FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Study and Implementation of Processing Offload Techniques for Virtualized Systems

Ricardo André Miranda Pimenta Castro

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Canas Ferreira

Co-Supervisor: João Monteiro

1 de outubro de 2022

Abstract

Testing is an integral part of project development and must be easy to manage to achieve timely software development. Today, the use of hardware development racks limits developers because their physical unity means they can exclusively be used for one purpose at a time. Virtualization can overcome this limitation by giving developers the freedom of availability. However, if the architectures of the target and host processors do not match, virtualization cannot be applied, and the system resorts to full emulation, which involves binary translation that adds significant overhead to the system, resulting in a loss of performance.

This dissertation aims to achieve cross-architecture virtualization of an ARM guest machine on an x86-64 host computer to mitigate performance degradation on different guest and host architectures. On the hardware side, the solution uses a hardware accelerator with the same architecture as the guest image to provide the virtualization functions of the hypervisor installed on it, which is connected to an x86-64 host via a high-speed communication bus. On the software side, the solution uses a software emulator that can leverage virtualization extensions and an efficient data mechanism for communication between the host and the hardware accelerator.

The developed solution is validated by confirming the displayed outputs and using CPU benchmarks to evaluate performance.

Keywords: KVM, PCIe, Computational Offloading, QEMU, Hardware Accelerator, Embedded Systems

Resumo

A testagem é uma parte integral no desenvolvimento de um projeto e deve ser de fácil manuseamento para atingir um desenvolvimento de software oportuno. Hoje em dia, o uso de *hardware racks* limita os desenvolvedores devido ao seu carácter físico, que condiciona o seu uso a uma pessoa de cada vez. A virtualização consegue ultrapassar estas limitação ao dar aos desenvolvedores liberdade em relação à disponibilidade de *hardware racks*. No entanto, se as arquiteturas do processador do anfitrião e alvo não forem semelhantes, a virtualização não pode ser aplicada e o sistema irá recorrer a *full emulation*, que envolve translação binária o que implica um aumento significativo do *overhead* do sistema, resultando numa perda de performance.

Esta dissertação visa alcançar uma virtualização *cross-architecture* de uma máquina alvo, com uma arquitetura ARM, num computador anfitrião, com uma arquiterura x86_64, para mitigar degradações de performance em anfitriões e alvos com arquiteturas diferentes. Do ponto de vista do *hardware*, a solução utiliza um acelerador de *hardware* com a mesma arquitetura que a imagem alvo para fornecer os funções de virtualização do *hypervisor* nele instalado, que é conectado ao anfitrião x86_64 através de um *bus* de comunicação de alta velocidade. Do ponto de vista do *software*, a solução usa um emulador de *software* que consegue aproveitar as extensões de virtualização e ainda usa um mecanismo de dados eficiente para a comunicação entre o anfitrião e o acelerador de *hardware*.

A solução desenvolvida é validada através da confirmação dos *outputs* exibidos e CPU *benchmarks* são usados para avaliações de performance.

Palavras-chave: KVM, PCIe, Computational Offloading, QEMU, Hardware Accelerator, Embedded Systems

Acknowledgements

Before anything else, I would like to express my gratitude to the Faculdade de Engenharia da Universidade do Porto (FEUP) and its teachers for their remarkable ability to impart the necessary knowledge and life lessons transmitted that encouraged me to envision the future I want. Individually, I would like to thank my dissertations supervisors, Professor João Canas Ferreira and Engineer João Monteiro, for their guidance, availability and cross-examination that led to the improvement of the solutions proposed.

Secondly, I would like to thank Critical TechWorks (CTW) for the warm welcome during the dissertation. Furthermore, I would like to thank the team members who assisted and encouraged me during the work development.

I would also like to express my sincere thanks to my girlfriend and wife-to-be, Cláudia Martins, for being my inspiration and moral support, never allowing me to give up and face the obstacles that arose. Therefore, thank you for your love that helped me be a better human being and made my day-to-day much more enjoyable.

To my father, that taught me life lessons that sparked my interest in engineering and for providing me with the freedom I needed to mature.

To my mother, that is always in my heart despite not always being by my side physically. Thank you for always supporting my decision and guiding me to a better future.

To my grandfather, whose fishing trips nourished me with patience which helped me during my academic journey and also for sparking my interest in electronics.

To my grandmother, a word of warm appreciation for taking care of me during my childhood and instructing me with household chores.

To my friends, thank you for the memories created and shared.

Finally, to everyone not mentioned here but who is an integral part of my life, thank you for the support you have given me until now.

Ricardo Castro

"To doubt everything, or, to believe everything, are two equally convenient solutions; both dispense with the necessity of reflection"

Henri Poincaré, "Science and Hypothesis"

Conteúdo

1 Introduction **1**

1.1 Context . . . 1

1.2 Motivation . . . 2

1.3 Objectives . . . 2

1.4 Dissertation Structure . . . 2

2 Background **5**

2.1 Virtualization . . . 5

2.1.1 Protection Rings . . . 6

2.1.2 Types of Virtualization . . . 6

2.2 Hypervisors . . . 8

2.3 Emulation . . . 9

2.4 QEMU . . . 10

2.5 KVM . . . 10

2.6 QEMU-KVM . . . 11

2.7 Hardware Accelerators . . . 11

2.8 Computation Offloading . . . 12

3 State of the Art **13**

3.1 Hardware Accelerator Applications . . . 13

3.1.1 Hardware Accelerators for General Purpose Computing . . . 13

3.1.2 Hardware Accelerators for Bioinformatics . . . 14

3.1.3 Hardware Accelerators for Robotics . . . 15

3.1.4 Hardware Accelerators for Inverted Index Search . . . 16

3.2 Computation Offloading Techniques . . . 17

3.2.1 Transparent Control Flow . . . 17

3.2.2 Offloading to FPGA Accelerators . . . 18

3.3 Conclusion 19

4 Problem Statement and Proposed Solution **21**

4.1 Requirements 21

4.2 Proposed Solution 22

5 Hardware Implementation **25**

5.1 Context 25

5.2 Accelerator High-level Design 27

5.2.1 Address Translation Tables 29

5.3 MPSoC Subsystem 30

5.3.1 Zynq UltraScale + MPSoC IP Configuration 31

5.4 PCIe Subsystem 34

5.4.1 DMA/Bridge Subsystem for PCI Express Configuration 37

5.4.2 Interrupts 39

6 Software Implementation **41**

6.1 Context 41

6.2 Host Software 42

6.2.1 Xilinx XDMA driver 42

6.2.2 Host QEMU instance Modifications Analyse 43

6.3 Accelerator Software 46

6.3.1 PCIe driver 47

6.3.2 Accelerator QEMU instance Modifications Analyse 48

6.4 Proposed Solution Software Analyze 50

7 System Validation and Performance Tests **57**

7.1 Context 57

7.2 System Validation 59

7.2.1 System Initialization Validation 59

7.2.2 System Execution Validation 62

7.3 Performance Tests 63

7.3.1 System Boot Performance 63

7.3.2 System Execution Performance 66

8 Conclusion and Future Work **69**

8.1 Conclusion and Requirements Fulfilment 69

8.2 Future Work 70

A QEMU Build Process **71**

A.1 Retrieve QEMU Source Code 71

A.2 QEMU Configuration Process 71

A.3 QEMU Building Process 73

B Building a custom OS using Petalinux **75**

B.1 Create a Petalinux Project using the BSP 75

B.2 Hardware Configuration 75

B.3 Kernel Configuration 76

B.4 Root Filesystem Configuration 77

B.4.1 Adding Existing Packages 77

B.5 Building and Packaging the Image 78

B.6 Copying Image Files to SD Card 79

C Building a Guest Image **81**

C.1 Creating and Building the INITRAMFS 81

C.2 Compile a Linux Kernel for ARM64 82

Referências **85**

Lista de Figuras

2.1 CPU protections rings 6
2.2 Full virtualization concept 7
2.3 Para-virtualization concept 7
2.4 Hardware-assisted concept 8
2.5 Type 1 hypervisors design concept 8
2.6 Type 2 hypervisors design concept 9
2.7 Emulation concept 9
2.8 Procedure of the dynamic binary translation 10

3.1 pvFPGA framework 13
3.2 pvFPGA architecture design 14
3.3 pvFPGA streaming pipeline stages 14
3.4 High-level architecture for the CPU-FPGA system 15
3.5 CPU-FPGA system implementation 16
3.6 CPU-IIU system architecture 17
3.7 Transparent accelerations transfer flow control process 17
3.8 TPC FPGA offloading flow 18

4.1 High-level simplified system 22
4.2 System activity diagram 24

5.1 ZCU106 top-level block diagram [31] 26
5.2 Accelerator architecture high-level block diagram (adapted from [7]) 27
5.3 High-level system implementation from Vivado 2020.2 28
5.4 DMA PCIe AXI master address map from Vivado 2020.2 29
5.5 DMA PCIe AXI Lite master address map from Vivado 2020.2 30
5.6 PS AXI master address map from Vivado 2020.2 30
5.7 MPSoC subsystem implementation from Vivado 2020.2 31

5.8 Zynq UltraScale + MPSoC configuration menu from Vivado 2020.2 32
5.9 Zynq UltraScale + MPSoC I/O configuration menu from Vivado 2020.2 33
5.10 Zynq UltraScale + MPSoC Clock configuration menu from Vivado 2020.2 34
5.11 Zynq UltraScale + MPSoC I/O configuration menu from Vivado 2020.2 35
5.12 PCIe subsystem implementation from Vivado 2020.2 36
5.13 DMA/Bridge Subsystem basic configuration from Vivado 2020.2 37
5.14 DMA/Bridge Subsystem BARs configuration from Vivado 2020.2 38
5.15 DMA/Bridge Subsystem BARs configuration from Vivado 2020.2 39

6.1 Overview of the Host software modules 42
6.2 Host QEMU modified version 43
6.3 QEMU INIT activity diagram 44
6.4 Overview of the Accelerator software modules 47
6.5 Accelerator QEMU modified version 48
6.6 Accelerator QEMU INIT activity diagram 49
6.7 System initialization sequence diagram 51
6.8 Read case sequence diagram 53
6.9 Command execution sequence diagram 55
6.10 Write case sequence diagram 56

7.1 System experimental setup 57
7.2 System experimental setup implementation 58
7.3 Host Startup validation without root access 60
7.4 Host Startup validation with root access 60
7.5 Accelerator Startup validation 61
7.6 Host execution validation 62
7.7 Accelerator execution validation 62

Lista de Tabelas

5.1 Color index of the Vivado 2020.2 connection wires 28

7.1 Host PC specifications 58
7.2 Full-emulation PC specifications 63
7.3 Boot performance 65
7.4 Performance of the different approaches on distinct benchmarks 66

List of Abbreviations

AMD	Advance Micro Devices
API	Application Programming Interface
APU	Application Processing Unit
ARM	Advanced RISC Machine
ASICs	Application-specific integrated circuits
BB	Basic Blocks
BSP	Board Support Package
CPU	Computer Processing Unit
CSR	Control and Status Registers
CTW	Critical TechWorks
DDR	Double Data Rate
DMA	Direct Memory Access
ECU	Electronic Control Units
FPGA	Field-Programmable Gate Arrays
GPIO	General Purpose Input/Output (GPIO)
GPU	Graphics Processing Unit
HW	Hardware
HLS	High-Level Synthesis
INITRAMFS	Initial RAM File System
IP	Intellectual Property
ISA	Instruction Set Architecture
JIT	Just-In-Time
KVM	Kernel Virtual Machine

MAI	Memory Address Interface
MSHR	Miss Status Handling Register
MPSoC	Multi Processor System on Chip
OS	Operating System
PC	Personal Computer
PL	Programmable Language
PCIe	Peripheral Component Interconnect Express
PS	Processing System
QEMU	Quick EMUlater
RIFFA	Reusable Integration Framework for FPGA Accelerators
SPI	Shared Peripheral Interrupts
TB	Translated Block
TCG	Tiny Code Generator
TLB	Translation Lookaside Buffer
TPC	ThreadPoolComposer
VM	Virtual Machine
VMM	Virtual Machine Monitor

Capítulo 1

Introduction

1.1 Context

In software engineering, test methods have an extreme significance in achieving efficient and reliable software development. Therefore, development teams must have access to robust environments that allow fast deployment and lessen the barriers between the realization of a test and the acquisition of data. However, the needs mentioned above cannot be achieved using the physical system due to the direct dependency between a hardware development rack availability, maintenance and physical access when a problem arises.

Although virtualized environments can tackle those demands, performance should come close to that of the hardware solution. Full-emulation was one of the possible solutions, but it has a significant performance overhead [1] resulting in the pursuit of a better solution. Virtualization is helpful for server and software development that requires service isolation, reproducible and scalable environments, operating-system independence or a reduced hardware vendor lock-in and faster development and test environment [1]. Hence, it frees the development team from the hardware setup availability and facilitates the debugging and issue reproduction. However, virtualization has some constraints that must be resolved to be considered a viable solution to the hardware equivalent.

Most virtualization constraints relate to performance issues at the hardware level due to the use of a general-purpose Computer Processing Unit (CPU) that has high overhead and wastes more energy, 10x to 4000x [2], than required. Despite that, this overhead was accepted back in the day when Moore's Law [3] was unquestionable. Unfortunately, Moore's Law is not perpetual, and some already claim it is no longer applicable [4]. On the other hand, some claim that Moore's Law is far from ending, but it is gradually ending [5]. Consequently, the world started looking for alternatives with lower overhead, such as hardware accelerators, to carry on the scaling of performance and efficiency [2].

Hardware accelerators can be classified as domain-specific accelerators as they are hardware

computer engines specialized for domain-specific applications that allow the system to achieve higher performances [2]. In the market, there are three-main, well-known hardware accelerators: Field-Programmable Gate Arrays (FPGA), Graphics Processing Unit (GPU), and Application-specific integrated circuits (ASICs) [6].

Succinctly, hardware accelerators can be used to nourish a system with virtualization features that could not be previously implemented, such as, hardware-assisted virtualization extensions which can give better performances compared to other solutions, on the assumption that both host and guest architectures are the same [1].

1.2 Motivation

This work was proposed by Critical TechWorks (CTW), a joint venture between Critical Software and BMW, which is dedicated to software solutions for the BMW group. Over the past years, BMW made a solid commitment to virtualizing its Electronic Control Units (ECU) and co-simulation environments test platforms. This progressive movement leads to a new work environment with higher scalability and testability. Despite of its benefits, their studies point to performance issues while using full-emulation, which is required due to specific ECU processor architectures. This would not be an issue if the ECU processor architecture would match the virtualization host CPU arch. As such, a solution based on dedicated hardware acceleration was considered, which is the central aspect of the current dissertation.

1.3 Objectives

This dissertation aims to develop a solution that will serve as a prototype of a complex system and indicate if it is worth putting effort into its development. It will further explore the technical solution paved by [7] and [8]. J. Pedro Vigário Garrido [7] proposed and developed the hardware accelerator for cross-architecture virtualization, and P. Casais da Silva e Sousa Gonçalves [8] studied and exposed QEMU software internals to use the external hardware accelerator. The previous work only ensures the communication between the Host's QEMU and the hardware accelerator, leaving all the bidirectional data transfer and QEMU offloading of the acceleration aspect to be done.

1.4 Dissertation Structure

In addition to the introduction, this dissertation has seven more chapters. In chapter 2, some fundamental theoretical aspects and an overview of the tools utilized are carried out. Chapter 3 refers to the state of the art of hardware accelerator applications in several domains and some techniques to achieve computation offloading utilized in similar works. Chapter 5 and 6 explore the

hardware and software implementation, respectively. Subsequently, the validation of the previous implementations and the performance measurements are exposed on chapter 7. Finally, chapter 8 closes the dissertation clarifying the main conclusions of the developed goal and the requirements verified. Additionally, it also points out the main areas that could be improved and explored in future work.

Capítulo 2

Background

This chapter provides basic knowledge necessary for understanding the field in which the dissertation is located.

Further understanding of virtualization (Sec. 2.1) and the role of the hypervisor (Sec. 2.2) is provided. Moreover, the different types of virtualization (Sec. 2.1.2), namely the one that most suits the project, are highlighted. Afterwards, emulation (Sec. 2.3) is addressed and the hypervisor (Sec. 2.5) chosen for the virtualization as well. Furthermore, the chosen machine emulator (Sec. 2.4, 2.6) is explained, and a brief introduction to hardware accelerators (Sec. 2.7) and computation offload (Sec. 2.8) is performed.

2.1 Virtualization

Virtualization corresponds to the creation process of a virtual instance of computer-related objects, such as computer hardware platforms, the device used for storage and computer networking resources [9]. Although, in terms of software (hypervisor-based) virtualization or platform virtualization, it is the abstraction of the underlying physical hardware so that it can be shared and used among different applications [1]. This introduces a new layer, the hypervisor layer, responsible for managing the physical machine resources. The hypervisor is also known as Virtual Machine Monitor (VMM) and will be explored later.

The advantages of virtualization take place in several areas relevant to a company. Universally, virtualization can achieve benefits, such as reduced capital and operating cost, minimized downtime, increased productivity and good disaster recovery [9]. In terms of specific advantages are service isolation, faster development and test environment, improved system security and reliability and OS and Hardware independence [1].

There are different methods to achieve virtualization, so it is crucial to understand the path that leads to them before advancing.

2.1.1 Protection Rings

Protection rings are mechanisms that protect data or faults based on the security enforced when accessing resources in a computer system. As shown in the figure 2.1, the protections rings are numbered from the most privileged, i.e, 0, to the least privileged, i.e., 3. In terms of Operating System (OS), ring 0 correspond to the kernel-mode/supervisor-mode and ring 3 to the user mode [1]. Applications run on ring 3 and cannot do anything external to them without calling the kernel, but they can do system calls to perform specialized functions [1].

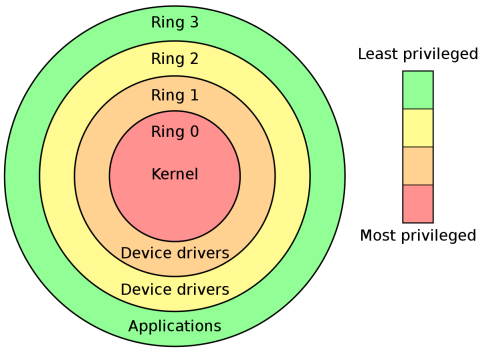


Figura 2.1: CPU protections rings (from [1])

Since the hypervisor/VMM needs to access the host's memory, CPU, and I/O devices, it should run at ring 0, the same ring as the kernel, as long as there is no specific hardware virtualization support [1]. The Virtual Machine (VM) operating system is unaware of the virtualization layer, so it also expects to access all the resources. However, since there is already a kernel running at ring 0 and only one kernel can run at a time at ring 0, the guest OS needs to run with fewer privileges or be modified to run in user mode [1]. This divergence introduces distinct virtualization approaches.

2.1.2 Types of Virtualization

Three main types of virtualization exist [1]:

- **Full virtualization**

  Full virtualization makes use of binary translation to trap and virtualize the execution of particularly sensitive and non-virtualizable instructions, privileged instruction, resulting in some system calls being interpreted and dynamically rewritten. Unprivileged instructions are executed directly, i.e., without the involvement of the guest OS (ring 1), by the hardware used by the hypervisor, as observed in figure 2.2, and as a consequence of hypervisor residing in ring 0, these instructions are executed at native speed.

  As mentioned before, the execution of privileged instructions requires the hypervisor to provide the CPU emulation to handle and modify the protected CPU operations, leading to

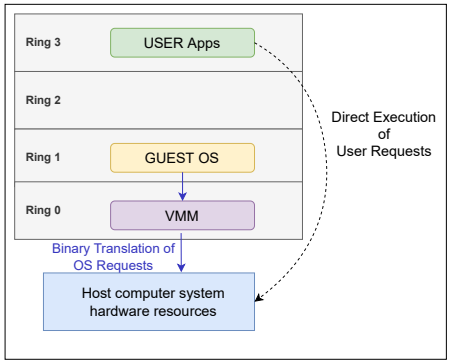


Figura 2.2: Full virtualization concept (adapted from [1])

poor performance due to the overhead, compared to other virtualization methods, such as para-virtualization.

However, full virtualization makes it possible to use unmodified guest operating systems, meaning that it is not necessary to modify the guest Kernel to run on a hypervisor.

- **Para-virtualization**

  Para-virtualization requires the modification of the guest OS to communicate between the hypervisor and the guest through hypercalls. This direct communication results in greater performance than full virtualization, but requires a specialized guest kernel.

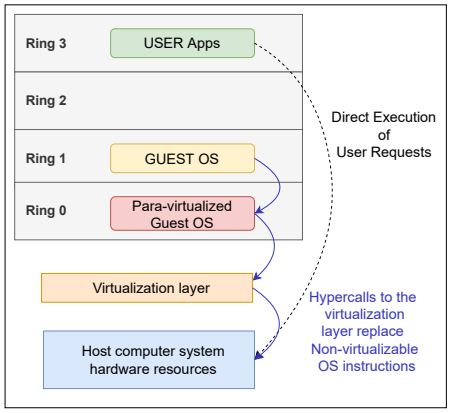


Figura 2.3: Para-virtualization concept (adapted from [1])

- **Hardware-assisted virtualization**

  Hardware-assisted virtualization uses virtualization acceleration extensions provided by Intel, Advance Micro Devices (AMD) and Advanced RISC Machine (ARM), to achieve better performance. This method introduces a new privileged access level, ring -1, where the hypervisor resides. Thus, the guest OS can run in ring 0, leading to direct access to resources without any emulation or OS modification.

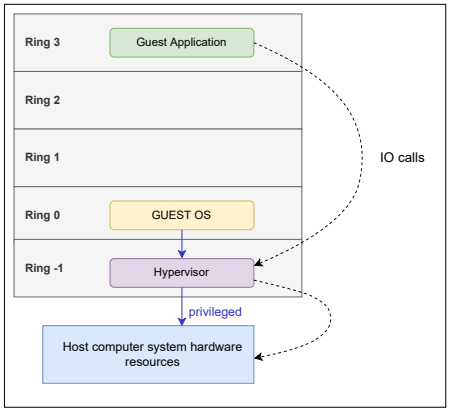


Figura 2.4: Hardware-assisted concept (adapted from [1])

There is a decrease in performance as a result of the dynamic translation used on full virtualization. Therefore, due to this overhead, the hardware-assisted virtualization is preferable. However, this solution cannot be always implemented as the Guest and Host processor architectures need to be the same.

2.2 Hypervisors

The hypervisor or VMM is one of the main components in a virtualization environment. The hypervisor works as an isolating layer responsible for managing the physical hardware and the VM's. Although there is no clear or standard definition, several authors classify them as [1]:

- **Type 1**

  Type 1 hypervisors are also known as Native or bare-metal hypervisors. These hypervisors are directly installed at the physical layer (Hardware) and interact directly with it. Furthermore, type 1 hypervisors do not need any OS.

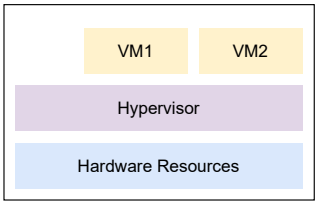


Figura 2.5: Type 1 hypervisors design concept (adapted from [1])

- **Type 2**

  Type 2 hypervisors are also known as *hosted* hypervisors. These are installed on top of an existing OS and run as an application. Therefore, type 2 hypervisors depend on the host OS system for their operations, leading to higher overhead performance.

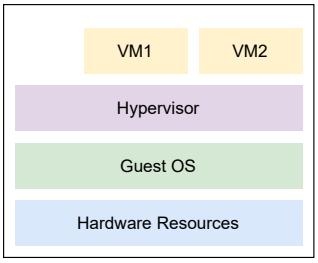


Figura 2.6: Type 2 hypervisors design concept (adapted from [1])

2.3 Emulation

Emulation is a technique that allows a computer system to behave as a different computer system, which consequently allows the execution of software intended for the computer system being emulated. In terms of instructions sets, emulation allows a machine to implement a specific instruction set to execute a software compiled to another instruction set. To accomplish this behaviour, emulation uses binary translation, where blocks of guest instructions are translated to the Instruction Set Architecture (ISA) of the host platform and saved for multiple executions [10].

Although full-virtualization also uses binary translation, they differ from each other because the full-virtualization only translates certain privileges instructions and, on the other hand, emulation requires the translation of all instructions. Therefore, the main difference consists on removing the direct execution of user request arrow from the figure 2.2, illustrated in figure 2.7. Consequently, emulation has a higher overhead than full-virtualization.

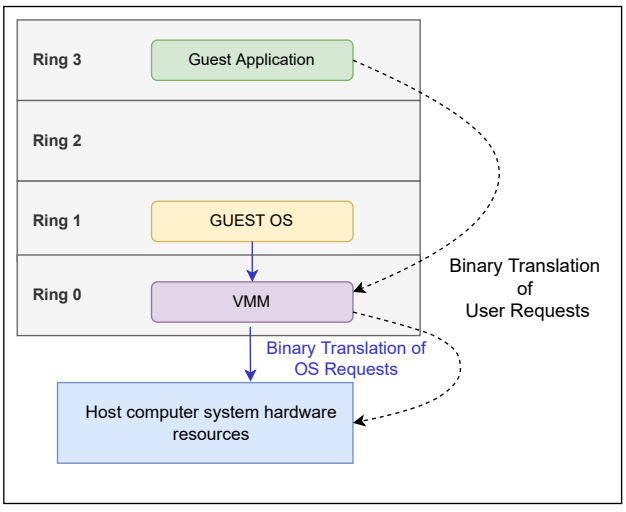


Figura 2.7: Emulation concept (adapted from [1])

2.4 QEMU

Quick EMUlater (QEMU) is an open-source machine emulator and virtualizer that supports two operating modes, full system simulation and user mode emulation [11].

In full system simulation, QEMU provides a virtual model of an entire machine to run a guest computer system. Furthermore, the CPU may also be fully emulated, or QEMU can work along with a hypervisor, KVM, that will be explored later, which results in a near-native performance when the host and target architectures are the same.

A process compiled for one target CPU can be run on another CPU in user mode emulation. In this mode, QEMU uses dynamic binary translations techniques to emulate CPUs, which is needed to execute the guest code in the host CPU. These binary translations are done by the Tiny Code Generator (TCG), which is a Just-In-Time (JIT) compiler that transforms the binary code written for a given processor to another one [1].

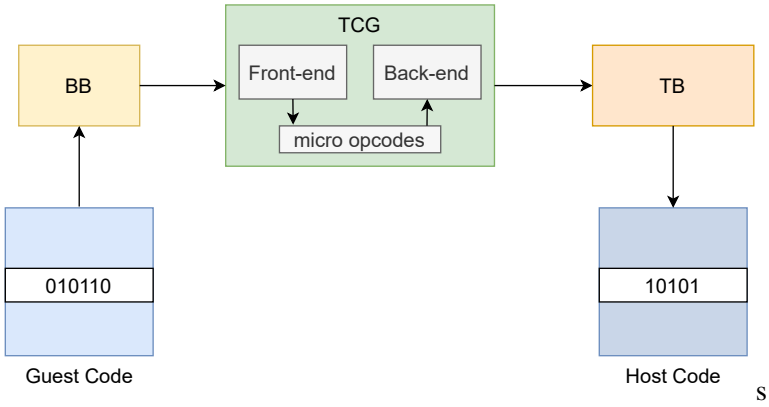


Figura 2.8: Procedure of the dynamic binary translation (adapted from [12])

The dynamic recompilation process to emulate a guest architecture performed by QEMU is described in figure 2.8. Summarizing, QEMU divides the target code into chunks of code called Basic Blocks (BB) that is decoded by the TCG front-end operations generating the micro opcodes. Then the TCG back-end operations hard encode the micro opcodes and generate the host executable instructions labelled as Translated Block (TB) and cache them in a TB hash table that will be directly executed [12].

2.5 KVM

The Linux-based Kernel Virtual Machine (KVM) is an example of a type 1 hypervisor implemented as part of the Linux Kernel [13], which supports full-virtualization on processors with hardware-assisted virtualization extensions (i.e. Intel VT or AMD-V). Basically, KVM is only an interface to the hardware virtualization extensions that turns the Linux kernel into a hypervisor when loaded.

KVM consists of two main components, a kernel module and a user space program. KVM's kernel module is responsible for the virtualization of hardware resources. On the other hand, the user space program uses the `/dev/kvm` interface exposed by KVM to create guest's VM address space and is responsible for I/O's virtualization [14]. Therefore, KVM alone cannot start a VM because an emulator is needed to emulate the hardware peripherals of the VM- [12]. The most common emulator used with KVM is the QEMU, mentioned earlier, and their relationship will be explored in the next section.

2.6 QEMU-KVM

KVM can be classified as a virtualization feature in the Linux Kernel, when the target architecture is the same as the host, which lets a program such as QEMU safely execute guest code directly on the host CPU [12].

KVM exposes the `/dev/kvm` interface to applications and QEMU communicates with KVM, using the `ioctls()` provided by the KVM kernel module, to create, initialise and manage the kernel mode context of virtual machines. When working with KVM, QEMU also does I/O emulation, I/O device setup, and so on. For example, any I/O request made by the guest operating system is trapped within the user space and simulated by QEMU [14]. Furthermore, the QEMU-KVM process has three types of threads [12]:

- Main thread that is responsible for main loop services, include file descriptors callbacks, bottom halves and timers
- Worker threads for virtual disk I/O backend
- One thread for each virtual CPU that runs the Guest VM and handles the communication between Host and Guest

So, the user part of the KVM hypervisor is supplied by QEMU, and it runs the guest code via the KVM kernel module, allowing the system to achieve better performance.

2.7 Hardware Accelerators

Over the last years, computer systems evolved towards heterogeneous architectures that comprise several types of processing elements and hardware accelerators, each of those corresponding to a specific task that leads to a distributed computational workload and consequently a high energy efficiency level [15]. Hardware accelerators are used in several areas, such as computer science [16], genetics [17] and big data [18] to correspond to the high computational demand. Hardware accelerators have a high cost associated, but recently FPGAs introduced a re-configurable factor

that adds flexibility to the implementation [15]. Compared to other hardware accelerators, such as ASICs or full-custom design, FPGAs offers a low-cost silicon chip area, increased performance, and low-power consumption [19].

However, hardware accelerators can introduce a new bottleneck to the system due to the I/O communication system. Therefore, the data transfer mechanisms and memory management solutions play a vital role in the domain of acceleration. One of the most efficient data transfer mechanisms is the Direct Memory Access (DMA) which can load or store large size data without involving CPU systems and in an FPGA-based system it is usually integrated as a software Intellectual Property (IP) core [20]. On the other hand, memory management solutions depend on the size of the data to be processed. According to [21], for small amounts of data, using DMA to fill up the accelerator memory in real time is a viable and efficient solution.

2.8 Computation Offloading

Computation offloading is the transfer of intensive computational tasks to a separate processor, such as a hardware accelerator. Hence, computational offloading is an important step to achieve acceleration. The main problem of offloading is knowing what tasks need to be offloaded and when should they be offloaded [22].

Capítulo 3

State of the Art

This chapter will focus on the current bibliography review of hardware accelerators (Sec. 3.1), mainly in the context of virtualization or other areas, and at the end provides a brief review of computation offloading techniques (Sec. 3.2).

3.1 Hardware Accelerator Applications

3.1.1 Hardware Accelerators for General Purpose Computing

In the area of virtualization and hardware accelerators, [21] proposes a solution for virtualizing an FPGA-based hardware accelerator using a hypervisor, pvFPGA. This approach results in an increase in the FPGA utilization and in the performance of applications running in different domains. This paper describes a basic architecture between the FPGA and an X-86 system host connected by Peripheral Component Interconnect Express (PCIe) bus, as shown in figure 3.1.

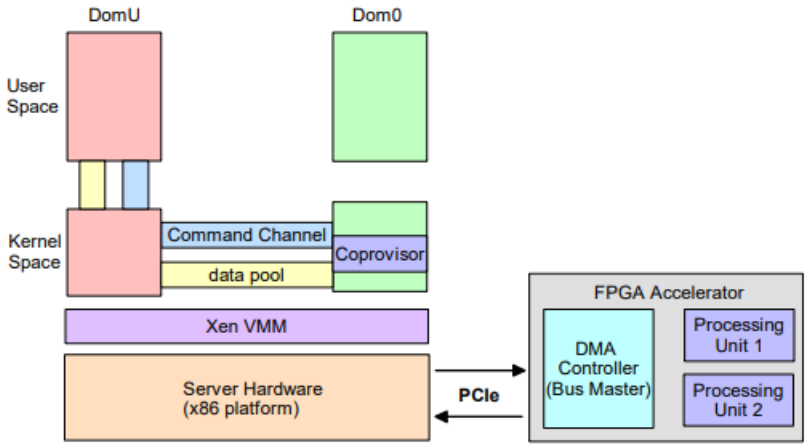


Figura 3.1: pvFPGA framework (from [21])

As mentioned in section 2.7 this paper follows a DMA approach to the data exchange between the host memory and hardware accelerator, which is a simple and efficient approach when the data sizes are small. Although, this approach uses two DMA channels to achieve a full duplex, which allows simultaneously DMA reads from the host memory and writes to the host memory, as observed in figure 3.2. This duplex channel allows the FPGA to take advantage of the streaming pipeline depicted in figure 3.3. However, a streaming pipeline can induce hazards in the FPGA accelerator computation, but [21] also introduces some solutions to fix those hazards with the consequence of a higher complexity added to the system.

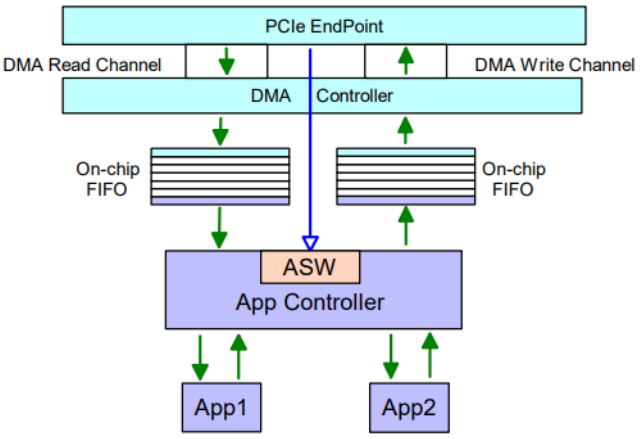


Figura 3.2: pvFPGA architecture design (from [21])

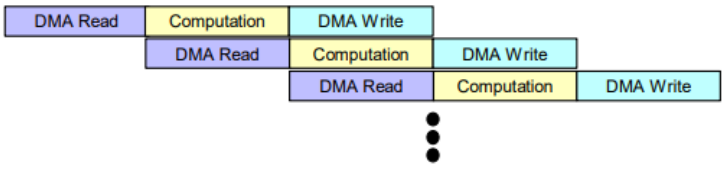


Figura 3.3: pvFPGA streaming pipeline stages (from [21])

3.1.2 Hardware Accelerators for Bioinformatics

To fulfil the need for significant computing resources to analyse genetics molecules [23] proposes a CPU/FPGA tandem. The FPGA receives its inputs from the CPU's main memory and sends its output back to the CPU's main memory in real-time streaming through the PCIe bus. Figure 3.4 describes the high-level architecture of the CPU-FPGA system, where the core engine proposed by [23] is wrapped by the Reusable Integration Framework for FPGA Accelerators (RIFFA) [24]. RIFFA is an open-source collection of software libraries and hardware designs to enable streaming between FPGAs and CPUs on top of PLCe. Notably, in the context of [23], RIFFA allows stitching the core engine to the PCIe endpoint using Rx and Tx engines which receive and generate PCIe packets. On the other side, on the host CPU side, the user application offloads the computation-intensive code to the FPGA. This offloading is executed with the help of RIFFA

Application Programming Interface (API), which includes send and receive functions that can be inserted by the developer directly into the code (as long as it is a C program). Then the RIFFA driver allows these API functions to access the CPU's main memory and the CPU's PCIe controller communicates the CPU's main memory data via the memory controller in the form of PCIe packets.

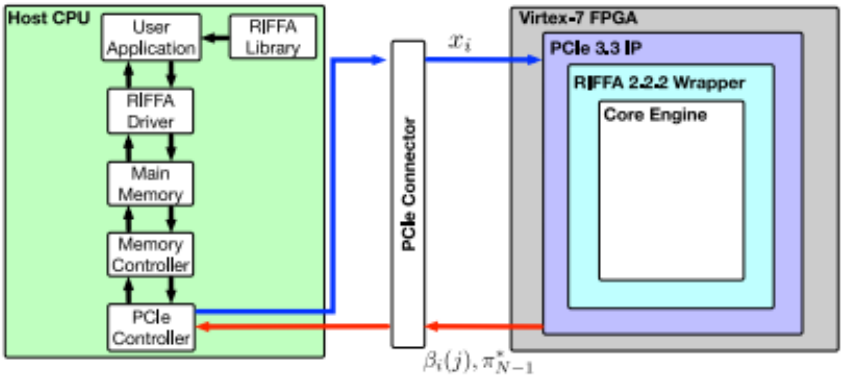


Figura 3.4: High-level architecture for the CPU-FPGA system (from [23])

3.1.3 Hardware Accelerators for Robotics

In robotic applications, there is a computational bottleneck in motion planning and control, the dynamics gradient, due to the performance gap between joint actuator response rate and state-of-the-art software solutions. To tackle this bottleneck, [25] introduces robomorphic computing, a methodology that transforms robot morphology into a custom hardware accelerator morphology. The methodology is achieved by generating a parameterized accelerator design for the gradient of rigid body dynamics and deploying it as a coprocessor to the host CPU for processing multiple dynamics gradient computations. Figure 3.5 illustrates the architecture design between the CPU and the FPGA. The CPU generates inputs and then calls the accelerator to compute the dynamics gradient, the communication between the CPU and FPGA is done with the aid of Connectal [26]. On the CPU side, the Connectal framework sets up links that act as DMA in C++, and on the FPGA-side hardware, it implements FIFO buffers in Bluespec System Verilog. However, the Connectal framework communication is limited to PCIe gen one on the FPGA, as used in [25]. Furthermore, it is used in the Xilinx IP cores to convert CPU floating-point data types to accelerator fixed-point data types.

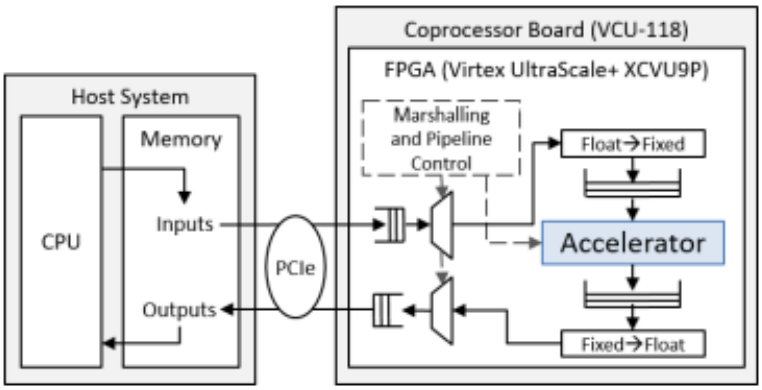


Figura 3.5: CPU-FPGA system implementation (from [25])

3.1.4 Hardware Accelerators for Inverted Index Search

Due to the rapid growth rate of storage requirement and query load for Inverted Index structures, [27] introduces IIU (hardware accelerator), a novel inverted index processing unit, to optimize the query performance while maintaining a low memory overhead for index storage. The solution goes through a CPU-Accelerator architecture where the accelerator processes a highly compressed inverted index for high throughput. Figure 3.6 describes the system architecture where IIU is designed to offload critical operations involved in a search query, and the host sends search queries to IIU via a command queue. Therefore, the host CPU-IIU interface provides two primitive functions for the host to communicate with the IIU. The first one, the `init` function, is used to configure the IIU and set up a communication pipe between the CPU and IIU using a memory-mapped register and loading the inverted index data structure from a file to a non-cacheable memory region. The second one is the `search` function that is used to send a query to IIU, and the results of this query are written to memory at the address specified in the `search` function. Notably, all memory requests from IIU are handled by the Memory Address Interface (MAI) located at the memory controller, which holds a table of 128 entries to store the requested address with a unique requestor ID. This behaviour is similar to Miss Status Handling Register (MSHR) in the host CPU. Later, the memory controller will issue the pending request to the DRAM, and when a response reaches the MAI, it stores the data in the corresponding requestor using its unique ID. Furthermore, due to the use of a virtual address for accessing the inverted index by the host CPU, duplicate Translation Lookaside Buffer (TLB) entries are stored within IIU.

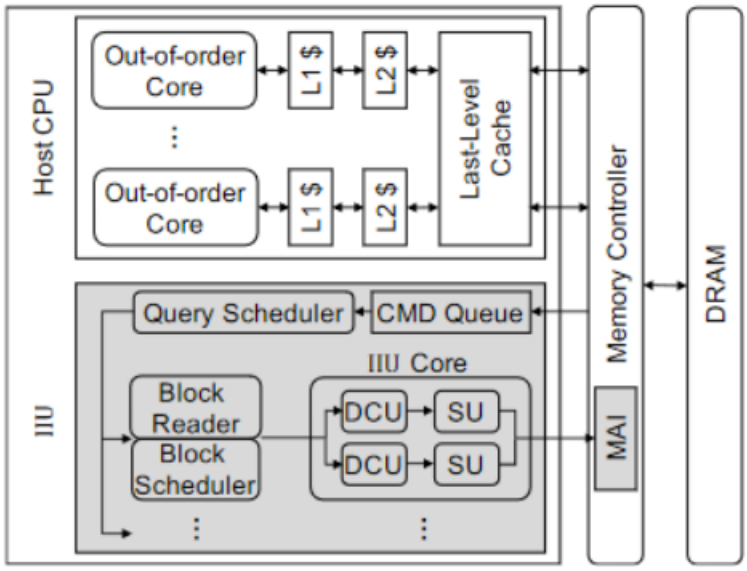


Figura 3.6: CPU-IIU system architecture (from [27])

3.2 Computation Offloading Techniques

3.2.1 Transparent Control Flow

A mechanism that can transparently transfer the control flow between CPU and FPGA is described at [28]. The proposed solution involves transferring the more intensive and regular computation parts known as "hot spots"to specialised hardware, such as an FPGA. The process can be seen in the figure 3.7. The Profiling block is responsible for analysing the program execution and detecting the possible hot spots, and the Accelerator Configuration block is responsible for retrieving or generating accelerator configurations for these hot spots.

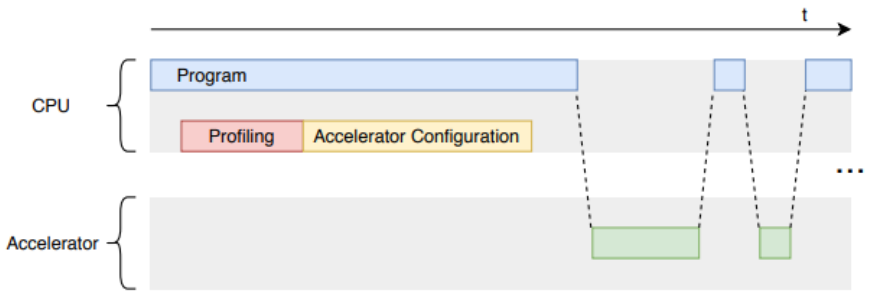


Figura 3.7: Transparent accelerations transfer flow control process (from [28])

This solution is developed on top of a processor tightly coupled to an FPGA on the same chip, interconnected by high-bandwidth communication links. The proposed mechanism depends on the Linux system call `ptrace` to control the process that is being accelerated and transfer the execution flow to the accelerator. In order to transfer data between the software and the accelerator,

[28] used Control and Control and Status Registers (CSR)) to transfer non-variable sized data, the accelerator is allowed to directly access the main memory when the data has a variable size, such as arrays. On the other hand, the accelerator uses the application's address space to access the main memory while preserving cache coherence.

3.2.2 Offloading to FPGA Accelerators

An OpenMP offloading solution for FPGA-based hardware accelerators integrated with the existing LLVM offloading infrastructure project' is proposed in [29]. The system architecture comprises a host CPU and an FPGA connected by PCIe GEn3 x8, and the FPGA accelerator has a 4 GiB onboard memory. This work utilizes the ThreadPoolComposer (TPC) [30], which automates the execution of the High-Level Synthesis (HLS) tool to synthesize hardware accelerators from kernel code and assembles multiple instances of accelerators in the top-level design called *threadpool*. Each *threadpool* is controlled by a two-layered software interface, consisting of TPC API and internal Platform API. The first one is responsible for providing basic functions to query the device bitstream for available hardware modules, transfer data to/from the device and launch jobs on the *threadpool*. On the other side, the second one is responsible for implementing the TCP API and isolating platform-specific code. Figure 3.8 illustrates the execution flow for TPC FPGA offloading. First, the execution starts on the host CPU, and the device execution is initiated by calling `libomptarget`, which is part of the LLVM OpenMP runtime. However, the device needs the data before executing. Therefore, the OpenMP standard allows data to be shared between host and device and for separate data spaces. The second approach is chosen by [29] because the FPGA and host CPU do not share memory.

Consequently, the need to allocate and transfer mapped data to the FPGA memory arises. This is solved by invoking the TPC device plugin. Meanwhile, the `libomptarget` is responsible for keeping track of the mapping between host and device pointers for each mapped variable. The TPC-specific device plugin uses data transfer functions provided by TPC API to initiate data transfers through the DMA engine in the FPGA bitstream. In the end, the data is copied back to the host memory, using TPC API calls and the DMA engine on the FPGA when the hardware execution completes.

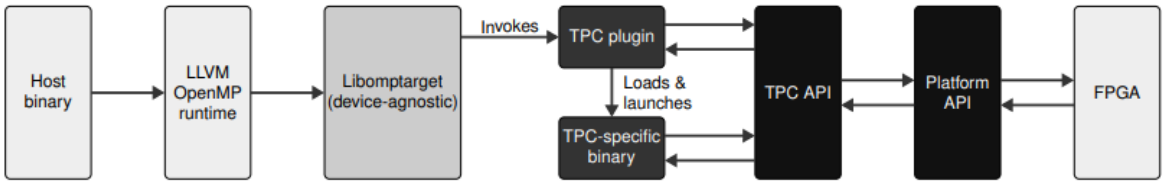


Figura 3.8: TPC FPGA offloading flow (from [29])

3.3 Conclusion

From the analysis of state of the art in hardware accelerators and computation offloading techniques, the efficient use of hardware accelerators and data transfer mechanisms is still being studied, but the domain of computation offloading, specifically instruction offloading, is currently unexplored. Although some workarounds like [28] and [25] can be used as starting points to achieve instruction offloading.

Capítulo 4

Problem Statement and Proposed Solution

Currently, the virtualization of ECUs is accomplished with full-emulation (Section 2.3). Although, as mentioned before, full-emulation has significant performance issues and solutions are arising as an alternative, such as full-virtualization (Section 2.1.2). Even better, full-virtualization solutions are achieving near-native performance due to using hardware-assisted virtualization extensions, such as KVM (Section 2.5). However, this is only possible when both CPU architectures, Host and Guest, are the same.

Thus, the idea of an external hardware accelerator to achieve higher performances occurred. It will be used so that the full-virtualization approach with near-native performance is always possible. As the solution scope is to validate the development worthiness of a complex system, the method used to achieve the near-native performance consists of making the accelerator take over the execution of the Guest Image and become the new "Host". However, the method will also go through offloading memory execution and instructions for the complex solution. Bear in mind that the hardware accelerator envisioned should have the same architecture as the Guest.

Therefore, the solution shall answer the following question:

How to achieve a virtualization solution in a cross-architecture environment without the overhead of full emulation?

4.1 Requirements

Before proceeding to the envisioned requirements, it is necessary to establish definitions that clarify some terms used in the requirements and problem statement.

Definition 4.1.1. (Host). The Host represents an x86-64 computer system that runs the initial instance of QEMU (Sec. 2.4) software. Furthermore, the Host has the hardware-assisted virtualization extensions needed for the full-virtualization.

Definition 4.1.2. (Guest). The Guest represents the software to be executed on the virtual platform.

Definition 4.1.3. (HW Accelerator). The Hardware (HW) Accelerator is an external device that is connected to the Host necessary to use the hardware-assisted virtualization extensions. As mentioned before, the Hardware Accelerator PS must have the same architecture as the Guest.

Definition 4.1.4. (QEMU Host). The QEMU Host represents the QEMU instance that is executed on the Host system.

Definition 4.1.5. (QEMU Acc). The QEMU Acc represents the QEMU instance that is executed on the Hardware Accelerator system.

The following list of requirements will be applied to the envisioned solution:

- The emulator shall execute all processing and memory management operations with the support of an external HW accelerator and any applicable virtualization extensions.
- The acceleration shall yield better overall computing performance compared to the standard full-emulation solution.
- The virtualized environment shall execute transparently, regardless of the availability of the HW accelerator

4.2 Proposed Solution

Aiming at the question envisioned at the beginning of chapter 4, this work aspires to design and implement a solution that will replace full emulation for full virtualization transparently from the user point of view. To do so, we devised a system based on a host and an accelerator platform that will exploit the hypervisor with hardware-assisted virtualization, which, as said in chapter 2, can reach near-native performances. Figure 4.1 illustrates the overall simplified system and highlights the modules we modify, highlighted in blue.

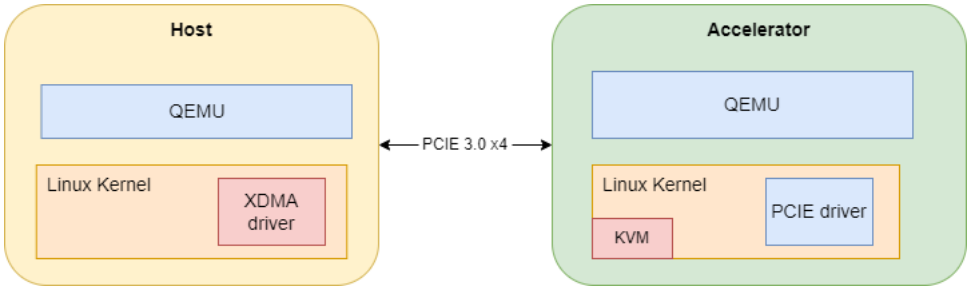


Figura 4.1: High-level simplified system

At first sight, the QEMU at the Host side is responsible for triggering the offloading of the Guest image to the Accelerator. The QEMU at the Accelerator side will receive the Guest image and emulate it. The XDMA driver will handle the low-level communication with the Host. On the other hand, the PCIe driver will be responsible for the low-level communication on the Accelerator.

Figure 4.2 depicts the overall system behaviour split into two parts, start-up and system execution. Furthermore, the diagram also specifies which actions belong to the Host and Accelerator and explicit shows the interactions between them. Note that, although not specified, the interactions in the diagram represent data sent through the PCIe interface and will be further detailed in the following chapters. Firstly, the system was conceived to be transparent to the user, which means that the execution starts on the Host through an external signal, a usual QEMU command, without modifications to it being needed. At the start, the Host sends to the Accelerators the Guest Image and the QEMU arguments and will later perform as an interface to interact with the Guest Image, allowing the Accelerator to take over the processing and memory operations of the emulation. This interface is accomplished by forwarding the serial communication to the PCIe interface, addressed in further chapters.

As stated in figure 4.1 the Accelerator Linux Kernel has the KVM extensions, which is the hypervisor that will be used by QEMU, QEMU-KVM, to benefit from the virtualization features, which implies that the accelerator has the same PS as the Guest VM, that in theory will make the Guest Image emulation faster. The upcoming chapters will elaborate on the hardware and software implementation.

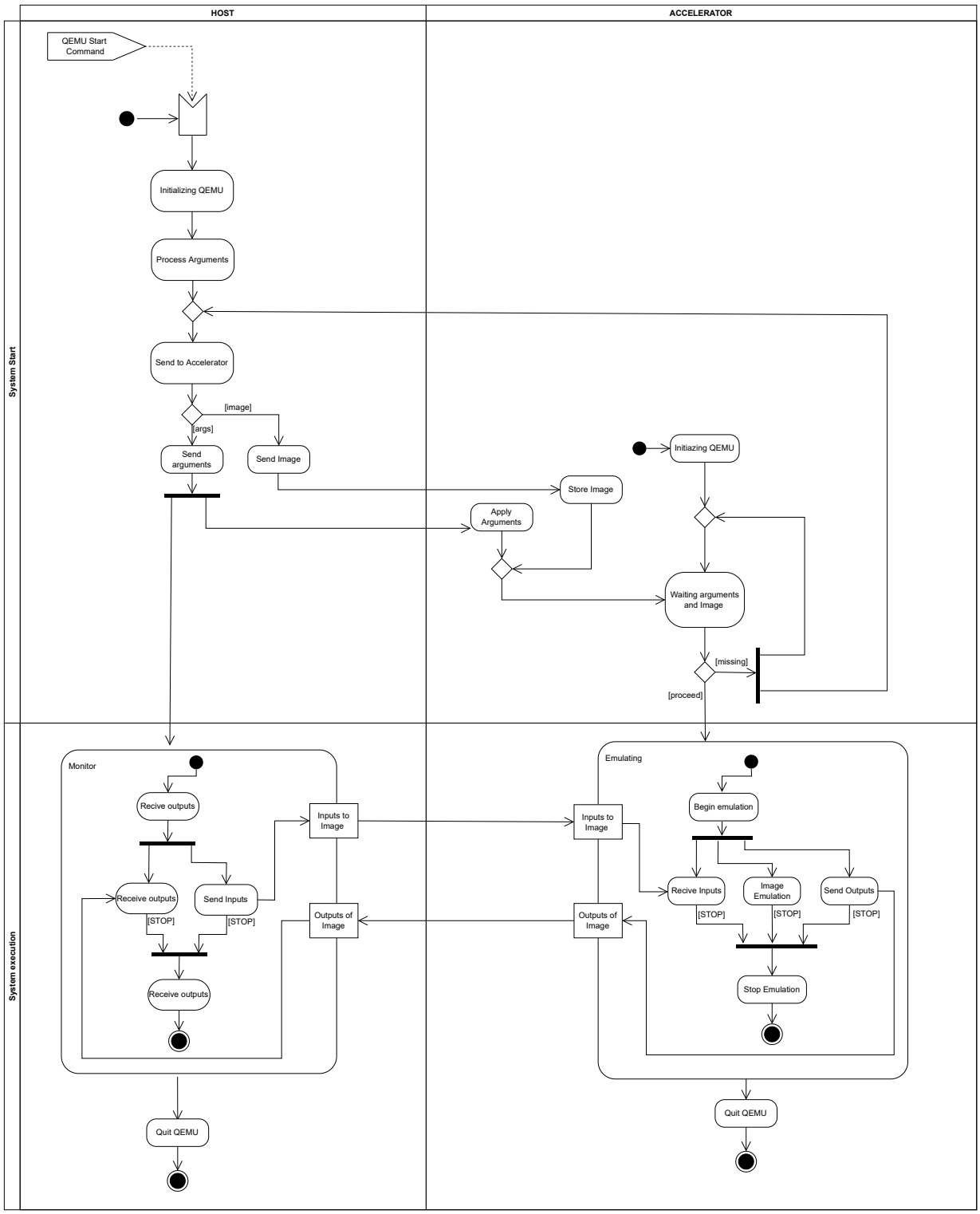


Figura 4.2: System activity diagram

Capítulo 5

Hardware Implementation

This chapter will go through the hardware implementation details and clarify certain decisions regarding its structure. First, a brief context is given to understand the modules that the Accelerator shall have. Subsequently, an examination of the high-level structure of the implemented hardware succeeds and is further analysed meticulous details of the processing subsystem and the PCIe subsystem used to communicate with the Host.

5.1 Context

The project uses the hardware accelerator to achieve hardware-assisted virtualization when emulating an ARM Guest system. When the Host is set up with the accelerator and the QEMU command is executed, the Guest image and arguments are steered to the accelerator. Therefore, firstly, the accelerator needs to have a communication protocol with the Host to exchange data. Second, the accelerator needs to run the QEMU Guest Image and have a Linux Kernel with KVM extensions so the overall system can use hardware-assisted virtualization, as stated in section 4.2. Consequently, the chosen accelerator must have an ARM CPU, in preference, equal to one of the Guest System.

The development platform chosen by CTW was the Xilinx Zynq Ultrascale+ MPSoC ZCU106, a Multi Processor System on Chip (MPSoC) development platform that comprises the characteristics presented in figure 5.1. However, not all are essential to the deemed Accelerator. For that reason, this section will assemble the main blocks that comprise the Accelerator. Firstly, it comprehends ARM Cortex-A53 quad-core multiprocessing CPU on its Application Processing Unit (APU). Moreover, the development platform also has the PCIe interfaces necessary for the exchange of data between the Host and the development platform. It also has an SD Card interface which can be used to install the Linux Kernel needed. Even though the development platform has more interfaces such as USB that could take over the PCIe, those decisions were already made in a previous dissertation [7]. The performances achieved were acceptable by CTW. For that reason,

the structure of the development platform will not be changed besides the shared register space used on the PCIe IP subsystem of [7] which will be mentioned afterwards.

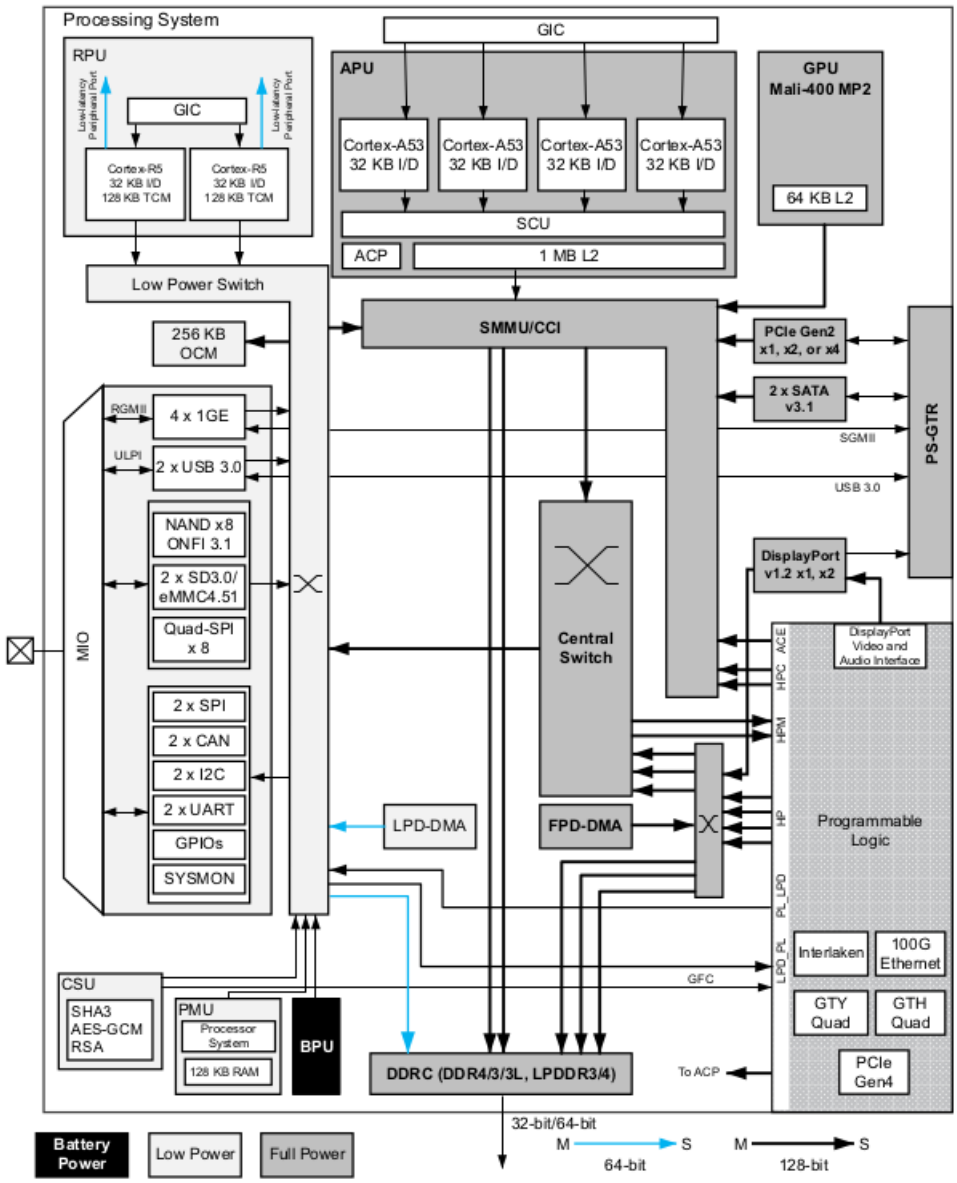


Figura 5.1: ZCU106 top-level block diagram [31]

Therefore, the Zynq106 board has everything needed to implement the proposed solution. However, it is still necessary to implement and develop those interfaces and blocks on our project.

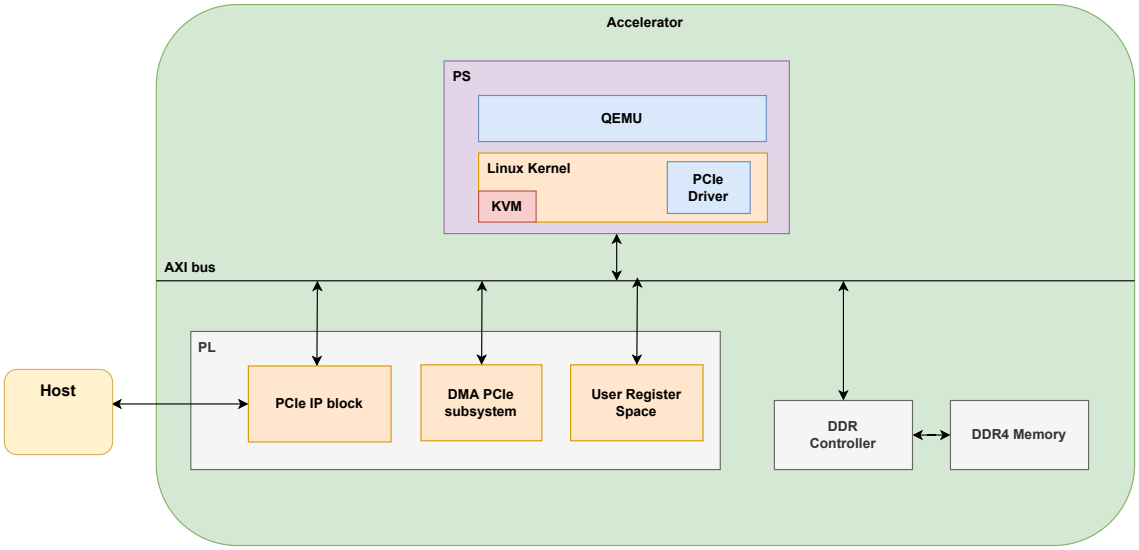


Figura 5.2: Accelerator architecture high-level block diagram (adapted from [7])

Figure 5.2 depicts a high-level block diagram of the accelerator architecture that uses the IP blocks of Zynq106 to fulfil the proposed solution demands. The subsequent sections will explore in more detail the blocks represented.

5.2 Accelerator High-level Design

The development tool chosen for the hardware implementation was Vivado 2020.2, the same used as the previous work [7] and the one provided by Xilinx, which ensures support for our accelerator. Using the development tool, the high-level system was built with the following elements:

- Clocking Wizard
- Processor System Reset
- AXI interconnect
- Processor System (PS) Subsystem
- PCIe Subsystem

Figure 5.3 depicts the high-level system implementation, which was based on the system used in the previous work [7] and from the Xilinx Zynq UltraScale+ MPSoC VCU TRD 2020.1 - PCIe examples [32]. The color index of the figure 5.3 is described at table 5.1.

The system is built with five elements, two subsystems and three auxiliary blocks. The auxiliary blocks, `clk_wiz_1`, `proc_sys_reset_1` and `axi_interconnect_0`, generate the clocks and reset signals required by the subsystems and serve as the subsystems' interconnection. The first primary subsystem, `mpsoc_ss`, represents the Processing System (PS) of the accelerator, and it has two AXI interfaces, master and slave. The slave interface is used to expose the accelerator memory to the PS, and the PS is granted access to the PCIe register space through the AXI master interface. Furthermore, the `pl_ps_irq` receives the interrupts sent by the PCIe subsystem and are mapped to the interrupt table of the ZynqMP IP block.

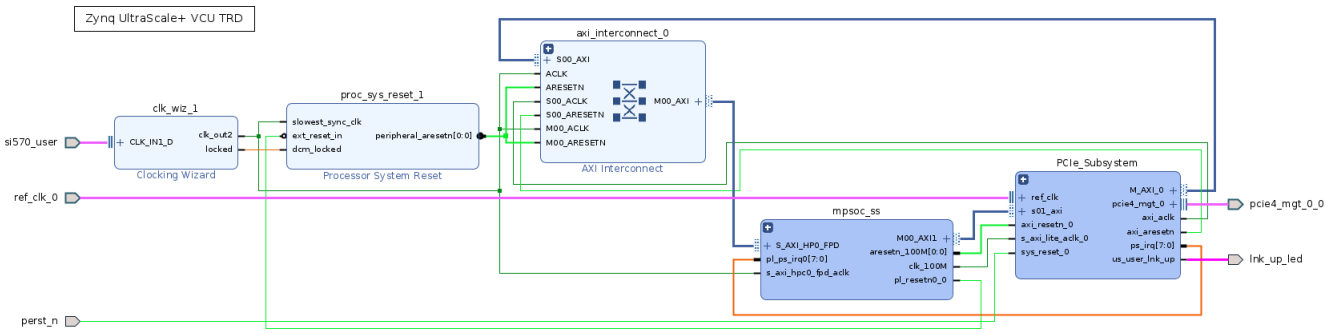


Figura 5.3: High-level system implementation from Vivado 2020.2

The second subsystem, `PCIe_Subsystem`, exposes the PCIe 3.0 x4 interfaces to the HOST PC and communicates to the PS subsystem through the AXI bus. Alike the previous subsystem, it also has two AXI interfaces. The slave interface grants shared register access to the PS, and the master interface allows the PCIe bus to access the memory of the PS through the `axi_interconnect_0` that makes use of an address translation table explained in the following subsection. The `us_user_lnk_up` is used for debugging purposes, which turns on a PL General Purpose Input/Output (GPIO) (GPIO) led when the PCIe data link-layer is on. As stated before, the PCIe subsystem sends interrupts to the HOST through the `ps_irq` that is connected to `pl_ps_irq`.

Tabela 5.1: Color index of the Vivado 2020.2 connection wires

Colors	Description
	Connection between AXI interfaces
	Connection between the PL output interrupt to the PS input interrupts
	AXI clock signals
	AXI resets signals
	Connection to inputs and output of the Xilinx Zynq Ultrascale+ MPSoC ZCU106

5.2.1 Address Translation Tables

Figure 5.2 depicts that an MPSoC can have multiple Programmable Language (PL) blocks beside the PS sharing the same AXI bus, and 5.3 reveals that blocks connected to the AXI bus have each one, usually, a master and slave interface. Therefore, it is necessary the presence of an address translation table to assign the address of each slave and master. Although figure 5.3 only shows two AXI masters, the current design has three AXI masters in total; the PCIe subsystem has an additional AXI master. Thus, it also exists three address translation tables corresponding to the three AXI masters.

Figure 5.4 corresponds to the address network formed by the AXI master of the DMA PCIe. Following the rules present in Zynq Ultrascale+ Technical Reference Manual [33] the address network connects directly to the main memory of the PS, the DDR memory. The lower part of the PS Double Data Rate (DDR) memory is addressed at the first 2 GB, and it is named `DR_LOW`. The following address space corresponds to the QUAD_SPI connected to the PS. The third address connects to the on-chip memory. The fourth and last address corresponds to the remnant of the DDR memory, named `DDR_HIGH` and goes up to 32 GB.



Figura 5.4: DMA PCIe AXI master address map from Vivado 2020.2

Figure 5.5 corresponds to the address network formed by the AXI Lite master of the DMA PCIe, which is connected to the slave found on the register space at the PCIe subsystem with an address range of 64k. The Host can read and write to the shared register space through the PCIe interface.

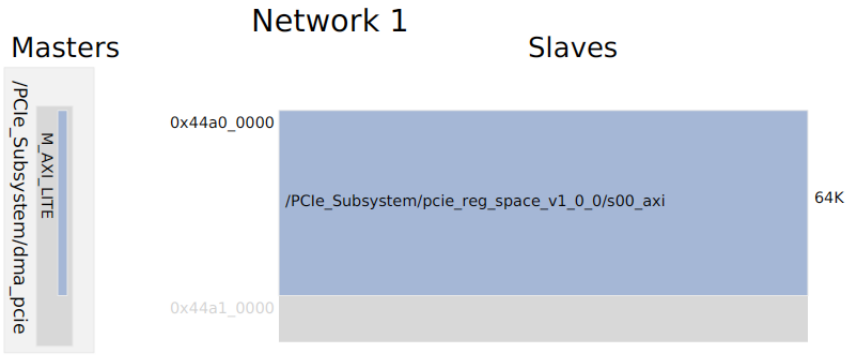


Figura 5.5: DMA PCIe AXI Lite master address map from Vivado 2020.2

Figure 5.6 corresponds to the address network formed by the AXI master of the PS and connects the PS to the register space allowing the PS to read and write to it.

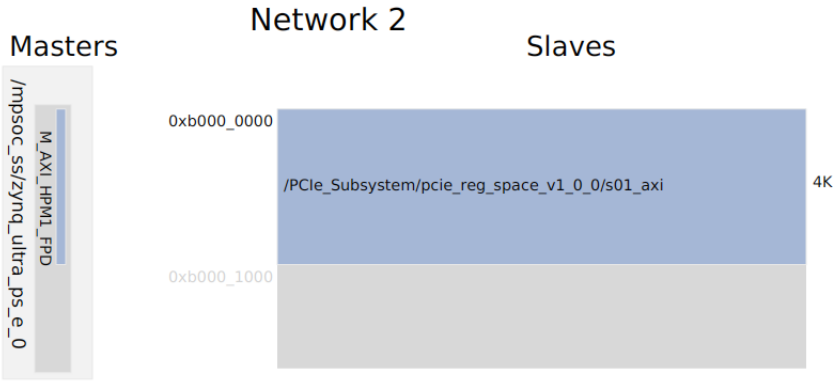


Figura 5.6: PS AXI master address map from Vivado 2020.2

The following sections will terminate the high-level system and focus on the two main subsystems, PS and PCIe.

5.3 MPSoC Subsystem

Figure 5.7 represents the MPSoC which, as stated before, represents the PS of the accelerator and is responsible for providing the Zynq UltraScale + MPSoC IP block that is endowed with the ARM Cortex-A53 CPU as well as interfaces that provide I/O within the SoC and peripherals and other processing units [34]. As illustrated, the PS subsystem has the following elements:

- **Processing system reset**: Allows the reset of the AXI bus and interconnect by the ZynqMP IP block;
- **Zynq UltraScale + MPSoC**: As stated before allows the overall configuration of the PS that is explored on the next subsection, 5.3.1;

- **AXI interconnect**: Connects the AXI master of the ZynqMP block, `M_AXI_HPM1_FPD`, to the global AXI address space.

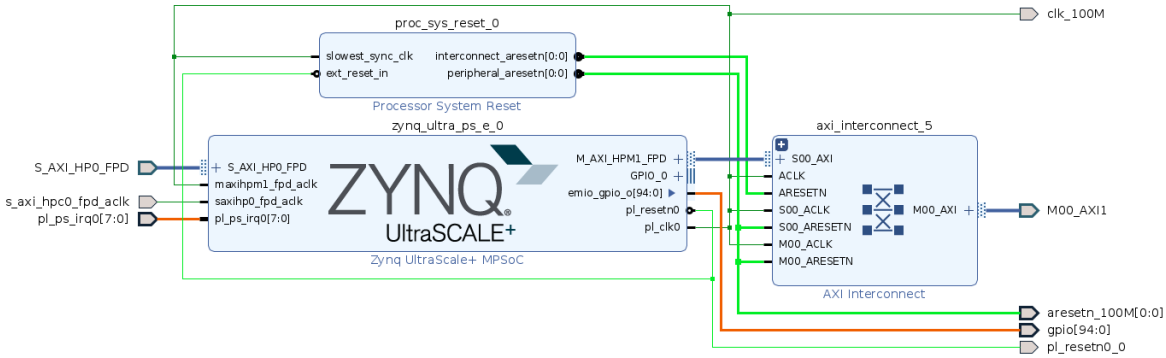


Figura 5.7: MPSoC subsystem implementation from Vivado 2020.2

5.3.1 Zynq UltraScale + MPSoC IP Configuration

This subsection aims to clarify the configuration process of the Zynq UltraScale + MPSoC IP Block to provide the functionalities demanded by the proposed solution presented.

Although the Zynq UltraScale + MPSoC IP Block has three central processing units, the Cortex-A53 APU, the Cortex-R5 real-time processing unit and the Mali-400 graphics processing unit, the proposed solution only embodies the APU.

Furthermore, the configuration also regards important communication options with the PL and peripherals. Figure 5.8 reveal the first page of the configuration menu, composed of several pages regarding each PS component. The first page exposes the system functionalities through a block diagram, where each block can be enabled or disabled. The blocks with a tick are enabled, and the ones without it are disabled.

As observed in figure 5.8 our proposed solution has some blocks enabled on this configuration tab, but most of them are due to debugging proposes and do not significantly impact the overall implementation, such as the USB, GEM, UART and and-so-forth. Therefore, the subsequent configuration pages are more relevant to our proposed design and are explored next.

Figure 5.9 depicts the I/O configuration of the Zynq UltraScale + MPSoC, where two main settings are selected alongside others. The first setting, SD 1, allows the PS to read from the SD card slot and boot and map to the filesystem the SD card found, which allows us to use a Linux Kernel Image. The second setting, GEM 3, is used for debugging purposes once it implements support for 10/100/1000 Mbps Ethernet, which can be helpful to connect to the accelerator via SSH and ease the software development on the accelerator.

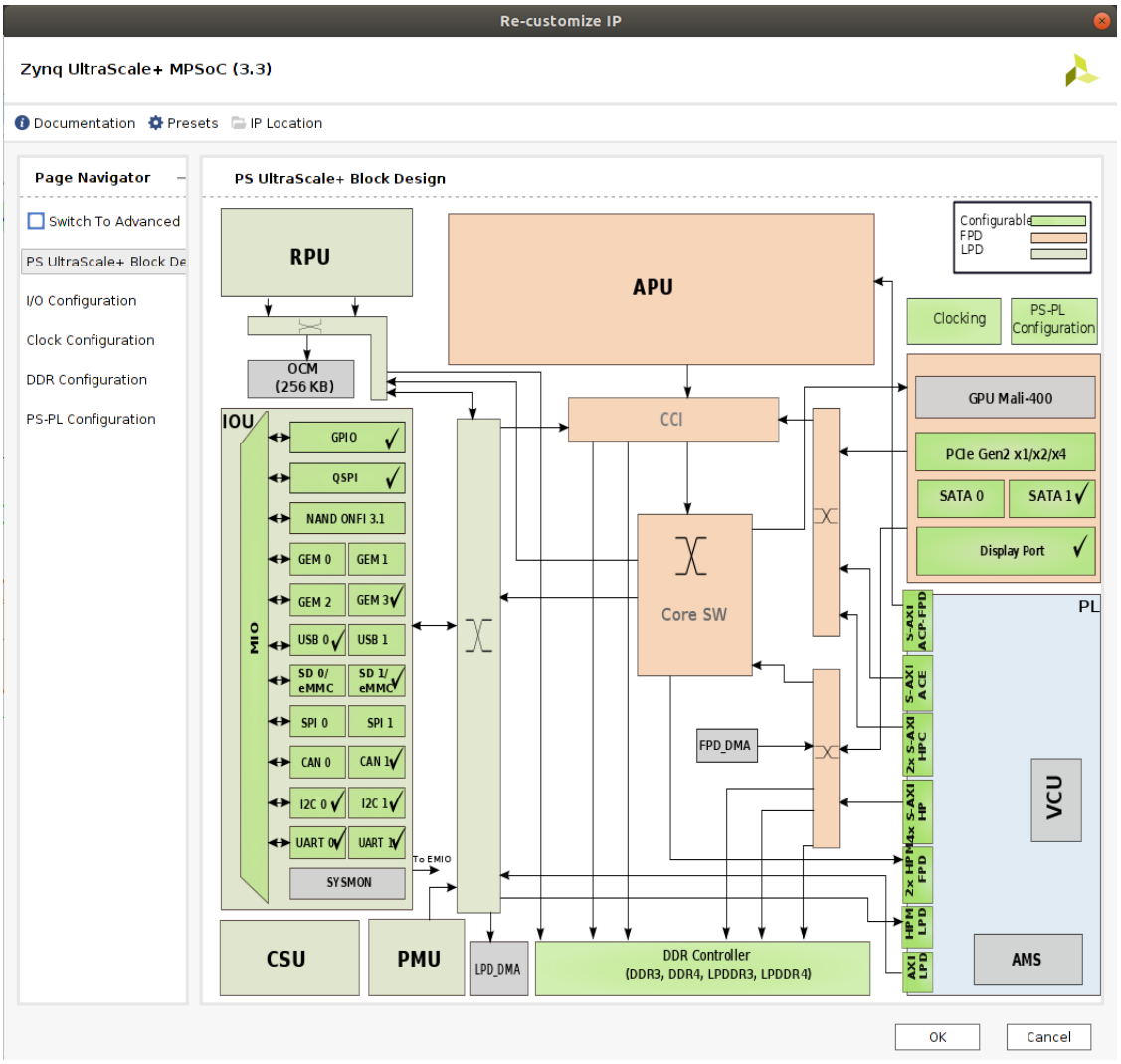


Figura 5.8: Zynq UltraScale + MPSoC configuration menu from Vivado 2020.2

Figure 5.10 corresponds to the clock configuration tab and allows us to modify the input and output clock. However, the Vivado tool automatically calculates the clocks to ensure that the tool meets the requested frequency, or the nearest possible value, from other settings of this IP block and connections to it, [34].

The last configuration tab concerns the PS-PL configuration and provides ways to communicate with the PL. Figure 5.11 portrays the PS-PL configuration and the configurations enabled. The first enabled configuration is the PL to PS interrupts that allow the PL interrupts to be mapped into the PS and be accessible by the Linux Kernel Image. The main difference from the previous work [7] is that the current proposed solution only has the first interrupt, `IRQ0`, enabled, while the previous one enabled `IRQ0` and `IRQ1`. This adjustment is due to the absence of a shared register space in the current project, which is replaced by other components, as described in the next section. Furthermore, the fabric reset enables the PS to send reset signals to other blocks, such as AXI interconnects and the AXI bus. The PS-PL interfaces also have selected settings relating

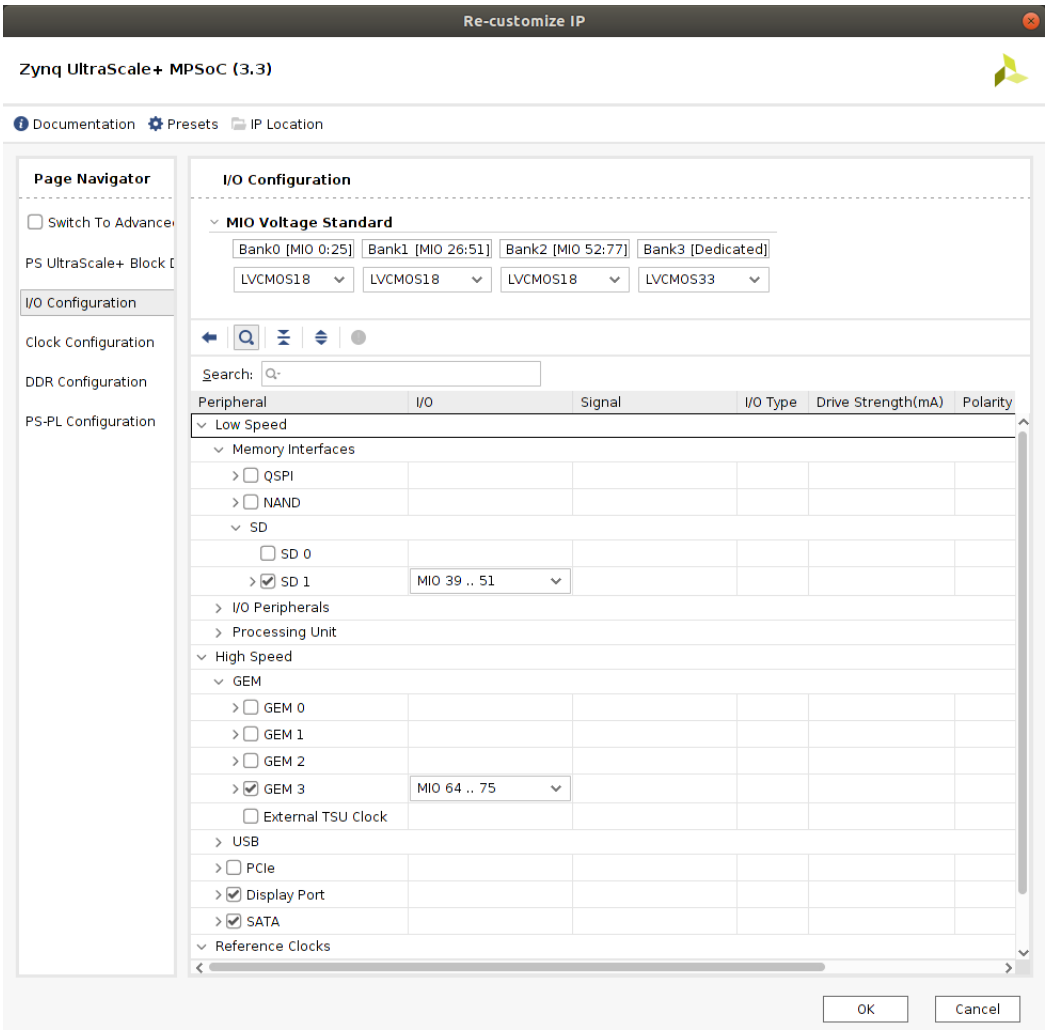


Figura 5.9: Zynq UltraScale + MPSoC I/O configuration menu from Vivado 2020.2

to AXI master and slave interfaces. The `AXI_HPM1_FPD` is selected as the master interface, and with 128 bits data width, the interface name maps to High-performance master 1 in the full-power domain. The `AXI_HP0_FPD` is the chosen slave interface and has a data width of 128 bits. The name maps to high-performance slave AXI interfaces in a full-power domain, which is used in a system where the I/O are coherent with contact closure inputs and no L2 cache allocation [34].

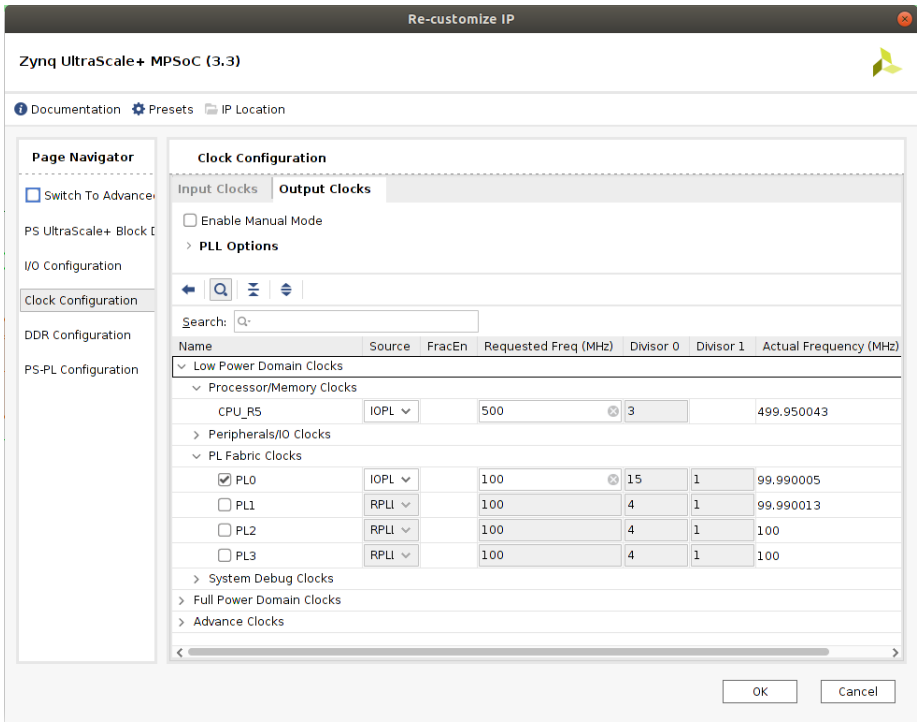


Figura 5.10: Zynq UltraScale + MPSoC Clock configuration menu from Vivado 2020.2

5.4 PCIe Subsystem

Figure 5.12 illustrates the PCIe subsystem, which combines seven blocks in total, four significant blocks, and three auxiliary blocks.

Our proposed solution implements a system where the UltraScale+ Integrated Block (PCIE4) for PCI Express, which offers the PCIe interface, is connected to the DMA/Bridge Subsystem for PCI Express, which offers the DMA capabilities and provides protocol conversion between PCIe transaction layer packets and AXI transactions [35]. Regardless, the second IP already provides the PCIe interface; Xilinx recommends using them together to donate us more control over the chosen settings [36, 32]. The utility buffer is used as a buffer for the PCIe reference clock, and the Constant block sets the value of the `usr_irq_function_number` input of the DMA/Bridge Subsystem for PCI Express. The PCIe register space provides AXI addressable register to store values from the PCIe subsystem and the PS. The two AXI slaves connected to it are related to the DMA/Bridge Subsystem for PCI Express IP Block through the auxiliary AXI interconnect, `s00_axi`, and to the PS subsystem, `s01_axi`. Considering that the outputs generated by this are interrupts from PL to PS and vice-versa, mapped into the respective interrupt tables. Lastly, the Concat IP block is used to concatenate the interrupts from the PL to the PS.

As the overview of the PCIe subsystem is done, the rest of the chapter will focus on the DMA/-Bridge Subsystem for PCI Express configuration since the UltraScale+ Integrated Block (PCIE4) for PCIe configurations are wrapped on the DMA/Bridge Subsystem for PCIe configurations.

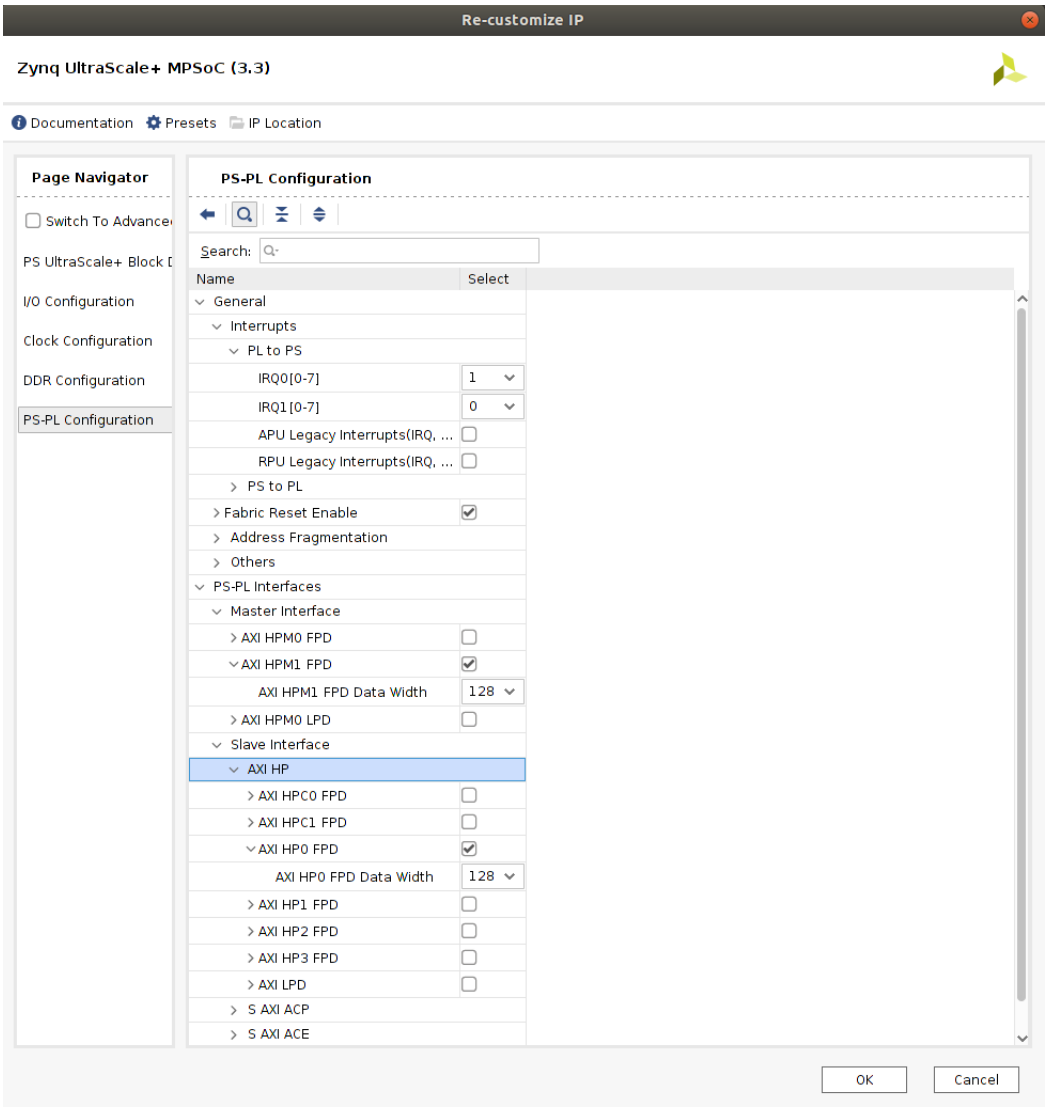


Figura 5.11: Zynq UltraScale + MPSoC I/O configuration menu from Vivado 2020.2

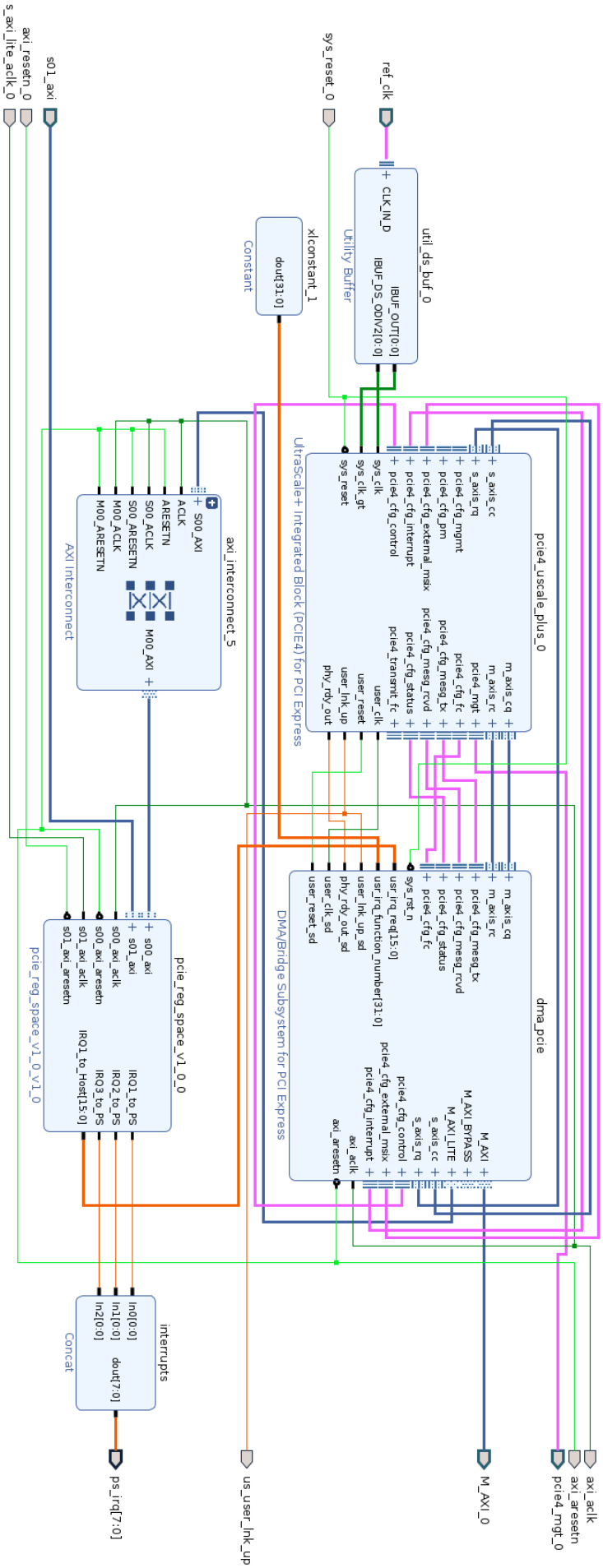


Figura 5.12: PCIe subsystem implementation from Vivado 2020.2

5.4.1 DMA/Bridge Subsystem for PCI Express Configuration

The first relevant configuration tab of the DMA/Bridge Subsystem allows us to specify several settings regarding the PCIe, AXI and DMA interfaces. The PCIe Interface provides us with several options, such as the lane width selecting x4, and maximum link speed, which allows choosing the most recent PCIe standard of 8.0 GT/s. The AXI interface provides the AXI data width, of which 128 bits are chosen to match the same data width of the PS AXI slave bus and clock frequency. The AXI Memory Mapped is selected as the DMA Interface option because, in the current solution, there is no use to the AXI Stream. These configurations can be observed in figure 5.13.

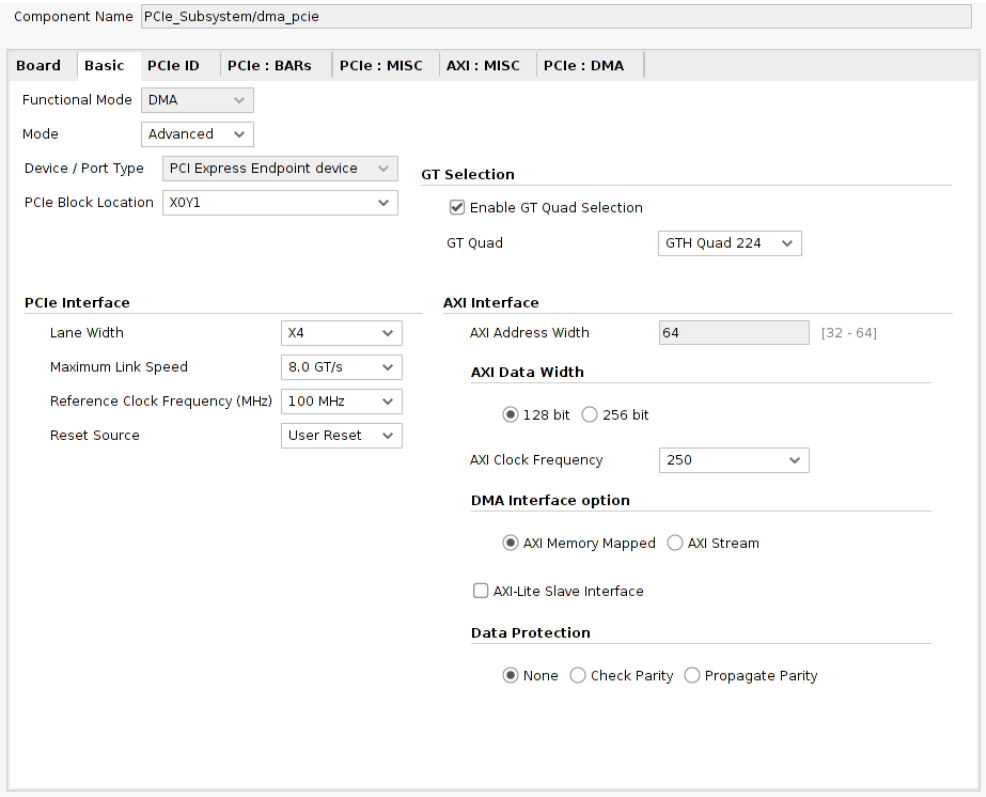


Figura 5.13: DMA/Bridge Subsystem basic configuration from Vivado 2020.2

The PCIe ID configuration tab does not have anything helpful to point out as it only allows changing PCIe ID, which can be used to identify our PCIe device at the Host and the default settings.

Figure 5.14 represents the BARs configuration where the PCIe to AXI Lite Master Interface is selected, so the Host can access our PCIe register space. Considering that the PCIe to DMA Bypass Interface could be used to get the same result, it would take a lot more consideration, and as 32 bits are more than enough for our current project, the AXI Lite Master Interface was chosen over this. Lastly, the PCIe to DMA Interface allows us to address more than 4 GB DDR memory when the 64 bit enable option is selected.

The PCIe:MISC configuration tab allows us to change the number of interrupt requests, which

in the current work is the same as the previous work [7]. Therefore, the number chosen for interrupts request is 16, and the INTA option is selected for the legacy interrupt settings.

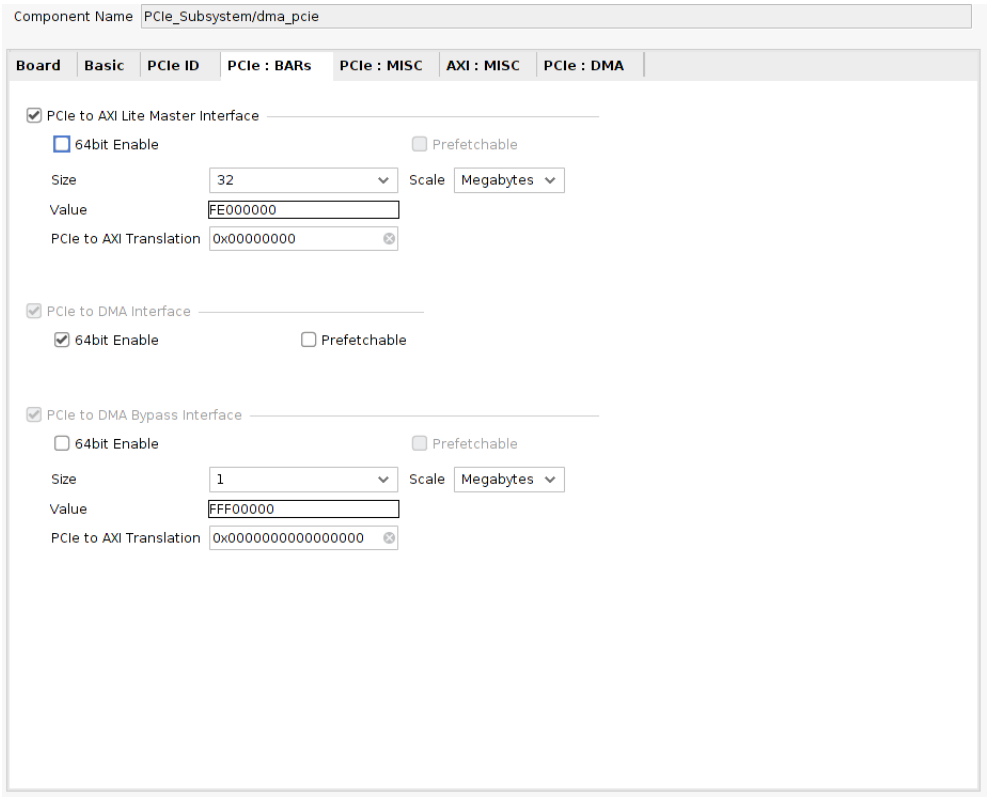


Figura 5.14: DMA/Bridge Subsystem BARs configuration from Vivado 2020.2

Lastly, the PCIe:DMA configuration tab depicted in figure 5.15 provides the DMA engines that can be up to eight. To lessen the doubts about the data flow that read and write could do, Xilinx dropped the typical read and write names for the engines and adopted the H2C and C2H, which means Host to Card and Card to Host, respectively. We can choose up to 4 of each DMA channel, but the proposed solution only implements one H2C and C2H DMA channel. For the rest of the settings, we used the default settings.

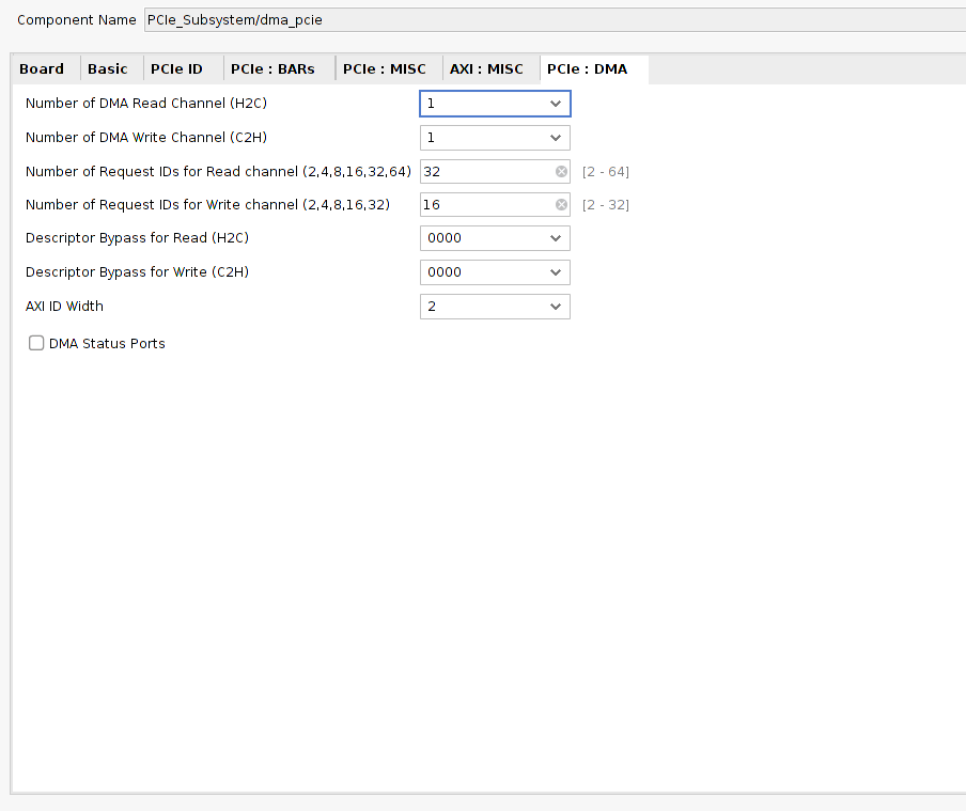


Figura 5.15: DMA/Bridge Subsystem BARs configuration from Vivado 2020.2

5.4.2 Interrupts

In addition to providing addressable registers to store values, the register space is also used to implement three Shared Peripheral Interrupts (SPI) called "read", "write"and "host interrupt"which are used for notifying the end of the read and write transfer and the end of the host logic, respectively. Therefore, listing 5.1 shows the register space hardware configuration defined on a *device tree source include* file that specifies the interrupts used.

```
1 &PCIe_Subsystem_pcie_reg_space_v1_0_0 {
2   interrupt-parent = <&gic>;
3   interrupts = <0x0 89 0x4 0x0 90 0x4 0x0 91 0x4 >;
4   interrupt-names = "read", "write", "host";
5 };
```

Listing 5.1: Register space hardware configuration

Line 2 of listing 5.1 specifies which interrupt controller to use. Therefore, the Generic Interrupt Controller is used to manage the interrupts defined on the next line. Line 3 has nine numbers, where every three numbers implement an interrupt. Line 4 specifies an alias for the interrupts

in the string format. For example, the read interrupt corresponds to the values 0x0 89 0x4. The first value, 0x0, is a flag that indicates if the interrupt is an SPI. The second value indicates the interrupt number. This meaning of these numbers can be found on the interrupt table defined in the Zynq Ultrascale+ Technical Reference Manual [33], where the values 89 to 96 are mapped to the PL to PS interrupt signal 0 to 7. Lastly, the third value indicates the interrupt type, which has three possible values, 0x0, 0x1 or 0x4. The interrupts implemented are all of type 0x4, which is a level-sensitive, active high interrupt.

Capítulo 6

Software Implementation

In this chapter, the details of the software implementation are explained. First, an overview of the context is given to clarify the main objectives and categorize the software to be developed. Then, the software of each part of the system is examined in detail. Finally, an analysis of the interactions between the host and accelerator software is made using sequence diagrams.

6.1 Context

The proposed solution comprises a Host connected to the PCIe interface exposed by the Accelerator. All the communication aspects of the PCIe interface are approached in the previous chapter 5, so this chapter will focus more on the low-level communication software and the software modules developed or modified to obtain the proposed solution. As the system is per se split into two parts, Host and Accelerator, so it is the software developed. The system's behaviour is previously specified in the proposed solution in section 4.2, which consists of the Host executing the QEMU software that will offload essential features, such as the arguments and the Image to be emulated, to the Accelerator. After the transfer completes, the Host QEMU will perform as a monitor connected to the PCIe interface, to which is forwarded the serial communication of the Guest image being emulated. This interface allows the user to input Linux commands and receive the output from the emulated Image originating at the Accelerator.

Therefore, the host software is responsible for data transactions to the Accelerator and providing the interface for its outputs and user inputs. On the other side, the accelerator software is responsible for accepting the data from the Host and emulating the Image received with the QEMU software. The following chapters will analyse the software and tools used during the development.

6.2 Host Software

As stated before, the Host software needs to implement structures that allow data transfer and upload to the Accelerator and an interface to allow users to interact with the Image being emulated. Therefore, the host software consists of an application running at user-level and software drivers running on kernel mode. The application will support the necessary interface and structures to access the software driver that communicates with the Accelerator through PCIe. To have a simple execution of the system and transparently to new users familiar with QEMU software, all the code developed is embedded into QEMU source code, as depicted in figure 6.1. Furthermore, the steps needed to compile and build the QEMU software can be found in Appendix A. Figure 6.1 depicts an overview of the Host software modules that will be explored in the following subsections.

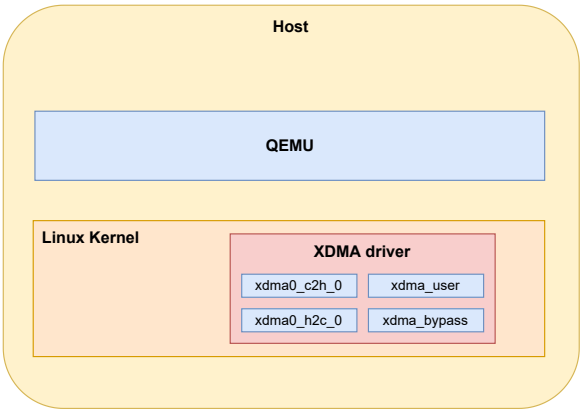


Figura 6.1: Overview of the Host software modules

6.2.1 Xilinx XDMA driver

A DMA driver is needed to implement the peripheral DMA transfers. Xilinx already provides a set of DMA drivers destined to use on a system where the Host communicates with the Accelerator through PCIe. Therefore, the proposed solution uses the Xilinx XDMA driver as the software driver responsible for the data transactions. According to the ZCU106 Design User Guide [36], the driver exposes five user-accessible devices:

- **xdma0_control**: Exposes the XDMA register
- **xdma0_user**: Exposes the AXI Lite Master Interface
- **xdma0_bypass**: Exposes the DMA bypass interface
- **xdma0_h2c_0**: Exposes the host to card channel
- **xdma0_c2h_0**: Exposes the card to host channel

Looking at the hardware implementation, elaborated in Chapter 5, we have a register space accessible by the AXI Lite Master Interface. Therefore, in the context of this work, the devices used were the `xdma0_user` and both channels, `xdma0_h2c_0` and `xdma0_c2h_0`, as the channels will be used to pass the features needed to emulate the Guest system, QEMU arguments and Guest Image. Although the documentation states that the driver exposes five devices, figure 6.1 only specifies four devices in the XDMA driver block. This is due to the XDMA driver source code, which assumes that the user and control device are the same. This applies to the initialization of the device, where the switch case used to initialize the modules has the same actions for user and controller, as can be checked in the 6.1 listing.

```
switch (type) {
  case CHAR_USER:
  case CHAR_CTRL:
    /* minor number is type index for non-SGDMA interfaces */
    minor = type;
    cdev_ctrl_init(xcdev);
    break;
```

Listing 6.1: Fragment of the XDMA driver source code

6.2.2 Host QEMU instance Modifications Analyse

QEMU is the chosen emulator to emulate the Guest System since it can resort to the hypervisor instead of full-system emulation. As previously mentioned, all the software developed at the user level was included in the QEMU source code to give a particular abstraction to the user. Otherwise, the user would need to run several commands to start the system or a script starting several programs, which would be significantly harder to debug. Therefore, additional source files were added to the base QEMU source to allow it to transfer information through PCIe to the Accelerator. Furthermore, modifications to the source code itself were made to accommodate the new PCIe transfer files.

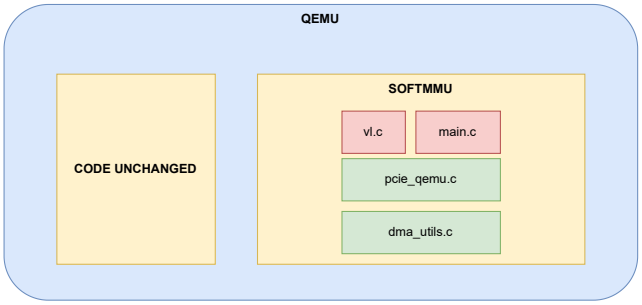


Figura 6.2: Host QEMU modified version

Figure 6.2 depicts the added source files to the QEMU, green blocks, and the files modified, red blocks. INIT's QEMU initialization process, located at `vl.c`, was modified to contain our system's initialization. For instance, figure 6.3 focus on the first part of the overall system behaviour, figure 4.2, and points out the necessary initialisation actions of our system within the QEMU INIT activity. The **Root Check** action is responsible for checking if the QEMU command was called with root properties. This is important because the next step that installs the XDMA drivers into the kernel requires root privileges. The **Parse Arguments** action handles the arguments parsing and withdraws the path to the Guest Image from them. Subsequently, the arguments parsed, and the Image are sent to the Accelerator using the routines provided by the `pcie_qemu.c`. Note that even though the **Send Arguments** is the last action represented, QEMU INIT does not end there, but there was no modification to the following INIT actions. Consequently, they are not mentioned.

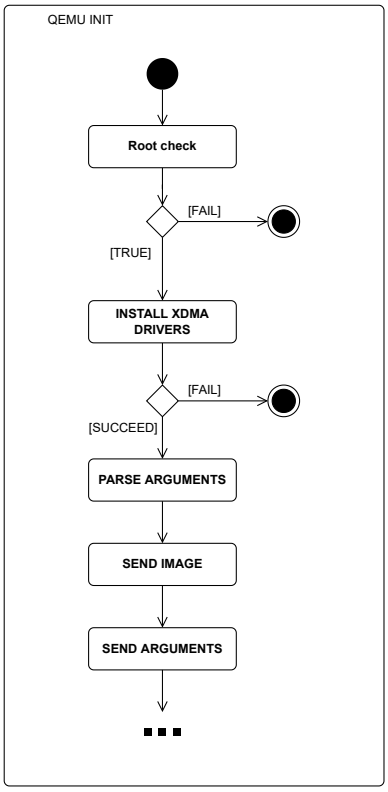


Figura 6.3: QEMU INIT activity diagram

The bidirectional input/output interface was implemented in the `main.c` by creating two threads: one for reading user inputs and one for displaying the standard output provided by the Accelerator. Considering that `main.c` holds the call to the QEMU main system loop, it seems to be the most suitable place to place threads without interfering with the overall QEMU system. Therefore, the thread is launched simultaneously with the QEMU main loop. The *read* thread listens for standard input from the user side, and sends them to the Accelerator calling the functions provided by `pcie_qemu.c`. Although it seems simple to implement, it required some attention

due to escape chars not being sent to the Accelerator, which led to developing a solution that would accept user input one by one and without the input being processed by the terminal. This could be done by setting the ICANON flag off:

```
char get_char_no_echo(void)
{
  struct termios old, new;
  char ch;

  tcgetattr(0, &old);

  new = old;
  new.c_lflag &= ~ICANON;
  new.c_lflag &= ~ECHO;
  tcsetattr(0, TCSANOW, &new);

  ch = getchar();

  tcsetattr(0, TCSANOW, &old);
  return ch;
}
```

Listing 6.2: Function to read user inputs

Furthermore, the solution implemented in listing 6.2 requires us to do the processing of the inputs after its call. Besides that, the solution guarantees that the new line, "`\n`"character, does not appear on the terminal and includes in the command sent to the Accelerator. On the other hand, the writing thread awaits a signal sent by the Accelerator, indicating that the Accelerator has data to transfer. The signal corresponds to a well-defined address in the register space whose value dictates the signal value. After the signal, the PCIe transfer functions are called, and the information received is displayed.

The `pcie_qemu.c` is responsible for providing the PCIe transfer routines that will interact with the XDMA drivers to communicate with the Accelerator, exposing to QEMU the functions listed at 6.3.

```
static int read_info_from_acc(char *c2h_device, char **info);
static int pass_args_to_acc(char *h2c_device, char *c2h_device, char *infname,
    size_t infname_buff_size);
static int pass_file_to_acc(char *h2c_device, char *c2h_device, char *infname);
static bool can_start_write(void);
```

Listing 6.3: `pcie_qemu.c` function declaritions

The `read_info_from_acc` allows the QEMU to receive standard output from the Accelerator. Briefly, it accesses the `xdma0_c2h_0` channel to read from the Accelerator and also accesses the interrupts of the register space on the Accelerator through the `xdma0_user` to know information about the transfer.

The `pass_args_to_acc` and `pass_file_to_acc` are used to pass the arguments and Guest Image to the Accelerator, respectively. The logic is similar to the previous one, but it accesses the `xdma0_h2c_0` channel to write to the Accelerator and accesses the interrupts of the register space to fill it with the transfer features.

The `can_start_write` indicates if the Accelerator wants to transfer data to the Host. This function is mostly used on the write thread.

Additionally, the `pcie_qemu.c` has two functions that are used internally, `pcie_dma_write` and `pcie_dma_read`. The notation used on these function follow the Accelerator point of view, which means that the writer indicates that the Accelerator writes to the Host and the read indicates that the Accelerator reads from the Host. Therefore, to send information to the Accelerator the `pcie_dma_read` is used and to read from it the `pcie_dma_write` is the appropriate one. These functions are responsible for setting the read/write transfer done flag to notify the Accelerator about the transfer status.

Lastly, the `dma_utils.c` as stated by the name provides the support for the DMA transfer. The `pcie_dma_read` and `pcie_dma_write` calls the correspondingly functions provided by this, `read_to_buffer` and `write_from_buffer`. It handles the case when the size to be transferred is bigger than the Linux traditional write() can handle, 0x7ffff000 (2,147,479,552) bytes.

6.3 Accelerator Software

The Accelerator Software is similar to the Host software, which means it has an application running on user-level and a driver running on kernel mode. The application must emulate a Guest Image, accept inputs from the Host and send outputs from the Guest Image to the Host. Thus, the application is a modified version of the QEMU, analogous to the one on the Host software. The driver corresponds to a modified PCIe user-space register device driver (**pcie_driver**) provided by Xilinx to access the register space located at the PL.

Consequently, in order to accommodate the application and the driver, the Accelerator needs an OS image. However, ARM CPUs Boards are heterogeneous, meaning the hardware and peripherals can be different from another board containing an ARM CPU. Therefore, a generic distribution may not work, and a custom one needs to be considered. Buildroot and Yocto are open-source projects that can build a custom OS image. However, Xilinx recommends using the Petalinux tools, which offer everything needed to build embedded Linux solutions on Xilinx PS. Furthermore, Petalinux also uses Yocto along with other tools. Therefore, the chosen development

tool to build our customized OS was the Petalinux due to supporting our development board and being easy to use. Figure 6.4 corresponds to the overall Accelerator Software that will be explored in the succeeding subsections. Note that when creating OS, the virtualization features must be enabled to use KVM within the Linux kernel and QEMU.

Appendix B does a step-by-step procedure for building the Petalinux custom OS.

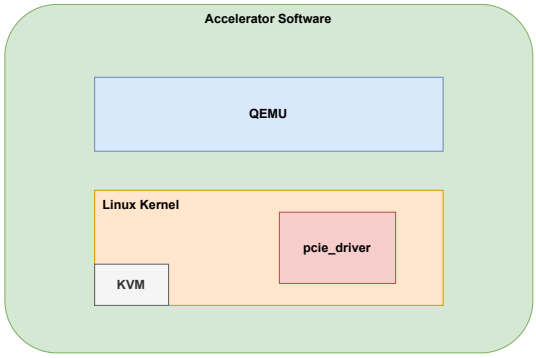


Figura 6.4: Overview of the Accelerator software modules

6.3.1 PCIe driver

The PCIe driver provided by Xilinx, xilinx_pcie_endpoint, and modified for our project is a PL Memory Mapping device that creates buffers for writing/reading data from user space and passes the buffer address to the Host triggering the transfer between Host and Accelerator. Listing 6.4 exposes the several IOCTL calls that the driver supports.

```
1 #define GET_FILE_LENGTH           0x0
2 #define SET_READ_OFFSET           0x1
3 #define SET_WRITE_OFFSET          0x2
4 #define SET_READ_TRANSFER_DONE    0x3
5 #define CLR_READ_TRANSFER_DONE    0x4
6 #define SET_WRITE_TRANSFER_DONE   0x5
7 #define CLR_WRITE_TRANSFER_DONE   0x6
8 #define GET_MODE                  0x7
9 #define GET_FILE_WRITE_LENGTH     0x8
```

Listing 6.4: PCIe driver IOCTL calls

Most IOCTL calls have self-explanatory names, but the `GET_MODE` IOCTL defines whether the data transferred by the Host is the Guest image or not.

Furthermore, the driver defines three interrupts (read, write and host) that can be mapped to the ones specified in the register space, Section 5.4. Each interrupt has a handler function that is called asynchronously each time the interrupt is triggered. The read interrupt states that the

reading transfer is finished. The write interrupt indicates the ending of the write transfer. Lastly, the host interrupt marks the ending of the host PCIe logic.

6.3.2 Accelerator QEMU instance Modifications Analyse

Due to limitations, such as, library versions of the recipes available when building the OS with Petalinux, the QEMU used on the Accelerator has a different version from the one on the Host: and therefore, the contents of the QEMU directory have some nuances. For example, the `softmmu` directory does not exist in older version and the code related to it is stored in the main directory itself and the `runstate.c` is integrated in the `vl.c`. Figure 6.5 highlights the modifications and additions to the QEMU source code on the Accelerator. Although, note that the `softmmu` directory and `runstate.c` are only represented due to consistency with the Host system. The blocks have the same meaning as before; the green blocks are additions to the QEMU, and the red blocks are parts of the modified QEMU code.

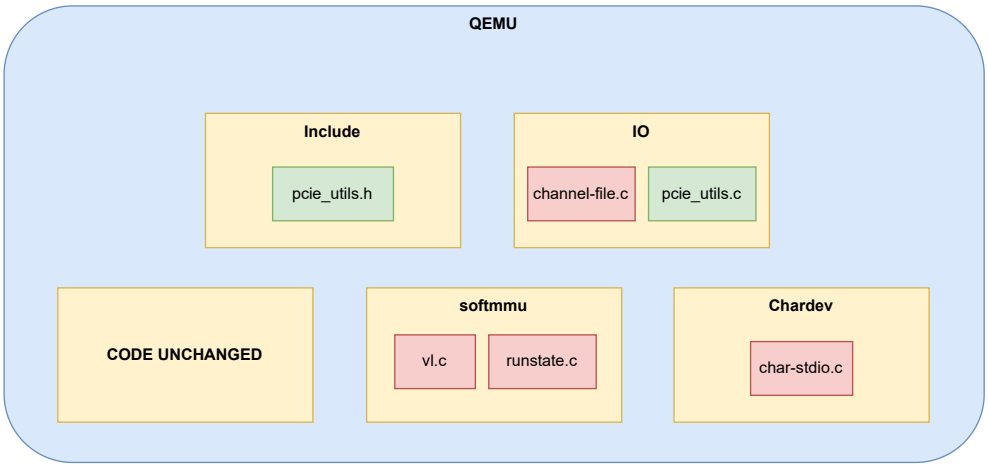


Figura 6.5: Accelerator QEMU modified version

Identically to the Host, the INIT's QEMU initialization process of the Accelerator, located at `vl.c`, was modified. Before proceeding to the usual initialization steps, QEMU waits for the Host's data. If it receives an Image while waiting for arguments, QEMU creates a new file and stores the Image there. Lastly, when the arguments are received, they are parsed to store the data in the variable used by QEMU to initialize its modules accordingly to the arguments specified. Figure 6.6 depicts the Accelerator QEMU initialization resorting to an activity diagram.

To support the PCIe transactions and access the driver, IOCTL calls the `pcie_utils.c` was added. Therefore, using the driver mentioned before, the `pcie_utils.c` compacts everything related to PCIe transfers and information retrieval from the PL register space. Although, unlike the Host QEMU where the PCIe utilities are used only on one directory, the Accelerator QEMU has several directories accessing the utilities. Therefore, a header file is created, `pcie_utils.h`, and added to the Include directory, so it can be accessible everywhere within QEMU.

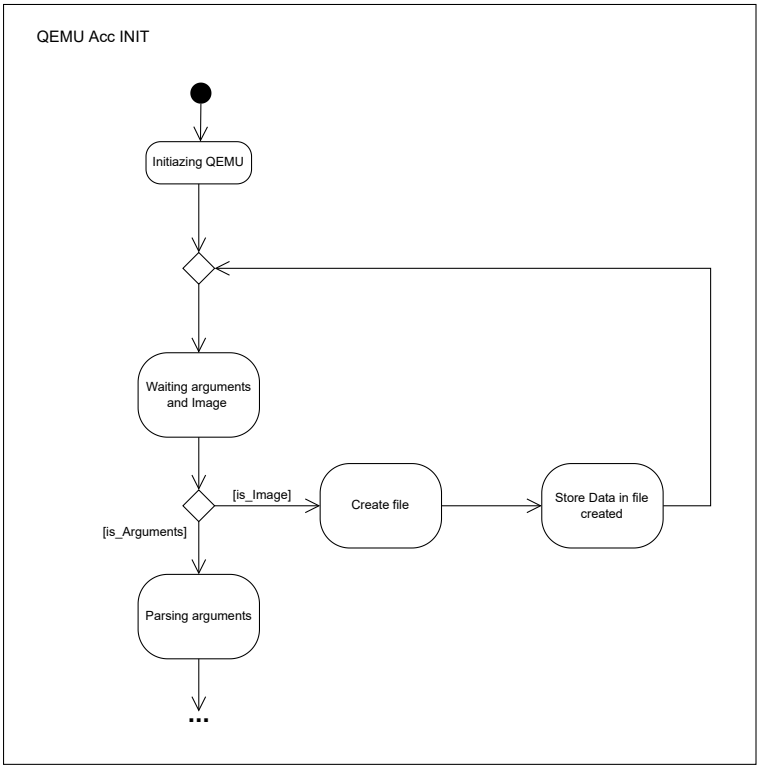


Figura 6.6: Accelerator QEMU INIT activity diagram

When initializing, QEMU creates char devices that are integrated within the emulated system depending on the arguments provided by the user. The most common device used by QEMU is the `char-stdio`, created when `-nographics` is provided as argument. This char device is responsible for providing QEMU with the file descriptors of the `stdin` and `stdout` of the guest image. Therefore, it is adjusted to redirect the file descriptor of `stdin` to a named pipe created by us on the Accelerator.

The main loop on `runstate.c` is used to check if commands are provided from the Host. The check condition uses the functions call established in `pcie_utils.h`. When a command is provided, it is allocated to the named pipe mentioned previously. Therefore, this is the same as writing to the `stdin` buffer of the Guest system.

Everything related to inputs and outputs of the QEMU emulation is inside the IO directory. Therefore, if we want to be able to access the output that QEMU displays, we need to find within this directory the file that corresponds to it. After an in-depth analysis, this file was found, and it corresponds to the `channel-file.c`. This file implements the QEMU I/O channels files driver, which implements the `qio_channel_file_writev` function, that writes everything related to the Guest Image `stdout` buffer. Hence, this function was modified so that it also sends to the Host the information received using the PCIe utilities. However, the first implementation had performance issues because it sends the information as it receives, which leads to sending only a char at a time. Thus, it a solution was developed where the chars would be stored in a static buffer,

and when some trigger event arrived, such as a new line ("\n"), the buffer would be sent. This solution led to better performances that can be validated in the next chapter.

Since the overview of both software components is completed, the next section will describe how they interact with each other using sequence diagrams.

6.4 Proposed Solution Software Analyze

This section will explore the diverse interactions between the Host and Accelerator, exposing the software's internal processing. Although, the system can have several unique interactions for each command sent by the user. Therefore, this section will only address two different cases, initialization of the system and the sending of a standard command, such as `ls`. The following diagrams will use notations, such as READ and WRITE, the same way they are defined in 5.4.1, which means it will be from the Accelerator point of view. Therefore, the READ indicates the Accelerator will receive data from the Host, and the WRITE indicates the sending of data to the Host from the Accelerator.

Figure 6.7 depicts the initialization process. The system is started by the user who invokes the QEMU INIT command, `qemu-system-aarch64`, with arguments. As stated before, in section 6.2.2, the first step checks if the program has root privileges so that the XDMA drivers can be installed. After the driver installs, the arguments are parsed, and the path to the Guest Image is stored and changed to the new path that is going to have on the Accelerator. The `pass_file_to_acc` is called with the Image path to transfer the Image to the Accelerator and during its execution the `open()` returns the file descriptor of the Image to be used within the `read()` call to the Accelerator. The `read()` call happens inside the READ_CASE reference that refers to the figure 6.8 which illustrates how the elements of the system interact when the Host transfers data to the Accelerator. The read case will be explored after the analysis of the system initialisation. Simultaneously, the Accelerator is executing the `pcie_dma_read` and waiting for the Image. Subsequently, to the Image transfer, the Host transfers the arguments received and parsed previously to the Accelerator. The transfer process is similar to the previous one, but without opening any file and on the Accelerator, the `pcie_dma_read` is also executing simultaneously to receive the Host arguments. The initialization process ends with the arguments being parsed to create the structure of the arguments that QEMU expects to be applied to its internals.

The READ CASE sequence diagram, figure 6.8, depicts the Host to Accelerator data transfer flow that refers to the arguments, Guest image and standard input. The transfer starts with executing the `pass_file_to_acc` or `pass_args_to_acc` calls on the Host and `pcie_dma_read` on the Accelerator. The Host starts by opening the XDMA devices available, exposed on section 6.2.1, and the mmap() call maps the PCIe BAR into user space for memory-like access, which allows the Host to change interrupts and access interrupts stored on the Accelerator register space. On the Accelerator, it starts by opening the PCIe driver, exposed on section 6.3.1, which also

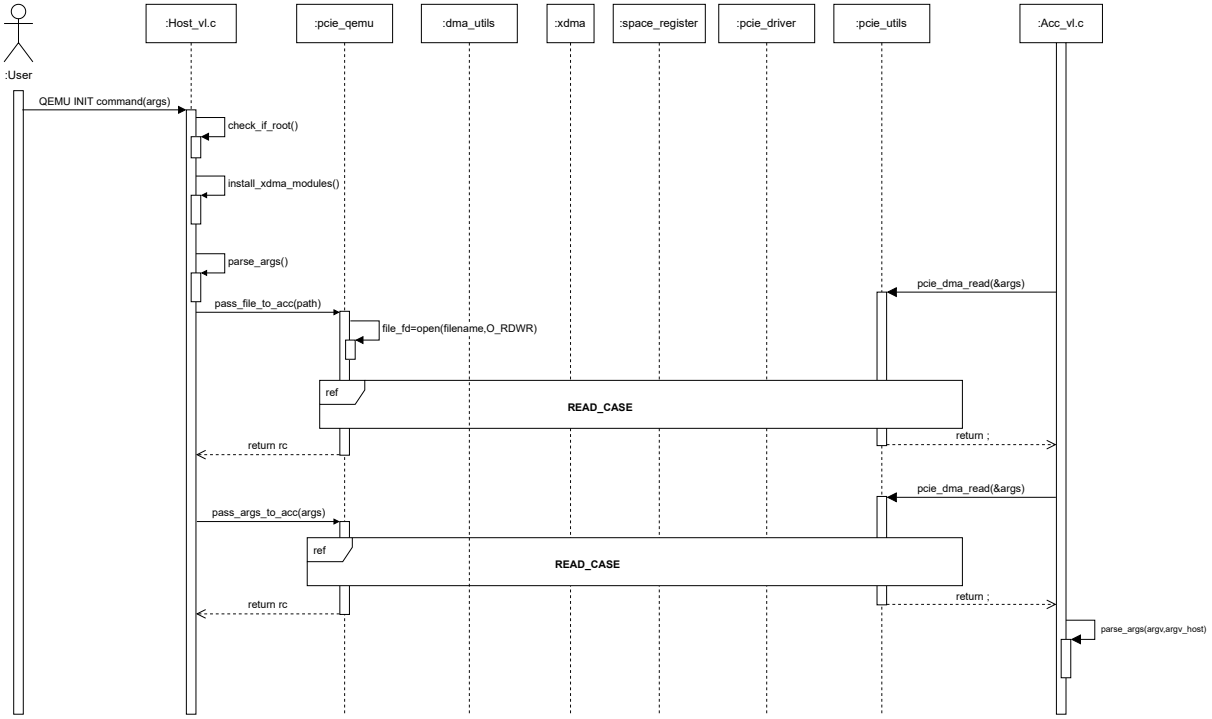


Figura 6.7: System initialization sequence diagram

allows the QEMU program to interact with the register space. The Accelerator enters a waiting state, interrupted when size is stored in the `PCIRC_READ_FILE_LENGTH` register. The Host writes to this register through `write_file_size` and the `set_mode_type()` call defines the mode of the transfer, which specifies if it is being transferred a file or arguments, and writes it to the space register. Subsequently, the Host awaits for the set of the `read_buffer_rdy` by the Accelerator. During these steps, the Accelerator left the waiting state and called the `read()` function of the PCIe driver. The PCIe driver `read()` starts by allocating a buffer based on the size of the application and stores its physical and virtual address. The physical address is stored on the `PCIEP_READ_BUFFER_ADDR` register. The buffer size is also stored in `PCIEP_READ_BUFFER_SIZE`. The Accelerator then signalizes that he is ready to receive the data from the Host by setting `PCIEP_READ_BUFFER_READY` to 0x1. After activating the ready flag, the driver waits for the read-done interrupt. The Host detected the `read_buffer_rdy` and proceeded with its execution, retrieving the buffer address and size values from the register space. The actions inside the **opt** frame are only executed when the data is to be passed to the Guest Image. The `file_fd` indicates the presence of Guest Image data or not. If it is bigger than 0, then the data to be transferred corresponds to the Image data. Therefore, if the data corresponds to an Image, first the Image data is stored in a user-space buffer, and this user space buffer is then transferred to the Accelerator through the `xdma0_h2c` device that uses the read buffer address to know the location of the buffer created on the Accelerator. The number of bytes transferred is returned by the `write()` call and can be used to check if the transfer was successful, after the `write()` call end, the Host raises the read done interrupt and waits for the complete transfer

flag that indicates that the Accelerator successfully accessed the data transferred and continued its operations. With the rise of the transfer done interrupt, the PCIe driver executes the read interrupt handler asynchronously, which ends the completion of the `read()` call and clears the read buffer ready register. For that reason, the PCIe driver can call the `copy_to_user` that stores the data transferred by the Host to a user-space buffer using the virtual address of the buffer created previously, ending the `read()` call that returns the size copied. Subsequently, the Accelerator accesses the mode by executing the `GET_MODE` ioctl. If the mode is equal to or bigger than one, the data represents an Image. Therefore, the Accelerator creates a file and stores the data in it. Otherwise, if the data corresponds to arguments, they are assigned by reference to the variable passed by `pcie_dma_read()`. This was the last step of the READ CASE logic of the Accelerator, and consequently, the `PCIEP_READ_TRANSFER_COMPLETE` register is filled with one's, which activates the read transfer complete flag freeing the Host that ends its READ CASE logic by clearing the read file size register and raising the Host done interrupt. Lastly, the PCIe driver detects the interrupt asynchronously, which executes the respective handler and clears the READ and WRITE complete registers.

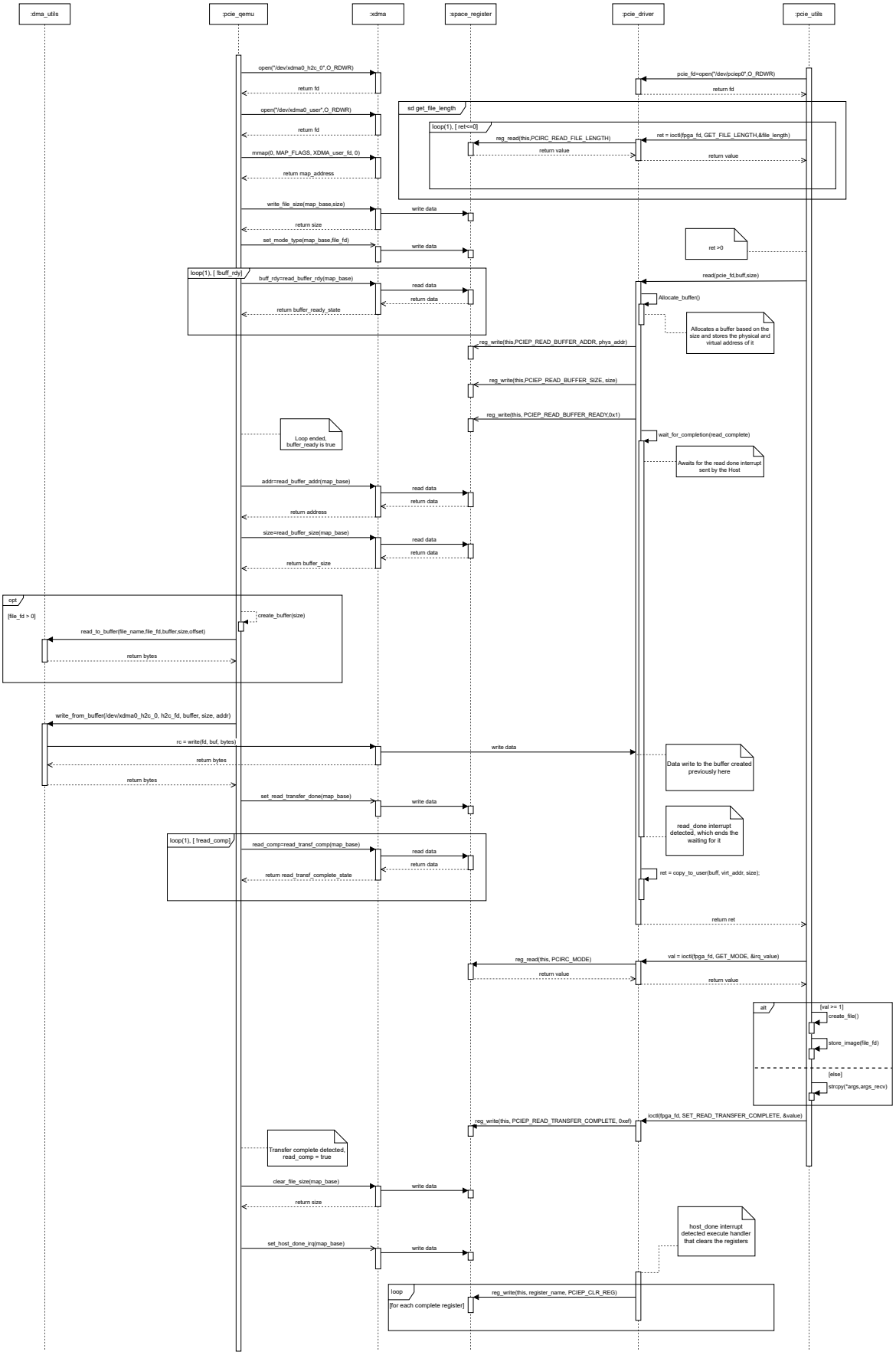


Figura 6.8: Read case sequence diagram

Figure 6.9 depicts the sequence diagram of a given command. The command chosen was the `"ls"` because it is widely known, but for other commands, the process would be similar to the one represented. The Host has two processes running from the start, `input_thread` and `output_thread` and the Accelerator only has one that is the `QEMU_main_loop`. The `input_thread` is responsible for reading the user inputs and is always waiting for inputs. On the other hand, the `output_thread` is responsible for handling and exposing the outputs from the Guest Image that is being emulated on the Accelerator. In order to keep the explanation simple and easy to understand, the sending of the command is explained first and subsequently is explained how the output generated by it is passed to the Host.

Firstly the `QEMU_main_loop` consists of a loop that only exits on the QEMU program end. During the loop, QEMU checks if the file size register has any value stored in it. If it does, the sequences inside the **opt** are executed; otherwise, it is ignored, and the loop starts again. Note that the `get_file_length` reference maps to a subpart of the READ CASE, figure 6.8. The command execution begins with the user inserting a command, which is detected by the `input_thread`. After detecting it and performing some manipulations, explained in the previous section 6.2.2, the `pass_args_to_acc("ls")` is called to pass the command to the Accelerator. The behaviour is similar to the previous one, specified by the READ CASE sequence diagram. Bear in mind that during the READ CASE the file size is set which leads to the `val` on the `QEMU_main_loop` being greater than 0 and proceeding to the `pcie_dma_read()`. After the command is successfully transferred to the Accelerator, it opens the `filename_in`, a named pipe currently being interpreted as the `STDIN` buffer of the emulated Image. Therefore, writing the command to it means writing the command to the Guest Image `STDIN` ending the command transfer sequence.

Similarly to the `input_thread` waiting for user inputs, the `output_thread` is also waiting for data from the Accelerator. The waiting loop checks if the write buffer ready flag is set. On the Accelerator, the output sending sequence only starts when the QEMU IO handlers within the `channel-file` detect output data from the Image. The parsing call corresponds to the mechanism created during the development described in section 6.3.2. The output data that is stored on a user-space buffer is then passed to the Host when the parsing calls return true. The transfers start with the Accelerator calling the `pcie_dma_write()` that takes the buffer as the argument to be used within the `write()` call to the Host. The `write()` call happens inside the WRITE_CASE reference that refers to the figure 6.10 which illustrates how the elements of the system developed interact when the Accelerator transfers data to the Host. The WRITE_CASE will be explored after the analysis of the command execution. Although, during the WRITE_CASE the size of the transfer is allocated to a register within the space register, which is detected by the Host freeing it from the waiting loop. The Host then calls the `read_info_from_acc` that will download the data from the Accelerator. The sequence ends after the Host receives the output data and prints it to the terminal that the user is currently using.

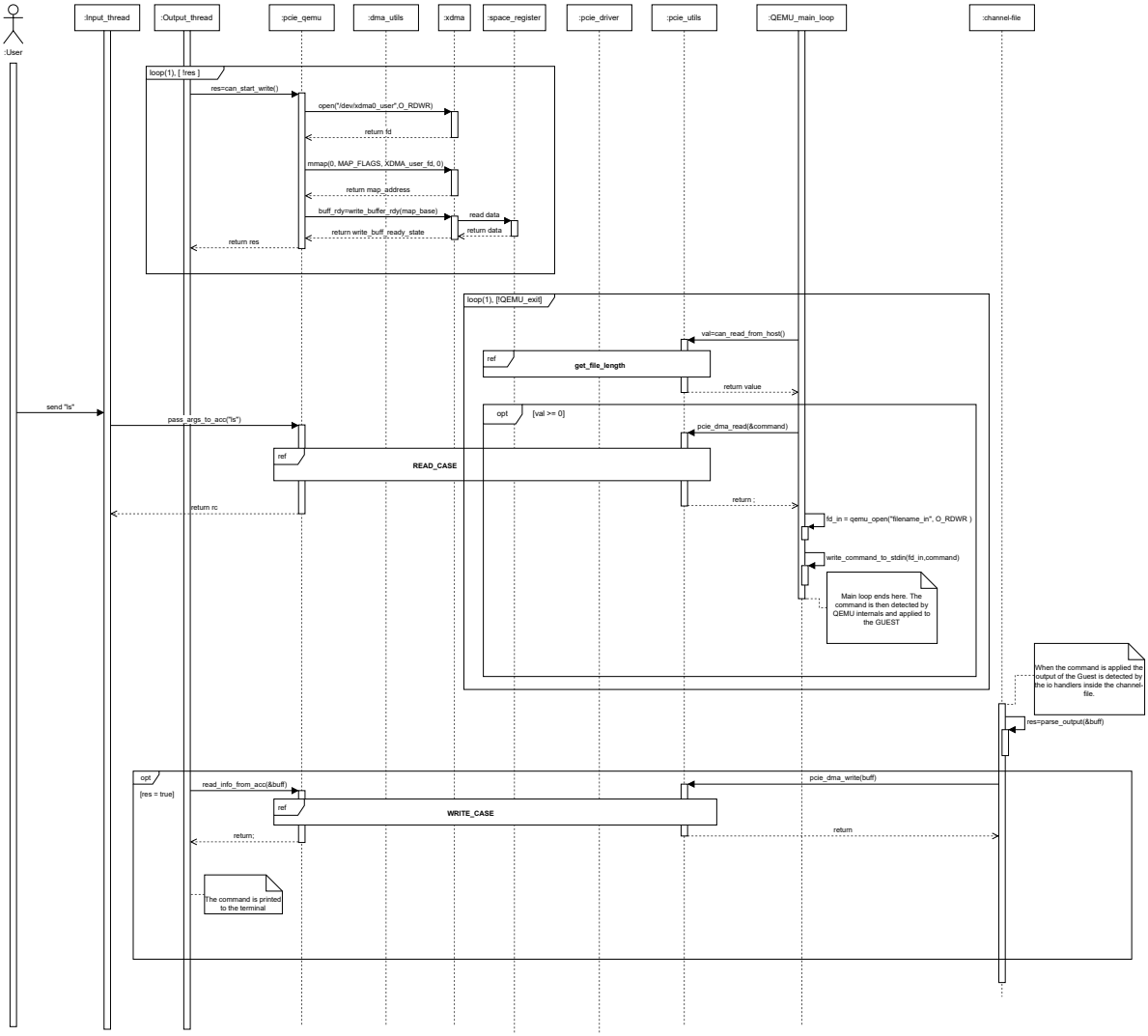


Figura 6.9: Command execution sequence diagram

Figure 6.10 illustrates the WRITE CASE sequence diagram, which has a lot of similarities with the READ CASE, figure 6.8. Therefore, the content that does not represent any change from the READ CASE will not be mentioned again to avoid content duplicity. The main difference is that the Accelerator, `pcie_utils`, is the one who starts the transfer, and the Host is waiting for the data. On the Accelerator, a buffer is allocated, and the data to be transferred is assigned to it. After the assignment, the Accelerator indicates he is ready for the transfer. Subsequently, the Host detects the ready signal and proceeds with the transfer calling the `read_to_buffer()` followed by a `read()` to the XDMA drivers, which reads the data from the buffer previously allocated on the Accelerator using the physical address of it. At the transfer completion, the Host rises the write-transfer-done interrupt, and the Accelerator PCIe driver detects it asynchronously, executing the respective handler. Lastly, the Host waits for the complete write flag, clears the buffer's size and raises the Host done interrupt detected by the driver on the Accelerator executing the handler that clears the complete registers.

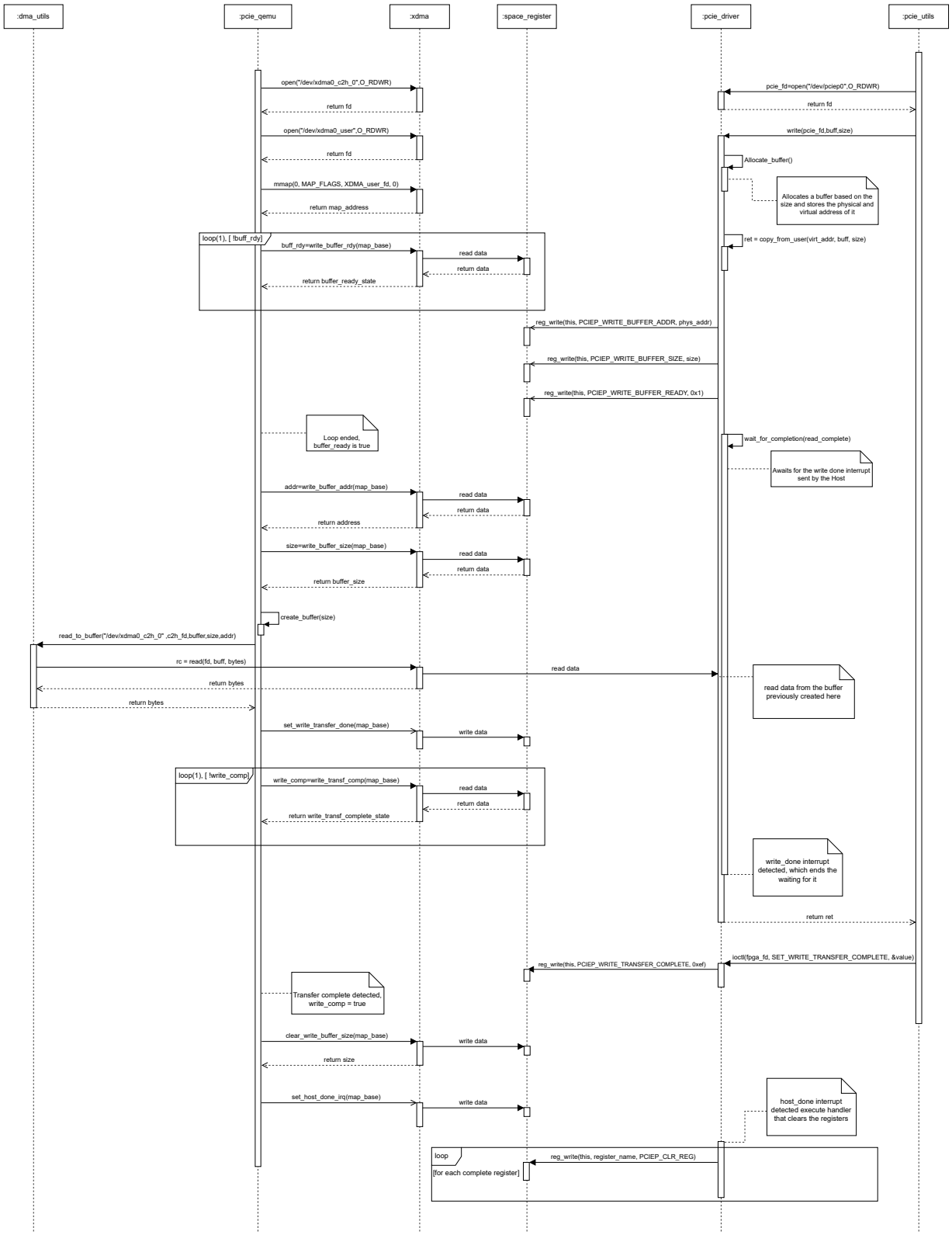


Figura 6.10: Write case sequence diagram

Capítulo 7

System Validation and Performance Tests

This chapter presents the validation of the developed system and its performance. First, the validation of the system is performed in two steps, system initialization and system execution. Then, the performance testing results for the developed solution are discussed. Finally, conclusions are drawn based on the obtained results.

7.1 Context

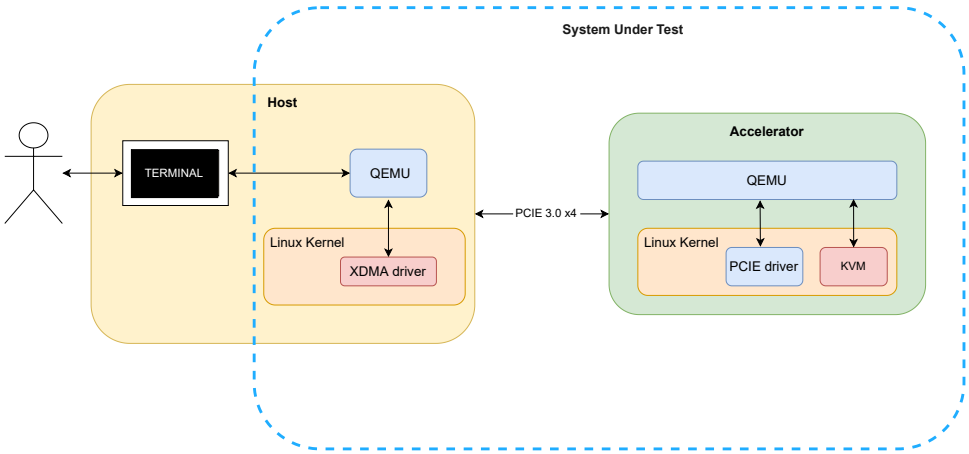


Figura 7.1: System experimental setup

As stated before, the system's behavior is well defined and deterministic and is depicted in figure 4.2. On startup, the Host will send the Guest Image and the arguments of the QEMU command created by the user. During system execution, the user will input commands on the Host terminal and see the results of the command on the same terminal. The Guest Image used

to validate and evaluate the system's performance was created by compiling the source code of Linux for the arm64 CPU, and the Initial RAM File System (INITRAMFS) was built using the Buildroot tool. Appendix C presents a step-by-step explanation of the process of compiling the Linux kernel process from its source code and building the INITRAMFS. Figure 7.1 depicts the system under test, which corresponds to all accelerator components and at the Host, to the QEMU software and the XDMA driver module.

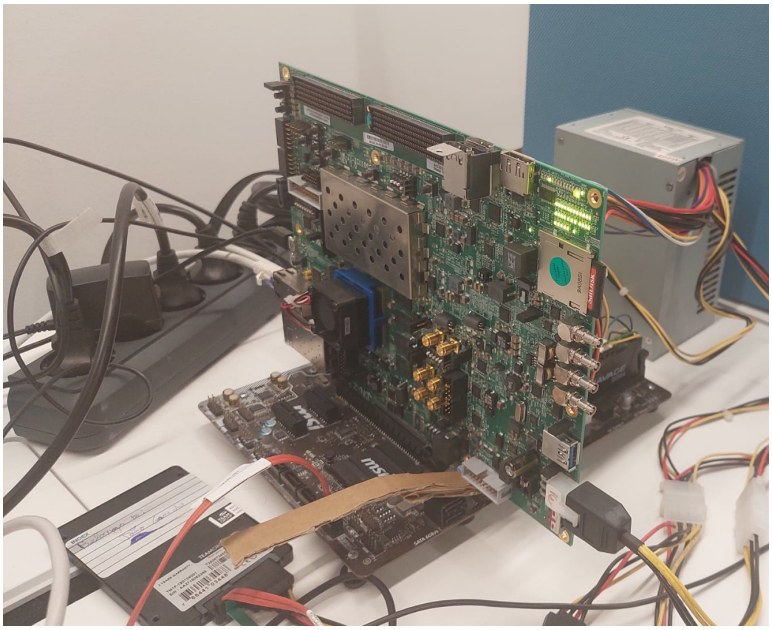


Figura 7.2: System experimental setup implementation

Figure 7.2 depicts the implementation of the experimental setup in the real world. The Host Personal Computer (PC) used on the experimental setup was connected via PCIe to the Accelerator using its PCIe slots, and the accelerator DIP switch was configured to boot from the SD card accordingly to the Board User Guide [31]. Furthermore, table 7.1 summarizes the bare the Host PC specifications.

Tabela 7.1: Host PC specifications

Motherboard	MSI B150M PRO-VH
CPU	Intel i5-6500 3.2 GHz
RAM	DDR3 8GB
Storage	SSD TeamGroup L5 Lite SATA III 240GB
OS	Ubuntu 18.04.5 LTS with 5.4.0-74-generic Kernel

7.2 System Validation

The system validation can be split into two parts:

- Initialization;
- System execution.

The initialization is responsible for starting up the system and getting everything ready for the next part, which is the execution of the system. To validate the initialization, a set of prints and conditional verification are used to confirm the steps defined in section 6.2.2. Furthermore, a set of assertions are also implemented to assure that the arguments received are equal to the arguments sent and that the Guest image received is the same as the one sent. Lastly, the system execution validation consists of observing the execution of a simple Linux command.

7.2.1 System Initialization Validation

This section will address the validation of the system initialization. This consists of checking the steps defined in Section 6.2.2 and later validating if arguments and Images are handed over to the Accelerator correctly. The listing 7.1 shows the command used to start the system with several QEMU flags.

```
#!/bin/bash

/home/Xilinx/qemu-7.0.0/build/qemu-system-aarch64 \
  -machine virt \
  -CPU cortex-a53 \
  -machine type=virt \
  -nographic -smp 1 \
  -enable-kvm \
  -m 2048 \
  -kernel /home/Xilinx/Desktop/Image_test \
  --append "console=ttyAMA0"

```

Listing 7.1: System initialisation script

Listing 7.2 is the conditional verification regarding the first initialization step, which checks if the command is run with root privileges by checking the return value of `geteuid()`. If it does not have root privileges, it shows an error message illustrated in figure 7.3.

```
if (geteuid() != 0) {
        printf("[ERROR] Command needs to be run as sudo\n");
```

```
3         exit(0);
4     }
```

Listing 7.2: Root conditional verification

Otherwise, if the command runs with root privileges, it will try to install the XDMA drivers needed for the PCIe transfer. The first three `INFO` messages at figure 7.4 are related to this step, and the last one validates that the driver is installed successfully. Subsequently, the Guest Image is sent to the Accelerator, the transfer size is displayed for debugging purposes, and the arguments are handed over after the image transfer ends.

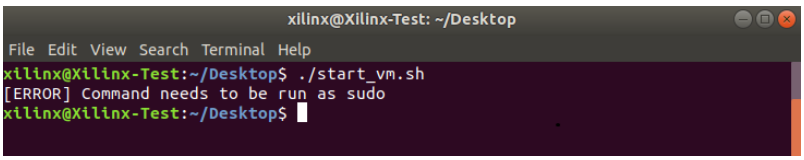


Figura 7.3: Host Startup validation without root access

Lastly, if the application is run with sudo, then the "[INFO] Initiating QEMU on the accelerator..."appears when the transfer of the arguments is completed. The message indicates that the QEMU on the Accelerator will start to emulate the Guest system, which is validated by the boot messages that are displayed at the end of figure 7.4. Parallel to the beginning of the emulation, two threads are created on the Host, creating an environment where the user can interact with the Guest system being emulated, as mentioned in section 6.2.2.

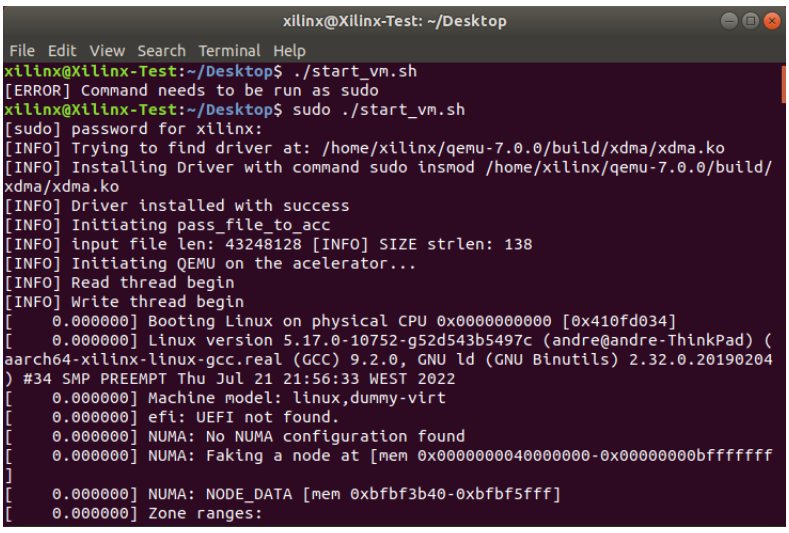


Figura 7.4: Host Startup validation with root access

Figure 7.5 represents the output displayed on the Accelerator operating system during the system initialization. First, it waits for the arrival of the image and arguments. Note that it always receives the image first and the arguments second. When the image is ready to be transferred to the

Accelerator, the size of it is displayed, so we can check if at least the size is the same as indicated by the message at the Host. After the PCIe read call, the interrupt value displayed indicates the type of the data received, being the identifier of an Image 0xEF, hexadecimal for 239, and 0 for the argument's type. After acknowledging the type, it performs actions for each type. Therefore, if the type corresponds to a file, the next step is creating a file to store the data in it, displayed by the creating file message.

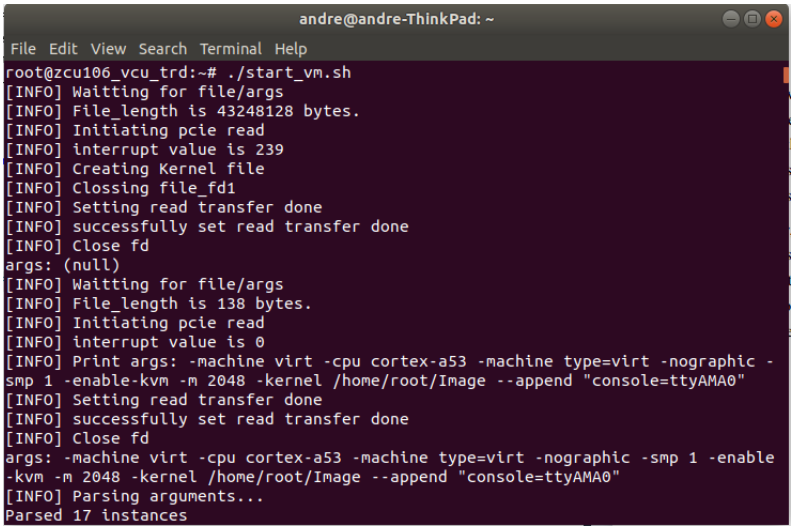


Figura 7.5: Accelerator Startup validation

As no error message is shown, the system is behaving as expected. Lastly, the image and arguments assertion takes place. The arguments assertion, listing 7.3, checks if the arguments received are equal to the expected arguments defined in listing 7.1. Although, note that the path of the kernel option varies and the new points to the new transferred image in figure 7.5.

```
1 assert(strcmp(args,ARGS_EXPECTED) == 0);
```

Listing 7.3: Arguments assertion

On the other hand, the Image assertion, listing 7.4, makes use of the *MD5* hash of a file, which is considered a digital fingerprint of a file. Therefore, functions are implemented to calculate the *MD5* hash of the image received, and the return value is used on the assertion.

```
1 assert(is_md5_match(file_hash, file_received_hash));
```

Listing 7.4: Guest image assertion

As the assertions were not triggered because they were correct, it is inferred that the initialization validation was successful.

7.2.2 System Execution Validation

The execution flow of the system is simple, but there can be infinite commands to test on a Linux environment. Therefore, to validate the execution flow, two commands are selected, `"cd /"` and `"ls"`, which are well-known commands and are enough to know if the system is responding to user inputs and see if the contents displayed by `"ls"` correspond to the ones specified when building the Guest Image. The Guest Image should have several files on Root Directory, such as the hello world binary and other performance binaries mentioned later in this section.

Figure 7.6 represents the execution flow of the system on the Host. After the Boot messages of the emulated image, a login prompt appears, asking for the username and password. As stated above, the command tested were the `"cd /"`, which sets the current directory to the Root Directory, and `"ls"` that lists the contents of the current directory. As expected, both commands behaved correctly, and the contents displayed validates the execution of our image.



Figura 7.6: Host execution validation

Although the accelerator visual flow is not necessary to validate the system execution, it exposes some important details worth mentioning. During system execution flow, the Accelerator kernel is always interrupted by the interrupts defined in the previous chapters used to notify the end of a read, write and host logic. Figure 7.7 illustrates the execution of the `"ls"` command and how its information is processed to the Host. Therefore, the information is handed over line by line, and the write interrupt indicates that the Host already received the information and the Host done interrupt means that the Host ended processing the information and is ready for more.

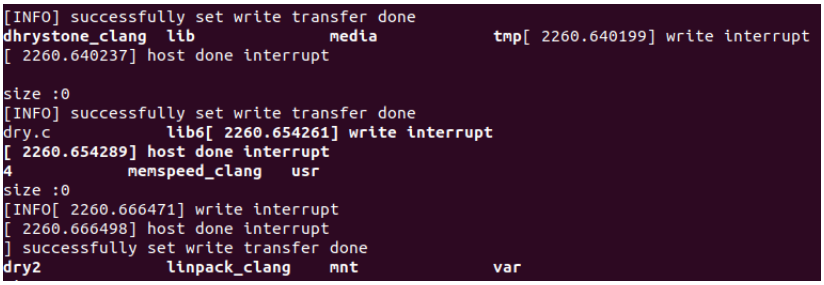


Figura 7.7: Accelerator execution validation

7.3 Performance Tests

This section will evaluate the performance of the developed solution against an x86_64 PC that resorts to full-emulation, but with greater computational power than the Accelerator. The specifications of the PC that uses full-emulation are shown in table 7.2.

Tabela 7.2: Full-emulation PC specifications

CPU	Intel® Core™ i7-10510U CPU @ 1.80GHz
RAM	DDR4 32GB
Storage	SSD 1TB
OS	Ubuntu 18.04.5 LTS with 5.4.0-122-generic Kernel

Following the same approach as the previous section, the performance tests are split into two groups:

- Start tests;
- Execution tests.

Start tests will evaluate the Guest Image's boot time, and execution tests will evaluate the system performance in terms of how long the system takes to execute CPU benchmarks. Furthermore, the tests will be applied to different approaches:

- Proposed solution;
- ARM without PCIe;
- Full-emulation Host.

The proposed solution represents the performance of our approach. The ARM without PCIe represents a solution based on the Accelerator of the chapter 4 with a Linux OS similar to the one at table 7.2 and is only used in the performance of the system execution. Lastly, the Full-emulation Host represents the solution based on an x86_64 PC mentioned earlier in this section that will resort to full-emulation when emulating the Guest image.

7.3.1 System Boot Performance

To measure the boot time with precision, we need to exit QEMU right after the boot ends, which cannot be achieved by a manual shutdown of the QEMU. Therefore a mechanism was

implemented within the QEMU source code that would stop it when the boot ended, including the `/init` process. Furthermore, a compile flag was added, `-enable-boot-perf`, so that the mechanism only actuates when the flag is set during the QEMU compilation. Listing 7.5 shows the mechanism implemented with the compile flag. The `buff` is a static buffer implemented in the proposed solution that stores the output values of the Guest Image. Therefore, the implementation follows a simple logic, if the image asks the user to insert the username, it must have ended the boot and executed the `/init` process so QEMU can exit.

```
1  static bool kernel_messages = true;
2  if(kernel_messages)
3      kernel_messages = !(strstr(buff,"buildroot login") != NULL);
4  #ifdef CONFIG_BOOT_PERF
5      if(!kernel_messages)
6      {
7          pcie_dma_write("quitt");
8          exit(0);
9      }
10 #endif
```

Listing 7.5: Mechanism that exits QEMU on the boot ending

As the need for human interaction to exit QEMU at boot ending is unnecessary, the remaining subsection will explore how we can effectively measure the boot time. Although the boot messages are displayed with a time stamp, this cannot be trusted because the emulated CPU and the real CPU could have different clock cycles. Consequently, the best way to measure the boot time is to measure the execution time of the QEMU command when the previous mechanism is implemented.

Several commands are available to measure a command's execution, such as `diff` and `time`, but a script was developed to measure multiple instances and display the average time in the end. The script is illustrated at listing 7.6 where line 10 represents the path to the QEMU execution script, listing 7.1, and in this particular situation, the script measures 25 instances of the QEMU boot.

```
1  #!/bin/bash
2
3  total=0
4  x=1
5  loop_times=25
6
7  while [ $x -le $loop_times ]
8  do
9    start_time=$(date +%s.%3N)
10   /home/andre/start_vm_aarch64.sh
```

```
  end_time=$(date +%s.%3N)
  elapsed=$(echo "scale=3; $end_time - $start_time" | bc)
  total=`echo $total + $elapsed | bc`
  (( x++ ))
done
echo "total:" $total
median=$(echo "scale=3; $total / $loop_times" | bc)
echo "average:" $median
```

Listing 7.6: Script to measure the boot time

Table 7.3 represent the average of the measured times on the system developed and the opposite PC approach. The developed system has three measures associated:

- System Initialisation;
- Boot;
- Total.

The System Initialisation represents the time the developed system takes in the initialisation steps specified in section 6.2.2, installing the driver modules and transferring the arguments to the Accelerator and Guest Image. The Boot measures indicate the time the Accelerator took to boot the Guest Image, which also considers the overhead of the data transfers through PCIe. Finally, the Total entry exposes the total time the developed system takes in the booting step.

Tabela 7.3: Boot performance

	Proposed solution			**Full-emulation Host**
	System Initalisation	Boot	Total	
Average time (s)	1,73	4.683	6.413	6.229

Due to the extra initialisation steps, it is expected that the total booting time of the developed system may take a longer time than the opposing solution. Nevertheless, the Accelerator should boot the Guest Image faster than the full-emulation approach. As stated in table 7.3, the Accelerator does improve the booting time of the Guest Image due to the KVM use. However, the overhead of the system initialisation significantly influences the total boot time, leading to worse performance of the proposed solution against the opposite approach. Although the total time taken by the proposed solution is not far different from the full-emulation approach, the developed solution can still be desirable depending on the results of the execution performance exposed in the following subsection.

7.3.2 System Execution Performance

The system execution performance is measured using predefined CPU benchmarks. For that reason, it is expected that when executing the CPU benchmarks on the emulated Guest Image, QEMU will benefit from full-virtualisation instead of full-emulation and achieve better performances. The first CPU benchmark chosen was the Dhrystone Benchmark [37], but the newest C version of it [38], 2.2a, due to it being simple and easy to understand. Although, it mainly focuses on integer performance and lacks floating-point operations. With this in mind, the second CPU benchmark chosen was the Livermore loops [39], which is also simple and easy to understand and heavily focuses on floating-point operations, compensating for the last benchmark.

Even though the benchmarks display the execution time, they are not to be trusted because, for the same reason, the boot timestamps could not be. Considering that these benchmarks will run on the emulated Guest system and not on the Accelerator itself. Therefore, to obtain a reasonable execution time, this needs to be calculated on the user level of the Accelerator and not on the Guest image being emulated. Although, as stated before, the QEMU software is running on the user level, it is possible to calculate the execution time on the QEMU part that is executing on the user level, but that also has access to the internal processing of the Guest Image being emulated. Accordingly, the file `channel-file.c` of QEMU was once again modified to contain the execution time calculation of a given command.

Tabela 7.4: Performance of the different approaches on distinct benchmarks

	Full-emulation Host	**Proposed solution**	**ARM without PCIe**
Dhrystone Benchmark (s)	43.774	20.536 (46.9 %)	20.056
Livermore loops (s)	103.131	81.200 (78.7 %)	80.833

Table 7.4 displays the performance measures of each benchmark for the different approaches. Furthermore, the proposed solution measures consider the PCIe overhead of the `printf` calls that the benchmarks implement. Even though the ARM without PCIe refers to the Accelerator used on the developed system, the measures are only taken within the Accelerator without being connected to a Host. Therefore, it only measures the execution time on the Accelerator, which means it withdraws the PCIe overhead.

Firstly, as expected, the ARM without PCIe measures are better than the proposed solution measures, but the performance cost is less than 2.4% approximately. Therefore, the proposed solution has almost the same performance as the standalone version of the Accelerator, which implies that the overhead of the PCIe transaction can be neglected. The almost native performance is since the QEMU PCIe transactions developed occur in parallel with the emulation of the Guest Image, which does not affect the execution of the benchmark.

Lastly, the proposed solution performance is higher than the commonly used full-emulation, which was the primary goal. However, these results may not validate the proposed solution as

desirable, but it makes a solid case on the potential of the system developed to reach higher performances.

Capítulo 8

Conclusion and Future Work

This chapter will summarize the contents and conclusions from the previous chapters and provide the final answer about the worth of developing a more complex solution. Lastly, new ideas and new development approaches are exposed as future work.

8.1 Conclusion and Requirements Fulfilment

Virtualization is already well-established in server virtualization and cloud computing, but it is still being studied and developed in the domain of embedded systems. Among all virtualization approaches, hardware-assisted virtualization is the most suitable for the project due to its near-native performance, but this comes at the cost that the host and guest architecture need to be equal.

A solution that requires offloading the emulated system to an external accelerator with the same architecture as the Guest was conceived to solve this requirement. Offloading the emulated system means offloading the QEMU process to the Accelerator, and the solution proposes that two versions of QEMU are created, one for the Host and one for the Accelerator, that would interact with each other to transfer data regarding the Guest system and user inputs. Therefore this solution is hardly dependent on low-level software that belongs to the communication between both devices. The low-level software comprehends the Host driver and the accelerator driver, both available by Xilinx, but certain modifications were needed to fit them into our system. The Accelerator architecture developed unifies the architecture developed on the previous work [7] and the architectures found on Xilinx documentation [32, 36]. The developed system mainly benefits from the PCIe interface and the ARM Cortex A-53 CPU. However, the PCIe interface forces the Host to reboot, so the BIOS detect the PCIe devices exposed, which results in the third requirement defined in chapter 4 being impossible to validate.

The validation of the proposed solution is effectuated in chapter 7, and it is reckoned valid. Moreover, the proposed solution performance was tested and compared to the commonly used solution on cross-architecture emulation, full-emulation. Firstly, due to the initialisation steps,

our system takes a little longer to boot than the other solution. However, it bestows higher system execution performance, which validates the second requirement defined in chapter 4. The higher performances come from the heavy use of state-of-the-art virtualization software and the low overhead achieved with the PCIe interface, validating the first requirement defined in chapter 4.

This solution paves the way for the complex system to be developed, and the performance measurements point out the potential to reach higher performances. Consequently, it validates the effort to create a more complex system.

8.2 Future Work

The developed solution fulfils its purpose as a prototype of the complex system, representing the best scenario of it when the offloading happens at the system start-up. Therefore, future work will focus on moving the developed solution into a more dedicated acceleration solution that would not require the image to pass to the Accelerator. A solution that would consist of accelerating essential parts, such as hot spots during the execution of QEMU, which would require the offloading of the CPU instructions and memory management to the Accelerator.

To offload the CPU instruction and memory operations we need to have access to the instructions that QEMU executed. One way to solve this would be to develop a tracing engine for QEMU, such as Cannoli [40], that would record all the code flow and memory operations and then transfer everything to the Accelerator. However, this may lead to a significant decrease in system performance.

On the other side, future work could pursue higher performances by implementing new mechanisms or upgrading existing ones. On the Accelerator, improvements could be made on the PL. The current solution only implements one channel for each operation (read and write), but it could be implemented more to decrease the transfer times. Furthermore, the bypass functionality can be implemented to achieve higher performance.

Anexo A

QEMU Build Process

The QEMU software is a crucial tool used on the developed solution for emulating the Guest System. Although it exists as a package on most Linux system, the developed solution required modifications of its internal. Therefore, this appendix will expose the steps necessary to access the QEMU source code and compile it. Take into account that the steps and commands here stated are only guaranteed to work on a Linux OS and can be applied on both QEMU used, Accelerator QEMU and Host QEMU.

A.1 Retrieve QEMU Source Code

The QEMU source code can be downloaded using the git software, issuing the following command:

```
1 $ git clone https://gitlab.com/qemu-project/qemu.git
```

A.2 QEMU Configuration Process

The QEMU needs to be configured before proceeding to the building. The configuration command allows QEMU to know what it should build and what compile flags are activated or not. Bear in mind that the command should execute on the main QEMU directory. For that reason it is necessary to run the following command before the configuration command:

```
1 $ cd qemu
```

Once in the main directory the configuration command can be executed. The intended command for the configuration for this work is:

```
$ ./configure --target-list=aarch64-softmmu --enable-kvm
```

The `target-list` flag indicates QEMU that it only needs to build the resources for the specified system. The current work aims to have QEMU only emulating an arm64 Guest system. For that reason the flag is called with `aarch64-softmmu` to later build only the arm64 resources. Otherwise, QEMU would build every available architecture which would increase the building time.

The `enable-kvm` flag instructs QEMU to include the KVM API to the build procedure, which allows QEMU to resort to virtualization instead of full-emulation.

Furthermore, the first configuration may fail due to it requiring additional packages that are not usually installed. The following command install the mandatory packages that are uncommon to be installed:

```
$ sudo apt-get install libglib2.0-dev libfdt-dev libpixman-1-dev zlib1g-dev ninja-
    build
```

Breaking down the packages installed:

- **libglib2.0-dev**: Install the development files for the Glib library, which provides the core libraries for Linux systems;
- **libfdt-dev**: Install the development files for the libfdt library, which contains functions to manipulate Flat Device Trees;
- **libpixman-1-dev**: Install the development files for the Cairo/X pixman library, which is a low-level library for pixel manipulation;
- **zlib1g-dev**: Install the development files for the compression library zlib1g, which implements gzip and PKZIP compression methods. Take into account that the package **libglib2.0-dev** automatically includes it, so it does not need to be included in the command. Although it is represented to highlight that QEMU requires it;
- **ninja-build**: Install the building system used by recent versions of QEMU.

A.3 QEMU Building Process

At the completion of the running command the QEMU software is ready to build the QEMU's binary by executing the following command:

```
$ make -j 8
```

The `-j` allows the execution of many recipes simultaneously improving the build time. The number associated should not be higher than the number of CPU available. Otherwise the result would be unfavorable.

Anexo B

Building a custom OS using Petalinux

The following steps assume that the following prerequisites have been satisfied:

- The OS used is Linux;
- The Board Support Package (BSP) for the Accelerator board is downloaded;
- The setup of the Petalinux working environment is completed.

B.1 Create a Petalinux Project using the BSP

The first step to create a custom OS with Petalinux is creating a Petalinux project using the BSP. The creation is achieved with the following command:

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_BSP> -n <PROJECT_NAME>
```

B.2 Hardware Configuration

The Petalinux has several configuration steps. The first of them is the hardware configuration, which enables the build of a Linux system, customized to a certain hardware platform specified. Bear in mind that the configuration commands need to be run in the project directory. Therefore, before executing any configuration command, the following command needs to be executed to enter the project directory:

```
$ cd <PATH_TO_PETALINUX_PROJECT>
```

The hardware configuration command requires the exported hardware platform (.xsa file) to proceed with the configuration. For that reason the path needs to be passed to the hardware configuration command:

```
$ petalinux-config --get-hw-description=<PATH-TO-XSA>
```

This command will launch a top system configuration menu. The modification required on this menu are the changing of the format specified for the Boot and Kernel Image and the Root filesystem type. Furthermore, the Ethernet settings are also modified to allow the Host to connect via SSH to the Accelerator, which improved the development experience. The modifications can be done under the following settings:

- Subsystem AUTO Hardware Settings -> Advanced bootable images storage Settings -> boot image settings -> image storage media -> primary sd
- Subsystem AUTO Hardware Settings -> Advanced bootable images storage Settings -> kernel image settings -> image storage media -> primary sd
- Subsystem AUTO Hardware Settings -> Ethernet Settings -> Obtain IP address automatically (remove the check)
  - Define the Static IP's required, for example:
    - (192.168.1.6) Static IP address;
    - (255.255.255.0) Static IP netmak;
    - (192.168.1.1) Static IP gateway.
- Image Packaging Configuration -> Root file system type -> EXT4 (SD/eMMC/SATA/USB)

After the modifications save and exit the menu.

B.3 Kernel Configuration

Petalinux offers the opportunity to change some Kernel configurations. The developed work makes use of this to enable the virtualization extensions, such as, KVM and make sure that the PCIe driver is installed. The Kernel configuration menu opens after executing the following command:

```
$ petalinux-config -c kernel
```

Therefore the only options needed to check are:

- Virtualization -> Kernel-based Virtual Machine (KVM) support
- Virtualization -> virtIO
- Kernel Configuration -> Device Drivers -> Misc devices

These option are the main reason behind the performance success of the developed solution, allowing QEMU to use the KVM capabilities on the Accelerator.

B.4 Root Filesystem Configuration

The last configuration step refers to the filesystem configuration, which does modification at user space level. Therefore the packages used within the Linux OS are added here. In order to do this Petalinux uses the recipes provided by Yocto to install the desired packages. The following command will open the top level menu configuration:

```
$ petalinux-config -c kernel
```

The desired packages can be added under the **user_packages ->** option. Take into account if the package does not appear under the option previously mentioned it needs to be added manually to the root filesystem configuration.

B.4.1 Adding Existing Packages

If the desired package is missing from the menu configuration, the first step to take is check if the package exists within the Petalinux project. In order to the following command should be executed:

```
$ find components/yocto/layers -name QEMU
```

The command looks for QEMU package under the yocto layers. The command should return the path to the packages, but if no path is shown it means that the package does not exist in the current layer and the package cannot be added automatically. Therefore, other layer should be added before executing this command again. Although during the development this action was not necessary as the layer had every package needed. Furthermore, the output produced by the previous command should be similar to:

```
components/yocto/layers/core/meta/recipes-devtools/qemu
components/yocto/layers/core/meta/recipes-devtools/qemu/qemu
components/yocto/layers/meta-xilinx/meta-xilinx-bsp/recipes-devtools/qemu
```

Therefore the qemu package can be added to the project. The following step is modifying the **user-rootfsconfig** file. This could be done by manually editing the file or executing the following command:

```
$ echo "CONFIG_qemu" >> project-spec/meta-user/conf/user-rootfsconfig
```

Lastly, the package will now appear on the root filesystem configuration menu and can be selected under the **user_packages ->** option. This process should be repeated for the following packages:

- dpkg;
- meson;
- ninja.

B.5 Building and Packaging the Image

Once the configuration step is completed the Image is ready to be built. The following command starts the Image building:

```
$ petalinux-build
```

This step may take some time to complete. On the completion of it , the Image is ready to be packaged, generating the files to boot on the Accelerator board and the root file system compressed file. The following command starts the Image packaging:

```
$ petalinux-package --boot --fsbl images/linux/zynqmp_fsbl.elf --fpga images/linux/
system.bit --pmufw images/linux/pmufw.elf --atf images/linux/bl31.elf --u-boot
images/linux/u-boot.elf
```

B.6 Copying Image Files to SD Card

This section explains what files need to be copy to an SD card and how it can be done. Therefore, this section assumes that the SD card has 2 partitions:

- **BOOT** - should be formatted as a FAT32 file system and at least have more than 500MB in size;
- **RootFS** - should be formatted as an EXT4 file system and can claim the rest of the SD card size.

Furthermore, it is assumed that they are mounted on `/media/BOOT` and `/media/RootFS`. The first partition will contain the boot loader, device tree and the kernel image. For that reason the following commands copy the necessary files to the BOOT partition:

```
1 $ cp images/linux/BOOT.BIN /media/BOOT/
2 $ cp images/linux/image.ub /media/BOOT/
3 $ cp images/linux/boot.scr /media/BOOT/
```

On the other hand, the second partition will hold the root file system. Therefore, the following command uncompressed the generated compressed file system to the second partition:

```
1 $ sudo tar xvf rootfs.tar.gz -C /media/rootfs
```

Bear in mind that the RootFS should be erased before copying another root file system. Otherwise data corruption may occur.

Anexo C

Building a Guest Image

The Guest Image is going to be emulated by QEMU software and should be an Linux Kernel Image of a Arm64 system. Additionally it should have already the benchmarks binaries on its root filesystem. The following creation and building steps are only guaranteed to work on a Linux OS.

C.1 Creating and Building the INITRAMFS

The Initial RAM file system (INITRAMFS) is the newest replacement for the INITRD, it is a `cpio` archive file that is mounted by the Linux Kernel as RootFS and stars the initialisation process. To create and build it the buildroot sofware is used. Therefore the buildroot software can be installed by executing the following command:

```
$ git clone https://github.com/buildroot/buildroot.git
```

At the download completion the following steps admit that they are executed on the buildroot root directory. For that reason the following command to enter the buildroot directory should be the first to be executed:

```
$ cd buildroot
```

As stated before, the INITRAMFS needs to have the benchmarks binaries that will test the system developed. For that reason a directory is created on the buildroot root directory using the following command:

```
$ mkdir buildroot_files
```

The binaries files are then copied into the `buildroot_files` directory. Subsequently, the following command can be executed at the buildroot root directory to open the top level menu:

```
$ make menuconfig
```

On the menu the following options need to be set to achieve the desired root file system for the guest Image:

- Target options -> Target Architecture -> AArch64 (little endian)
- Toolchain type -> External toolchain
  - ***Toolchain External Options***
    - Toolchain (ARM AArch64 <version>)
- System Configuration -> Enable root login with password
- System Configuration -> Run a getty (login prompt) after boot -> TTY port -> "ttyAMA0"
- Target packages -> Show packages that are also provided by busybox
- Target packages -> Debugging, profiling and benchmark -> strace
- Filesystem images -> cpio the root filesystem (for use as an initial RAM filesystem)
- System configuration -> Root filesystem overlay directories -> "/home/andre/buildroot/buildroot_files"

After checking the previous options save and exit the menu. The INITRAMFS is now ready to be built. The following command starts building the INITRAMFS:

```
$ make
```

The `cpio` file will be placed at **/output/images** at the end of the building. Take into account that this path will be used later when configuring the Linux Kernel.

C.2 Compile a Linux Kernel for ARM64

The last step to achieve the Guest Image is creating the Linux Kernel Image that implements the INITRAMFS created previously. Firstly, the following command download the Linux Kernel source code from the Linux Mainline kernel development source:

```
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

At the download completion the following steps admit that they are executed on the Linux root directory. For that reason the following command to enter the Linux directory should be the first to be executed:

```
$ cd linux
```

As the Guest Image is suppose to have a ARM64 architecture the default configuration of the ARM64 is loaded into the Kernel configuration using the following command:

```
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make defconfig
```

After the completion of the previous command the Linux Kernel Image will have the default configurations for creating a Image for the Arm64 architecture. Therefore, most of the options are already selected and just a few need to be manually selected. To manually select the desired options without resorting to the top level menu, the following command is issued to access the `.config` file directly:

```
$ nano .config
```

During the file edition the following modifications must be done:

- CONFIG_CMDLINE="console=ttyAMA0"
- CONFIG_INITRAMFS_SOURCE="/home/andre/buildroot/output/images/rootfs.cpio.gz"

After the modifications save and exit the file. Lastly, the Linux Kernel is ready to generate the desired Guest System by executing the following command:

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j 8
```

At the completion of the previous command the Guest Image will be placed at **/arch/arm64/boot/** with `Image` as its name.

Referências

[1] Humble Devassy Chirammal; Prasad Mukhedkar; Anil Vettathu. *Mastering KVM Virtualization*. Packt Publishing, 1st edição, 2016.

[2] W.J. Dally, Y. Turakhia, e S. Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020.

[3] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[4] H. Esmaeilzadehy, E. Blemz, R. St. Amantx, K. Sankaralingamz, e D. Burgerfi. Dark silicon and the end of multicore scaling. páginas 365–376, 2011.

[5] T.N. Theis e H.-S. Philip Wong. The end of Moore's law: A new beginning for information technology. *Computing in Science and Engineering*, 19(2):41–50, 2017.

[6] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, e D. Marr. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. páginas 77–84, 2017.

[7] J. Pedro Vigário Garrido. *Study and Implementation of a Hardware Cross Architecture Virtualization Accelerator*. Master's thesis, University of Porto, School of Engineering, 2021.

[8] Pedro Sousa Gonçalves. *A QEMU-based Approach to Hardware-Assisted Virtualization*. Master's thesis, University of Porto, School of Engineering, 2021.

[9] F.A. Khaji, S.V. Potluri, e A.K. Kakelli. A performance analysis of virtual mail server on type-2 hypervisors. *Walailak Journal of Science and Technology*, 18(13), 2021.

[10] James E. Smith e Ravi Nair. Chapter two - emulation: Interpretation and binary translation. Em James E. Smith e Ravi Nair, editores, *Virtual Machines*, The Morgan Kaufmann Series in Computer Architecture and Design, páginas 27–82. Morgan Kaufmann, Burlington, 2005.

[11] F. Bellard. QEMU, a fast and portable dynamic translator. páginas 41–46, 2005.

[12] X. Bian. Implement a virtual development platform based on QEMU. Em *Proceedings - 2017 International Conference on Green Informatics, ICGI 2017*, páginas 93–97.

[13] T. Spink, H. Wagstaff, e B. Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization*, 13(4), 2016.

[14] Fatma Bazargan, Chan Yeun, e Jamal Zemerly. State-of-the-art of virtualization, its security threats and deployment models. *International Journal for Information Security Research*, 3, 2013.

[15] P. Babu e E. Parthasarathy. Reconfigurable FPGA architectures: A survey and applications. *Journal of The Institution of Engineers (India): Series B*, 102(1):143–156, 2021.

[16] S. Yu. Neuro-Inspired Computing with Emerging Nonvolatile Memorys. *Proceedings of the IEEE*, 106(2):260–285, 2018.

[17] T. Robinson, J. Harkin, e P. Shukla. Hardware acceleration of genomics data analysis: Challenges and opportunities. *Bioinformatics*, 37(13):1785–1795, 2021.

[18] Junjie Hou, Yongxin Zhu, Sen Du, e Shijin Song. Design and implementation of reconfigurable acceleration for in-memory distributed big data computing. *Future Generation Computer Systems*, 92:68–75, 2019.

[19] I. Kuon e J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[20] Zhangqin Huang, Shuo Zhang, Han Gao, Xiaobo Zhang, e Shengqi Yang. A configurable multiplex data transfer model for asynchronous and heterogeneous FPGA accelerators on single DMA device. *Microprocessors and Microsystems*, 77:103174, 2020.

[21] W. Wang, M. Bolic, e J. Parri. Pvfpga: Paravirtualising an FPGA-based hardware accelerator towards general purpose computing. *International Journal of High Performance Computing and Networking*, 10(3):179–193, 2017.

[22] O. Beaumont, L. Eyraud-Dubois, e A. Shilova. Optimal GPU-CPU offloading strategies for deep neural network training. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12247 LNCS:151–166, 2020.

[23] Z. Wu, K. Hammad, E. Ghafar-Zadeh, e S. Magierowski. FPGA-Accelerated 3rd generation DNA sequencing. *IEEE Transactions on Biomedical Circuits and Systems*, 14(1):65–74, 2020.

[24] M. Jacobsen, D. Richmond, M. Hogains, e R. Kastner. Riffa 2.1: A reusable integration framework for FPGA accelerators. *ACM Transactions on Reconfigurable Technology and Systems*, 8(4), 2015.

[25] S.M. Neuman, B. Plancher, T. Bourgeat, T. Tambe, S. Devadas, e V.J. Reddi. Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology. páginas 674–686, 2021.

[26] Myron King, Jamey Hicks, e John Ankcorn. Software-driven hardware development. Em *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, página 13–22. Association for Computing Machinery, 2015.

[27] Jun Heo, Jaeyeon Won, Yejin Lee, Shivam Bharuka, Jaeyoung Jang, Tae Jun Ham, e Jae W. Lee. IIU: Specialized architecture for inverted index search. Em *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, página 1233–1245. Association for Computing Machinery, 2020.

[28] D. Granhão e J.C. Ferreira. Transparent control flow transfer between CPU and accelerators for HCP. *Electronics (Switzerland)*, 10(4):1–12, 2021.

[29] L. Sommer, J. Korinth, e A. Koch. OpenMP device offloading to FPGA accelerators. páginas 201–205, 2017.

[30] J. Korinth, D. De La Chevallerie, e A. Koch. An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. páginas 195–198, 2015.

[31] Xilinx. *UG1244 - ZCU106 Evaluation Board User Guide*, Outubro 2019.

[32] Ronak Shah. Zynq UltraScale+ MPSoC VCU TRD 2020.2 - PCIe. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1010368833/Zynq+UltraScale+MPSoC+VCU+TRD+2020.2+-+PCIe, 2020. Accessed: 2022-05-13.

[33] Xilinx. *UG1085 - Zynq Ultrascale+ Technical Reference Manual*, Dezembro 2020.

[34] Xilinx. *PG201 - Zynq UltraScale+ MPSoC Processing System Product Guide*, Novembro 2015.

[35] Xilinx. *PG195 - DMA/Bridge Subsystem for PCI Express v4.1 Product Guide*, Junho 2022.

[36] Xilinx. *UG1250 - Zynq UltraScale+ MPSoC ZCU106 Video Codec Unit Targeted Reference Design User Guide*, Outubro 2019.

[37] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, oct 1984.

[38] R. P. Weicker. Dhrystone benchmark: Rationale for version 2 and measurement rules. *SIGPLAN Not.*, 23(8):49–62, aug 1988.

[39] Frank H. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. 1986.

[40] Brandon Falk. CANNOLI: THE FAST QEMU TRACER. https://margin.re/blog/cannoli-the-fast-qemu-tracer.aspx, Maio 2022. Accessed: 2022-06-25.