard, MSI-X seems to be the best option for performing interrupts [16].

However, the increased I/O traffic in the interrupt controllers and the concurrent software overhead caused by multiple context switches [146] can make the interrupt-driven approach inefficient [128]. Therefore, several solutions have been proposed, such as interrupt coalescing [18] and NVMe polling [128, 146]. Interrupt coalescing is a technique which groups events that would trigger a hardware interrupt, holding back until either a specific number of events occurs or a timeout is exceeded. NVMe polling is a method which generates periodic memory requests at the head of an NVMe completion queue until the status and phase fields of the returned value indicate a new entry. Thence, the NVMe polling continues pointing at the new head of the queue. Moreover, Shin et al. [129] presented a low latency I/O completion scheme, indicating a hybrid solution of polling and interrupt methods as appropriate for scalable SSDs [129].

2.4 NVMe-over-Fabrics

The number of NVMe I/O queue pairs is bounded by the number of physical cores in the host system. Hence, there is spare capacity of I/O queues, that could be exploited and increase performance even further. Grouping hardware resources under the hood of a network protocol is a common practice in data centers for increasing their utilisation, widely known as disaggregation. However
2.4. NVME-OVER-FABRICS

NVMe disaggregation is considered more challenging [23, 37, 86, 101, 118], because NVMe drives perform faster than the existing network storage protocols (e.g. Internet Small Computer Systems Interface (iSCSI)), therefore the additional latency for protocol processing and copying data between kernel and user space can be critical to the overall performance [76].

In recent years, many storage vendors have focused on defining a specification that enables NVMe drives to be accessed through a network interface layer, resulting in the establishment of the NVMe-over-Fabrics (NVMeoF) specification [4]. NVMeoF is designed to support a range of networking protocols, called fabrics, such as Ethernet, Fibre Channel and InfiniBand. The main goal of NVMeoF is to provide remote connectivity to NVMe SSDs, with negligible additional latency (up to 10 µs [4]) against a local NVMe drive inside a server. NVMeoF currently supports two types of fabrics: Remote Direct Memory Access (RDMA) based fabrics and Fibre Channel. The RDMA-based fabrics are widely employed in the high performance computing domain and modern data centres. On the contrary the Fibre Channel fabric requires deploying a dedicated Fibre Channel network or encapsulating it over a consolidated network such as Ethernet [39, 57, 61, 103, 134].

This thesis currently focuses on local communication with NVMe drives, however, it is designed to be extended with a FPGA-based network fabric in the future. Such a system can be essential to investigate efficient NVMe remote access.

2.4.1 RDMA-based Fabrics

RDMA is a remote memory management unit that allows direct data transmission between two machines over a network interface, without any processor interference [132]. NVMeoF systems leverage the RDMA unit because it offers zero-copy networking data transfers from the application memory with no kernel involvement [62]. Figure 2.9 shows how an NVMe system architecture can perform both local and remote I/O requests. The local path is illustrated with black arrows and presents the host system accessing the local NVMe drives via PCIe. On the other hand, the remote path is illustrated with red arrows and depicts how I/O requests are forwarded remotely from the host system (Figure 2.9a) to the target system (Figure 2.9b) via the RDMA-capable Network Interface Card (NIC).

The remote path comprises eight main steps. Once a remote request is generated at the NVMe driver in the host system, at first it traverses the RDMA
stack (Step 1) and then moves to the current network-specific driver (e.g. Infini-
bond, Ethernet) (Step 2). Thence the command is transmitted in packet form via
the NIC to the remote machine (Step 3), in which it traverses the RDMA stack
in the target system (Step 4), before arriving at the NVMe driver layer (Step
5). Whenever the NVMe driver in the target system receives remote requests, it
forwards them back to the block I/O layer (Step 6), which then generates I/O
requests back to NVMe driver for submission (Step 7). Finally, the NVMe driver
in the target system synthesises the NVMe commands and submits them locally
to an RDMA dedicated NVMe queue (Step 8). Before a transfer is made, the
host communicates with the NVMe controller in the target system and they co-
operatively determine a one-to-one mapping between the NVMe queues and the
RDMA queues (dedicated for this transfer). Both NVMe driver layers in host
and target systems share the RDMA queues to facilitate the remote service of
requests [62]. The total route of remote I/O requests is depicted in Figure 2.9 in
red arrows.

Popular network fabrics integrating RDMA capabilities are InfiniBand [99],
RDMA over Converged Ethernet (RoCE) [38] and Internet Wide Area RDMA
Protocol (iWARP) [41]. Although InfiniBand has been the original network in-
terface associated with the RDMA approach, it requires special purpose intercon-
nect. The remaining fabrics use Ethernet; RoCE offers RDMA over Ethernet and
the UDP/IP protocol, while iWARP offers RDMA over the TCP/IP protocol.
2.4. Current Limitations of NVMe-over-Fabrics

NVMeoF is introduced to enable low overhead remote access to NVMe SSDs over a network fabric. Although current NVMeoF systems allow applications to target remote NVMe drives with high I/O rates by providing distinct hardware queues and namespaces as isolation schemes, they cannot mitigate the overhead due to the interference between multiple remote client systems [87]. Thus, the NVMeoF systems face lack of performance isolation in a multiple clients scenario, while NVMe capabilities of the storage device remain underutilised [87].

2.4.3 NVMe-over-Fabrics vs Software-based Solutions

Unlike NVMeoF hardware approaches, related work has also focused on investigating software-oriented solutions for flash disaggregation.

Klimovic et al. [86] employed iSCSI to investigate flash disaggregation. The proposed system is evaluated on the same Intel-based host and client servers against the RocksDB key-value store database [11]. The results showed that although the remote access over iSCSI protocol has a 20% reduction in throughput, the overall performance of the disaggregated system increases due to the CPU and flash scaling. As an extension to this work, they explored a light-weight dataplane kernel, integrating both storage and networking interfaces to build a software-based flash server. The evaluation was undertaken on the same machine as previous work against RocksDB and FlashX graph processing framework [2]. The results showed that the proposed system suffered up to a 4% slowdown against local flash, whilst the baseline remote system over iSCSI performed up to 32% slower [87].

On the contrary, Guz et al. [62] conducted a similar experiment to Klimovic [87], using an NVMeoF system above the RoCE fabric, to explore flash disaggregation. The evaluation was undertaken on an Intel-based machine against RocksDB database. The results showed that the conventional remote access over iSCSI had up to 40% performance degradation, whilst the NVMeoF system performed up to 2% slower against the local storage access.

Finally, Trivedi et al. [139] used RDMA to create a direct data path between the client and the remote flash storage drive. The presented storage and network stack co-design approach, comprised an RDMA controller, along with the flash controller and a file system on a cluster with 17 Intel CPUs. The performance
evaluation of the proposed system against the baseline system (over sockets and files) showed 43.5% lower latency and 2x faster performance when running the Aerospike Key-Value store database [1].

2.5 Summary

This chapter presented how contemporary storage evolved from hard disks to PCIe-based SSDs and introduced the NVMe system architecture from a software point of view.

Firstly, it described the Linux OS storage software stack, including all the software layers in which an I/O request crosses. User I/O requests traverse through the File System or Direct I/O, then pass to the Block I/O and end up at the NVMe driver. The NVMe driver forms low-level commands, then writes them into the NVMe queues and notifies the NVMe controller to start fetching the commands. Secondly, this chapter presented the NVMe commands in conjunction with a thorough description of the NVMe specification. Finally, it discussed the NVMe-over-Fabrics concept as a way to transport I/O requests remotely, across a network fabric. The most widely used fabrics in the high performance computing domain and modern data centres are based on RDMA (e.g. InfiniBand, RoCE, iWARP).

The next chapter analyses the hardware view of the NVMe system architecture. At first, it presents how NVMe drives operate, including two basic hardware components, the NVMe controller and flash memory. Finally, it overviews the PCIe system and discusses the implementation of the PCIe Root Port both in the processing system (i.e. CPU) and the programmable logic (FPGA).
Chapter 3

NVMe System Architecture: A Hardware Perspective

3.1 Introduction

Non-volatile memory technology is adopted on various storage devices (e.g. Flash cards, USB Flash drives, SSDs). Each device has a different communication user interface and can be connected via separate I/O ports, based on the requirements of each application. For example, smart-phone devices, constrained by battery power and size, require small sized storage media, such as microSD cards, while high-performance servers utilise the PCIe-attached SSDs.

The SSD architecture is comprised of two main components: the NVMe controller, and the flash memory (Figure 3.1). This chapter describes the architecture of contemporary flash memory-based drives, from the NVMe controller (Section 3.2) to the flash memory (Section 3.3), and provides an overview of the PCI Express host interface [5] in Section 3.4.

3.2 NVMe Controller

The NVMe controller is an embedded processor in the SSD, and it is responsible for ensuring the correct operation of the flash drive. Thus, the NVMe controller executes software operations including:

- Establishment of the host interface (i.e. PCIe, SATA, SAS), along with the initialisation of the necessary data structures in the host memory.
• Initialisation of the Flash File System (i.e. Wear Leveling, Garbage Collection, Bad Block Management).

• Error correction of the cells in the flash drive.

• Execution of submitted NVMe commands.

• Notification back to the driver of successful or unsuccessful completion of the submitted commands.

The following sections analyse the aforementioned operations in more detail.

3.2.1 Host Interface and Initialisation of Data Structures

The host interface is the first block presented in the NVMe SSD block diagram in Figure 3.1 and enables communication between the NVMe SSD and the host computer. Both, the host computer and host interface in the SSD, set up a communication link which uses an industry-standard protocol (i.e. PCIe, SATA, or SAS). In the case of PCIe the host interface block is a combination of hardware buffers and low-level drivers for the packetising/depacketising of the PCIe packets, and device-specific firmware for the correct operation of the storage device. The
3.2. NVME CONTROLLER

PCle subsystem is discussed in detail in Section 3.4. Finally, the NVMe controller executes driver operations, such as the configuration of the NVMe queues, as mentioned in Section 2.2.3.

3.2.2 Flash File System

The Flash File System (FFS) is a module in the flash drive that enables the flash memory-based SSDs to be used identically to a magnetic disk. As a file system, the FFS hides the information about physical memory blocks from the kernel driver and the user application, thereby enabling storage sharing among multiple local and remote applications [49]. In essence, the FFS holds a mapping between the address of a logical block and the corresponding physical block of the data, as illustrated in Figure 3.2. However, this is a complex task that will be discussed in more detail in Section 3.2.2.2. The FFS performs three main operations, Wear Leveling, Garbage Collection and Bad Block Management, as shown in Figure 3.1.

3.2.2.1 Wear Leveling

Wear Leveling deals with the aging of the storage system. Modern SSDs use Non-Volatile Memory blocks to write data in blocks. Hence, some blocks of data can be frequently updated, while other blocks remain unmodified for longer periods of time. This results in having some blocks highly stressed with multiple write/erase cycles, while other blocks operate under less stress. This inconsistent usage of memory blocks can lead in part to the aging of the flash drive, and eventually to hardware failure [49]. Thus, it is essential to balance the number of writes and erases among all the available blocks.

![Figure 3.2: Block management in flash memory.](image)
3.2.2.2 Garbage Collection

Due to flash limitations, data updates in flash drives do not necessarily mean overwriting the contents on the physical memory block [49]. On the contrary, the current physical block is invalidated, and a new physical block that stores the latest data, is allocated and mapped to the same logical block address. However, the continuous allocation of blocks can reduce the number of free physical blocks to be allocated, thereby resulting in significant performance loss.

Hence, the blocks are split into sectors, and every invalidated sector should be recycled to be used transparently. This task is known as Garbage Collection [34], and is performed by the Garbage Collector module, as shown in Figure 3.3. The Garbage Collector keeps track of every block containing invalid sectors, such as Block A in Figure 3.3a, and uses a threshold based on the number of remaining available sectors. This threshold triggers the collection of every invalid sector in Block A, while the Garbage Collection process is in the initial state (Figure 3.3a). The sectors with valid data are duplicated to a new empty block such as Block B in Figure 3.3b and the logical to physical address mapping is updated, as illustrated in Figure 3.2. The remaining sectors in Block A are firstly invalidated (Figure 3.3c), and then the whole Block A is erased and added in the free block pool as depicted in Figure 3.3d.

When the majority of the sectors within a block are valid, the overhead of moving further these sectors into a new block can be significant. This phenomenon is called write amplification and is examined by storage researchers [92, 104, 135]. Contemporary flash drives perform the Garbage Collection task in the background to reduce the impact on performance [49].

3.2.2.3 Bad Block Management

As mentioned earlier, several memory blocks in flash-based SSDs can be damaged because of early aging, caused by high stressing or other hardware issues related
to technology. These blocks are called *Bad Blocks* and must be excluded from the list of the available blocks of memory. This operation is performed by the *Bad Block Management (BBM)* module, which creates and maintains a map of bad blocks. It is noticeable that this map is generated during the manufacturing of the flash memory (it can include some already present bad blocks) and it is maintained during device lifetime \[30, 49\].

Considering that the number of bad blocks increases during the aging of flash-based SSDs and that the required block address translation tables reside in the flash memory, the overall storage capacity can be significantly reduced.

### 3.2.3 Error Correction Code

As mentioned in Section 3.2.2, overwriting the data in flash memories is a complex task, requiring the current sector to be duplicated and invalidated, and the address mapping to be updated to point to the new sector. Then the old sector is marked to be erased and reused. The *erase process* involves hitting the flash cell with a relatively large charge of electrical energy. This process can cause the semiconductor layer to degrade and result in increased bit-error rates. The Error Correction Code (ECC) module can mitigate this issue by correcting the raw bit errors \[137\]. However, multiple erasures could make the ECC module unable to keep up with the error rates, making the flash memory unreliable \[25\].

### 3.2.4 Execution of Submitted Commands

As mentioned in Section 2.2.3, the NVMe controller is notified by the OS device driver, through a mechanism called *ringing the bell*. Whenever the NVMe driver submits a new command to the NVMe queues, the respective doorbell register is updated with the new tail. Consequently, the NVMe controller becomes aware of the new submitted command and starts fetching the command from the NVMe queue. Every fetched command is executed based on the operation that it is asked to perform. No matter the type of the NVMe queue (Admin or I/O), the execution of the submitted commands follows the same procedure.

### 3.2.5 Acknowledgement of the Requests

The execution of every submitted NVMe command is followed by an NVMe response. The NVMe controller forms responding messages, based on the success
or failure of the executed commands, and issues them in the tail of the associated NVMe completion queue. In case of faulty execution, the NVMe response contains an error code, pointing out the reason that caused the failure.

3.3 Flash Memory

Flash memory is arranged as an array of floating gate transistors, called as memory cells, each one capable of storing digital data. In addition, the placement of the memory cells determines the flash memory as a NAND or NOR flash memory. These two types of memories have significant performance differences. For example, NOR-based memory performs faster reads than NAND-based memory, while the latter performs significantly faster write operations than the first [121].

This section presents the internals of flash, starting with a description of the memory organisation (Section 3.3.1) and then outlining flash at the transistor level (Section 3.3.2). Consequently, Section 3.3.3 elaborates on the structure of NAND-based flash\(^1\) since the experiments presented in this thesis (Chapter 6) are made on such device. Finally, Section 3.3.4 presents the concept of caching data in DDR memory.

3.3.1 Flash Memory Organisation

Flash-based SSDs are designed to be functional with existing storage software stack. Hence, Flash-based SSDs are arranged in blocks, each of which contains multiple sectors, as shown in Figure 3.4. A sector is composed of multiple memory cells in adjacent NAND strings, sharing the same wordlines. Thus, the sector size of the latest NAND devices ranges from 512B to 4KB [45]. However, the amount of the transistors contained in a sector is related to the number of bits stored per memory cell. Flash memory devices are split into:

- **Single Level Cell (SLC)** devices, where the memory cell capacity is one bit of data.

- **Multi-Level Cell (MLC)** devices, where the memory cell capacity is two bits of data.

- **8LC** devices, where the memory cell capacity is three bits of data.

\(^1\)Information on NOR-based flash can be seen in [56].
1 NAND Flash memory = 512 K Blocks 1 Block = 32 Sectors 1 Sector = 4KB 1 B = 8 SLCs

Figure 3.4: Flash memory organisation: from a NAND device (left) to a Single Level memory Cell (right).

• 16LC devices, where the memory cell capacity is four bits of data.

In addition, a flash memory block is a group of sectors, as shown in Figure 3.4. The number of sectors per block is associated with the number of the wordlines passed through the NAND strings. For example, an SLC NAND device with 4KB as sector size will occupy 32,768 memory cells for each sector. Considering that this device comprises 32 wordlines per NAND string (32 sectors per block), that leads into 128 KB sized blocks and the overall capacity size is 64 GB.

3.3.2 Flash Memory Cell

Every flash memory cell is a floating gate transistor, that is either ‘‘switched on’’ or ‘‘switched off’’. The floating gate transistor is a Metal-Oxide Semiconductor field-effect transistor (MOSfet) with two layered gates, as illustrated in Figure 3.5.

Figure 3.5: Schematic representation of a floating gate memory cell (left) and the corresponding capacitive model (right). Diagram taken from [49].
The Control Gate (CG), surrounded on both sides by oxide, provides an isolation layer which guarantees charge retention for years. Whereas, the Floating Gate (FG) is used to modify the operation of the MOS transistor. Once the voltage applied on the floating gate is higher than the threshold voltage, electrons start flowing from source to drain, and the transistor is ‘‘switched on’’ to write “1”. The part in the bulk (substrate) below the floating gate, where the electrons flow from source to drain is called the conductive channel or tunnel oxide as shown in Figure 3.5. The length of the conductive channel is used in many microarchitectures and hardware devices to identify the underlying transistor technology.

### 3.3.3 NAND Flash

The NAND-based flash is a two dimensional array (NAND Array) comprised of NAND strings, as shown in Figure 3.6. A NAND string (Figure 3.6a) is a sequence of 32, 64, or 128 memory cells. The memory cells on the horizontal

![Diagram of NAND String and NAND Array](image)

Figure 3.6: NAND string (a) and NAND array (b). Diagram taken from [49].
axis are called wordlines (WLs), while the cells on the vertical axis are called bitlines (BLs). Besides the memory cells associated with a word line, the NAND string has two selection transistors (MSSL and MBSL), through which the NAND string is physically connected to a source line (SL) and a BL, respectively.

The WLs are connected to the gate of the memory cells, thereby enabling all the memory cells on this line to “switch on” or “switch off”. Whereas, the BLs are connected at the drain of the MBSL transistor of the vertical NAND strings and allow information to be written/read to/from the memory cells. The size of the WLs and BLs, along with the power of the voltage capable of driving a finite number of transistors, make flash vendors deal with the arising trade-offs, between energy efficiency and the overall capacity of flash memory.

3.3.4 DDR Controller - Data Caching

Following the advancements of the memory cache hierarchy in computer architecture, storage architects have integrated fast DDR memory next to the flash memory chips, to perform data caching as presented in Figure 3.1. Data caching includes the storage of data in DDR memory via a DDR controller, before writing data back to flash memory. In case of overwriting data in the flash memory, the Garbage Collection task is performed in the background.

3.4 PCI Express Subsystem

Peripheral Component Interconnect Express (PCIe) is a high-speed interconnect, that emerged as an extension to previous bus standards, such as PCI or PCI-x, and Accelerated Graphics Port (AGP). Various computer systems, such as enterprise servers, communication systems, and personal computers, have integrated PCIe buses due to the high I/O bandwidth.

PCIe is a bus standard based on point-to-point communication, through multiple serial links connecting either a peripheral device to the host or two PCIe endpoint devices [5]. Every serial link includes two bidirectional paths (signaling pairs/signals) between two devices, allowing a simultaneous data flow from one point to the other. That link is widely known as a lane, as shown in Figure 3.7.

In recent years, many versions of the PCIe standard have emerged, demonstrating a continuous increase in the maximum I/O bandwidth, as shown in Table 3.1. One PCIe link can comprise up to 32 parallel lanes, allowing two devices
Figure 3.7: PCIe bus composed of two lanes with a bidirectional pair of signals.

Table 3.1: The I/O bandwidth of different PCIe generations.

<table>
<thead>
<tr>
<th>PCIe version</th>
<th>Year Introduced</th>
<th>Throughput per lane</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe 1.0 (Gen 1)</td>
<td>2003</td>
<td>250 MB/s</td>
</tr>
<tr>
<td>PCIe 2.0 (Gen 2)</td>
<td>2007</td>
<td>500 MB/s</td>
</tr>
<tr>
<td>PCIe 3.0 (Gen 3)</td>
<td>2010</td>
<td>984.6 MB/s</td>
</tr>
<tr>
<td>PCIe 4.0 (Gen 4)</td>
<td>2017</td>
<td>1969 MB/s</td>
</tr>
</tbody>
</table>

to interact with up to 64 GB/s I/O rate (PCIe Gen 4 - x32 lanes). The performance of PCIe interconnect against other interconnects (e.g. PCI, USB), has driven multiple high performance computers to use it for interconnecting peripheral accelerators, such as GPUs and FPGAs. Contemporary computers include the main card (motherboard), on which every hardware component is integrated, including the processor, the main memory, and multiple PCIe slots. Figure 3.8 illustrates three types of PCIe bus slots on a (motherboard) of a personal computer (from top to bottom: x4, x16, x1). All the PCIe slot types are compatible; therefore a single lane PCIe device (PCIe x1) can be plugged in an x16 bus slot, and operate with the one lane capable bandwidth [49].

Chapter 6 presents two hardware platforms (Xilinx Zynq-7000 SoC and Xilinx Zynq UltraScale+ SoC) in which the proposed system is evaluated. Both systems use PCIe Gen 2 interconnect to communicate with the flash-based SSD.

Figure 3.8: Three PCIe slots (from top to bottom: PCIe x4, PCIe x16, PCIe x1).


3.4. PCI EXPRESS SUBSYSTEM

3.4.1 PCIe Tree Topology

Modern computers are capable of accommodating multiple PCIe devices. A reasonable question that arises is how computers keep track of every physically-attached PCIe peripheral. Current computer systems have addressed this question by keeping track of the devices and arranging them in a specific order. This task can be performed either by the Basic Input/Output System (BIOS) during the system initialisation, or by the OS during the “hot-plugged” probing process. Every PCIe device is registered in a tree structure, as shown in Figure 3.9. The PCIe tree structure includes three main components: a) the Root Complex (RC), b) the PCIe switch and c) the PCIe endpoint device (EP) [90].

The root of the tree is the RC component, and it operates as a bridge among the PCIe devices and host resources (i.e. the processor and main memory). The PCIe switch is an intermediate connector that allows multiple PCIe EPs to be connected to the RC, either directly or indirectly through another PCIe switch, as illustrated in Figure 3.9.

Figure 3.9: PCIe tree topology (The main components are: Root Complex, PCIe switches and PCIe endpoints).
3.4.2 PCIe Protocol

PCIe is a bus standard based on a packet-based layer protocol, consisting of three layers, the transaction layer, the data link layer, and the physical layer [5], as shown in Figure 3.10.

Figure 3.10a shows the communication between the host system (software driver runs on the processor) and a PCIe device, via the RC. In this case, the software driver, that runs on the processor, issues read/write requests to the endpoint device. The generated requests are passing through the PCIe layers, which transform the requests into PCIe packets, called as Transaction Layer Packets (TLPs). These packets contain the information about the data and the control flow of the generated transaction. Finally, the physical link transmits the TLPs before arriving at the endpoint device, in which they are decoded and executed.

On the other side, Figure 3.10b illustrates the PCIe layers, in the case of point-to-point communication between two PCIe devices through a PCIe switch. In this case, a PCIe device (Device A) generates the read/write requests, which are transmitted via the physical link in the form of TLPs and received by the receiver endpoint (Device B).

**Transaction Layer** The transaction layer is responsible for encoding and decoding incoming requests from the software driver or the device core into TLPs. At the transaction layer the TLP packet has information derived from the request regarding the data and the address in which the data will be read/written, as shown in the top part of Figure 3.11.

![Figure 3.10: The structure of PCIe layers.](image)
3.4. PCI EXPRESS SUBSYSTEM

Data Link Layer  Subsequently, the data link layer is in charge of the reliable fulfillment of a transaction. Therefore, the primary role of this layer concerns reliability. For example, the data link layer in the transmitter (Device A Figure 3.10b) receives the generated TLPs, and appends them with a unique packet identifier (Packet Id) and a cyclic redundancy check (CRC) code so that the transaction remains uncorrupted. Then, the TLPs are ready to enter into the physical layer, as shown in Figure 3.11.

In contrast, the data link layer in the receiver (Device B Figure 3.10b) accepts the received TLPs, acknowledges them and sends them to be decoded in the transaction layer. Whenever the transmitter performs a transaction, the data link layer monitors the progress of the transaction and issues an automatic replay, in case of failure. Consequently, PCIe obtains low error rate communication between two endpoints.

Physical Layer  The physical layer is at the bottom of the PCIe layered structure (Figure 3.10), and constitutes the bond between the digital and analogue part of a PCIe device. In particular, this layer monitors the utilisation of the lanes on the PCIe link, and aims to distribute the generated TLPs among the available physical lanes, resulting in high I/O transfer rate. For example, the second version of PCIe (Gen 2) bus achieves up to 5 Giga Transactions per second (GT/s), while the latest PCIe version (Gen 4) performs up to 16 GT/s [6]. Figure 3.11 depicts that the physical layer append the starting and ending points of the packet, to the TLPs.

3.4.3 PCIe Integration in Computer Systems

Legacy Computer Systems  Conventional personal computers comprise multiple hardware chips, including the Processor, the Memory and other intermediate hubs, such as the Southbridge-I/O Hub (IOH) and the Northbridge-Memory.
CHAPTER 3. NVME: A HARDWARE PERSPECTIVE

Controller Hub (MCH) in the x86 architecture [75] (Figure 3.12).

The Southbridge and Northbridge hubs are used in legacy computer systems (e.g. Intel Pentium 4 Processors) as two bridges that link multiple hardware resources. Furthermore, Figure 3.12 shows that a PCIe peripheral device can communicate with the main memory by traversing two hubs, the IOH and the MCH, respectively.

Contemporary Computer Systems Unlike conventional systems, computers that integrate modern processors, such as high-performance Intel Core i7 processors, have absorbed the Northbridge on the same chip with the processor, aiming for low latency transactions, and energy efficiency by avoiding moving out of the chip [75].

Figure 3.13 illustrates the high performing hardware units, such as the Graphics Processing Unit (GPU) and the latest PCIe (Gen 4) peripheral devices, which are packaged in the same silicon with the processor to sustain their maximum performance. On the other hand, the devices based on former PCIe versions, USB and SATA, operate at the same I/O rate with the intermediate interconnect (i.e. Direct Media Interface (DMI) and QuickPath Interconnect (QPI)), thereby remaining plugged on the I/O Hub Chip, as shown in Figure 3.13.

System on Chip (SoC) Following the trend of moving hardware units next to the processor, computer architects integrated accelerator devices on the same
die with the cores of the processor. These systems are widely known as Systems-on-Chip (SoCs), delivered either with an on-chip GPU or FPGA [46]. Figure 3.14 illustrates a SoC design with two separate packages on the same silicon die: the processing system and the programmable logic. The processing system integrates a GPU along with the cores of the processor, while the programmable logic includes an FPGA device. Xilinx and Intel are two of the FPGA vendors that have fabricated such systems (e.g. Xilinx Zynq UltraScale+ MPSoC and Intel Stratix 10 SoC).
Figure 3.14a presents the PCIe \textit{Root Complex} on the subsystem on a \textit{SoC}, as described above. The \textit{Root Complex} component resides in the \textit{processing system}, allowing any PCIe peripheral to be plugged in the system. On the contrary, Figure 3.14b depicts the PCIe \textit{Root Complex} component as an intellectual property (IP) core in the \textit{programmable logic}, allowing systems that do not have a PCIe Root Port on the processing area to communicate with PCIe peripherals. Chapter 6 will present the experimental setup in two Xilinx Arm-FPGA \textit{SoCs} configured with the PCIe subsystem to reside in either the \textit{processing system} or the \textit{programmable logic}.

### 3.5 Summary

This chapter provided a hardware insight to NVMe SSDs, comprising of various hardware blocks, such as the \textit{NVMe controller}, the \textit{flash memory}, and the \textit{DDR memory}. The \textit{NVMe controller} performs various operations, including: 1) establishment of host interface, 2) initialisation of FFS (i.e. Wear Leveling, Garbage Collection, Bad Block Management), 3) error correction of flash memory cells, 4) execution of NVMe commands, and 5) acknowledgment of executed NVMe commands. The \textit{flash memory} is arranged as a two dimensional array of memory cells, capable of storing multiple bits of data in the form of \textit{blocks}. While \textit{DDR memory} is used as a cache memory for in-storage data to improve storage performance.

Finally, this chapter presented the PCIe subsystem, which is the foundation of host and storage communication. Contemporary computer systems integrate the PCIe subsystem in a tree structure composed of three elements: a) the \textit{Root Complex (RC)}, b) the \textit{PCIe switches} and c) the \textit{PCIe endpoints}. The PCIe subsystem establishes a packet-based communication between two devices via three layers: the \textit{transaction layer}, the \textit{data link layer}, and the \textit{physical layer}.

The next chapter surveys the related work on memory and storage systems and introduces the concept of key-value store databases. It also familiarises the reader with the concept of “Near-Data Processing”, which is an important research area in data centres.
Chapter 4

Memory & Storage Systems: State-of-the-Art

4.1 Introduction

The previous two chapters discussed the NVMe system architecture from both software and hardware perspectives. This chapter aims to familiarise the reader with key-value store databases (Section 4.1.1) and the Near-Data Processing concept (Section 4.1.2), as both have been studied in literature [60, 78, 98, 126] to analyse novel architectural ideas. In addition, Section 4.2 presents related work on tackling the overhead of the OS storage software stack and how to enhance memory and flash architectures with Near-Data Processing functionality. The related work is classified into five groups, based on the level at which the proposed technique is applied. Section 4.3 presents two classification schemes of the aforementioned related work, aiming to provide the reader with a better understanding of each body of scientific work. Finally, Section 4.4 discusses how this thesis extends the existing literature.

4.1.1 Key-Value Store Database

The key-value store (KVS) databases store data as a group of values composed of multiple fields. Figure 4.1 illustrates the internals of a KVS database in which every group of values is associated with a unique key. Thus the structure of the KVS database is similar to a hash table data structure configured with a one to one hash mapping.
4.1.2 Near-Data Processing

In recent years, several researchers have identified that the traditional computing model that requires data to be moved into main memory to be processed by conventional processors is insufficient, as data movement at large scale impacts performance, energy efficiency, and reliability [22].

Thus, the aforementioned traditional computing paradigm is modified to add computational capability near data sources. This concept is widely known in the literature as *Near-Data Processing (NDP)* [22, 113], and aims to eliminate data movement by processing it at the most appropriate location in the hierarchy, either in caches, main memory, or persistent storage.

4.2 Related Work

There is a broad range of studies that propose either software/hardware-based tools to assist architects to deal with important decision trade-offs [27, 77, 127, 144], or novel architectural designs to mitigate the inefficiency in state-of-the-art flash storage systems [60, 78, 98, 95, 118, 126].

The following subsections classify the related work into five groups based on the level at which the proposed technique is applied. The classification groups in a top-down order from the application level to the architectural level are:

- The first level comprises related work at the **application level** of the host system in the form of *Application Programming Interfaces (APIs)*.

- The second group refers to work made in **virtualised or containerised environments**, in which multiple guest operating systems request access to storage devices.
4.2. RELATED WORK

- The third group of work conveys work in the OS level, aiming to optimise particular OS software stack layers, such as BIO, Driver.

- The fourth group presents work that modified firmware software.

- The final group contains the related work for novel architectural designs, that aim to reduce the I/O latency and increase performance.

4.2.1 Application Level I/O Storage Software Stack

The first group of related work has created application-specific I/O software stacks at the user level, aiming to reduce latency by adopting a light-weight stack model. Thus, many APIs have been exposed, enabling direct access to the NVMe from the user space [84, 95, 147].

NVMeDirect [84] and SPDK [147] aim both to relocate driver functionality from the OS kernel to user level, thereby bypassing completely the OS kernel. Consequently, overall latency is improved, as applications avoid the context switch overhead (performed by a Linux system call), and they do not have to traverse the Linux OS storage software stack.

SPDK is a framework introduced by Intel to provide users with profiling information for benchmarks targeting both local and remote NVMe devices via NVMeoF. SPDK demonstrates between 6x and 10x better performance than the traditional OS kernel, showing that the SPDK user driver uses fewer CPU cycles to submit and complete I/O requests. However, the asynchronous execution of I/O requests has no lock support, requiring each application thread to have exclusive access of the user level driver [147].

On the other hand, NVMeDirect shows up to 30% improvement in latency against the standard kernel system. NVMeDirect is implemented as a user-level library enhanced with driver level functionality (including operations and structures), thereby allowing applications not only to access the NVMe device directly but also to support dynamic queue management [84]. In the first version of NVMeDirect the system supported exclusively raw data access to the SSD, while the second version was enhanced with file system support [68].

In addition, Lee et al. [95] proposed the Application-Managed Flash (AMF) architecture, which exposed flash capability at the user level, instead of keeping it hidden through multiple intermediate software layers, such as the file system, the block I/O and the FFS. The AMF system introduced a new block I/O interface
and improved the performance of a standard file system by 80% compared with the baseline system. Furthermore, Seshadri et al. [126] proposed exposing the programmability of the storage device to the applications. The proposed system is named Willow and uses DRAM along-with FPGA-based memory controllers to emulate the NVM system and allow applications to leverage the FPGA-based emulation system, presenting up to 4x speedup against the baseline system. However, this study did not mention the effect of Wear-Leveling and Garbage Collection, which are the foundations of flash-based SSDs.

Recent work has also focused on addressing the storage inefficiency in the Android I/O stack, which restricts storage performance in smartphone devices [74, 73, 83, 114].

4.2.2 Virtualised and Containerised NVMe SSDs

NVMe storage technology has penetrated into the data center market, driving both academic and industrial research to evaluate storage performance in virtualised environments such as containers and virtual machines (VMs).

Bhimani et al. [24] considered a persistent storage scenario, in which multiple containerised applications in Docker containers [109] require storage access concurrently. This work showed that both application I/O bandwidth and PCIe utilisation could scale up with containers, and highlighted that a MySQL database showed up to 50% performance improvement against the baseline system. Likewise, both Yang et al. [148] and Kim et al. [85] examined the overhead introduced by the hypervisor layer, when VMs require access to an NVMe drive. The former work proposed H-NVMe, a hybrid framework that provides two VM I/O stack modes, the parallel mode, and the direct mode. The parallel mode allows multiple VMs to perform simultaneous I/O requests to the hypervisor, while the direct mode enables VMs to bypass the hypervisor layer and access directly the NVMe driver on the host system [148]. The latter work followed a similar approach, proposing each VM to be related with an independent I/O thread. Every I/O thread is responsible for performing I/O requests in a particular hardware queue exclusively without requiring any atomic locking method, thereby improving performance by up to 80% [85].

Finally, Jun and Shin [77] presented a scheduling algorithm that extracts profiling information for the FFS layer, when multiple virtual machines (VMs) are
accessing the same SSD. That information can be useful for arranging the execution of multiple VMs and securing performance isolation and fairness. The proposed algorithm replaced the in-storage scheduling algorithm, and was evaluated on the SSDsim [67] simulator with single-root I/O virtualisation (SR-IOV) [119] capability. The SR-IOV is a mechanism that allows multiple VMs to share a PCIe physical device without any intervention from the hypervisor layer.

### 4.2.3 OS-Level Optimisations

The third category of related work has focused on accelerating the I/O software stack at the kernel level. For example, Caulfield et al. [32] proposed to bypass the BIO layer and implemented a separate driver, as a way to increase performance. Lee et al. [94] presented a novel queue isolation scheme, considering the write interference and increasing the read performance in heavy read workloads. Bjørling et al. [26] redesigned the BIO layer in order to enable scalability and exploit the spare hardware capabilities by implemented multi-queue support, as discussed previously in Section 2.2.2. The parallel block I/O implementation has been embraced in the Linux kernel since version 3.16.

**Delegating OS Functionality to Accelerators** A part of this group of work has focused on utilising accelerator devices, such as FPGAs and GPUs, to tackle the OS overhead and enhance the computational power of the system using NVMe SSDs.

GPUs show an extra challenge in accelerating storage data processing, due to their programming model. Hence, HippogriffDB [97] was proposed to address this challenge, by implementing GPU-based components capable of transferring data transfers directly from an NVMe drive to GPU memory. HippogriffDB combines the direct PCIe-based peer-to-peer communication with GPU-based kernels, which perform compression/decompression allowing the overall system to scale up and process Terabytes of input data.

On the other hand, several companies focused on developing boards with FPGA-based NVMe Intellectual Property (IP) cores, capable of delivering I/O transfers directly from/to FPGAs [48, 69, 117]. Design Gateway implemented an FPGA-based IP that uses PLDA PCIe Gen3, and performs 2GBps for write and 3.2 GBps for read operations, while being portable on both Xilinx and Altera.
FPGA devices [48]. Intelliprop [69] designed an FPGA IP core for building powerful Storage accelerator PCIe cards (e.g. Sidewinder-100 PCIe NVMe Storage Controller). While many of these systems demonstrate near wire-speed performance, they have not reported the underlying challenges of the NVMe architecture.

4.2.4 SSD Firmware Modifications

The fourth category relates to interfering with firmware software. This software is an in-storage operating system used by storage vendors to perform device-specific tasks, such as wear-leveling or garbage collection.

Several researchers have identified that Flash File System (FFS) (discussed in Section 3.2.2), makes crucial decisions that can affect both storage performance and lifetime, whilst being unaware of the application characteristics [63, 36, 82, 118]. Therefore, the FFS functionality could perform more efficiently if it knew what type of application is accessing the flash drive. Such an example is the aforementioned AMF [95] system, that provides applications with flash functionality. The AMF architecture is prototyped on the BlueDBM system [78], which offloads in-storage flash architecture onto an FPGA and combines a network of the composed FPGA-flash nodes to aggregate multiple flash drives. Hence, the FPGA architecture can perform wear-leveling and garbage collection tasks to perform more efficiently. The BlueDBM system architecture sustains up to 2.7x higher bandwidth when using three nodes in contrast to the single node architecture.

On the contrary, Gu et al. [60] leveraged the cores of the processor in the storage device, and proposed a system named Biscuit, which installs a runtime system, as software running on the SSD Controller. Biscuit was capable of processing data on the controller, without requiring the costly transfers to the processors of the servers, achieving up to 15x average speedup against conventional host-SSD systems.

Finally, Ouyank et al. [118] exposed the hardware SSD channels to the user level, achieving almost 300% higher I/O bandwidth than the conventional system. The proposed Software-Defined Flash (SDF) architecture deploys FPGAs to accommodate the lightweight memory controller, which does not perform common flash operations, such as Wear-Leveling and Garbage Collection. However an erasure command is provided to applications which can erase blocks in the flash memory. The SDF system emerged as a Baidu’s project with the aim of rethinking
the overhead in both software and hardware, and creating an optimal software interface that allows applications to leverage flash technology.

### 4.2.5 Novel Memory & Flash Architectures

In recent years, the advancements of memory systems with the emergence of 3D-stacked memory architectures [102] (e.g. Hybrid Memory Cube (HMC)), along with the conjunction of flash memory with high-speed interconnect, motivated work to study how to exploit emerging storage technology to efficiently process data at the scale that big data and social networks are operating.

**Evaluated work on KVS Databases** A significant amount of related work proposed novelties on a key-value store (KVS) in-memory database scenario (e.g. memcached [51], Redis [9]) with the aim of scaling up performance, while handling the significantly higher flash latency compared with DRAM. KVS databases have become one of the central appliances for scalable web applications with several industrial deployments, such as Facebook, Twitter, Wikipedia, Github, etc.

Blott *et al.* [27] and Xu *et al.* [145] focused on how to sustain high I/O bandwidth and low latency on scalable memcached servers, by proposing hybrid memory architectures that combine conventional DRAM memory with serial-attached Flash memory. The former work was able to increase the KVS capacity to 40 Terabytes while performing at 80 Gbps I/O bandwidth [27]. Whereas, the latter work proposed the BlueCache system, which comprises a network of FPGAs, used to: 1) accelerate the KVS protocol and 2) cache the flash-based data [145]. BlueCache obtained at least 4x higher throughput and consumed 25x less power than a flash-based KVS software implementation. Likewise, Fukuda *et al.* [53] used a network interface card (NIC) with embedded FPGA to cache the data of memcached, as the network interface is the nearest place to the web server on which memcached is running. Their evaluation showed a latency improvement by an order of magnitude over the software implementation of memcached.

In addition, Lim *et al.* [98] attributed the lack of performance to the network stack, thereby proposing the Thin Servers with Smart Pipes (TSSP) architecture. TSSP architecture couples low-power cores with FPGA-based NIC and a memcached accelerator [33], which offload both network handling and caching on the FPGA. Thus, the proposed system achieves up to 16x power-performance improvement against current server baselines.
Finally, Facebook presented **memcache**, an architecture that enables memcached software to scale up and sustain the enormous volume of data that Facebook steers [116].

**Adopting Near-Data Processing**  Gao *et al.* [54] seized the opportunity of existing processing capability on 3D stacked memory technology and proposed an architecture that integrates multiple general simple in-order cores that can communicate either with other cores via a 2D mesh network-on-chip (NoC) or with the stacked memory via high-speed serial links. Their evaluation was implemented on **zsim** simulator and outperformed performance and energy over conventional approaches by up to 16x. As an extension, Gao and Kozyrakis [55] proposed the **Heterogeneous Reconfigurable Logic (HRL)** architecture, which combines general-purpose cores with an array of FPGAs. Similarly, the HRL system was evaluated on **zsim** under MapReduce, graph processing and deep neural network benchmarks, and showed that HRL achieves almost 92% of the peak anticipated performance on an ASIC while improving performance per Watt by 2x over FPGAs. Additionally, Choe *et al.* [35] simulated an in-storage processing system that trains machine learning models for a specific algorithm (i.e. the stochastic gradient descent); widely used in logistic regression and neural networks.

Lee *et al.* [93] proposed **ExtraV**, a framework composed of FPGAs and SSDs both attached to a **Coherent Accelerator Processor Interface (CAPI)** POWER8 processor with 20 cores and 4 GBs of memory. **ExtraV** used the FPGA to accommodate compression/decompression functionality and accelerate various applications, such as MapReduce PageRank and graph traversal, etc. **ExtraV** was compared with previous out-of-memory graph processing frameworks, showing up to 4.7x and 2.62x speedup for the **FlashGraph** and the **Llama** frameworks, respectively. The overall performance improvement was attributed to the compression functionality, which leads to reducing the buffer management overhead.

Likewise, Jun *et al.* [79] accelerated large scale (Terabyte) data sorting in storage. The proposed system coupled FPGA-based accelerators for **MergeSort** and a flash-based SSD with 1 TB capacity, outperformed by two times the power efficiency of previously stated **Joulesort** system, thereby showing that by utilising accelerators in fine granularity the overall performance can be bounded only by the flash storage bandwidth. Finally, IBM’s **Netezza** [130] is an example from
industry, which offloads data processing functionality, such as filtering operations, to a reconfigurable device near storage.

4.3 Taxonomy of Related Work

Beyond the study of related work, this thesis aims at providing further understanding on the approaches that researchers have followed for demonstrating novel ideas. The following sections present a taxonomy of related work based on two criteria. Section 4.3.1 groups work according to the platform used for experimental evaluation, while Section 4.3.2 distinguishes work based on the memory hierarchy level at which every work contributes.

4.3.1 Classification of the related work based on the hardware platform

Table 4.1 groups the related work into three categories based on the platform used to evaluate the experiments. The first group of works uses simulation \(^1\) while the second group of works uses emulation \(^2\). The last group shows studies that evaluate the proposed systems on real hardware devices.

<table>
<thead>
<tr>
<th>Platform</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator</td>
<td>[35], [54], [55]</td>
</tr>
<tr>
<td>Emulator</td>
<td>[32], [77], [85], [106], [126]</td>
</tr>
<tr>
<td>Hardware</td>
<td>[24], [27], [48], [53], [60], [68], [69], [78], [79], [84], [93], [95], [97], [98], [100], [116], [117], [118], [127], [130], [142], [144], [145], [147], [148]</td>
</tr>
</tbody>
</table>

\(^1\)A simulator is a system that enables a machine to imitate the behavior of a different system and allows developers to conduct conclusions on how would the system operate on real hardware. Simulation is not as accurate and fast as emulation or native running; however it allows system developers to test their modifications, without requiring to run on real hardware.

\(^2\)An emulator is an existing system that behaves similarly to the target system. Emulation is a faster method to test a system than simulation. In many cases, the FPGAs have been incorporated to emulate CPU micro-architectures and memory models, due to the reconfiguration ability.
4.3.2 Classification of the related work based on the interposing memory hierarchy level

Furthermore, Table 4.2 separates the related work into four categories, based on the memory hierarchy level, in which each work interposes. The first category shows work that proposes novel architectural modifications in memory architecture. The second category comprises work that enhances memory with in-memory processing power to adopt the NDP concept. The third category relates to work using storage devices, as a secondary storage media device to extend limited memory space and enhance the key-value store scenario on a higher scale. The last category introduces the NDP concept on a storage device, either by moving functionality on an FPGA next to storage or by utilising in-store processors at runtime.

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>[27], [53], [98]</td>
</tr>
<tr>
<td>In-Memory Processing</td>
<td>[54], [55]</td>
</tr>
<tr>
<td>Storage</td>
<td>[24], [68], [77], [79], [84], [93], [94], [95], [97], [100], [116], [118], [126], [127], [130], [142], [144], [145], [147], [148]</td>
</tr>
<tr>
<td>In-Storage Processing</td>
<td>[35], [60], [78]</td>
</tr>
</tbody>
</table>

4.4 Thesis Extensions to the Literature

This thesis extends the related work and contributes to addressing three main questions:

1. how to tackle the inefficiency of the Linux I/O software stack?
2. how to enable NVMe SSDs operating independently from the processor?
3. how to enhance conventional flash-based storage with NDP capability?

Despite prior work in Section 4.2.1 managed to bypass a significant part of the kernel’s software stack by offloading kernel’s functionality in the user level, additional user-level code is required to perform various functionality. This thesis, on the other hand, attempts to offload and accelerate the whole process onto the FPGA.
The related work in Section 4.2.2 and Section 4.2.4 is used to present research in virtualized environments and under the hood of SSD devices, respectively. Although this thesis does not cross with these two groups, it is necessary to understand them for providing a system with respect to them.

Consequently, the work mentioned in Section 4.2.3 concerns various optimisations in the OS level; along with the use of accelerator-based NVMe controllers for delegating OS functionality. In contrast, this thesis presents an insight to the overhead of the software stack and aims not only to accelerate storage performance but also to ease programmability for multiple applications via a novel API. The proposed API is implemented as a software library allowing applications to access NVMe SSDs without being aware of the underlying architecture or the storage specification.

Finally, the fifth group (Section 4.2.5) discusses work in memory and storage systems that have used experimental evaluation with a use case of key-value store or near-data processing. This thesis follows the experimental paradigm by evaluating the proposed system in a key-value store case scenario. In particular, Redis [9] is selected as it is an in-memory database widely used in Github and Twitter. Additionally, the thesis creates an opportunity for FPGA-based near-data processing by enabling the integration of further functionality (e.g. encryption, decryption, sorting, user-defined functions) directly into the storage data path, while bypassing completely the main processor. However, this aspect of the system belongs to the future work of the thesis.

4.5 Summary

This chapter familiarised the reader with key-value store databases and the near-data processing concept. In addition, it provided a classification of the related work based on two criteria, a) the platform used to evaluate the systems, and b) the level of the memory hierarchy at which the systems interfere. Finally, a new classification scheme was proposed, grouping related work into five categories: the application level I/O storage software stack, the virtualised and containerised NVMe SSDs, the OS level optimisations, the SSD firmware modifications and the novel memory & flash architectures.

The next chapter introduces four implemented systems which will be evaluated in Chapter 6 with regards to performance and power dissipation. These systems
include one software-based system (Software_raw) that bypasses part of the Linux storage software stack and three incremental versions of FastPath, an FPGA-based storage path.
Chapter 5

FastPath: A Direct NVMe FPGA-based Storage Path

5.1 Introduction

As mentioned in Section 1.3, the Linux OS storage software stack adds a significant performance overhead when accessing modern SSDs, using NVMe requests. Thus, bypassing or accelerating this stack is an opportunity towards achieving wire-speed performance.

This chapter introduces FastPath. FastPath accelerates access to NVMe SSDs through the following techniques: 1) bypassing the Linux software stack by designing and exposing an API to the user level that enables applications to place NVMe requests directly to the FPGA, and 2) implementing the BIO and driver logic (mentioned in Chapter 2) into the FPGA for acceleration, achieving near wire-speed performance.

Section 5.2 discusses the design objectives that were set during the design and implementation of the proposed system. Section 5.3 explains how the FastPath architecture can be accessed using both the FastPath_API (Section 5.4.1), and the Linux NVMe driver. Section 5.4 describes the FastPath programming model and the Application Programming Interface (API), that allow multiple applications to communicate with storage through the FastPath system. For simplicity, Section 5.4 refers to the latest version of the FastPath API. Section 5.5 presents FastPath as an NVMe I/O path that operates transparently with the conventional Linux NVMe path. Section 5.6 presents Software_raw; a software implementation that uses the ioctl system call exposed by the NVMe driver.
to perform asynchronous I/O request, while bypassing part of the Linux I/O storage software stack. Finally, Section 5.7 presents the FastPath architecture, which has been implemented through three design versions. The FastPath_v1 design offloads the submission and completion functionality into the FPGA. The FastPath_v2 design is an optimised design that discards the locks and utilises FPGA logic instead, while the most recent design (FastPath_v3) extends the second version to use more NVMe queues, to scale I/O performance.

5.1.1 FPGAs on Computer Systems

In recent years, many big corporations such as Baidu [118] and Microsoft [120] have adopted FPGAs in their data centres as FPGAs combine high performance and energy efficiency for particular type of workloads (e.g. loop unrolling, pipelining). Figure 5.1 shows how FPGAs are connected in current computer systems and highlights how an NVMe SSD device interfaces with them through the PCIe bus. The most widely used interconnect for FPGA connectivity in existing servers is PCIe. Figure 5.1a presents the block diagram of a server in which the FPGA is a PCIe-based peripheral for acceleration. Nevertheless, recent advancements in computer architecture show a trend that computer systems combine processors along with FPGA accelerators on the same silicon die for low power and low

![Diagram](attachment:diagram.png)

Figure 5.1: The overview of two systems that comprise FPGAs and NVMe SSDs. Figure 5.1a shows the FPGA connectivity on traditional systems via PCIe, while Figure 5.1b shows the connectivity on Processor-FPGA SoC via a Network-on-Chip interconnect.
5.2. FASTPATH OBJECTIVES

Latency communication. Figure 5.1b presents the block diagram of a Processor-FPGA SoC, in which the FPGA is connected with the processor via a network on chip (NoC) and the PCIe Root Complex resides in the FPGA. Thus the NVMe SSD is connected via the PCIe bus that interfaces through the FPGA. Note that the memory controller typically resides in the processor in traditional systems but is shared by both processor and FPGA in novel SoCs.

5.2 FastPath Objectives

FastPath has been designed with the following main objectives:

1. **Avoid data copying** which can impact the efficiency of the system [22]. FastPath accomplishes this objective, by allowing user applications to modify data directly in the DMA memory regions, via the FP\_DIRECT flag in FastPath API (Section 5.4.4).

2. **Increase performance** by completely bypassing the Linux kernel. FastPath achieves this objective by offloading both submission and completion paths on the FPGA, without any need to interact with the main processor in the system (Section 5.7).

3. **Preserve compatibility** by allowing the conventional Linux kernel-based system (baseline) to coexist with the FPGA FastPath system. FastPath is designed to operate in parallel with the kernel-based I/O system, without adding further overhead in the NVMe driver layer (Section 5.5).

4. **Support multi-threading**, thereby enabling concurrent applications to access the NVMe drive in isolation. FastPath API has been implemented as a thread-safe standard C library “libfnvme”, that interfaces with the FPGA-based FastPath architecture (Section 5.4.1) and allocates a separate logical block address space for each application.

5.3 FPGA Interface for Applications & Driver

FastPath is prototyped on two ARM-FPGA SoC boards (*Xilinx Zynq-7000 SoC and Xilinx Zynq UltraScale+ SoC*). Both systems require during OS boot, a
board-specific structure called the *device tree*, which is a representation of the physical memory addresses, at which I/O is mapped.

Thus, the FPGA fabric is physically associated with a part of the physical memory\(^1\), which can be accessed from user applications and kernel modules if it is memory mapped in an address region in virtual memory, as illustrated in Figure 5.2. User applications can map physical devices in the virtual memory address space by calling the standard Linux `mmap` system call and passing the file descriptor of the device as a parameter. Likewise, kernel modules map physical devices by calling the Linux kernel `ioremap` function, which accepts as parameters the physical address and the size of memory to map. The return value of both functions is a virtual address, that can be stored in a pointer.

During the initialisation of **FastPath** the NVMe driver communicates with the FPGA-based architecture via a memory-mapped interface. At first, the driver maps the physical address of a particular FPGA module, such as `FastPath_Submit`

\(^1\)The physical address in Xilinx Zynq-7000 SoC is \(0x53C0\_0000\), while in Xilinx Zynq Ultra-Scale+ SoC is \(0xA000\_0000\).

---

**Figure 5.2:** The memory mapping of FPGA-based modules into user level applications and NVMe driver.
(red arrow in Figure 5.2). Thereafter, the memory mapped registers in the FastPath_Submit module can be accessed and modified with a read and write operation to the pointer that stores the virtual address of the hardware module.

5.4 FastPath Programming Model

FastPath has been deliberately designed with raw performance in mind. Therefore, it has not implemented any filesystem underneath but focused on a high-performance direct read/write path to NVMe SSDs in raw form. This approach is most suited for applications that require disk persistence such as in-memory databases [84]. For simplicity, this section discusses the programmability of the latest version of FastPath. More detail is given in Appendix B, which presents an example of a program that accesses the NVMe SSD device through FastPath.

5.4.1 FastPath_API

FastPath has been designed with programmability and portability in mind. Therefore, the software component of the system has been developed as a thread-safe standard C library (“libfnvme”) that interfaces with user-level programs via the API shown in Table 5.1, and has been validated against multithreaded workloads. The library abstracts away implementation details from the user and exposes only a lightweight API that allows easy integration with existing applications safely and securely (e.g. only eight lines of code modified to run the fio benchmark with FastPath).

<table>
<thead>
<tr>
<th>API Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fastpath *fp= fastpath_alloc(int size);</code></td>
<td>Allocates disk space, maps a DMA buffer into the application’s virtual address space, and returns a <code>fastpath</code> struct.</td>
</tr>
<tr>
<td><code>void fastpath_write(fastpath *fp, void *buffer, int size, FP_FLAGS);</code></td>
<td>Sends an I/O write request.</td>
</tr>
<tr>
<td><code>void fastpath_read(fastpath *fp, void *buffer, int size, FP_FLAGS);</code></td>
<td>Sends an I/O read request.</td>
</tr>
<tr>
<td><code>void fastpath_polling(fastpath *fp);</code></td>
<td>Blocks until all requests are fulfilled.</td>
</tr>
<tr>
<td><code>void fastpath_free(fastpath *fp);</code></td>
<td>Releases the allocated disk space.</td>
</tr>
</tbody>
</table>

5.4.2 Disk Allocation and Release

Applications that require read/write access to the NVMe drive must explicitly request disk space. The `fastpath_alloc` function, shown in Table 5.1, accepts
as input parameter the size of the requested disk space, and returns a descriptor fp_id of type fastpath (shown in Listing 5.1).

Listing 5.1: The definition of the fastpath struct type

```c
typedef struct {
    char *dma_address;
    int fp_id;
    int block_size;
} fastpath;
```

The fastpath struct is composed of the following three fields:

1. **dma_address**: A pointer to the DMA address for direct read/write to the disk, avoiding any extra data copying.

2. **fp_id**: A unique incremental identifier of the current disk partition.

3. **block_size**: The block size of the requests. Initially, this field has the 4KB as default value but can be updated by the user before calling the FastPath read/write I/O functions. This way we can have a flexible block size per-request.

The fastpath_alloc function may fail either due to inability to allocate a logical partition or to communicate with the FastPath IP on the FPGA. In case of failure (denoted by a returned NULL), the application has to repeat the same call. Finally, the release of an allocated partition is performed by the fastpath_free function.

### 5.4.3 Memory Allocation

In contrast to the baseline system, FastPath enables zero copy of data during disk I/O, by mapping DMA buffers into the virtual address space of the applications. Every FastPath_Submit module (described later in Section 5.7.2), that resides in the FPGA, has an assigned DMA buffer. For write operations, the data to be written on disk are placed into the DMA buffer associated with the FastPath IP, and sent to the NVMe drive. Similarly, for read operations, the data fetched from NVMe disk are placed into the DMA buffers where applications can directly access it. In addition, the addresses of the DMA buffers are used as the Physical Region Pages (PRPs) during NVMe command creation.
5.5. **THE FASTPATH I/O PATH CONFIGURATION**

The DMA buffers are pre-allocated in the driver’s memory address space during the initialisation phase of the device. As shown in Figure 5.8 on page 95 (Step 1), when the application calls the `fastpath_alloc` function, the FastPath library internally performs an `mmap` system call (Step 2) to the Linux kernel to map the buffers into the application virtual memory (this is the only system call performed in FastPath). The invoked `mmap` function (Appendix A.5) is a custom implementation inside the NVMe driver that allows us to expose the DMA buffers to the FastPath library. However, this function is never exposed to the user and is called transparently by the FastPath library upon initialisation.

5.4.4 **NVMe Request Submission**

In FastPath, the submission of requests to the NVMe device is performed directly to the FPGA, bypassing the OS stack (Figure 5.8, Step 3) completely. As depicted in Figure 5.8 applications can submit requests directly to the FastPath controller in the FPGA, through the `fastpath_read`/`fastpath_write` functions (Table 5.1). These functions accept as arguments: 1) a pointer to the `fastpath` struct, 2) a pointer to the `buffer` where the user’s data is read/written from/to, 3) the size of the request, and 4) the `FP_FLAGS`.

Regarding the `FP_FLAGS`, FastPath supports direct or indirect calls (`FP_DIRECT` flag) and synchronous or asynchronous requests (`FP_BLOCKING` flag). These two flags can be combined resulting in four different combinations of I/O requests. If `FP_DIRECT` flag is set, the application can use the DMA buffers directly through the `dma_address` field of the `fastpath` struct. If the `FP_DIRECT` is not set, the FastPath library will internally perform a `memcpy` of the data pointed by `buffer` to the DMA buffers. The `FP_BLOCKING` flag indicates whether the application will block until all requests are completed. If set, the `fastpath_read`/`fastpath_write` functions will not return until all requests are served. Otherwise, they will return immediately leaving the programmer the ability to inquire about the successful request completion, by using the `fastpath_polling` function.

5.5 **The FastPath I/O path Configuration**

FastPath is implemented to coexist with the current Linux kernel I/O path, without adding further overhead. As mentioned in Section 2.3.1, the NVMe system architecture provides `namespaces` to allow multiple I/O paths to access
the NVMe device. Storage administrators can either obtain exclusive control or share the namespace between each other. In case of shared namespaces storage administrators can operate simultaneously with command atomicity [16]. The NVMe namespace sharing along with the I/O virtualization is the foundation for enhanced enterprise storage systems, such as Storage Area Network (SAN), virtualization and hyperconvergence use cases. Thus, the rest of this thesis proceeds with a shared namespace between the Linux I/O path and FastPath, as shown in Figure 5.3. However, this configuration can be altered in future. The following section presents the modifications required in the NVMe driver to enable FastPath running as an individual system.

5.5.1 Linux NVMe Driver Modification

The driver modifications presented below are executed during the probing phase when the NVMe driver allocates specific data structures to establish the communication between the main processor and the NVMe controller. The source code is presented in detail in Appendix A.

1. The number of the NVMe queue pairs is equal to the processor’s cores in the system, as mentioned in Section 2.2.3. Considering that the maximum allowed number of pairs in the NVMe specification is 64k, the
Linux NVMe driver is modified to allocate extra queue pairs for FastPath. Enabling FastPath to operate on separate queues from the main processor allows no congestion as no need for atomic accesses on the NVMe queues is required. Additionally, the final version of FastPath (FastPath_v3) enables multiple paths to use separate queues in parallel, to enable scalable performance.

2. **The physical region pages (PRPs)** are built as DMA memory regions allocated with a maximum size of 64 MB. Every FastPath submission queue uses one DMA memory region as PRP field in NVMe commands. The DMA physical address is stored in a register in the FastPath_Submit module via the memory mapped interface and can be reused, once the submission queue is idle.

3. **The submission queue address** is stored in a register in the FastPath_Submit module via the memory mapped interface. This is the submission base address used for submitting NVMe commands.

4. **The submission doorbell address** is stored in a register in the FastPath_Submit module via the memory mapped interface. This is the doorbell base address used for writing the new submission queue tail, thereby notifying the NVMe Controller of the new submitted command in the submission queue.

5. **The completion queue address** is stored in a register in the FastPath_Complete module via the memory mapped interface. This is the completion base address used for polling the completion queue entries.

### 5.6 Software raw: Bypassing Linux Storage Stack

The first implemented system in this work is a software-based system, called Software_raw, developed to explore the anticipated latency reduction, bypassing part of the Linux I/O storage software stack. Although, standard Linux OS does not recommend the use of bypassing paths, the NVMe Linux driver exposes device-specific input/output operations via system calls to user applications [43]. Applications can leverage those system calls and invoke driver-level functions, through the following steps:
1. Call the `open` system call to retrieve the file descriptor of the required block device.

2. Call the `ioctl` system call, which accepts as parameters: a) the file descriptor from the previous step, b) the request code according to the executed I/O operation and c) a struct type, that includes various input/output fields passed from/to the kernel side.

3. Check the returned value. The completion of `ioctl` system call (consider this is not the completion of the request) returns an integer value, which can be zero on success, or else a request-specific error code.

Figure 5.4 illustrates the lifetime of an I/O request in the Software_raw system, including submission (Steps 1-4), processing (Step 5) and completion (Step 6) time.

Figure 5.4: The I/O request path on the Software_raw NVMe system.
5.6.1 Submission

At the software layer, user applications submit requests (Step 1) directly to the NVMe driver, based on the aforementioned ioctl paradigm. In particular, applications running on the Software_raw system use the NVME_IOCTL_SUBMIT_IO request code and pass information about the number of transactions, along with the thread identifier (tid) of the application that submits the request. Consequently, the ioctl system call invokes the nvme_user_io function in driver source code (Step 1). This function injects user-level I/O requests into the BIO request queue (Step 2), whose role is to forward pending I/O requests to the NVMe driver for submission (Step 3). Afterwards, the driver starts submitting NVMe commands into the NVMe queues in memory (Step 4), which will be fetched and executed by the NVMe Controller, via the Root Complex component (Step 5).

5.6.2 Completion

The previously discussed submission system call (ioctl) informs the NVMe driver about a) how many transactions will be submitted on behalf of each application, and b) what the application process identifier (pid) should be notified after the successful completion of the requests. Beyond that, each application checks the returned value of the system call. If the system call succeeded, applications call the sigwait function, which suspends the execution of the current application and waits until the kernel or any other process sends a signal to this application. This method is called signals [21, 28] and is an inter-process communication method used on Unix, Unix-like and other POSIX-compliant OSs. Thus, the Software_raw system uses the method of signals, to send a notification to an application about successful completion, as soon as the interrupt handler in the driver is triggered (Step 6).

5.7 FastPath Architecture

Based on previous studies [144, 147], that attributed the software overhead to a) context switching, b) data copy (from kernel to user and back), c) interrupt handling and d) lock contention of shared resources such as request queue, the FastPath architecture is developed through three design versions, as described below:
1. **FastPath_v1** moves the submission and completion functionality, including the locks, from the NVMe driver to the FPGA logic.

2. **FastPath_v2** optimises the previous version, and removes all the locks by creating a **FastPath_Control** layer that accepts and distributes every user sent I/O request [133].

3. **FastPath_v3** quadruples the number of parallel submission and completion FastPaths used in the second version.

Please note that the disk’s block size is device specific; ranging from 512B to 8KBs in our case. The aforementioned systems currently support 4KB as modern OS and disk management software are 4KB sector aware. Nevertheless, it is anticipated in the future work to extend them to various block sizes. Finally, all hardware blocks in the FPGA have been designed in the Bluespec [20] hardware description language.

### 5.7.1 FastPath_v1 Design

The FastPath IP block in this version consists of the `axi_nvme_submit` and the `axi_nvme_complete` modules, as illustrated in Figure 5.5.

The `axi_nvme_submit` module is responsible for forming the NVMe commands and submitting them into the NVMe submission queue. The `axi_nvme_complete`, on the other hand, polls the respective completion queue and reports statistics regarding the latency of the completion path.

The design of both modules accommodates multiple user threads competing to get access to the FPGA-accelerated NVMe channels. This requirement is accomplished by integrating a hardware lock in each FPGA module, following the NVMe queue locks in the NVMe driver. The lock is a 32-bit wide memory mapped register that stores the thread identifier `tid` of the calling thread, through the FastPath API. The submission modules have different locks from the completion modules in order to remove congestion between the two paths. Furthermore, both hardware modules are configured with the corresponding base addresses of the submission and completion queues and doorbells during the initialisation phase of the device driver.

The `axi_nvme_submit` module (Figure 5.6) manipulates the memory for I/O requests in a 4KB block size, and performs the following actions:
5.7. FASTPATH ARCHITECTURE

Figure 5.5: The I/O request path on the FastPath_v1 NVMe system.

Figure 5.6: Design of axi_nvme_submit module.
• **Synthesises the NVMe commands**, based on the inputs of the `fastpath_read` and `fastpath_write` API functions (Synthesis phase in Figure 5.6). As discussed in Section 2.3.2, the NVMe command is a 512-bit wide field that contains information about the submitted operation (e.g. read/write), the DMA address of the data to be processed, the starting logical block address in the volume, and a command identifier.

• **Submits NVMe commands onto a specific NVMe queue stored in memory** (Submission phase in Figure 5.6). The tail of this circular queue is stored in a register, which is updated according to the actions of the NVMe driver. Whenever the tail reaches the depth size of the queue, it resets to the start.

• **Notifies the NVMe controller about the pending submission** (Doorbell phase in Figure 5.6). This is achieved by storing the new tail in the “doorbell” register, which is a memory mapped PCI register (unique for each submission queue).

The `axi_nvme_complete` module (Figure 5.7) is responsible for:

• Calculating the number of NVMe commands associated with the current I/O request.

![Figure 5.7: Design of axi_nvme_complete module.](image-url)
• Polling the completion queue until the submitted commands of a request have finished. Similarly to the submission queue, the completion queue is also a circular queue. The head of the completion queue is stored in a register, which is updated based on the new completion entries.

Finally, contrary to the conventional Linux kernel-based system (baseline), which is using an interrupt-driven mechanism to complete the I/O requests, the FastPath_v1 architecture implements a polling completion method for the two following reasons: a) keep the processor completely uninvolved in the I/O process, and b) avoid any traffic occurred by the interrupt controllers [146].

### 5.7.2 FastPath_v2 Design

The FastPath IP block in this version consists of three modules (FastPath_Control, FastPath_Submit, and FastPath_Complete) [133]. The FastPath_Control module receives I/O requests from the applications through the fastpath_read and fastpath_write API functions (Figure 5.8, Step 3) and forwards them to the FastPath_Submit module.

![Figure 5.8: The I/O request path on the FastPath_v2 NVMe system.](image-url)
The FastPath_Submit module is responsible for forming the NVMe commands and submitting them onto the NVMe submission queue, similar to the axi_nvme_submit module (Step 4). Subsequently, the FastPath_Complete polls the completion queue to signal the completion of a request as well as reports statistics to the FastPath API regarding the latency of the total request (Step 5), identical to the axi_nvme_complete module. The only difference between the two versions of modules (axi_nvme and FastPath) is that the latter has replaced the lock registers in the former version with the FastPath_Control module. Hence, the FastPath API in this version does not communicate directly with the FastPath_Submit and FastPath_Complete modules, but uses an upper level module (FastPath_Control) to collect the user requests.

The submission and completion queues reside in the main memory of the system (DRAM) and store NVMe commands and completion entries respectively. These queues are accessed by the NVMe controller of the SSD, and their corresponding base addresses and doorbells are assigned during the initialisation phase of the device driver.

FastPath_Control The FastPath_Control module, as shown in Figure 5.9, holds a request FIFO queue and serves two roles. The first is to accept NVMe
requests from the FastPath API and forward them to the 
\texttt{FastPath Submit} module. The second role is to bookkeep the number of NVMe commands that have to be completed to signal a completion message for a specific request (please note that the mapping between NVMe requests to NVMe commands is $1...N$ based on the configuration). The number of the NVMe commands in this version of \texttt{FastPath} is used to express the depth size of the queue allowing the depth exploration, as presented in Section 6.3.1. Every submitted request is forwarded by the \texttt{FastPath Control} module in the \texttt{Submit FIFO} of the \texttt{FastPath Submit} module, as shown in Figure 5.10. For every forwarded request, the \texttt{Command Accumulator} is updated to submit the desired number of commands. Accordingly, the number of pending commands is passed to the \texttt{NUM_CMDS_REG} in the \texttt{FastPath Complete} module, as shown in Figure 5.11.

\textbf{FastPath Submit} The \texttt{FastPath Submit} module manipulates the memory for I/O requests in a 4KB unit size, and performs the following actions:

- \textbf{Synthesises the NVMe commands} based on the arguments of the \texttt{fastpath read} and \texttt{fastpath write} API functions (Synthesis phase in Figure 5.10). An NVMe command is a 512-bit wide field that contains information about the submitted operation (e.g. read/write), the DMA

![Figure 5.10: Design of FastPath Submit module.](image-url)
address of the data to be processed, the starting logical block address in the volume, and a command identifier. The command identifier is used to indicate a data dependency between two commands. If the device supports NUMA-optimised NVMe drivers [80], the command identifier will be combined with the submission queue identifier, to distinguish the commands submitted by different queues.

- **Submits NVMe commands onto a specific NVMe queue stored in memory** (Submission phase in Figure 5.10). The tail of the depicted circular queue is stored in a register and is updated according to the actions of the NVMe driver. Whenever the tail reaches the depth size of the queue, it resets to the start.

- **Notifies the NVMe controller about the pending submission** (Doorbell phase in Figure 5.10). This is achieved by storing the new tail in the “doorbell” register, which is a memory mapped PCI register (unique for each submission queue).

**FastPath_Complete** The FastPath_Complete module (Figure 5.11) is responsible for:

![Figure 5.11: Design of FastPath_Complete module.](image-url)
5.7. FASTPATH ARCHITECTURE

- Polling the completion queue until the submitted commands of a request have been finished. Similarly to the submission queue, the completion queue is also circular. The head of the completion queue is stored in a register which is updated based on the new completion entries.

- Calculating and reporting time statistics.

Similar to FastPath_v1 architecture, the FastPath_v2 architecture adopts a polling completion method in contrast to the interrupt-driven baseline system.

5.7.3 FastPath_v3 Design

The final version of FastPath described in this thesis (FastPath_v3) extends FastPath_v2 architecture to submit concurrently NVMe commands in multiple NVMe queues. Figure 5.12 presents FastPath_v3 architecture, designed to accommodate four fast paths, due to the four parallel memory ports on the Xilinx Zynq SoC. The FastPath_Submit and FastPath_Complete logic of four paths is an exact replica of the FastPath_v2 architecture. Hence, during the

![Figure 5.12: The I/O request path on the FastPath_v3 NVMe system.](image-url)
initialisation phase of the system, multiple FastPath IP blocks on the FPGA are configured with the base addresses of the associated submission and completion queues. The main extension in the FastPath_v3 architecture resides in the FastPath_Control_v3 module, which is enhanced to schedule concurrent multiple requests for submission. At the current state of the system, the FastPath_Control_v3 module monitors the state (i.e. busy, idle) of the FastPath IP blocks and uses the round-robin scheduling to distribute requests to the appropriate FastPath IP blocks. Please note that the investigation of appropriate scheduling methods is beyond the scope of this thesis. The FastPath_v3 system is used in the next chapter to investigate performance scalability based on the number of FPGA-based parallel fast paths.

**FastPath_Control_v3** The FastPath_Control_v3 module, follows the same two roles as in the FastPath_v2 architecture: 1) to accept from the FastPath API NVMe requests and forward them for submission, and 2) to bookkeep the number of NVMe commands that have to be completed. In addition, the incoming requests are submitted in 4KB block size, as in the previously presented architectures. However, the main difference is that the FastPath_Control_v3 module can forward multiple I/O requests for submission to four FastPath IP blocks, as shown in Figure 5.13. The FastPath_Control_v3 module pops requests out of the request queue and schedules them, based on the incoming request size.

![Diagram](image.png)

Figure 5.13: Design of FastPath_Control_v3 module.
Thus, in case a request has 8KB size, the \texttt{FastPath\_Control\_v3} module will forward two parallel requests in the \textit{Submit FIFO} of two \texttt{FastPath\_Submit} modules, in contrast to \texttt{FastPath\_v2} architecture, in which requests were submitted and executed sequentially. For every forwarded request, the associated \texttt{Command Accumulator} is updated to submit the desired number of commands. Accordingly, the number of pending commands is passed to the \textit{NUM\_CMDS\_REG} in every \texttt{FastPath\_Complete} module (Figure 5.11). Considering that the smallest block size that the \texttt{FastPath} architecture supports is 4KB, the maximum request size that can be served simultaneously by the \texttt{FastPath\_Control\_v3} module is 16KB.

5.8 Summary

This chapter presented a holistic view of \texttt{FastPath}, an FPGA-based architecture that accesses NVMe SSDs directly without involving the main processor in the system. In particular, Section 5.2 discussed the objectives of the proposed system, while Section 5.4 described the programming model that user applications should follow if low latency is required. Section 5.5 presented the modifications in the Linux NVMe driver, so as to configure the \texttt{FastPath} architecture with the correct base addresses. Section 5.6 presented \texttt{Software\_raw}; a software-based system that allows user applications to bypass the \texttt{BIO} layer and submit commands via the NVMe driver. Finally, Section 5.7 presented three versions of \texttt{FastPath}, including:

1. \texttt{FastPath\_v1} which moves the submission and completion functionality, including the locks, from the NVMe driver to the FPGA logic.

2. \texttt{FastPath\_v2} that optimises the first version, and removes all the locks, by creating a \texttt{FastPath\_Control} layer that accepts and distributes every user I/O request.

3. \texttt{FastPath\_v3} that quadruples the number of parallel submission and completion FastPaths, used in the second version.

The next chapter presents the experimental results concerning \texttt{FastPath} versus the baseline system for two cases: a) asynchronous I/O requests generated by the \texttt{fio} benchmark, and b) \texttt{Redis} in-memory database application that uses the NVMe SSD for persistence.
Chapter 6

Evaluation & Results

6.1 Introduction

The previous chapter presented four implemented systems, including a software-based system (Software\textsubscript{raw}) that bypasses part of the Linux storage software stack (Section 5.6) and three incremental versions of \texttt{FastPath} (Section 5.7).

This chapter presents a comparative evaluation of \texttt{FastPath} versus the baseline Linux kernel I/O path to showcase efficiency, regarding performance and energy. Section 6.2 provides information about the hardware platforms, the benchmarks used to conduct the experimental setup, along with the metrics used for performance evaluation. Section 6.3 presents performance, power and area analysis by running the Flexible I/O tester benchmark (\texttt{fio}) [71] for both validation and evaluation on a Xilinx Zynq-7000 SoC device board. In addition, Section 6.4 presents performance, power and area analysis for \texttt{fio} on a Xilinx Zynq UltraScale+ MPSoC. Finally, Section 6.5 presents an in-memory database case study, in which the high-performing version of \texttt{FastPath} is evaluated against the baseline system to move data from Redis database [9] to NVMe SSD for persistence.

6.2 Experimental Methodology

To evaluate \texttt{FastPath} against the baseline Linux kernel I/O path and Software\textsubscript{raw}, an experimental methodology is followed concerning the hardware platforms (Section 6.2.1), the application benchmarks (Section 6.2.2), and the performance metrics (Section 6.2.3). The experiments presented in this chapter
Table 6.1: The experimental systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>The unmodified Linux system that uses the mainline kernel I/O path.</td>
</tr>
<tr>
<td>Software_raw</td>
<td>Bypasses the file system layer and calls directly the NVMe driver with</td>
</tr>
<tr>
<td></td>
<td>information about the I/O request.</td>
</tr>
<tr>
<td>FastPath_v1</td>
<td>Uses FastPath submission and completion paths with FPGA-based hardware</td>
</tr>
<tr>
<td></td>
<td>locks for both paths.</td>
</tr>
<tr>
<td>FastPath_v2</td>
<td>Introduces a FastPath controller that discards the FPGA-based hardware</td>
</tr>
<tr>
<td></td>
<td>locks of FastPath_v1.</td>
</tr>
<tr>
<td>FastPath_v3_p</td>
<td>Uses parallel fast paths similar to FastPath_v2, interfacing with</td>
</tr>
<tr>
<td></td>
<td>FastPath_Control_v3 module.</td>
</tr>
</tbody>
</table>

imply that all systems presented in Table 6.1 are evaluated on the same hardware platforms to provide a fair comparison. In addition, the given results are the arithmetic means of 15 iterations of each experiment, as this number is considered sufficient to indicate the standard deviation of the experimental measurements.

6.2.1 Hardware Platform

The results presented in this chapter are conducted on two Arm-FPGA SoCs, as illustrated in Figure 6.1.

**Xilinx Zynq-7000 SoC** FastPath was initially prototyped on a Xilinx SoC that consists of a dual-core Cortex A9 MPcore processor and an FPGA. Table 6.2, contains the detailed characteristics of the hardware system. The FPGA

Table 6.2: The Xilinx Zynq-7000 SoC characteristics.

<table>
<thead>
<tr>
<th>Processor Core</th>
<th>Dual-core Arm Cortex-A9 MPcore @667 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Device</td>
<td>Xilinx Zynq-7000 SoC (Device Name: Z-7035)</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB Instruction, 32 KB Data per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512 KB</td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>256 KB</td>
</tr>
<tr>
<td>External Memory</td>
<td>1 GB DDR3</td>
</tr>
<tr>
<td>DMA Channels</td>
<td>8 (4 dedicated to Programmable Logic)</td>
</tr>
<tr>
<td>PCI Express</td>
<td>Xilinx Root Complex IP operating at 125 MHz</td>
</tr>
<tr>
<td></td>
<td>(PCIe-Gen2 speeds, up to 8 lanes)</td>
</tr>
<tr>
<td>NVMe Storage</td>
<td>Samsung PM953 SSD 480 GB NVMe v1.1 (attached via U.2 connector)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux Kernel 4.4</td>
</tr>
</tbody>
</table>
Figure 6.1: The layout of a Processor-FPGA SoC.

accommodates both the FastPath architecture and the PCIe Root Complex on the programmable logic, as illustrated in Figure 6.1a. The PCIe Root Complex is a Xilinx Root Complex IP and determines the clock frequency of the overall hardware design to be 125 MHz. The NVMe SSD is attached to the SoC via a U.2 connector.

Xilinx Zynq UltraScale+ MPSoC  

FastPath is designed to be portable on different hardware platforms with FPGA devices. Therefore, FastPath is ported and validated in the Xilinx Zynq UltraScale+ MPSoC, which integrates on the same chip both the programmable logic (i.e. FPGA) and the processing system (i.e. quad-core Cortex A53 MPcore processor). Table 6.3 contains the detailed characteristics of the hardware system. Figure 6.1b shows that the PCIe Root

<table>
<thead>
<tr>
<th>Processor Core</th>
<th>Quad-core Arm Cortex-A53 MPCore @1.5 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Device</td>
<td>Xilinx Zynq UltraScale+ MPSoC (Device Name: ZU9EG)</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB Instruction, 32 KB Data per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1 MB</td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>256 KB</td>
</tr>
<tr>
<td>External Memory</td>
<td>4 GB DDR4</td>
</tr>
<tr>
<td>DMA Channels</td>
<td>8</td>
</tr>
<tr>
<td>PCI Express</td>
<td>PCIe-Gen2 speeds, 4 lanes (1 lane used)</td>
</tr>
<tr>
<td>NVMe Storage</td>
<td>Samsung PM953 SSD 480 GB NVMe v1.1 (attached via U.2 connector)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux Kernel 4.9</td>
</tr>
</tbody>
</table>
Complex module is now on the processing system, allowing the FastPath architecture to target higher clock frequency than it did on the Xilinx Zynq-7000 SoC. The maximum frequency that satisfied the timing constraints of FastPath on the Xilinx Zynq UltraScale+ MPSoC is 250 MHz. The experiments conducted on Xilinx Zynq UltraScale+ MPSoC use the same NVMe SSD device, which is also attached to the SoC via a U.2 connector.

6.2.2 Benchmark Applications

**Flexible I/O Tester**  The Flexible I/O tester (fio) [71] is a standard I/O benchmark in Linux system; used for measuring the I/O performance of storage infrastructures and evaluating the impact in the OS software stack and in the underlying storage architecture. The fio has been used in literature [24, 27, 77, 84, 85, 95, 144, 147, 148] to characterise performance and latency of several storage systems. Hence, the first experiment uses fio\(^1\) to evaluate both latency and I/O performance of every system shown in Table 6.1. Fio benchmark is configured to use the libaio Linux library to generate asynchronous I/O traffic (read and write requests) to the NVMe drive. To ensure fair comparisons, the time spent for generating I/O requests in the user space is not factored into the result metrics. In the baseline system, the I/O requests traverse the Linux software layers, whereas in the FastPath system the I/O requests are submitted in the FPGA-based FastPath architecture.

**Redis**  In addition, several studies [27, 53, 84, 98, 116, 145, 147] have evaluated novel storage systems on a key-value store (KVS) database scenario, as presented in Section 4.1.1. In particular, Redis\(^2\) [9] was selected, which is an in-memory No-SQL database widely used in Github and Twitter. Being a memory-based database, the system software may reach the memory capacity of the host system, thereby requiring key-value data to be moved back to the secondary storage device for persistence. Persistence is a significant feature in KVS databases (e.g. Redis [9], RocksDB [11]), because allows customers to recover their data without loss in the unfortunate case of a fatal system error. Nevertheless, in case persistence is not required by customers, it can be disabled. Redis has two methods for persistence: redis database file (RDB) and append-only file (AOF) [10].[1]

---

1. All the experiments carried out with fio use the fio-2.99 version.
2. All the experiments carried out with Redis use the Redis 4.0.8 version.
CHAPTER 6. EVALUATION & RESULTS

The RDB method captures snapshots of the data in the database at specific periods of time and saves them in a redis specific format (rdb). While the AOF method stores every write operation received by Redis server to replay these operations and restore the same state in case of a server startup. Both methods have advantages and disadvantages. The RDB file looks ideal for storing backup versions of the database because it is more compact than the AOF file, thereby allowing the Redis server to perform faster restarts. On the other hand, the AOF file is more robust than RDB and enables no data loss in case of power outage, as only the write operations are recorded [10]. Even in the unfortunate event of a power outage, the AOF method has a recovery mechanism to restore the database. Despite, AOF performing slower when loading big datasets in Redis server, it uses a rewrite background method to remove any duplicated records associated with the same key automatically.

Consequently, the AOF method was selected to compare FastPath performance with the conventional Linux kernel I/O path, when using a real application benchmark, such as the Yahoo! Cloud Serving Benchmark (YCSB) [42]. The YCSB benchmark includes five workloads configured with four types of operations (read, insert, update and scan), as presented in Table 6.4. The read operation is used to read data from the database. The insert operation is used to write data for the first time in the memory database, and consequently in the storage device for persistence. Unlike insert which writes data for the first time in the database, the update operation is used to modify the fields of an existing entry in the database. Finally, the scan operation can be considered a more complex operation than read, as it scans a specified number of entries in the database memory and returns all the data fields of a value associated with a particular key. The YCSB workloads are configured with various patterns of

<table>
<thead>
<tr>
<th>Workload</th>
<th>Operation</th>
<th>Application example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50% read</td>
<td>Session store recording recent actions in a user session.</td>
</tr>
<tr>
<td></td>
<td>50% update</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>95% read</td>
<td>Photo tagging; add a tag is an update, but most operations (95%) are for reading tags.</td>
</tr>
<tr>
<td></td>
<td>5% update</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>100% read</td>
<td>User profile cache, where profiles are constructed elsewhere.</td>
</tr>
<tr>
<td>D</td>
<td>95% read</td>
<td>User status updates; people want to read the latest statuses.</td>
</tr>
<tr>
<td></td>
<td>5% insert</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>95% scan</td>
<td>Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id).</td>
</tr>
<tr>
<td></td>
<td>5% insert</td>
<td></td>
</tr>
</tbody>
</table>
the aforementioned four operations, which can be used to characterise database performance, according to the application requirements.

6.2.3 Performance Metrics

The I/O performance of a storage system is evaluated by using three main metrics: the latency, the bandwidth, and the I/O rate in the form of I/O operations per second (IOPS) [112].

Latency  The remaining of the chapter will mention three types of latency:

1. The submission latency is the time spent for submitting an I/O request to the underlying system. In the baseline system the submission latency is the time for the Linux system call execution. While in the FastPath system the submission latency is the time for executing the FastPath API function calls (e.g. fastpath_write, fastpath_read).

2. The completion latency is the time taken to forward the submitted I/O request to the NVMe controller, process it, and complete.

3. The overall latency is the summary of submission and completion latencies, excluding the time spent for generating I/O requests in the user space in order to achieve more precise comparisons.

Bandwidth  The bandwidth is the rate that data are transferred from storage device to memory and from memory back to storage.

I/O Operations per Second (IOPS)  IOPS is used to express the throughput in terms of the number of block-sized I/O requests performed within a second and is strongly connected to both the latency and the block sizes of the system; small block sizes lead to higher number of I/O requests served by the system within a time window.

6.3 FIO - Results on Zynq 7000 SoC

This section presents a comparative evaluation of the five systems, presented earlier in Table 6.1. The following experiments are conducted using the fio [71]
benchmark to generate I/O requests aiming at an NVMe drive. *Fio* generates two types of I/O read/write operations: *sequential* and *random* operations. Both types aim to inform users about the performance characteristics of a storage system, indicating if this system meets the specified requirements of the user application.

6.3.1 Performance Analysis

**Latency** As shown in Figure 6.2, FastPath_v1 presents the highest mean latency among the five systems for single I/O requests. The reason is that FastPath_v1 uses FPGA-based registers accessed from user level as locks to ensure atomic operations between multiple applications sharing an NVMe queue via the FastPath system. The Software_raw system achieves slightly lower latency compared with the baseline system, as part of the software layers is bypassed. The lowest latency is achieved by FastPath_v2, which contains no FPGA-based lock and is the fastest way of accessing one NVMe queue. FastPath_v3 has higher latency than FastPath_v2, as it contains additional logic in the FastPath_Control_v3 module for distributing I/O requests to multiple NVMe queues.

However, experiments conducted on larger data sizes with many 4KB blocks show that the average block latency differs from single block requests. The reason is that the NVMe controller is capable of processing I/O operations in parallel, thereby lowering the overall latency. Table 6.5 presents the overall latency for

![Figure 6.2: The overall latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for single 4KB block size I/O request, on Zynq 7000 SoC. The latency is reported in microseconds (μs).](image)
6.3. FIO - RESULTS ON ZYNQ 7000 SOC

Table 6.5: The overall latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds ($\mu$s).

<table>
<thead>
<tr>
<th></th>
<th>seq-read</th>
<th>rand-read</th>
<th>seq-write</th>
<th>rand-write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>27,808</td>
<td>28,335</td>
<td>28,365</td>
<td>24,507</td>
</tr>
<tr>
<td>Software_raw</td>
<td>43,283</td>
<td>44,667</td>
<td>25,074</td>
<td>25,998</td>
</tr>
<tr>
<td>FastPath_v1</td>
<td>10,151</td>
<td>10,368</td>
<td>10,390</td>
<td>11,332</td>
</tr>
<tr>
<td>FastPath_v2</td>
<td>4,915</td>
<td>5,699</td>
<td>5,561</td>
<td>6,151</td>
</tr>
<tr>
<td>FastPath_v3</td>
<td>5,270</td>
<td>5,730</td>
<td>5,602</td>
<td>6,308</td>
</tr>
</tbody>
</table>

1MB$^3$ of data per configuration, while Figure 6.3 shows the average latency for a block. As shown in Figure 6.3, Software_raw performs the highest latency for read operations, due to the inefficient way of signaling the completion of the I/O requests in the user level. All the versions of FastPath reduce both the overall latency and the average block latency of read and write operations by up to 82% compared with the baseline system. The locking overhead in FastPath_v1 noticed on a single I/O request is negligible in this case, as locking is performed once before the submission of the first I/O request. Finally, both FastPath_v2 and FastPath_v3 present the lowest latency, showing that the modifications

---

$^3$Experiments were conducted with larger data sizes, but we did not notice any performance differences.

Figure 6.3: The average block latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds ($\mu$s).
Table 6.6: The average submission block latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds (µs).

<table>
<thead>
<tr>
<th></th>
<th>Submission Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>seq-read</td>
</tr>
<tr>
<td>Baseline</td>
<td>69.4</td>
</tr>
<tr>
<td>Software_raw</td>
<td>155.8</td>
</tr>
<tr>
<td>FastPath_v1</td>
<td>22.4</td>
</tr>
<tr>
<td>FastPath_v2</td>
<td>7.7</td>
</tr>
<tr>
<td>FastPath_v3</td>
<td>7.95</td>
</tr>
</tbody>
</table>

applied to the control module for integrating more fast paths introduce negligible overhead.

Submission Latency Table 6.6 presents the average submission block latency on the evaluated systems and shows that all the versions of FastPath outperform the other two systems. The reason is that FastPath optimises the submission path, by replacing the system calls (i.e. submit io in baseline, and ioctl in Software_raw) with a lightweight copy of the request into the FastPath logic.

Completion Latency Table 6.7 presents the average completion block latency of the evaluated systems. All FastPath systems outperform the software-based systems (baseline and Software_raw). The reduction of the completion latency ranges from 58% in FastPath_v1 (rand-write) up to 90% in FastPath_v2 (seq-read). As the number of I/O requests is equal for every system to ensure fairness, the difference in the completion latency among the evaluated systems in Table 6.7 is attributed to the following reasons.

Table 6.7: The average completion block latency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC. The latency is reported in microseconds (µs).

<table>
<thead>
<tr>
<th></th>
<th>Completion Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>seq-read</td>
</tr>
<tr>
<td>Baseline</td>
<td>106.55</td>
</tr>
<tr>
<td>Software_raw</td>
<td>166.27</td>
</tr>
<tr>
<td>FastPath_v1</td>
<td>34.47</td>
</tr>
<tr>
<td>FastPath_v2</td>
<td>16.72</td>
</tr>
<tr>
<td>FastPath_v3</td>
<td>18.24</td>
</tr>
</tbody>
</table>
• The completion latency of multiple I/O requests is strongly related to the submission rate of the requests, as the completion latency is the time spent to serve all the submitted I/O requests from the first till the last. Thus, the software-based systems, which use two 32-bit Arm cores to submit the I/O requests, are incapable of saturating the processing power of the NVMe controller. On the other hand, the FastPath systems achieve significantly lower latency, as the submission of new I/O requests can take up to three FPGA cycles (three pipeline stages in the FastPath_Submit module).

• FastPath bypasses the software layers in the Linux OS, in contrast to the baseline system that needs to propagate the completion from the NVMe driver to the BIO, and from BIO to the user-level application. Software_raw achieves the highest completion latency among the evaluated systems, as it requires the Linux kernel to notify the user-level applications via a signal. The signal method is similar to an interrupt and is not recommended for high performance and low latency operations because the transition from kernel mode to user mode is inefficient, due to the context switch.

Bandwidth  Figure 6.4 presents the I/O bandwidth for each evaluated system on 4KB block size (equal to the Linux kernel’s page size). Although the current state of FastPath supports 4KB block size, the architecture is designed to be parameterized for various sizes. Thus the FastPath design should not be the bottleneck in future experiments with various block sizes. As shown in Figure 6.4, all FastPath versions outperform both the baseline and Software_raw systems, due to the elimination of all the expensive system calls to the kernel I/O stack, as well as the acceleration of the creation and issue of the NVMe commands performed on the FPGA. The FastPath_v2 system achieves the highest throughput compared with other FastPath systems, ranging from 203 MBps (seq-read) to 162 MBps (rand-write).

The maximum reported I/O bandwidth of the particular SSD used in the experiments, is 1GBps for read operations and 800MBps for write operations [96]. However, these numbers have been measured under different conditions, as presented in Table 6.8. In addition, the maximum reported I/O bandwidth is performed when larger block sizes are used, ranging up to 128KB. The main advantage when using large block sizes is that it is equivalent to fewer user level system calls. Subsequently, the BIO layer generates fewer I/O requests which
CHAPTER 6. EVALUATION & RESULTS

Figure 6.4: The average I/O bandwidth on five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) for 1MB of data with 4KB block size, on Zynq 7000 SoC.

Table 6.8: The experimental conditions between the FastPath evaluation board and the specifications of the Samsung SM953 NVMe SSD.

<table>
<thead>
<tr>
<th>Condition</th>
<th>FastPath</th>
<th>SM953 White Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Architecture</td>
<td>Arm v7</td>
<td>x86_64</td>
</tr>
<tr>
<td>PCIe Interface (Bandwidth per Lane)</td>
<td>PCIe Gen2 (500 MB/s)</td>
<td>PCIe Gen3 (984.6 MB/s)</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux kernel 4.4</td>
<td>Windows kernel 6.5</td>
</tr>
<tr>
<td>I/O Benchmark</td>
<td>fio</td>
<td>IOMeter</td>
</tr>
</tbody>
</table>

pack data into larger chunks, capable of processing more data than the 4KB requests. However, FastPath at the current stage supports only 4KB block size, with future directions aiming at extending this to larger block sizes. Thus, it was decided to investigate potential performance improvement by undertaking a two-way exploration: in depth (FastPath_v2) and in space (FastPath_v3). The depth exploration aims to use the FastPath_v2 system to submit multiple NVMe commands per I/O request in the same queue. On the other hand, the space exploration aims to allocate multiple FastPath_v3 IP blocks, which submit multiple NVMe commands in parallel via different memory ports. The number of the FastPath_v3 IP blocks decided to be four, as this is the number of parallel memory ports on the Zynq 7000 SoC.

**Depth Exploration**  FastPath_v2 uses the block size parameter to indicate the depth of the FastPath IP block, in order to provide a fair comparison against
6.3. FIO - RESULTS ON ZYNQ 7000 SOC

Figure 6.5: The average I/O bandwidth on FastPath_v2 system for various depth sizes, ranging from 1 to 32, on Zynq 7000 SoC.

the baseline system. For example, a block size of 8KB generates two times fewer I/O requests from the user level API to the FastPath_Control module. So, it will indicate the FastPath IP block to operate with a depth size of two, instructing the FastPath_Submit module to submit two NVMe commands per I/O request fetched from FastPath_Control. Accordingly, a block size of 128KB indicates the FastPath_Submit module to submit 32 NVMe commands per I/O request. The depth exploration aims to increase the number of I/O operations processed by the NVMe controller, by allowing the FastPath_v2 architecture to submit multiple NVMe commands sequentially with a few FPGA cycles latency. Figure 6.5 shows that the I/O bandwidth is up to 3 times higher when submitting up to 32 sequential NVMe commands per I/O request.

Space Exploration  On the other hand, the FastPath_v3 uses the block size parameter to indicate the number of parallel FastPath IP blocks used to serve one I/O request. For example, a block size of 8KB indicates two FastPath_Submit modules to submit one NVMe command each in parallel, leading to the same computation to provide a fair comparison. Accordingly, a block size of 16KB indicates the FastPath_Control module to use the four available FastPath_Submit modules to serve one I/O request. Figure 6.6 shows that the I/O bandwidth is up to 2.3 times higher when using four FPGA-based fast paths. The results show that the I/O bandwidth increases along with the number of FPGA-based
fast paths, and it is estimated that if the system provided the FPGA design with enough parallel memory ports, then the number of **FastPath** IP blocks that could achieve the maximum theoretical bandwidth would be sixteen.

**I/O Operations per Second (IOPS)** Figure 6.7 presents the normalised IOPS of four **FastPath** versions against the baseline system on the Zynq 7000 SoC. As shown in Figure 6.7, **FastPath** outperforms both the baseline and Software_raw systems, with the latest version which operates on four NVMe queues,
achieving up to twelve times and nine times more IOPS for read and write operations respectively. The throughput of the write operations is related to the internal path in the storage device (from the NVMe controller to the flash memory). Consequently, the highest FastPath performance is achieved with the read operations because FastPath accelerates the storage path from the host computer to the NVMe controller.

Furthermore, Figure 6.8 presents the IOPS for all configurations with 4KB block size for every evaluated system. The number of the IOPS is correlated with the block size because the number of I/O operations depends on their size. The fulfilment of the same data size with larger block sizes can lead to a lower number of IOPS since a smaller number of I/O requests is created within a second.

The maximum theoretical throughput in Figure 6.8 is adjusted based on the specifications of the Samsung SM953 NVMe SSD [96] for the PCIe generation on the current testbed. As shown in Figure 6.8, FastPath reaches up to 73% of the peak theoretical throughput, while the baseline system performs up to 7% of the peak theoretical throughput. However, there are opportunities for further performance tuning that will enable FastPath to perform at wire-speed. Section 7.2.1 presents several design optimisations that can maximise the overall FastPath performance. Please note that both the metrics of IOPS and bandwidth are linked to the performance of the processor since the time spent to create I/O requests is factored into the results; as per the fio methodology of reporting performance numbers.
Scalability  As mentioned previously in Chapter 2, the NVMe system architecture has been designed to enable scalable performance by supporting up to 64k NVMe queues with one queue dedicated per processing core. FastPath is designed to satisfy this requirement. Thus, the latest version (FastPath_v3) allocates further NVMe queues, which are operated by the FastPath_Control_v3 module. Figure 6.9 presents the normalised I/O rate of one fast path, two fast paths, and four fast paths compared with the baseline Linux kernel path. The results show that performance scales up with the number of FPGA-based fast paths.

![Figure 6.9: The normalised IOPS of FastPath_v3 system against the baseline for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4) on Zynq 7000 SoC.](image)

### 6.3.2 Power Analysis

FPGA-based systems are usually associated with energy efficiency [40, 46, 65]. The tradeoff between performance and energy savings is very common in the data centers, as electrical power is an important regulator for the budget of data center facilities. Thus, this thesis provides a power analysis between the evaluated systems (i.e. Baseline, Software_raw, FastPath_v1, FastPath_v2 and FastPath_v3_1).
Table 6.9 presents the total on-chip power usage for every system on Zynq 7000 SoC, as extracted from Xilinx Power Estimator (XPE) [143] by using the vector less estimation mode. XPE is a software tool developed by Xilinx to provide hardware developers with accurate power analysis, modeling both device static power, design static power, and design dynamic power.

Energy is defined as the quantitative property that must be transferred to an object in order to perform a work. While, power is defined as the rate of energy capable of performing a work for a particular time interval. Thus the energy of transferring 1MB of data can be calculated based on the reported power in Table 6.9 and the total execution time in every system. Figure 6.10 presents a comparative evaluation of energy efficiency between the evaluated systems against the baseline system, that performs the standard Linux kernel I/O path. As shown in Figure 6.10 the random write operations are the least efficient in every system, as writing involves the wear-leveling and garbage collection processes. The Software_raw system seems to perform as energy efficient as the baseline system.

Table 6.9: The total on-chip power of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3) on Zynq 7000 SoC. The power is reported in Watt (W).

<table>
<thead>
<tr>
<th>System</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>3.483</td>
</tr>
<tr>
<td>Software_raw</td>
<td>3.483</td>
</tr>
<tr>
<td>FastPath_v1</td>
<td>3.744</td>
</tr>
<tr>
<td>FastPath_v2</td>
<td>3.842</td>
</tr>
<tr>
<td>FastPath_v3</td>
<td>4.155</td>
</tr>
</tbody>
</table>

Figure 6.10: The energy efficiency of five systems (Baseline, Software_raw, FastPath_v1, FastPath_v2, FastPath_v3 with 1, 2, 4 fast paths) against the baseline on Zynq 7000 SoC.
as the baseline, while \texttt{FastPath\_v1} achieves up to 2.5x higher efficiency. The second version of \texttt{FastPath} that removes the FPGA-based locking from the first version and performs the lowest latency (Figure 6.3), presents up to 5x more energy efficient performance. In addition, the \texttt{FastPath\_v3} system is evaluated for three configurations based on the number of FPGA-based fast paths. The system with one \texttt{FastPath} (\texttt{FastPath\_v3\_1}) performs higher power consumption compared with the \texttt{FastPath\_v2} system, as more FPGA logic is consumed in the \texttt{FastPath\_Control} module to enable the multiple fast path configurations. Nevertheless, the remaining systems (\texttt{FastPath\_v3\_2} & \texttt{FastPath\_v3\_4}) perform up to 7x and 10x higher efficiency, respectively. Finally, Figure 6.10 shows that the energy efficiency of the \texttt{FastPath} system architecture is increased by the number of fast paths accommodated in the FPGA device.

### 6.3.3 Area Analysis

One more important aspect of evaluating the FPGA-based architecture is the utilisation of the FPGA resources. Table 6.10 presents the utilisation of three basic types of FPGA resources such as the Look-Up Tables (LUTs) as logic and as memory, and the CLB registers, for every evaluated system [46]. Please note that it was deliberately decided the \texttt{FastPath} architecture not to occupy the maximum of the hardware resources, and deliver free space, including block memory, to application-level functions that require FPGA acceleration. Nevertheless, the source code of the \texttt{FastPath} architecture in Bluespec can be modified to convert the LUTs that used for distributed memory into block ram, if required.

Considering that the FPGA architecture for both the Baseline and Software raw systems (Table 6.10) accommodates only the Root Complex IP, it is

<table>
<thead>
<tr>
<th>System</th>
<th>LUTs as Logic</th>
<th>LUTs as Memory</th>
<th>CLB Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used</td>
<td>Available</td>
<td>Used</td>
</tr>
<tr>
<td>Baseline</td>
<td>20,793</td>
<td>171,900</td>
<td>887</td>
</tr>
<tr>
<td>Software_raw</td>
<td>20,793</td>
<td>171,900</td>
<td>887</td>
</tr>
<tr>
<td>FastPath_v1</td>
<td>33,518</td>
<td>171,900</td>
<td>4,595</td>
</tr>
<tr>
<td>FastPath_v2</td>
<td>42,443</td>
<td>171,900</td>
<td>24,931</td>
</tr>
<tr>
<td>FastPath_v3</td>
<td>60,549</td>
<td>171,900</td>
<td>48,375</td>
</tr>
</tbody>
</table>
6.3. **FIO - RESULTS ON ZYNQ 7000 SOC**

Rational to mention that the Root Complex IP block occupies 12.1% of the logic resources, 1.26% of memory LUTs and 5.82% of CLB registers on the FPGA. By subtracting this portions of each slice unit from the total percentage numbers reported in Table 6.10, the occupied resource units for **FastPath** are obtained as follows:

- The **FastPath v1** architecture presented in Section 5.7.1, occupies 7.4% of the logic resources for the control path and 5.27% of the LUTs used as memory for the internal FIFOs in the submission and completion modules. In addition, there is 5.30% increase in the slice registers, which are the units that the FPGA-based lock are mapped to.

- The **FastPath v2** architecture replaces the aforementioned lock registers with a control module, as presented in Section 5.7.2. The control module forwards requests to the submission module and waits until the completion module indicate the fulfilment of the executed requests. Then, it notifies the user-level “libfnvme” library for the outcome of the request. As mentioned in Table 6.10, the **FastPath v2** IP block employs 12.4% of the available LUTs as logic, 32.44% LUTs as memory, and 5.62% of CLB registers on the FPGA. The increase in memory LUTs is justified due to the FPGA request FIFO introduced in the FastPath_Control module.

- The **FastPath v3** architecture modifies the control module of **FastPath v2**, to support up to four replicated submission and completion units, and operate on four NVMe queues concurrently. This parallelism requires 23.10% of LUTs as logic, 67.44% of LUTs as memory, and 12.22% of the CLB registers. It would be anticipated the occupied resources to be at least four times higher, due to the three further replicas of submission and completion modules. However, as shown in Table 6.10 the resource units are increased up to 2.1x compared with **FastPath v2**. The reason is that the FPGA request FIFO in the control module is not replicated, and exceeds the size of the internal FIFOs in submission and completion modules.

### 6.3.4 Discussion

In summary, the evaluation of **FastPath** architecture has been undertaken in terms of performance analysis, power analysis, and area analysis. Table 6.11
Table 6.11: The normalised report of throughput, energy efficiency and the percentage of the unutilized LUTs for logic and memory. The comparison is between four systems (Software\textsubscript{raw}, FastPath\textsubscript{v1}, FastPath\textsubscript{v2}, FastPath\textsubscript{v3}) against the Baseline on Zynq 7000 SoC. The FastPath\textsubscript{v3} is configured with one, two and four fast paths (FastPath\textsubscript{v3}_1, FastPath\textsubscript{v3}_2, FastPath\textsubscript{v3}_4).

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput</th>
<th>Energy Efficiency</th>
<th>Spare LUTs as Logic (%)</th>
<th>Spare LUTs as Memory (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1</td>
<td>1</td>
<td>87.9</td>
<td>98.74</td>
</tr>
<tr>
<td>Software\textsubscript{raw}</td>
<td>0.9</td>
<td>0.94</td>
<td>87.9</td>
<td>98.74</td>
</tr>
<tr>
<td>FastPath\textsubscript{v1}</td>
<td>2.73</td>
<td>2.54</td>
<td>80.5</td>
<td>93.47</td>
</tr>
<tr>
<td>FastPath\textsubscript{v2}</td>
<td>5.6</td>
<td>5.12</td>
<td>75.31</td>
<td>64.59</td>
</tr>
<tr>
<td>FastPath\textsubscript{v3}_1</td>
<td>5</td>
<td>4.24</td>
<td>64.78</td>
<td>31.29</td>
</tr>
<tr>
<td>FastPath\textsubscript{v3}_2</td>
<td>8.28</td>
<td>6.94</td>
<td>64.78</td>
<td>31.29</td>
</tr>
<tr>
<td>FastPath\textsubscript{v3}_4</td>
<td>11.85</td>
<td>9.93</td>
<td>64.78</td>
<td>31.29</td>
</tr>
</tbody>
</table>

presents a comparison in terms of performance, power and resource occupancy between all the versions of FastPath and the baseline Linux system on the Zynq 7000 SoC. As noticed, the FastPath\textsubscript{v3} system configured with four fast paths achieves the highest performance and energy efficiency. In particular, FastPath\textsubscript{v3}_4 presents up to 11.85x higher throughput, 9.93x more energy efficiency, while utilising up to 36% and 69% of the available LUTs for logic and distributed memory, respectively.

Although FastPath has mitigated the performance overhead of the software-based systems (i.e. Baseline and Software\textsubscript{raw}), the architecture can be tuned to achieve the maximum theoretical performance on the system. Chapter 7 presents future hints for reducing the area consumption on the FPGA and increasing performance.

6.4 Portability to Modern SoC and Evaluation

In the aftermath of the evaluation of FastPath system on the Zynq 7000 SoC, it was decided to port the FastPath\textsubscript{v3} system into the latest Zynq UltraScale+ MPSoC, that couples an FPGA along with four Arm Cortex-A53 cores and integrates four times higher memory capacity. The characteristics of the hardware board are given in Section 6.2.1. However special emphasis should be given to the following:

- The Linux kernel supported on the Zynq UltraScale+ MPSoC is kernel 4.9\textsuperscript{4},

\textsuperscript{4}The 4.9 kernel in the Xilinx Github repository is compatible with the Vivado 2017.2 version.
which is a recent kernel, including a different implementation of the NVMe driver in Linux kernel 4.4 that was executed on Zynq 7000 SoC. Nevertheless, the modifications in the NVMe driver were exactly as described in Section 5.5. Thus, the implementation of FastPath can be considered adjustable to the version of the Linux kernel.

- The Root Complex IP does not reside in the FPGA logic, as it is not supported by the current Zynq UltraScale+ MPSoC (Device Name: ZU9EG). Hence, the Root Complex in the processing subsystem is used instead, which supports PCIe-Gen2 performance. The Linux kernel is built following the Xilinx guidelines for the Xilinx Petalinux software tool (Petalinux 2017.2). The current configuration in the petalinux tools could not support four lanes; therefore the experiments presented in this section are conducted with one PCIe lane.

- Due to the transition of the Root Complex IP from the FPGA to the processing subsystem, the FastPath architecture is not forced to run with the PCIe reference clock produced by Root Complex IP. The operational frequency of the FastPath architecture is 250 MHz.

### 6.4.1 Performance Analysis

**Latency** Table 6.12 presents the overall latency for 8MB of data per configuration, while Figure 6.11 shows the average block latency. FastPath_v3 achieves the lowest latency compared with the baseline system for various numbers of fast paths (FastPath_v3_1, FastPath_v3_2, FastPath_v3_4). However, the difference

<table>
<thead>
<tr>
<th></th>
<th>Overall Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>seq-read</td>
</tr>
<tr>
<td>FastPath_v3_1</td>
<td>20,229</td>
</tr>
<tr>
<td>FastPath_v3_2</td>
<td>20,154</td>
</tr>
<tr>
<td>FastPath_v3_4</td>
<td>20,152</td>
</tr>
</tbody>
</table>

Table 6.12: The overall latency of FastPath_v3 against the Baseline for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).
between the three configurations of FastPath_v3 system is not easy to distinguish, showing that the FastPath_v3 does not require four paths to saturate the PCIe lane. The reason is also that the 64-bit Arm Cortex-A53 processor has higher I/O performance than the 32-bit Arm Cortex-A9 on Zynq 7000 SoC.

**Bandwidth** Figure 6.12 presents the I/O bandwidth for each evaluated system on 4KB block size (equal to the Linux kernel page size). As shown in Figure 6.12:

Figure 6.11: The average block latency of FastPath_v3 system against the Baseline for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).
6.12, all FastPath_v3 configurations outperform the baseline system, due to the elimination of all the expensive system calls to the kernel I/O stack, as well as the acceleration of the creation and issue of the NVMe commands performed on the FPGA. The highest I/O bandwidth ranges from 397 MBps (seq-read) to 363 MBps (rand-write). Due to the usage of one PCIe-Gen2 lane, the maximum theoretical I/O bandwidth is 500 MBps. Figure 6.12 shows that FastPath can reach up to 80% of the available PCIe interconnect performance, in contrast to the baseline system that reaches up to 36%.

**I/O Operations per Second (IOPS)**  Figure 6.13 presents the IOPS for all configurations with 4KB block size for both FastPath_v3 and baseline systems. FastPath outperforms the baseline systems, by up to 2.27x and 1.86x times more IOPS for read and write operations respectively. As mentioned in the evaluation on the Zynq SoC (Section 6.3), the highest FastPath performance is shown for the read operations because the write operations are related to the internal path in the storage device (from the NVMe controller to the flash memory).

The maximum theoretical throughput in Figure 6.13 is adjusted based on the specifications of the Samsung SM953 NVMe SSD [96] for the PCIe generation and the number of lanes on the current testbed. As noticed in Section 6.3.1, FastPath

![Figure 6.13: The I/O operations per second of FastPath_v3 and Baseline systems for 8MB of data with 4KB block size on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).](image)
reaches up to 77\% of the maximum theoretical throughput. Also, the baseline system shows increased throughput up to 39\% of the maximum theoretical value due to the quad-core 64-bit processing system on the Zynq UltraScale+ MPSoC. Nevertheless, further performance tuning can be obtained by applying several design optimisations, as presented in Section 7.2.1.

### 6.4.2 Power Analysis

Table 6.13 presents the total on-chip power usage for every system on Zynq UltraScale+ MPSoC, as extracted from Xilinx Power Estimator (XPE) [143]. The XPE is a software tool developed by Xilinx to provide hardware developers with accurate power analysis, modeling both device static power, design static power, and design dynamic power.

As mentioned previously in Section 6.3.2, energy is deducted by using the reported power in Table 6.13 and the total execution time in every system.

Table 6.13: The total on-chip power FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC. The power is reported in Watt (W).

<table>
<thead>
<tr>
<th>System</th>
<th>Baseline</th>
<th>FastPath_v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power (W)</td>
<td>3.902</td>
<td>6.328</td>
</tr>
</tbody>
</table>

![Figure 6.14: The energy efficiency of FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC. The FastPath_v3 system is evaluated for three different configurations of fast paths: 1 path (FastPath_v3_1), 2 paths (FastPath_v3_2), and 4 paths (FastPath_v3_4).]
Figure 6.14 presents a comparative evaluation for energy efficiency between the FastPath_v3 and the baseline system, that performs the standard Linux kernel I/O path. As shown in Figure 6.14 the sequential write operations are the less efficient in every system, as writing involves the wear-leveling and garbage collection processes. The FastPath_v3 system is evaluated for three configurations based on the number of FPGA-based fast paths. All the configurations (FastPath_v3_1, FastPath_v3_2 and FastPath_v3_4) exceed the baseline system regarding energy efficiency by up to 1.4x (rand-read) and 1.15x (rand-write).

6.4.3 Area Analysis

Table 6.14 presents the utilisation of three basic types of FPGA units such as the LUTs as logic, the LUTs as memory and the CLB registers, for every evaluated system. Considering that the FPGA device is not used in the baseline system, the contents of the utilised FPGA units shown in Table 6.14 are zero. On the other hand, the FastPath_v3 architecture accommodates a control unit that distributes user requests, along with four replicated submission and completion units, which allow operating on four NVMe queues concurrently. This parallelism required 19.49% of LUTs as logic, 37.71% of LUTs as memory, and 8.45% of the CLB registers.

<table>
<thead>
<tr>
<th>FPGA Resources</th>
<th>Baseline Used</th>
<th>Baseline Available</th>
<th>FastPath_v3 Used</th>
<th>FastPath_v3 Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs as Logic</td>
<td>0</td>
<td>274,080</td>
<td>53,414</td>
<td>274,080</td>
</tr>
<tr>
<td>LUTs as Memory</td>
<td>0</td>
<td>144,000</td>
<td>54,301</td>
<td>144,000</td>
</tr>
<tr>
<td>CLB Registers</td>
<td>0</td>
<td>548,160</td>
<td>46,312</td>
<td>548,160</td>
</tr>
</tbody>
</table>

6.5 Redis I/O - Results on Zynq UltraScale+ MPSoC

This section presents the evaluation of FastPath system in a real application scenario: the Redis in-memory database [9]. The configuration of Redis settings is discussed in detail in Section 6.2.2. However, it is important to describe the
flow of the experiment. The Redis server is modified and compiled to use the FastPath library functions to write the append-only file into the NVMe SSD. Thus, FastPath is added as an option in the configuration file managed by the server administrator.

As soon as the Redis server is active; it initialises a connection with the client processes that require to communicate with the Redis server. Thence, the Redis server starts the service of the incoming requests from the client processes. The conducted experiment uses five workloads from the Yahoo! Cloud Serving Benchmark (YCSB) [42], as presented in Section 6.2.2. These workloads include four types of operations, such as read, insert, update and scan, and they follow different patterns of the aforementioned four operations, based on the application requirements.

### 6.5.1 Performance Analysis

Figure 6.15 presents the throughput at which the Redis server executes the operations issued by the YCSB clients. The Redis server is configured to write the append-only file using two systems, the baseline, and FastPath. Each workload presented in Figure 6.15 runs separately on the Zynq UltraScale+ MPSoC, in order to provide a fair evaluation of the Redis performance on both the baseline and FastPath systems. As shown in Figure 6.15, FastPath outperforms the baseline system in most of the workloads, including Workload A, Workload B, Workload D. The maximum increase in throughput is noticed in Workload A, and it is up to 15% higher.

### 6.5.2 Latency

The percentile is a statistical term used in memory and storage systems to represent the tail of the latency distribution. For example, the 90th percentile defines the latency threshold which separates the fastest 90% from the slowest 10% of the total samples. The range of values for percentile is used to indicate the distribution of the I/O rate for the requests served by the memory controller or the disk controller. The following paragraphs present the outgrowth latency in three percentiles (90%, 99%, 99.9%) on baseline and FastPath systems for four type of operations: read, insert, update and scan.
Read Operation  Figure 6.16 presents the read latency of the Redis server when four YCSB workloads (Workloads A-D) are used as clients to perform read request on the database. Figure 6.16 reports three snapshots of latency in the 90%, 99% and 99.9% of the completion. As shown in Figure 6.16, there is no significant difference between the baseline and FastPath systems for 90th and
99th percentiles, while for 99.9th percentile FastPath outperforms the baseline system for most of the workloads (A, B, D). In particular, FastPath achieves up to 60% tail latency reduction for Workload A, that is considered a heavy workload with both read and update operations [42]. On the other hand, on Workload C FastPath performs up to 33% higher tail latency than the baseline system. The reason is that this workload includes only read operations and the block size used to perform the operations is important. The baseline system can perform the read operations in the granularity of 512 Bytes, while the minimum supported block size in FastPath is 4KB. Thus, FastPath currently is not intended for block sizes smaller than 4KB, but in future, it is planned to evaluate performance with smaller block sizes.

Insert Operation The insert operation is used to write data for the first time in the memory database. As shown in Figure 6.17 FastPath outperforms the baseline system for insert operations. The 90th percentile of latency shows that FastPath achieves up to 13% lower latency than baseline, while in the 99th percentile FastPath shows a 12% increase on Workload D and 36% reduction on Workload E. The final percentile shows that FastPath outperforms the baseline systems, with latency reduction ranging from 4.5% (Workload D) up to 26% (Workload E).

![Figure 6.17](image.png)

Figure 6.17: The insert latency of YCSB benchmark workloads (D,E) when Redis server uses the FastPath_v3 and Baseline systems on Zynq UltraScale+ MPSoC.

Update Operation Unlike the insert operation, which writes data for the first time in the database, the update operation is used to modify the fields of an
existing entry in the database. As shown in Figure 6.18, FastPath outperforms the baseline system in every percentile of both Workload A and Workload B, and the maximum latency reduction is up to 48% in Workload A.

**Scan Operation** The scan operation can be considered a more complex operation than read, as it scans a specified number of entries in the database memory and returns all the data fields of a value associated with a particular key. As shown in Figure 6.19, the baseline and FastPath systems have similar latency for the scan operation, with FastPath reducing the latency up to 3%.
6.6 Summary

This chapter presented the results obtained by the evaluation of FastPath (Chapter 5) compared to the baseline system that used the standard Linux kernel I/O path to access the NVMe storage device. At first, all the incremental designs of FastPath were evaluated against the baseline system on the Xilinx Zynq 7000 SoC, by using the fio benchmark to generate I/O requests to the storage. The conducted experiments showed that FastPath achieves up to 82% lower latency, up to 12x higher throughput, and up to 10x more energy efficiency than the baseline system. In addition, FastPath showed scalable performance as the number of NVMe queues increase.

The most efficient design of the FastPath architecture was ported into a state-of-the-art Xilinx Zynq UltraScale+ MPSoC. This platform contains 64-bit Arm cores capable of delivering higher I/O performance than previous SoC. The results of fio showed that FastPath can achieve up to 75% lower latency, up to 2.2x higher throughput, and up to 1.4x higher energy efficiency than the baseline system.

Finally, as the Xilinx Zynq UltraScale+ MPSoC has higher memory capacity, FastPath was evaluated against the baseline system by using an in-memory database application such as Redis. Five workloads from the Yahoo! Cloud Serving Benchmark (YCSB) were used as clients in the database. The latency of both systems was analysed based on the percentile statistics provided by YCSB, and showed that FastPath can achieve up to 60% lower tail latency than the baseline system, while the throughput gains were up to 15%.
Chapter 7
Conclusion and Future Work

In recent years, storage research has pushed effort towards new technologies (e.g. NVMe SSDs) aiming to tackle the inefficiency of conventional hard disks. Although the adoption of PCIe interconnect has enabled contemporary disks to deliver higher performance, there are still performance bottlenecks to be tackled. Following novel computer systems which have enabled the network interface cards to operate independently from the processors, this thesis proposes FastPath, a direct FPGA-based storage path, that demonstrates the benefits of a storage element. As presented in the previous chapter, FastPath facilitates storage devices to achieve up to 82% low latency, 12x higher throughput and 10x higher energy efficiency compared with the storage path in the Linux kernel executed on Arm-FPGA SoC.

This chapter presents a summary of each chapter in this thesis in Section 7.1. Finally, Section 7.2 overviews the emerged opportunities to extend the current work in future.

7.1 Summary

This thesis can be summarised as follows:

Chapter 1 presented the importance of storage technology in many domains. Although recent advancements in storage technology have reduced the gap in latency between storage devices and main memory, there is still performance overhead due to the inefficiency of the software layers in the OS. This motivated this thesis to enable a direct storage path by employing an FPGA device, which can allow the storage device to be accessed with no interference by the processor.
Chapter 2 overviewed the Linux OS software layers, which are required to be traversed during the service of an I/O operation on an NVMe SSD. Chapter 3 presented the internal view of contemporary SSD disks and analysed the PCIe specification. Chapter 4 discussed recent approaches followed by work in the literature to tackle the inefficiency of the software layers for storage devices and enable applications to obtain lower latency and high performance. Chapter 4 also discussed how this thesis extends the related work, by deploying an FPGA-based system that allows applications to have direct access to an NVMe SSD via a lightweight API.

Chapter 5 presented a software implementation that bypasses part of the Linux storage stack, and analysed the FastPath architecture though three incremental versions. The four implemented systems were evaluated in Chapter 6 against the baseline storage path in the Linux kernel. Chapter 6 showed that FastPath can achieve up to 82% lower latency, up to 12x higher throughput, and up to 10x more energy efficiency for standard microbenchmark on an ARM-FPGA Zynq 7000 SoC. Finally, FastPath was ported on the Zynq UltraScale+ MPSoC, and evaluated for a real application, such as the Redis in-memory database, using requests from the Yahoo! Cloud Serving Benchmark (YCSB). The experimental evaluation showed that FastPath achieved up to 60% lower tail latency and 15% higher throughput than the baseline storage path in the Linux kernel.

7.2 Future Directions

The work presented in this thesis can be further extended to facilitate various functionality such as remote accessing and accelerating data in NVMe SSDs. This functionality can be essential for future computer systems to satisfy their requirements for low latency, high performance, and energy efficiency. The following sections present future directions for extending the current work. Section 7.2.1 points various optimisations and future functionality that can be added in the FastPath architecture. Section 7.2.2 discusses how FastPath can be extended to facilitate remote access to an NVMe SSD, while Section 7.2.3 describes how the current system can support accelerated processing of data residing in the storage device.
7.2. FUTURE DIRECTIONS

7.2.1 FastPath Optimisations

The design of FastPath pointed out several challenges that can be tackled in the future to enable the system to reach the maximum theoretical performance. The challenges are:

- The FastPath IP block was implemented in Bluespec, and the results in Chapter 6 showed that the overall performance could be scalable with the increase in the number of NVMe queues. However, designing an FPGA-based system that dynamically administers multiple NVMe queues is a complex task, due to the limitation of the hardware resources on the FPGA device. The current design presented in Chapter 5 can be optimised to use more NVMe queues without occupying many FPGA resources for replicating the FastPath_Submit and FastPath_Complete modules for each queue. Instead, the FastPath_Control module should monitor the status of each NVMe queue and decide the scheduling of the user I/O request among the available NVMe queues. In addition, the registers in the FastPath_Submit and FastPath_Complete modules should be configured as vector registers that store multiple values according to the queue that is operating. The vector registers in the FastPath_Submit module should be the: TAIL_REG, SQ_ADDR_REG, DB_ADDR_REG, and DMA_ADDR_REG registers. While, the vector registers in the FastPath_Complete module should be the: HEAD_REG, CQ_ADDR_REG, PHASE_REG, NUM_CMDS_REG, and completed_cmds registers.

- Currently FastPath supports 4KB block sizes and the NVMe commands contain the prp field as the pointer to the DMA buffers allocated for the I/O request. However, it can be extended to use smaller block sizes and the data can be structured in a scatter-gather list [16]. The prp field, in this case, stores the address of the first segment descriptor of the data.

- At the moment, FastPath supports raw I/O transactions and interferes with blocks at the logical block address space. Previous work [95] has identified the Flash File system (Section 3.2.2) as a low-level file system and the legacy file systems supported by the Linux OS as duplicated functionality that adds extra overhead. Thus, future work can explore the best match between file systems and FastPath.
CHAPTER 7. CONCLUSION AND FUTURE WORK

7.2.2 FastPath: An FPGA-based alternative to NVMeoF

Following the trend of converting hardware components into active elements, the current work can be extended to access storage devices remotely. The active elements can facilitate storage and network components (e.g. NVMe SSDs, NIC) to cooperate transparently while lowering the overall storage and network latency. Figure 7.1 presents a block diagram of future computer systems, composed of independent active elements (e.g. processing element (PE), network element (NE), storage element (SE)) which can communicate in a transparent manner.

The primary contribution of this thesis is the design and implementation of a SE by using an Arm-FPGA SoC and leveraging the programmable logic as the viable technology for accessing NVMe SSDs with no dependency on the processor cores in PE. One future direction to extend the current system is the integration of a communication interface (e.g. SE2NE) between the SE and NE. This interface is depicted in blue in Figure 7.1 and would allow remote disk operations, while bypassing the PE completely. Such a system can be used to investigate further the benefit of the proposed approach against existing NVMe-over-Fabrics solutions.

![Block Diagram](image)

Figure 7.1: The block diagram of future computer systems, which comprise multiple active elements for processors (PE), storage (SE), and network (NE).

7.2.3 Near-Data Processing

This work introduced FastPath, a low overhead and energy efficient storage path, which does not only improve the storage performance but also enables the implementation and pipelining of extra functionality (e.g. encryption, decryption,
sorting, user-defined functions). For example, the cryptography feature can be substantial for applications, such as cloud services, which have high-security requirements for data that reside in the storage devices.

**FastPath** submits direct operations to the disk drives, and the data are transferred from one edge to the other via DMA buffers. Consequently, FPGA-based acceleration functions could process the data in the DMA buffers, bypassing the processor completely and without any necessary copy to the memory, thereby making **FastPath** a system capable of delivering *near-data processing*. 
Bibliography


139


[57] Dror Goldenberg. Accelerating Ceph with Flash and HighSpeed Networks. Flash Memory Summit 2015.


[62] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan.


Appendix A

NVMe Driver Modifications

This chapter presents the source code modifications in the Linux NVMe driver that allow FastPath to operate independently from the Linux kernel storage I/O path, as mentioned in Section 5.5.1.

A.1 Allocate NVMe Queue Pairs for FastPath

Listing A.1 presents the code that modifies the number of the NVMe queue pairs. The NVME_NR_FAST_PATHS constant is configured to hold the value of the FastPath I/O queue pairs, while the nr_io_queues variable has the summary of the physical processing cores and the number of parallel FPGA-based fast paths.

Listing A.1: The initialisation of the NVMe I/O queue pairs.

```c
#define FPGA_ACCEL
#ifdef FPGA_ACCEL
    #define NVME_NR_FAST_PATHS 4
#else
    #define NVME_NR_FAST_PATHS 0
#endif

static int nvme_setup_io_queues(struct nvme_dev *dev)
{
    /* Skipped code */
    int nr_io_queues;
    nr_io_queues = num_online_cpus() + NVME_NR_FAST_PATHS;
}
```
A.2 Allocate DMA Regions for FastPath Blocks

Listing A.2 presents the code that allocates the DMA regions that are used from FastPath as PRP field within NVMe commands.

Listing A.2: The allocation of the NVMe DMA regions for the PRPs.

```c
#ifdef FPGA_ACCEL
    #define DMA_BUF_SIZE 67108864
#endif

static int nvme_probe(struct pci_dev *pdev, const struct pci_device_id *id)
{
    /* Skipped code */
    #ifdef FPGA_ACCEL
        dev->fpga_ptr = ioremap(XPAR_AXI_FPGA_BASEADDR, 131072);
        fastpath_dma_alloc_buffers(dev,&dev->ctrl);
    #endif
}

static int fastpath_dma_alloc_buffers(struct nvme_dev *dev, struct nvme_ctrl *ctrl)
{
    int i;
    #ifdef FPGA_ACCEL
        ctrl->fastpath_dma_cpu = (uint32_t **) kzalloc((NVME_NR_FAST_PATHS) * sizeof(uint32_t **), GFP_KERNEL);
        ctrl->fastpath_dma_bus = (dma_addr_t *) kzalloc((NVME_NR_FAST_PATHS) * sizeof(dma_addr_t *), GFP_KERNEL);
        for(i=0;i<NVME_NR_FAST_PATHS;i++) {
            /* Create and Pass FastPath DMA address */
            ctrl->fastpath_dma_cpu[i] = dma_alloc_coherent(dev->dev, DMA_BUF_SIZE, &ctrl->fastpath_dma_bus[i], GFP_KERNEL);
            *(dev->fpga_ptr + DMA_ADDR_REG +(i<<11)) = ctrl->fastpath_dma_bus[i];
        }
    #endif
}
```
A.3 Configure FastPath_Submit Module

Listing A.3 presents the code that stores the submission queue address in the memory mapped register that resides the FastPath_Submit module. While Listing A.4 presents the code that stores the submission doorbell address in the memory mapped register that resides the FastPath_Submit module.

Listing A.3: The allocation of the NVMe submission queues and configuration of the base address in the FastPath_Submit modules.

```c
static int nvme_alloc_sq_cmds(struct nvme_dev *dev, struct nvme_queue *nvmeq, int qid, int depth)
{
    #ifdef FPGA_ACCEL
    /* Pass FastPath Submission Queue address */
    int index;
    if(qid>num_possible_cpus()) {
        index = (qid-num_possible_cpus())-1;
        *(dev->fpga_ptr + SQ_ADDR_REG +(index<<11)) = nvmeq->sq_dma_addr;
    }
    #endif
    return 0;
}
```

Listing A.4: The configuration of the doorbell address in the FastPath_Submit modules.

```c
static struct nvme_queue *nvme_alloc_queue(struct nvme_dev *dev, int qid, int depth)
{
    /* Skipped code */
    /* Set the FastPath Submission Doorbell addresses on the FPGA */
    #ifdef FPGA_ACCEL
    int index = (qid-num_possible_cpus())-1;
    struct pci_dev *pdev = to_pci_dev(dev->dev);
    dma_addr_t sq_bar = (pci_resource_start(pdev, 0) + 4096 + qid*8);
    *(dev->fpga_ptr + (index<<11) + SQ_DB_ADDR_REG ) = sq_bar;
    #endif
}
```
A.4 Configure FastPath_Complete Module

Listing A.5 presents the code that stores the completion queue address in the memory mapped register that resides the FastPath_Complete module.

Listing A.5: The allocation of the NVMe completion queues and configuration of the base address in the FastPath_Complete modules.

```c
static struct nvme_queue *nvme_alloc_queue(struct nvme_dev *dev, int qid, int depth)
{
    /* Skipped code */
    /* Set the address of Completion Queue on the FPGA */
    #ifdef FPGA_ACCEL
        int index = (qid-num_possible_cpus())-1;
        if(qid>num_possible_cpus()){
            index = (qid-num_possible_cpus())-1;
            *(dev->fpga_ptr + CQ_ADDR_REG + ((index)<<10) + ((index+1)<<10)) = nvmeq->cq_dma_addr;
        }
    #endif
}
```

A.5 The nvme_mmap function

Listing A.6 presents the code that maps the PRP DMA addresses into virtual addresses via FastPath_API. The vm_pgoff variable is passed from the FastPath_API and it is the queue identifier of the associated FastPath queue that has already been reserved by FastPath_API function fastpath_alloc.

Listing A.6: The mmap implementation in the NVMe driver that maps the PRP DMA addresses into virtual addresses via FastPath_API.

```c
static int nvme_mmap(struct file *filp, struct vm_area_struct *vma)
{
    #ifdef FPGA_ACCEL
        unsigned long size;
        struct nvme_ctrl *ctrl = filp->private_data;
        int ret = -1;
```
size = vma->vm_end - vma->vm_start;

if(vma->vm_pgoff==0) {
    vma->vm_pgoff = 0;
    ret = dma_mmap_coherent(get_device(ctrl->dev), vma,
                            ctrl->fastpath_dma_cpu[0], ctrl->fastpath_dma_bus[0], size);
} else if(vma->vm_pgoff==1) {
    vma->vm_pgoff = 0;
    ret = dma_mmap_coherent(get_device(ctrl->dev), vma,
                            ctrl->fastpath_dma_cpu[1], ctrl->fastpath_dma_bus[1], size);
} else if(vma->vm_pgoff==2) {
    vma->vm_pgoff = 0;
    ret = dma_mmap_coherent(get_device(ctrl->dev), vma,
                            ctrl->fastpath_dma_cpu[2], ctrl->fastpath_dma_bus[2], size);
} else if(vma->vm_pgoff==3) {
    vma->vm_pgoff = 0;
    ret = dma_mmap_coherent(get_device(ctrl->dev), vma,
                            ctrl->fastpath_dma_cpu[3], ctrl->fastpath_dma_bus[3], size);
}
#endif

return 0;
}
Appendix B

Example of Using the FastPath_API

This chapter presents an example of a user-level program in the C programming language to facilitate applications that require to obtain direct access to the NVMe SSD via the FastPath_API functions. The functions mentioned in this chapter concern the FastPath_v2 and later versions. The following sections present the FastPath_API functions in the order that the user-level program is supposed to use them.

B.1 Allocate a FastPath Block

At first, the user program calls the fastpath_alloc function, as shown in Listing B.1. This function informs the FastPath_Control module for the size of the transaction. If there is any FastPath submission block free, this function will reserve it and return the identifier (fp->ctx) to the “libfnvme” library, otherwise a NULL value will be returned and the program has to repeat this process until there will be a free FastPath submission block.

Listing B.1: The allocation of a FastPath block.

```c
fastpath *fp = NULL;
int size = 4096;

while(fp==NULL)
    fp = fastpath_alloc(size);
```
B.2 Use the DMA regions for Direct Access

In case of a write operation, the user-level program can use the DMA pointer returned in the `fastpath` structure (`fp->dma_addr`) to move data to the NVMe blocks, without requiring any intermediate copies that can reduce the overall performance. Listing B.2 shows how the program can modify and write data in the DMA region that will be transferred later to the disk. On the other hand, in the case of a read operation, the program can access the returned data in the DMA region after the completion of the I/O request.

Listing B.2: The user program uses the DMA regions for the I/O transaction via FastPath.

```c
int *dma_addr=NULL;
dma_addr = fp->dma_addr;

for(i=0; i<size/4; i++) {
    if(write==1)
        dma_addr[i] = 2*i;
}
```

B.3 Submission of an I/O request

The program can submit a write/read request to the NVMe SSD by using the `fastpath_write`/`fastpath_read` functions provided by the FastPath_API. Consequently, these functions create I/O requests to the `FastPath_Control` module that resides in the FPGA. The program can use the available `FP_Flags` to indicate information about the I/O request. For example, if the program uses the DMA pointer returned by the `fastpath_alloc` function, then the `FP_DIRECT` is required to indicate the “libfnvme” library to process directly in the submission of the request. Otherwise, the library will map the user data into the DMA address and then move to the submission. In addition Listing B.3 shows how a program submits a direct non-blocking request via FastPath, while Listing B.4 makes use of both flags to indicate the submission of a direct and blocking request. Finally, the FastPath_API provides a timing function, called `gettime_fp` which can be used by the program to calculate a time interval.
B.4. RELEASE THE FASTPATH BLOCK

Listing B.3: The submission of one non-blocking and direct I/O request to the Fast-Path Control module.

```c
start_time = gettime_fp();
if(write==1)
    fastpath_write(fp,dma_addr,size,FP_DIRECT);
else
    fastpath_read(fp,dma_addr,size,FP_DIRECT);

// Blocking call until the I/O request is completed
fastpath_polling(fp);
stop_time = gettime_fp();
```

Listing B.4: The submission of one blocking and direct I/O request to the Fast-Path Control module.

```c
start_time = gettime_fp();
if(write==1)
    fastpath_write(fp,dma_addr,size,FP_DIRECT|FP_BLOCKING);
else
    fastpath_read(fp,dma_addr,size,FP_DIRECT|FP_BLOCKING);
stop_time = gettime_fp();
```

B.4 Release the FastPath Block

The final step required by the FastPath programming model is the call of the `fastpath_release` function, as shown in Listing B.5. This function notifies the FastPath Control module that the program does not require any further I/O requests, thereby releasing the FastPath Block to be used by another user-level application.

Listing B.5: The allocation of a FastPath block.

```c
fastpath_free(fp);
```