Bachelor's Degree Final Project

Virtualization of programmable switches on top of an FPGA board

Albert Vilardell Barnosell

Supervised by:
Eric Keller[1]
Tamara Silbergleit Lehman[1]
Jordi Guitart[2]
Joan Sardà[3]

Boulder, United States
Defended on June 30, 2022

Within the framework of:

Bachelor Degree in Informatics Engineering — Information Technology specialization

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

UNIVERSITY OF COLORADO BOULDER (CU Boulder)

[1]Electrical, Computer & Energy Engineering - University of Colorado Boulder
[2]Departament d'Arquitectura de Computadors - Universitat Politècnica de Catalunya
[3]Departament d'Organització d'Empreses - Universitat Politècnica de Catalunya

Abstract

Network traffic has greatly increased in the last few years. Processing large amounts of data without compromising the system's performance is a challenge.

A powerful technology to manage network traffic is XDP which, combined with eBPF programs, offers the possibility to handle network packets before they are processed by the Linux network stack. Controlling data at a low level makes these types of programs ideal to be offloaded to an FPGA board. However, the sequential behavior of eBPF programs makes the synthesis process tricky.

This project presents a solution to offload eBPF programs to FPGA boards, getting benefits like scalability and security. The resulting design is loaded in the NetFPGA-SUME. Furthermore, an open-source library that contains optimized common eBPF functions has been developed in order to make the process of synthesizing eBPF programs easier for anyone interested in implementing this design.

The design has been compared to hXDP, the main existing solution to offload eBPF programs that consists of an eBPF soft-processor developed in the NetFPGA-SUME. The result of taking full advantage of hardware acceleration is a speedup between 3.72x and 26.04x.

Keywords: XDP, eBPF, HLS, FPGA

Resumen

El tráfico en la red ha aumentado considerablemente en los últimos años. Procesar grandes cantidades sin comprometer el rendimiento del sistema es todo un reto.

Una tecnología muy poderosa para administrar el tráfico de la red es XDP que, combinada con programas eBPF, ofrece la posibilidad de gestionar los paquetes antes de que estos sean procesados por la pila de red de Linux. Tratar datos a bajo nivel hace que este tipo de programas sean ideales para ser descargados a una placa FPGA. Sin embargo, el comportamiento secuencial de los programas eBPF hace que el proceso de síntesis sea delicado.

Este proyecto presenta una solución para descargar programas eBPF a una placa FPGA, obteniendo así beneficios como escalabilidad y seguridad. El diseño resultante es cargado a la NetFPGA-SUME. Además, se ha desarrollado una librería de código abierto que contiene funciones eBPF comunes optimizadas, para facilitar el proceso de síntesis de los programas para quien quiera implementar este diseño.

Este diseño ha sido comparado con hXDP, la principal solución existente para descargar programas eBPF, i que consiste en un procesador de instrucciones eBPF en la NetFPGA-SUME. El resultado de aprovechar al máximo la aceleración hardware es una ganancia de entre el 3.72x y el 26.04x.

Palabras clave: XDP, eBPF, HLS, FPGA

It is with genuine gratitude and warm regard that I dedicate this work to Laia and to my parents who, without understanding a single word of my research even after explaining it to them a million times, still managed to help and support me throughout this project.

Acknowledgements

I would like to express my thanks of gratitude to Professor Eric Keller and Professor Tamara Lehman for giving me the opportunity to work on this project and for guiding me in the right way. Their immense knowledge has helped me overcome every obstacle and give the best of myself.

I am also thankful to Professor Jordi Guitart and Professor Joan Sardà for helping me to fulfill all the formal requirements of a TFG.

Contents

List of Figures . . . **xii**

List of Tables . . . **xiv**

List of Abbreviations . . . **xv**

1 Introduction . . . **1**
- 1.1 Terms and concepts . . . 1
- 1.2 Structure of the document . . . 2

2 Context of the project . . . **4**
- 2.1 General contextualization . . . 4
- 2.2 Problem to be resolved . . . 4
  - 2.2.1 Stakeholders . . . 4
- 2.3 State of the art . . . 5
  - 2.3.1 hBPF . . . 5
  - 2.3.2 hXDP . . . 5
  - 2.3.3 Netronome . . . 6
- 2.4 Proposed solution . . . 6
- 2.5 Analysis of alternative solutions . . . 7

3 Scope of the project . . . **9**
- 3.1 Scope . . . 9
  - 3.1.1 Objectives and sub-objectives of the project . . . 9
  - 3.1.2 Functional and non-functional requirements . . . 9
  - 3.1.3 Other objectives . . . 10
  - 3.1.4 Potential obstacles and risks . . . 10
- 3.2 Methodology and rigor . . . 11
  - 3.2.1 Methodology . . . 11
  - 3.2.2 Monitoring tools . . . 11
  - 3.2.3 Validation methods . . . 12
- 3.3 Changes to the initial methodology . . . 12
- 3.4 Integration of knowledge . . . 13
- 3.5 Laws and regulations . . . 14

4 Time planning . . . **17**
- 4.1 Project duration . . . 17

4.2 Description of tasks . . . 17
4.3 Gantt chart . . . 24
4.4 Risk management . . . 26
4.5 Modifications from the original work plan . . . 27

5 Budget . . . 31
5.1 Identification and estimation of costs . . . 31
5.1.1 Human resources . . . 31
5.1.2 Generic costs . . . 34
5.1.3 Total costs . . . 37
5.2 Management control . . . 37
5.3 Impact of the modifications from the original work plan . . . 39

6 Background . . . 42
6.1 FPGA . . . 42
6.1.1 NetFPGA-SUME . . . 43
6.2 eBPF and XDP . . . 44
6.3 Hardware Modeling . . . 47
6.4 PyMTL3 . . . 48
6.5 Vitis HLS . . . 50
6.5.1 Xilinx . . . 50
6.5.2 The tool . . . 50
6.5.3 Synthesize process . . . 51
6.5.4 Metrics . . . 52
6.5.5 Bug . . . 53
6.5.6 Pragmas and directives . . . 54

7 Design and implementation . . . 56
7.1 Code rewriting to make it synthesizable . . . 56
7.1.1 Vitis unsupported C/C++ Constructs . . . 56
7.1.2 Required transformations of the code . . . 58
7.2 Code arrangement . . . 63
7.3 Vitis HLS interfaces . . . 63
7.3.1 Memory . . . 64
7.3.2 Streams . . . 64
7.3.3 Registers . . . 65
7.3.4 Default interfaces . . . 65
7.3.5 The interface pragma . . . 66
7.3.6 Best interfaces for eBPF programs . . . 67

7.4 Vitis HLS optimizations . . . 67
7.4.1 Loop unrolling . . . 67
7.4.2 Pipelining . . . 69
7.4.3 Array partitioning . . . 70
7.4.4 Transformation of the code . . . 71
7.5 Loading eBPF programs in the NetFPGA-SUME . . . 77
7.5.1 Vivado . . . 77
7.5.2 Base design . . . 78
7.5.3 Adapters for eBPF programs . . . 79
7.5.4 Validating the design . . . 81

8 Performance evaluation . . . 83
8.1 Methodology . . . 83
8.2 Impact of the optimizations . . . 85
8.3 Comparison with hXDP . . . 87
8.4 Performance of the loaded IP . . . 89

9 Sustainability and social commitment . . . 91
9.1 Environmental dimension . . . 91
9.2 Economic dimension . . . 93
9.3 Social dimension . . . 94

10 Conclusions . . . 97
10.1 Future work . . . 98
10.2 Completion of technical competencies . . . 98

Bibliography . . . 101

Appendix A. Example of an eBPF program . . . 109

Appendix B. Device support in Vivado in different licenses . . . 114

Appendix C. Creation of a Bitfile from a Vitis project . . . 116

Appendix D. Loading and testing a Bitfile in the NetFPGA-SUME . . . 118

List of Figures

1 Gantt chart . . . 25
2 Final Gantt chart . . . 30
3 Formula of the cost of an element used for a certain amount of time . . . 34
4 Human resources deviation formulas . . . 38
5 Amortization deviation formula . . . 38
6 Incidental cost deviation formula . . . 38
7 Total cost deviation formula . . . 38
8 Proportion of the total cost deviation formula . . . 39
9 Task ME3 deviation . . . 39
10 Task PP6.2 deviation . . . 39
11 Task PP7 deviation . . . 40
12 Task PP9 deviation . . . 40
13 Task FD1 deviation . . . 40
14 Total task deviation . . . 41
15 Final cost of the project . . . 41
16 The NetFPGA-SUME board . . . 44
17 PyMTL3 code that creates and configures a multiplexer . . . 49
18 Error that Vitis generated when trying to export the RTL Implementation . . . 54
19 Alias to execute Vitis HLS using another time . . . 54
20 Tag to remove syscalls from the synthesis process . . . 56
21 Example of typical pointer casting in eBPF programs . . . 57
22 Defined function to use printf only in C simulation . . . 58
23 Example of casting between different data types using unions . . . 62
24 Example of casting between 8-bit and 64-bit data types . . . 62
25 The interface pragma . . . 66
26 The unroll pragma . . . 68
27 The pipeline pragma . . . 69
28 The array_partition pragma . . . 71
29 Example of a loop with a variable bound . . . 72
30 Modified loop to enable pipelining the design . . . 72
31 Example of a conditional task . . . 74
32 Conditional task converted to unconditional . . . 74
33 Example of a program that writes an IPv4 header if there is one . . . 75
34 Example of hardcoding the access to an array . . . 75
35 Reading a map and increasing its value if it is diferent from zero . . . 76
36 Reducing the access to a map by using a local variable . . . 76

37 Storing an array using malloc only in the C simulation 77
38 Reference NIC block diagram 79
39 IP block of the loaded eBPF program 80
40 Integration of the IP block of the eBPF program in the design 81
41 Formula to calculate the throughput given the II 85
42 Speedup formula 85
43 Throughput of Linux's XDP programs in hXDP 88
44 PC consumption 91
45 Workstation consumption 91
46 Monitor consumption 92
47 Total consumption of the project 92
48 Example of an eBPF program that modifies the packet length by adding or removing a VLAN header 109
49 Function that parses an Ethernet and VLAN headers 110
50 Function that evaluates if the protocol of the next header is VLAN 111
51 Function that removes the outermost VLAN header 112
52 Function that adds a VLAN header 113
53 Diagram of the steps to "convert" an eBPF program to a bitfile 117
54 Initialization of the drivers 118
55 Initialization of the testing environment 119
56 Ping command to test the design of the board 119

List of Tables

1 Summary of the tasks . . . 23
2 Summary of risk management . . . 27
3 Roles of the project and its salaries . . . 32
4 Cost of each role for each task . . . 32
5 Total cost of the human resources . . . 34
6 Total cost of the hardware resources . . . 35
7 Total cost of the software resources . . . 35
8 Total cost of the general expenses . . . 36
9 Incidentals costs . . . 36
10 Total cost of the project . . . 37
11 Default interfaces in Vitis HLS . . . 66
12 Programs used to evaluate the performance of the project . . . 83
13 Performance of the XDP1 program . . . 86
14 Performance of the XDP2 program . . . 86
15 Performance of the XDP_ADJUST_TAIL program . . . 86
16 Summary of the performance of the tested programs optimized version . . . 87
17 Estimation of the throughput of Linux's programs in hXDP . . . 88
18 Comparison of the performance of the optimized programs with hXDP . . . 89
19 Performance of the loaded IP block . . . 90
20 Device support in Vivado ML Standard and in Vivado ML Enterprise . . . 114

List of Abbreviations

AXI4: Advanced eXtensible Interface 4 is an interface specification from ARM. Xilinx standardized this protocol for its boards [1].

BSD: Berkeley Source Distribution is a group of free code permissive licenses, which means that derived code can be licensed with any other license.

CL: Cycle-Level model defines a hardware implementation in an abstract way that cannot be directly synthesized.

CLB: Configurable Logic Blocks are a basic unit of an FPGA and they are a group of FF and LUTs.

CPU: Central Processing Unit is a processor. The fact that they are designed to be for general-purpose computing is especially relevant in this project.

DSP: Digital Signal Processing are blocks of an FPGA dedicated to arithmetic and logic operations.

DDR: Double Data Rate means that a bus is able to transfer data on both the rising and falling edge of the clock. In this report, this abbreviation is used to describe DDR RAM memory, which is a type of memory that has the properties of DDR.

eBPF: extended Berkeley Packet Filter is a technology that runs programs in kernel space that are loaded at runtime.

ELF: Executable and Linkable Format is a file format for executables, shared libraries, object code, etc.

FF: Flip-Flops are a fundamental element of FPGA, and they are used to store one bit of data.

FIB: Facultat d'Informàtica de Barcelona is the home school of the developer of the project.

FL: Functional-Level model defines a hardware implementation in an abstract way that cannot be directly synthesized.

FPGA: Field-Programmable Gate Array is a device in which its circuit can be redefined multiple times after manufacturing.

GPL: GNU General Public License is a group of free code restrictive licenses, meaning that work derived from another GPL licensed work must have a similar license.

GUI: Graphical User Interface manages the interaction between the system and the user through icons and visual elements. It is the opposite of a command-line interface.

HLS: High-Level Synthesis consists of converting a high-level description of hardware into actual hardware.

hXDP [2]: hardware eXpress Data Path is a project that has developed a solution to offload eBPF programs to the NetFPGA-SUME board.

ICMP [3]: Internet Control Message Protocol is a standard defined in RFC 792 that specifies a protocol designed to provide feedback about the status of Internet communications.

II: Initiation Interval is a metric used to evaluate how many clock cycles are taken between the start of a program execution and the start of the following one [4].

IP: Intellectual Property are the blocks used in an FPGA. They are an essential element of design reuse [5]. This abbreviation may be confused with the Internet Protocol, so this report will always use IP as Intellectual Property and IPv4 and IPv6 to refer to Internet Protocol.

IPv4 [6]: Internet Protocol version 4 is one of the core protocols of the Internet, used to transmit datagrams from a source to a destination. Version 4 is the original and active version of the protocol, which appears to be slowly replaced with version 6.

IPv6 [7]: Internet Protocol version 6 is an upgraded and newer version of IPv4. It is active but not used as much as IPv4. However, it is expected to replace it in the future.

LUT: Lookup Tables (LUT) are used to implement all the combinational logic of the FPGA as truth tables [8].

Mpps: Millions of packets per second is a metric used to evaluate the throughput of a network device.

NIC: Network Interface Card is a component that is added to a computer so that it can connect to a network.

PCIe: Peripheral Component Interconnect Express is a high-speed bus standard. It is an interface commonly used to connect the graphic card and the NIC to the motherboard.

RAM, BRAM, DRAM, URAM: Random-Access Memory, Block RAM, Dynamic RAM, Ultra RAM are different types of RAM, which is a type of memory that can be accessed in any order. The differences between them are not explained as it is not relevant to the project.

ROM: Read-Only Memory is a type of memory that can only be read. It is only written once, when it is manufactured.

RTL: Register-Transfer Level model defines a hardware implementation in an accurate way that can be derived to actual wiring.

SRL: Shift Register LUT are a unit of FPGAs, and they are an alternative to LUTs.

soft-CPU: soft-Central Processing Unit is a processor that is described using HLS. In other words, it is a CPU implemented in an FPGA.

tc: traffic control is a Linux tool used to manage the traffic from the network.

VLAN [9]: Virtual Local Area Network is a network protocol standardized under IEEE 802.1Q that is used to create isolated environments in Local Area Networks.

XDP [10]: eXpress Data Path is a technology that enables eBPF programs to run at a low point of the Linux network stack.

1 Introduction

Network traffic has increased at a compound annual rate of 29% during the last five years [11]. Data centers need a solution capable to handle the increase of bandwidth without compromising the whole system's performance.

The goal of this project is to propose a solution to this problem by virtualizing programmable switches using an FPGA board. The virtualization consists of offloading a task in a potentially virtual environment, eBPF programs take the part of programmable switches and the board used is the NetFPGA-SUME.

Offloading the processing of network packets to an FPGA board can potentially increase the speed of this task as it can use specific hardware that makes the programs run faster than a general-purpose CPU. Furthermore, the kernel would be free from doing those tasks, that would be completed in an isolated environment, adding a layer of security.

There is a project, hXDP [2], that has developed a solution to offload eBPF programs to the NetFPGA-SUME. However, the design is a soft-CPU that fails to take full advantage of hardware acceleration. Furthermore, it is completely dependent on the board. Both of these concerns are addressed in this project.

1.1 Terms and concepts

The reader should be familiarized with the following concepts in order to effectively follow this thesis.

eBPF:
The extended Berkeley Packet Filter (eBPF) is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner [12]. The code is loaded into the kernel from user space, so a kernel verifier evaluates the program before allowing it to go to kernel space to make sure that it is not harmful. The idea of eBPF is that it adds new capabilities to the Linux kernel at runtime [13].

XDP:
The eXpress Data Path (XDP) [10] is a technology that was designed to run eBPF programs in a device driver context. It avoids the overhead added by the kernel stack because the programs operate with the packet at the lowest point of the stack when they have not yet been processed.

XDP also defines when the eBPF program is executed. It is a technology that is incorporated into the Linux kernel.

FPGA:
A Field-Programmable Gate Array (FPGA) is a circuit designed to be configured multiple times after its production. This means that it can create almost any electrical circuit with the available resources of the board. Silicon-based hardware would require to be manufactured again in order to change its design.

PyMTL3:
PyMTL3 [14] is an open-source Python-based hardware generation, simulation, and verification framework. Instead of coding with a Hardware Description Language, Python is used to describe hardware, in a *user-friendly* way. This tool converts the Python code to Verilog, making it easier for a programmer to design hardware.

HLS:
High-Level Synthesis (HLS) consists of converting an abstract behavioral description of the hardware, which in this project is an eBPF program written in C, into an actual hardware model. This kind of model is known as Register-Transfer Level (RTL) Model, and it can be derived to actual wiring.

1.2 Structure of the document

The structure of the document allows a sequential reading of the report. It can be divided into three blocks.

The first part is related to the **management of the project**. Section 2 analyses the State of the Art of this solution and justifies the proposed solution. Section 3 discusses the objectives of this project and the methodology followed to achieve and validate them. Section 4 defines the different tasks that the development of this solution is divided into. Section 5 makes an economical estimation of the cost of the development of this project.

The second block is about the **development of the research**. Section 6 explains the research that was performed before starting to implement the solution. Section 7 implements the solution itself, which includes synthesizing the eBPF programs using Vitis HLS and loading them into the NetFPGA-SUME board.

The last phase is the **results**. Section 8 evaluates the performance of the synthesized eBPF programs and compares it to the main existing solution, hXDP. Section 9 examines the impact of this project in different dimensions. Section 10 concludes the document by presenting the conclusions and how the project could be expanded.

2 Context of the project

2.1 General contextualization

The Degree Final Project (TFG) is a mandatory subject included in the syllabus of the Bachelor's Degree in Informatics Engineering. The project covers a topic related to the specialization and has the goal of applying the acquired competencies during the degree.

The Europe-Colorado Mobility Program [15] granted me an opportunity to do research at the University of Colorado Boulder (CU Boulder). I am developing the project at the Department of Electrical, Computer & Energy Engineering. I have the pleasure of having Professor Eric Keller and Professor Tamara Lehman as my directors and my technical advisors at CU Boulder, and having Professor Jordi Guitart and Professor Joan Sardà as the generic advisors from my home school, the Facultat d'Informàtica de Barcelona (FIB). This thesis is within the scope of the Information Technology (IT) specialization.

2.2 Problem to be resolved

The use of the Internet does not stop growing, and Data centers receive more network traffic than they ever have. There is a need to process all of that information in a faster and more efficient way.

NetFPGA-SUME, an open-source type of FPGA board widely used in research, has the goal to eventually support network bandwidth of 100 Gbps [16]. This way, all of the network processing work would be transparently offloaded (and thus virtualized) from the kernel to the FPGA to enable hardware acceleration.

However, virtualizing low-level network functionalities in an efficient way is a challenge. This project seeks to achieve the goal by combining technologies like XDP and eBPF with an FPGA board

2.2.1 Stakeholders

The success of this project may benefit the following parties:

Data Centers:
This project is aimed at data centers as the solution could highly benefit them. I believe that

being able to offload network functionalities from the Linux kernel would greatly improve the performance of the data center.

Advisors of the project:
The four advisors of this project would benefit from its success. The directors are direct stakeholders, as the outcome of the research will be used to improve some other ongoing projects in which they are involved. The generic advisors are indirect stakeholders, as they are interested in the correct completion of the project.

Myself:
I am highly interested in developing an outstanding project as it is a great way to learn many new things, create a positive impact in the scientific community, to complete my Bachelor's Degree and, most importantly, enjoy working in the industry I am passionate about.

2.3 State of the art

The existing projects that relate the most to this thesis revolve around offloading network processing tasks to an external device using XDP or eBPF.

2.3.1 hBPF

hBPF [17] is an eBPF CPU in hardware, written in python. The aim of this project is to process network packets the same way eBPF does it. So the FPGA could become a form of smart NIC, offloading the tasks from the host CPU.

Even though hBPF implements most of eBPF features, the stack is not yet supported. It is not clear how eBPF maps are implemented, as it is not explained in the documentation.

2.3.2 hXDP

hXDP [2] is a project that has developed a system to run unmodified XDP programs in an FPGA board, the NetFPGA-SUME. It is a large project that has developed a new Instruction Set Architecture (ISA), a compiler, a CPU design and an FPGA infrastructure of maps and function helpers to fully support eBPF programs.

However, hXDP acts as a processor. It has been optimized to achieve the best possible performance, but it is limited as its hardware cannot accelerate eBPF programs. Though it does not require any modification of the code, thus being very straightforward to use.

This project is probably the one that relates to this one the most in terms of the end goal, which is offloading eBPF programs to an FPGA. But the solution that hXDP follows is different from the one that this project seeks.

2.3.3 Netronome

Netronome [18] is a company that develops networking solutions such as virtualization, security, hardware offloading, etc. They have created an infrastructure that allows virtualizing eBPF with XDP, and it only works with the drivers of the SmartNIC that they produce [19], [20].

When XDP is used to load a program, an operation mode has to be chosen. There are three supported modes, which are `xdpdrv`, `xdpoffload` and `xdpgeneric`. The second one offloads XDP to a SmartNIC. However, these SmartNIC require drivers that support it, and they are property of Netronome.

Netronome allows offloading eBPF only on SmartNICs, which are not re-programmable, and the hardware circuit of the network card can not be modified or adapted. FPGAs offer more flexibility, which can result in taking better advantage of hardware acceleration.

2.4 Proposed solution

This project is about accelerating common eBPF functions on an FPGA, as an open-source code in a library like a repository. It will extend the NetFPGA-SUME project to include the set of accelerated functions taking full advantage of the FPGA benefits. However, the resulting code will be independent of the NetFPGA-SUME project and applicable to any FPGA.

The reason to offload eBPF programs as the network functionalities is that eBPF is nearly "self-contained" and acts at a low level when it is combined with XDP [21]. A low-level manipulation of the packet is ideal to offload as it does not require any prior modification, which would end up adding an overhead. The wide programmability of eBPF makes it possible to implement many different network functionalities. Another advantage of eBPF is that it runs on kernel space, which is delicate. Offloading the program to the FPGA would add security as it would run on the board, which is away from the kernel environment.

The three existing solutions that have been mentioned in 2.3 State of the art have something in common: they have designed a type of CPU that receives an eBPF program as input and that is

afterward executed on an FPGA or a SmartNIC. This could be seen as an *eBPF general-purpose* CPU because the hardware that runs the eBPF programs is always the same for every different program. In other words, **these solutions fail to take full advantage of the hardware acceleration because the underlying architecture still acts as a processor and not as an accelerator**. This project is looking to accelerate eBPF programs using all of the potential an FPGA offers to design custom hardware. The outcome may be the same as the existing solutions - offloading eBPF programs - but the path taken to achieve it is different, and that may result in some final differences worth exploring.

The way to synthesize the eBPF programs will be using PyMTL3 or HLS. Both options could be valid to do the job. They will be exhaustively used and mastered to determine which one fits the goal of the project better.

The chosen FPGA board to load the programs is the NetFPGA-SUME as it is a powerful board (it has capabilities to support 100 Gbps traffic) and it has a large project behind it. The fact that there is a big infrastructure and community with open-source projects makes working with the board a better experience.

2.5 Analysis of alternative solutions

In relation to synthesizing eBPF programs, two different tools will be explored to determine which one is more suitable, PyMTL3 and Vitis HLS. Each one of them offers a different way to solve the problem, so exploring them to wisely choose the best option is very important. However, both PyMTL3 and Vitis HLS have alternatives in the market.

A substitute for PyMTL3 would be a Hardware Description Language (HDL). The two major ones are VHDL and Verilog. PyMTL3 transforms Python code into HDL, so directly programming in HDL would produce similar output. The reason why PyMTL3 was chosen is that it simplifies the job for both developing this project and for anyone else that may want to use it or extend it. Learning how to program in HDL would take a large amount of time which would make this project not viable because of its limited duration. Furthermore, every eBPF program to be offloaded has to be manually programmed, so it would take a lot more time to program them in HLS rather than in PyMTL3. And that could be a reason that could make this project less practical.

Vitis HLS is a type of High-Level Synthesis (HLS) tool. There are many vendors that offer HLS tools to synthesize C programs. Comprehensive research [22] was conducted to identify the different available options market. The one that was chosen was Vitis HLS, which belongs to a company named Xilinx [23]. The reasons are the following:

- **Great documentation:** Vitis HLS has a very detailed user guide [24] and it has many examples [25], [26]. This source of information speeds up the process of learning and solving potential problems with the tool.
- **Widely used:** There are many non-official tutorials, projects and books that use this tool like [27], [28], [29].
- **NetFPGA-SUME is manufactured by Xilinx:** The board that will be used in this project incorporates Xilinx Virtex-7 690T FPGA [30]. Vitis HLS is compatible with this board, and using the software from the company that develops the board ensures maximum compatibility.
- **Stable and under constant development:** Vitis HLS has a group of developers that are constantly updating and improving the tool. This characteristic is important as an unsupported tool may have more bugs.
- **Software family:** Xilinx has other popular software tools, like Vivado. Vitis HLS has an option to export the RTL design as blocks that can later be used by other tools of the same company. This portability inside the Xilinx environment is also a plus.
- **Previous work:** One of my advisors has previously worked with this tool, and he strongly believes that it is the best option.

However, in order to load the programs to the NetFPGA, Vivado has to be used to obtain the file that will be loaded into the FPGA, the bitfile. Using Vitis HLS, exporting the design and loading it in Vivado may be easier as both tools are developed by the same company, but it is possible to load designs to Vivado from third-party tools.

Regarding an alternative to the NetFPGA, no other hardware open-source platforms oriented to networking have been found. The NetFPGA project itself is quite unique, and finding a reasonable alternative outside of it is unlikely. The NetFPGA project offers another board apart from the NetFPGA-SUME, which is the NetFPGA-PLUS. It is newer and very similar to the NetFPGA-SUME, but this last was the newest in the market when it was acquired.

3 Scope of the project

3.1 Scope

3.1.1 Objectives and sub-objectives of the project

The main objective of the project is to document and evaluate how to offload the processing of network packets to an FPGA board and to gain performance thanks to using specific hardware.

Some sub-objectives are the following:

- Improve the execution of eBPF programs by using hardware acceleration.
- Integrate the synthesized eBPF programs to the FPGA design.
- Optimize the programs to obtain the best possible performance.
- Compare the performance of the optimized version against the original one.

3.1.2 Functional and non-functional requirements

Other functional requirements of this project are the following:

- Transform eBPF programs that are relevant and useful in order to synthesize them so that they can be used in other projects in the future.
- Determine which tool is better to synthesize eBPF programs: PyMTL3 or HLS.
- Always follow the same procedure when analyzing the performance to ensure that the comparison is under the same environment.

On the other hand, some non-functional requirements not mentioned already are:

- Have a high test coverage for each modification of the code to be able to easily find any error.
- Write code that can be easily reused and recycled to be as effective as possible.
- Learn how to use the text editor vi/vim.
- Learn how to use the version control git.

3.1.3 Other objectives

This project will be conducted in Boulder, in the United States. I have set some other goals to accomplish because of the environment I will be in:

- Improve my English skills.
- Learn about the American culture.
- Interact with other students of my directors (mostly PhDs) and learn about their research.

3.1.4 Potential obstacles and risks

Deadlocks:
One of the risks of this project is the deadlocks. Doing research implies trying out new things or ways to solve a problem. One can easily take the wrong path and waste some time without even realizing it. If it were to happen, it would negatively affect the timing of the project. The only solution to this is to carefully think about every step taken and to continuously evaluate the status of the project. The deep experience of the directors of the project will also be very important to avoid deadlocks.

Being unable to add modules to the NetFPGA design:
Some PhD students of my directors are currently working on adding modules to the NetFPGA design. They are confident they will accomplish it soon, but there is a chance they will not be able. My ability to load the eBPF programs to the FPGA depends on their development. I will simulate every eBPF program with Vitis, so loading it in a physical FPGA is not something strictly necessary to be able to successfully complete the project.

Inexperience in the field:
The topic and the tools used on the project are new for me, so there will be a chance I could need more time to adequately learn how to use these new tools. However, my directors and their PhD students have been working with them for quite some time, so I will be able to ask them any questions I have.

Bugs:
Almost every IT project can face this issue. Working with tools developed by someone else has this risk. If it were to happen, I would need to try to fix it myself, change the path or

even contact the developers of the tool. Consequently, I would need to readjust the planning accordingly.

Hardware incidents:
I will be using many physical tools that could be broken. The worst-case scenario would be if my computer stopped working. I would then need to buy a new one and install all the tools that are needed to develop the project. It is very unlikely to happen, and the most important thing, which is the data of the project, would not be lost. I plan on doing daily backups on an external USB flash drive and on Google Drive, which is on the Cloud. I will also upload some of the code to a GitHub repository. On the other hand, if my mouse, my keyboard or the external monitor stopped working, I would purchase a new one.

3.2 Methodology and rigor

3.2.1 Methodology

I believe that choosing a proper methodology is a very important factor that can positively affect the development of a project. The two options I have considered are the Waterfall and the Agile Model.

On the one hand, the Waterfall Model is a sequential method where each phase starts after the previous one has finished [31]. It is not very flexible but it ensures a strict order of tasks.

On the other hand, the Agile Model is an iterative way of developing a project. A phase is called "sprint", which has a fixed duration of time (usually some weeks) with a list of objectives planned at the start of the "sprint" [32]. These objectives are prioritized by their value. It has great flexibility.

The methodology that I have chosen is the Agile Model. I have decided to follow this model because the project is open and it could require making multiple decisions during its development. Furthermore, I will be learning about many different technologies that are new to me and I will need to master the basics about all of them to later be able to dive into each one of them. A Waterfall Model would not be optimal due to its inflexibility and its sequential approach.

3.2.2 Monitoring tools

The project will be monitored using a specific tool and via meetings with the advisors of the project.

The code control version that will be used is git. It will be combined with GitHub to have the code uploaded to the Cloud. The repository will be shared with the directors so that they can have another way to keep track that the objectives of the project are met. Some of the benefits of this tool are that a copy of the project is on the Cloud and that a list of the modifications will be saved sorted by time, so it is easy to track any potential error. However, I intend to only upload code that has been tested and that appears to have the expected behavior.

I will be on a weekly-meeting basis with my technical advisors with the objective of discussing the progress achieved every week. I believe that this is the best way to monitor the state of the project. At every meeting, I will explain the work done during the past week and how the objectives have been accomplished. I will also review the list of all the objectives of the project and I will mention the current state of each one of them. This way, we will realize the current state of the research and whether the objectives are being met or not.

Furthermore, I will be in touch with my generic advisors via email to ask them any questions that I will potentially have and to also update them about the current state of the project. This way, the requirements of the Degree Final Project will surely be met.

3.2.3 Validation methods

This project will perform incremental testing for the code. Every function of the library will have its own testing. The goal of this will always be to seek efficiency and correctness. Finding any errors as soon as possible will be helpful during the development of the project.

The synthesized code will also be simulated and tested to make sure that the RTL implementation also behaves as expected. Vitis HLS offers a C/RTL Cosimulation, where the RTL implementation is simulated with the user-defined C testing files.

One key aspect of the project is the performance. There are different metrics to analyze it, and it will be very important to be rigorous with the samples and the conclusions. The same procedure will be always followed to ensure to simulate the exact same condition when obtaining the results.

3.3 Changes to the initial methodology

There were no changes applied to the work methodology. However, a minor addition has been added to the monitoring tools, and the validation method has been enhanced.

Google Calendar has been used to establish deadlines and to write down the meetings and other tasks related to the project. It is a great tool for visually organizing and managing the different

jobs. It has helped in setting the weekly objectives. The idea to use Google Calendar came because the weekly meetings appointment through Zoom with the technical advisors was added to the calendar automatically, so I realized that it could be a useful tool and I decided to give it a try.

The original method to analyze the resulting performance of the project consisted of comparing the version of the program before it was optimized and after the optimizations were applied. This will remain unchanged, but a new competitor will be added: hXDP. As it was mentioned in 2.3 State of the art, hXDP is the most similar project to this one. So one way to complement the validation of the results of the project is to compare the achieved performance with the main current existing solution.

3.4 Integration of knowledge

This is the last project that I will develop for my bachelor's degree. It is the conclusion of a four-year academic experience. I started my degree in Informatics Engineering with *almost* no knowledge about the field. This project has been possible because of all of the competencies I have acquired through taking different subjects and doing research on my own.

The courses that I have taken - all of them at FIB - and that have been the most relevant for this project are discussed below.

Programming 1 (PRO1), Programming 2 (PRO2), Data Structures and Algorithms (EDA):
My basis of programming comes from these subjects. I have programmed a lot on this project, and both the usage of data structures and my coding style were mastered in these programming courses. Specifically, I have applied it when I have developed eBPF programs. Furthermore, eBPF programs are written in Restricted C, and these three subjects teach C++, so both languages are quite close to each other.

Computer Organization (EC), Computer Architecture (AC), Parallelism (PAR):
EC, AC and PAR gave me a general idea of how a computer works. Some concepts were directly applied to the project - like memory mapping, pipelining and parallelism - while others contributed to my background - like endianness. I have applied the most these concepts when I used Vitis HLS.

Introduction to Computers (IC):
This subject is not included in the set made of EC, AC and PAR as IC has provided an overview

of digital circuits, both the basic concepts and the design of one. When I explored the different options to approach how to synthesize eBPF programs, I had to either directly design digital circuits (in PyMTL3) or make direct decisions about the resulting digital circuit (in Vitis HLS and in Vivado). Both options required the contents of this course.

Operating Systems (SO), Advanced Operating Systems (SOA), Advanced Concepts on Operating Systems (CASO):
The subjects that I took that revolved around operating systems are these ones. They gave me a background about what an OS does and how it works, and I also did some OS programming on some of them. These competencies have been useful because eBPF programs run on kernel space, so special care has to be taken to ensure that it cannot damage such a critical part of the system.

Computer Networks (XC), Internet Protocols (PI):
XC and PI taught me about networking. I have applied multiple concepts and ideas to this project. For example, I developed an eBPF program that consisted of implementing network functionalities, like NAT or a firewall, which I had learned about in these courses.

Computer Network Technology (TXC):
An action that is common in eBPF programs to perform is parsing protocol headers. This requires an idea of how they are organized. These concepts were learned in TXC. However, it has been the first time I have programmed with these kinds of data structures, and applying the theoretical knowledge coding has been a very pleasant experience.

Free Software and Social Development (SLDS):
This is the only optional subject that I have included in this list. Topics like licensing and regulations were addressed in this course, and they have been applied in this project, especially in the following section.

3.5 Laws and regulations

There are no laws or regulations that directly affect this project. However, most of the software used is licensed. Moreover, this project will produce an open-source library of eBPF functions that has been properly licensed.

The programming language mainly used is C, which has an MIT license. It is a permissive license, and that allows the project to distribute C code freely. Redistributed code written in C will be able to have any license as the MIT license is not a copyleft one.

Regarding XDP and the tutorial that was followed (xdp-tutorial), they are licensed under GPL. It is a free-code restrictive license, which means that any work derived from it has to have the same license. I do not plan to include programs from the xdp-tutorial in the repository as I prefer writing my own programs. If I used modified programs from the xdp-tutorial, I would have to license it under GPL (or an equivalent license). As I have mentioned, I will not do it, so I will be free to license them as I like.

PyMTL3 has a BSD 3 license. It is a permissive license free to use. However, any work derived from it has to attach a license and copyright notice. But any Python code that uses PyMTL3 does not require attaching that information. It only applies to projects that modify the source code of the tool. So this license does not affect at all the project.

Regarding Vitis HLS, the working environment installed is the Xilinx Unified Installer 2021.2, which was the newest at the moment the thesis started. It requires a private license. Vitis HLS is a subset of the Vivado Design Suite, so the license that it needs is the one from Vivado Design Suite. There are two versions of Vivado, the Vivado ML Standard, which is free (it still requires a license, but it has no cost), and the Vivado ML Enterprise, which requires purchasing a commercial license. This project has been using the Enterprise model. The two differences between the standard and the enterprise version are the device support they have and that only the enterprise version supports a certain functionality of the environment, called RTL Implementation. This functionality allows the RTL Implementation to be exported so that it can be used in other tools also owned by Xilinx, and it also produces a report of the RTL Implementation performance, which is the most accurate one that Vitis HLS produces. Regarding the device support, the board that this project uses is exclusive from the Enterprise Version.

In relation to the NetFPGA project, the code is licensed in a BSD-style license [33]. It is a free license that permits to use, copy, modify, distribute, sell copies of the Software, etc. It only has two restrictions: the copyright notice has to always be attached and the names of the copyright holders may not be used to promote products derived from this software without prior permission.

The open-source library is licensed under GPLv2[1]. The reason is that some programs used to evaluate the performance of the project are samples of the Linux kernel GitHub tree [34]. They are licensed under this same license and, as they have been modified and published,

[1] The license can be found at: `https://github.com/aviba2000/eBPF_NetFPGA-SUME/blob/main/LICENSE.md`

they have to keep a similar license, as GPLv2 is restrictive. Furthermore, licensing the code also protects the developer of the project, as no Liability or Warranty is given about the code.

4 Time planning

4.1 Project duration

This project will start on Monday 17th of January 2022. It is a holiday in the US, Martin Luther King, Jr. Day. The campus will be closed, but my directors have kindly agreed to meet me online and start the project. The campus will be closed because of the impacts of COVID-19 Omicron variant and the Marshall Fire. However, it is expected to reopen soon.

The research will end on Friday 24th of June 2022. I will present my thesis during the following week, someday between the 27th of June and the 1st of July. I have booked my flight back to Barcelona on the 25th, so I will arrive on the evening of the 26th, one day before the first potential day to present this thesis. This is a total of 23 weeks to complete the project. I will be working roughly 40 hours per week, so that adds up to a total of 920 hours. I have divided the project into blocks and tasks, and I have estimated the amount of time each one of them will need. In total, I have assumed that the project will take 926.25 hours. It is six hours more than the 920 hours I have agreed to work, but I do not think that I will have any problem finding a spot to fit this extra time.

Both the start and the end date of the project are quite close to the final exams of the previous semester and to the presentation of the project. However, I decided to lengthen the project duration as much as I could to make the most of it.

4.2 Description of tasks

The project is divided into several blocks. All of them are identified by an abbreviation (a unique code that identifies each task) and they will be described and sorted in the chronological order of the start of each block.

Table 1 summarizes the tasks and can be found at the end of the section. Figure 1 is a Gantt chart that shows the project schedule.

Introduction:

This is going to be the first block of the project and it is named Introduction [INT]. I expect it to take place during the first week.

- **[INT1]** Initial Meeting (1h): In this first meeting with Professor Keller and Professor Lehman, we will discuss the topic of the project and they will update me regarding the

status of their current research. We will also talk about the working environment I will have on campus.

- **[INT2]** Reading related papers (39h): I will spend the rest of the week reading papers about the research topic. This task has INT1 as a dependency as I will need to know a little bit more about which is the topic of the research before reading about it.

The total amount of time planned to complete this block is 40 hours, one entire week of work.

The resources needed to complete this block are a computer with Internet access and the Zoom application to have the online meeting.

Defining the project:

This is going to be the second block of the project and it is named Defining the project [DEF]. It will consist of investigating the state of the art of the project and then exploring different alternatives to achieve the objectives of the research. All of this block will be developed strictly after the previous one, as a general knowledge of this specific field is required before diving into it.

- **[DEF1]** State of the Art (10h): I will investigate about different projects that have focused on the same topic, to realize what kind of novel approach this project should take.
- **[DEF2]** Exploring PyMTL3 (25h): One of the potential tools to be used for this project is PyMTL3, so I will explore it to see if it could be integrated into the project. I will follow an introductory tutorial [35] that has been developed by Professor Christopher Batten, the creator of this tool. This task will be developed after DEF1, as I prefer to first learn about the different options to be able to compare them to PyMTL as I program with it.
- **[DEF3]** Exploring HLS (25h): The other potential tool is HLS. I will read an introductory book [4] to get an idea of its potential. For the same reasons as with DEF2, this task has DEF1 as a dependency.

The total amount of time planned to complete this block is 60 hours.

The resources needed to complete this block are a computer with Internet access, the PyMTL3 python package and the digital access to the HLS introductory book.

Theoretical part:

This third block is the Theoretical part [TP] and is about investigating the different technologies that I will work with. All of this block has the DEF block as a dependency, as a general idea

of the topic of the project is going to be required before starting this part.

- **[TP1]** FPGA (10h): I will review again some of the papers read on INT2, and I will also review some other material to widen my knowledge about it. My starting point will be the GitHub repository of NetFPGA [36].

- **[TP2]** eBPF reading (25h): In this part, I will read some documents about this technology. There is an extensive list of different websites and papers related to eBPF [37] that I will check out.

- **[TP3]** XDP (30h): XDP is usually programmed with eBPF, so this block will follow the same tutorial [38] as the two following ones. It has TP2 as a dependency because an idea of eBPF is required in order to be able to program eBPF and, consequently, XDP.

- **[TP4]** eBPF basic lessons (35h): As I have already mentioned, this block will be developed in parallel with TP3. It will consist of completing the basic and the packet assignments of the tutorial that combines XDP with eBPF. This block also has TP2 as a dependency for the same reasons that TP3 has it.

- **[TP5]** eBPF advanced lessons (30h): The XDP eBPF tutorial has some advanced assignments that I will complete in order to master this technology. This task will be developed after TP3 and TP4, as I will need to have some knowledge about both XDP and eBPF in order to complete these assignments.

The total amount of time planned to complete this block is 130 hours.

The resources needed to complete this block are a computer with Internet access, Virtual Box to be able to explore these technologies in a safe environment and a text editor.

Learning about new tools:

This is going to be the fourth block of the project and it is named Learning about new tools [NT]. It will consist of learning the basics about some popular and useful tools I have never used. This block has no dependencies - these tools can be learned anytime. However, I plan them to take place at the time I will first make use of them.

- **[NT1]** vi/vim (3h): This text editor is known to be quite powerful. I have never used it, and I believe that this project is the perfect excuse to master it. I will learn the basics during this task so that I can productively program using it. I will first use this tool when I start programming with XDP and eBPF as I will use a virtual machine without a graphical user interface.

- **[NT2]** git/GitHub (2h): This control of versions is widely used. I have briefly experimented with it in the past but using the Desktop version. I want to use the command-line version as I believe it is faster to use once one gets used to it, thus increasing productivity. I will first use this tool when I start the Practical Part.

The total amount of time planned to complete this block is 5 hours.

The resources needed to complete this block are a computer with Internet access and a text editor.

Project management (GEP):

This fifth block is the Project management [PM] and has the objective to help structure the project in order to successfully complete the thesis. It will be developed after completing the DEF and the TP block, as I will then have a clear idea of the approach of this project and it will coincide with the beginning of the semester and the GEP classes. The amount of time required to complete each task has been obtained in the syllabus of the GEP subject. The four tasks of these blocks will be developed sequentially, as each one of them requires the previous one to be completed.

- **[PM1]** Context and Scope (24.5h): The first part of the GEP assignment will define and justify the topic of the project and how it is going to be developed.
- **[PM2]** Time Planning (8.25h): The time planning will be used to define the different tasks of the project, estimate the duration of each one of them, and build dependencies between them.
- **[PM3]** Budget and Sustainability (8.25h): In this task, I will estimate the economic cost of the project and the impact it will have on the economic, environmental and social levels.
- **[PM4]** Delivery and correction (18.25h): After the previous three tasks have been completed, I will send the report to my GEP tutor, Professor Joan Sardà, so that he will then briefly review and suggest any modification to improve the quality of the GEP document before uploading the final version. I will estimate some extra days to complete this task as I have to take into account the time that Professor Joan Sardà may take to review the document.

The total amount of time planned to complete this block is 59.25 hours.

The resources needed to complete this block are a computer with Internet access, Overleaf to write the document in LaTeX and the slides of GEP.

Practical part:

This sixth block is the Practical part [PP] and it is the part where I will produce the outcome of my research. It will be started after the TP block has been finished, as this part will directly apply all the gained knowledge about the different technologies. The tasks will be sequentially started as each one of them requires the results of the previous one.

- **[PP1]** Synthesize trivial eBPF programs (30h): This first task will have the objective of making the minimal necessary modifications to a simple eBPF program to synthesize it. I will test the synthesized programs using the Vitis simulator.
- **[PP2]** Synthesize complex eBPF programs (50h): After synthesizing simple eBPF programs, I will start doing the same with more complex ones. The objective of this task is to realize if the minimal modifications applied in PP1 are enough to be able to synthesize a complex eBPF program.
- **[PP3]** Optimize eBPF programs - Research (50h): Before applying any optimization to the programs, I will do some research about it. I will briefly review again the book read on DEF3 and I will get deeper into this topic.
- **[PP4]** Optimize eBPF programs - Memory access (90h): I believe that the main optimization I will be able to do to an eBPF program is the memory access as this programs perform a lot of them. I will rewrite the programs to optimize them as much as I can. I have estimated that this task will take a lot of time because there are many options and alternatives to treat memory with Vitis HLS. Furthermore, I will have to develop a methodology to compare each option and decide which one is better.
- **[PP5]** Optimize eBPF programs - Further optimizations (50h): After optimizing the memory access, I will try different approaches to optimize them even more, especially with loops.
- **[PP6]** Compare the performance of the optimizations (50h): Once I finish the optimizations, I will test the performance of programs before and after applying all the optimizations.
- **[PP7]** Rewrite some core eBPF functions (90h): In this task, I will rewrite some eBPF programs I consider relevant by applying all the optimizations previously discovered. I have planned that this task will take me a huge amount of time as I will rewrite as many programs as I can.
- **[PP8]** Review the NetFPGA design (15h): I will review NetFPGA's documentation and then meet with the PhD students that are investigating how to add modules to the FPGA design so that we discuss how to add the eBPF programs.

- **[PP9]** Load programs to the NetFPGA (50h): I will load the synthesized eBPF programs in the NetFPGA and I will also test their performance.

The total amount of time planned to complete this block is 475 hours.

The resources needed to complete this block are a computer with Internet access, the Vitis programming environment and simulator and the NetFPGA board.

Progress meetings:

This seventh block is named Progress meetings [ME] and it describes the different tasks related to the interaction I will have with my technical and generic advisors.

- **[ME1]** Meetings with the technical advisors (22h): I will have a weekly meeting with Professor Eric Keller and Professor Tamara Lehman to discuss the progress of the project. They will be held online at first but they may later be on-person. It has INT1 as a dependency as this task is the subset of all the weekly meetings except the first one, as I consider it just an introduction.
- **[ME2]** Preparing the Meeting with the technical advisors (33h): Some of the meetings will require some preparation. I believe that I will prepare for one out of two meetings and that three hours may be needed to make and practice each presentation. It has INT1 as a dependency for the same reason as the previous task does.
- **[ME3]** Meeting with the generic advisor (2h): I could meet with my generic advisor, Professor Jordi Guitart, to discuss the progress of the project if it is needed. It is not mandatory, but I am taking it into account as it is likely to happen. This task will be developed after the PM block has been completed because the meeting will be arranged after delivering the GEP final document.

The total amount of time planned to complete this block is 57 hours.

The resources needed to complete this block are a computer with Internet access, Google Drive to develop the presentations and online platforms to have the online meetings, which may be Zoom and Google Meet.

Final document and presentation:

This eighth and last block is the Final document and presentation [FD] and it will consist of writing the thesis final document and developing the presentation.

- **[FD1]** Documentation (80h): The documentation of this project is going to be a very important matter as my directors and I intend that the results of the research are used afterward. This is why high-quality documentation will be required. This task has no dependencies because I will develop it in a weekly manner. Even though one could think that it should be started after the PM block, I have decided to document every step I take from the very beginning of the project.

- **[FD2]** Presentation (20h): Once the documentation task (FD1) has been completed, I will develop the presentation of the project and practice it.

The total amount of time planned to complete this block is 100 hours.

The resources needed to complete this block are a computer with Internet access, Overleaf to write the documentation and Google Drive to develop the presentation.

Table 1: Summary of the tasks. *Source: own elaboration*
Note: C = Computer with Internet access, O = Overleaf, Z = Zoom, PyMTL = PyMTL3 python package, HLS B = HLS introductory book, VB = Virtual Box, TE = Text editor, SG = Slides of GEP, V = Vitis programming environment, GD = Google Drive, GM = Google Meet

Code	Task	Time(h)	Dependencies	Resources
Introduction		40		C, O, Z
INT1	Initial meeting	1		
INT2	Reading related papers	39	INT1	
Defining the project		60		C, PyMTL, HLS B
DEF1	State of the Art	10	INT	
DEF2	Exploring PyMTL3	25	DEF1	
DEF3	Exploring HLS	25	DEF1	
Theoretical part		130		C, VB, TE
TP1	FPGA	10	DEF	
TP2	eBPF reading	25	DEF	
TP3	XDP	30	TP2	
TP4	eBPF basic lessons	35	TP2	
TP5	eBPF advanced lessons	30	TP3, TP4	
Learning about new tools		5		C, TE
NT1	vi/vim	3		
NT2	git/GitHub reading	2		
Project Management (GEP)		59.25		C, O, SG
PM1	Context and Scope	24.5	TP	
PM2	Time Planning	8.25	PM1	

Continued on the next page

Table 1 – continued from the previous page

Code	Task	Time(h)	Dependencies	Resources
PM3	Budget and Sustainability	8.25	PM2	
PM4	Delivery and correction	18.25	PM3	
Practical part		475		C, V
PP1	Synthesize trivial eBPF programs	30	TP	
PP2	Synthesize complex eBPF programs	50	PP1	
PP3	Optimize eBPF programs - Research	50	PP2	
PP4	Optimize eBPF programs - Memory access	90	PP3	
PP5	Optimize eBPF programs - Further optimizations	50	PP4	
PP6	Compare the performance of the optimizations	50	PP5	
PP7	Rewrite some core eBPF functions	90	PP6	
PP8	Review the NetFPGA design	15	PP7	
PP9	Load programs to the NetFPGA	50	PP8	
Progress meetings		57		C, GD, Z, GM
ME1	Meetings with the technical advisors	22	INT1	
ME2	Preparing the Meeting with the technical advisors	33	INT1	
ME3	Meeting with the generic advisor	2	PM	
Final document and presentation		100		C, O, GD
FD1	Documentation	80		
FD2	Presentation	20	FD1	
TOTAL:		926.25		

4.3 Gantt chart

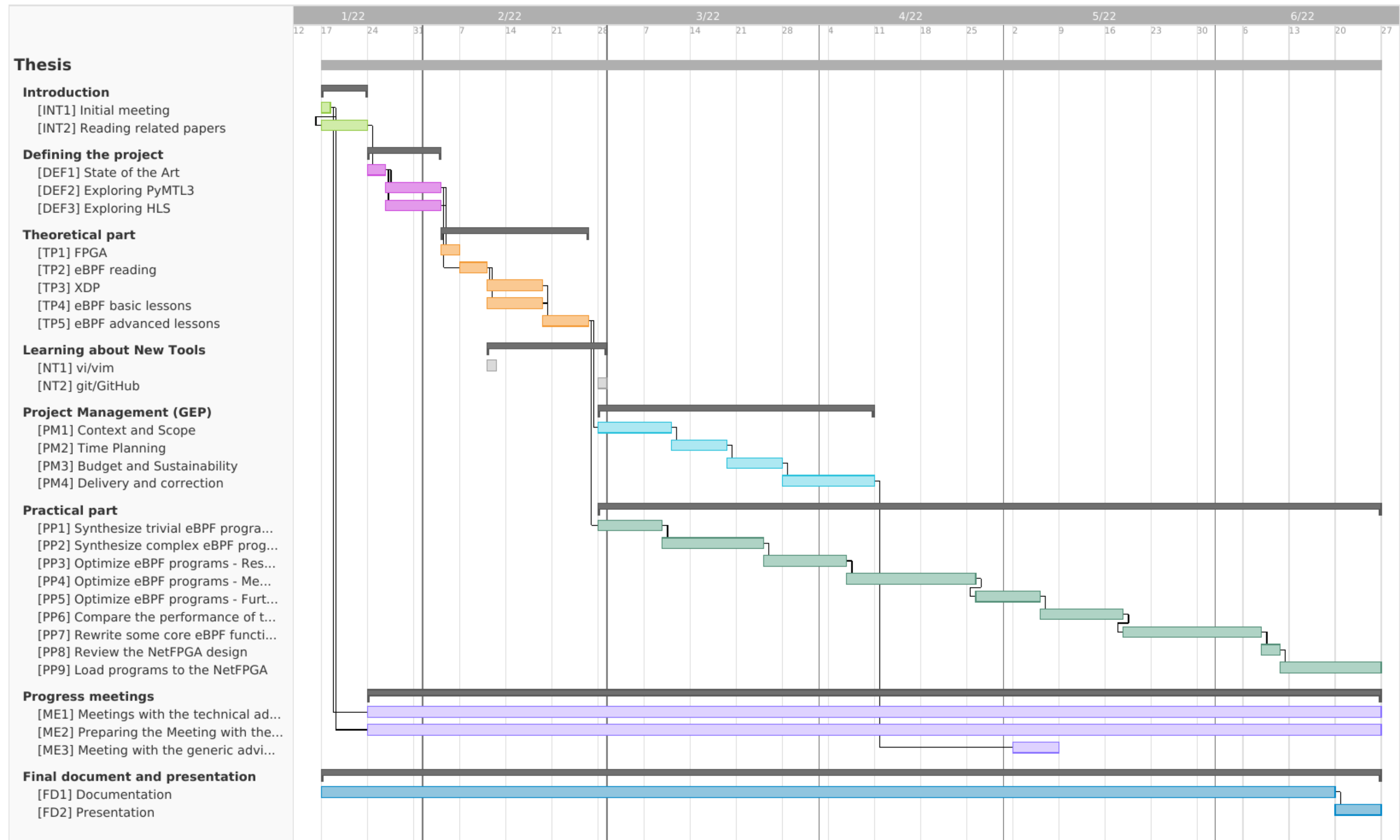

Figure 1: Gantt chart. *Source: own elaboration*

4.4 Risk management

In this section, each risk mentioned in 3.1.4 Potential obstacles and risks is estimated for its potential impact on the project. Table 2 shows a summary of the probability that each risk has to occur, and the number of extra hours that would be required to solve it, in its best and worst-case scenarios.

- **Deadlocks:** There is a medium chance that the project suffers a deadlock. If it were to happen, the impact could last from 8 hours (one working day) to 40 hours (one week). It would be hard to detect it and find a solution, so I would really expect to lose an entire week of work. That would affect the timing of the project, so I would have to shorten the estimated amount of time for PP7 if the PhD students have found a way to add the modules, or to shorten the PP8 and PP9 tasks if they have not.
- **Being unable to add modules to the NetFPGA design:** The probability of being unable to add modules to the NetFPGA design is medium as it depends on the PhD students' progress. They have been working on the project since September, and they are confident they will be able to do it. If they were unable, tasks PP8 and PP9 would be removed and task PP7 would be enlarged, as I would rewrite more eBPF functions. Consequently, the scope of the project would be slightly modified. However, the main goal of the project would remain unchanged.
- **Inexperience in the field:** The most probable risk is this one. All the technologies I will be working with are completely new to me, so it is very likely that I could need more time to learn about them. I estimate that it would take from 5 hours to 20 hours. If it affected the total duration of the project, I would take the same approach described with the deadlocks - shorten the amount of PP7, PP8 or PP9.
- **Bugs:** In the unlikely event that I face a bug, I would contact the developers of the tool, which could make me waste 5 hours, or I could try to work around it, which I estimate could take up to 16 hours. If it affected the total duration of the project, I would take the same approach described with the deadlocks - shorten the amount of PP7, PP8 or PP9.
- **Hardware incidents:** I do not expect to face any hardware incident. If my computer stopped working, I would spend 8 hours going to a shop to buy a new one if the damage was severe or going somewhere to get it fixed. I have also taken into account the amount of time I would need to install and customize my working environment back. I would not lose any data as I have multiple backups. If my keyboard or my mouse stopped working, I would ask for one from my advisors, and after the work day I would order one online or I would come by a shop to purchase a new one. It would take me one hour at most. I

do not think the impact of this risk would force me to heavily change the time planning. However, I would take that time from task PP7.

Table 2: Summary of risk management. *Source: own elaboration*

Risk	Probability	Lowest impact (h)	Highest impact (h)
Deadlocks	Medium	8	40
Load FPGA module	Medium	-	-
Inexperience	High	5	20
Bugs	Low	5	16
Hardware incidents	Low	1	8

4.5 Modifications from the original work plan

The project has faced some obstacles that have modified the original time planning, but they have not affected the core objectives of the project. The changes have also affected the estimated budget explained in 5.1 Identification and estimation of costs, and the impact of the changes on the costs is discussed in 5.3 Impact of the modifications from the original work plan.

The project has progressed as it was scheduled in the original planning. The Practical Part of the project has been completed, hence achieving the main objectives of the research. Overall, the final total amount of time dedicated to this project has been 922.25 hours, 4 hours less than originally planned.

The definitive work plan can be appreciated in figure 2, which is the final Gantt chart, an updated version from the one in 4.3 Gantt chart.

Some problems have affected the original time planning of the project:

COVID:
On Monday, April 4, I took a COVID test and the result was that I had the virus. I was sick for five days, during which the project had no progress. I decided that I would fit those 40 hours during that weekend and the following weeks. Task PP3 was completed at the weekend, and the part of task PP4 that was scheduled for that week was developed during the following two weeks.

No meeting with the generic advisor:
Task ME3 consisted of a meeting with Professor Jordi Guitart. However, after discussing the

progress of the project, we decided that it was not necessary. This task has been removed from the definitive Gantt chart.

Compare the performance of the project with an existing solution:
During the development of the project, it was clear that a validation method, other than the testing and the comparison of the performance of the optimizations, was needed. The chosen method was a formal comparison of the performance of this solution with the existing solutions. As it has been explained in 2.3.2 hXDP, hXDP is the closest project to this one, and it will be the one used for the comparison. So a new task, PP6.2, was created. Some extra hours will be needed to complete it, so the last two weeks of May will require 50 hours of work instead of 40 hours. A definition of the task, in a similar fashion as the one in 4.2 Description of tasks, is the following one:

- **[PP6.2]** Compare the performance with hXDP (20h): After comparing the performance of the optimizations, the project will be compared with hXDP. In the hXDP paper [2], the performance of some programs is evaluated. Some of those will be selected, synthesized and analyzed.

Being unable to add modules to the NetFPGA design:
This has been the risk that has affected the project the most. While task PP6 was being developed, the developer of this project was told that the students that had been working with the NetFPGA had not been able to successfully load modules. Task PP9 should either require more time or be removed. After some discussions with the directors of the project, it was decided that task PP7 would be removed to have more time to complete task PP9. The goal of task PP7 was to transform more common eBPF programs. However, a lot of functions had already been transformed during previous tasks of the Practical Part, so removing task PP7 has not affected the completion of the objectives of the project. The final time spent on task PP9 has been 105 hours. A consequence of enlarging task PP9 is that no documentation was developed during the three weeks that tasks PP8 and PP9 were developed, so the week of June 13 was exclusively dedicated to writing this report.

Deadline to deliver the report:
The project will be defended on June 30, so the deadline to deliver the report is June 23. However, due to the fact that the project is developed with an eight-time zone difference and that the generic advisor of the project has to review the final document without validating it, this report will be delivered before June 21. That adds one more day to task FD1, and task FD2 will be started one day later. Task FD1 has taken more time than originally expected, as

the directors of the project have insisted on writing the documentation with a lot of detail so that the project can be expanded in the future. In total, 25 more hours have been dedicated to task FD1.

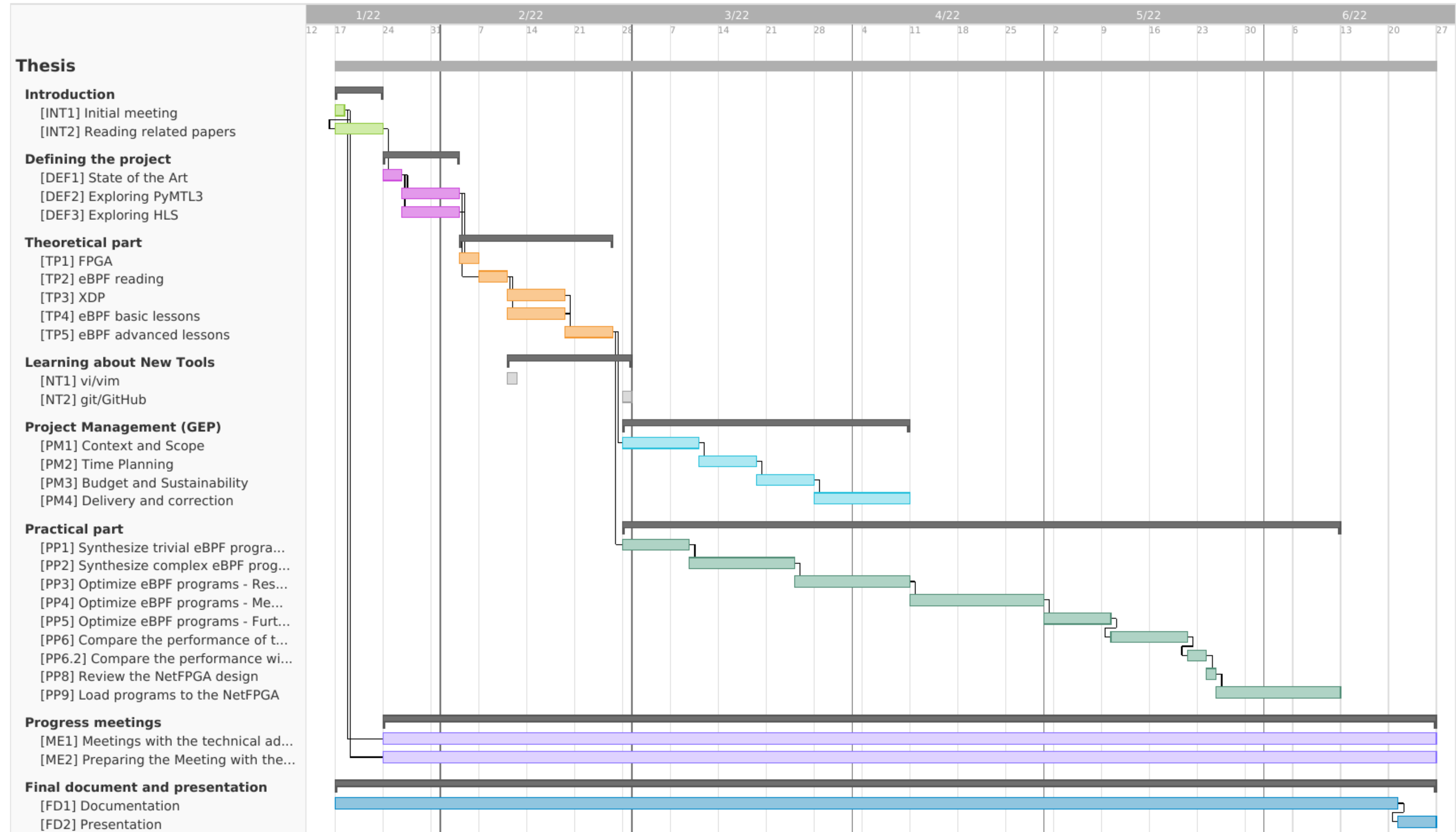

Figure 2: Final Gantt chart. *Source: own elaboration*

5 Budget

5.1 Identification and estimation of costs

As the project is being developed in Boulder, the estimates will be according to the costs here, and they will be expressed in the currency used in the US, the United States dollar.

In order to determine the costs associated with each item, the following websites have been used:

- **Payscale** [39] to obtain the average wage in Boulder for different jobs.
- **Amazon** [40] to obtain the costs of the hardware equipment, as it is the website where I buy most of the hardware here.

5.1.1 Human resources

This section estimates the costs related to human resources. First of all, the following roles have been identified:

- **IT Architect:** A considerable part of the project will require an IT Architect as it will involve integrating different software and hardware technologies.
- **Project Manager, IT:** This role will develop the task of Project Management (PM1 to PM4).
- **Senior Project Manager, IT:** This project is supervised by senior engineers.
- **Software Tester:** The project heavily depends on the quality of the testing, and this role ensures outstanding testing.
- **Technical Writer:** The documentation is also a very important task in this project. It is developed by a Technical Writer.

Table 3 contains the average wage per hour for each of the previously identified roles.

Table 3: Roles of the project and its salaries. *Source: Payscale*

Role	Wage ($ / hour)
IT Architect	44.23
Project Manager, IT	37.5
Senior Project Manager, IT	45.19
Software Tester	35.58
Technical Writer	28.85

Table 4 contains the amount of time that each role will take part in the project. Note that there are two technical directors, so some tasks double the total hours of the Senior Project Manager, IT role.

Table 4: Cost of each role for each task *Source: own elaboration*
Note: IT_A = IT Architect, PM = Project Manager, S_PM = Senior Project Manager, ST = Software Tester, TW = Technical Writer

Code	Task	Time (h)	Hours / role				
			IT_A	PM	S_PM	ST	TW
Introduction		40	40		2		
INT1	Initial meeting	1	1		2		
INT2	Reading related papers	39	39				
Defining the project		60	60				
DEF1	State of the Art	10	10				
DEF2	Exploring PyMTL3	25	25				
DEF3	Exploring HLS	25	25				
Theoretical part		130	130				
TP1	FPGA	10	10				
TP2	eBPF reading	25	25				
TP3	XDP	30	30				
TP4	eBPF basic lessons	35	35				
TP5	eBPF advanced lessons	30	30				
Learning about New Tools		5	5				
NT1	vi/vim	3	3				
NT2	git/GitHub reading	2	2				
Project Management (GEP)		59.25		59.25			
PM1	Context and Scope	24.5		24.5			
PM2	Time Planning	8.25		8.25			

Continued on the next page

Table 4 – continued from the previous page

Code	Task	Time (h)	Hours / role				
			IT_A	PM	S_PM	ST	TW
PM3	Budget and Sustainability	8.25		8.25			
PM4	Delivery and correction	18.25		18.25			
Practical part		475	360			115	
PP1	Synthesize trivial eBPF programs	30	20			10	
PP2	Synthesize complex eBPF programs	50	35			15	
PP3	Optimize eBPF programs - Research	50	50				
PP4	Optimize eBPF programs - Memory access	90	60			30	
PP5	Optimize eBPF programs - Further optimizations	50	35			15	
PP6	Compare the performance of the optimizations	50	35			15	
PP7	Rewrite some core eBPF functions	90	60			30	
PP8	Review the NetFPGA design	15	15				
PP9	Load programs to the NetFPGA	50	50				
Progress meetings		57	57		46		
ME1	Meetings with the technical advisors	22	22		44		
ME2	Preparing the Meeting with the technical advisors	33	33				
ME3	Meeting with the generic advisor	2	2		2		
Final document and presentation		100	20				80
FD1	Documentation	80					80
FD2	Presentation	20	20				
TOTAL:		926.25	672	59.25	48	115	80

Table 5 contains the total amount of time that each role will be required and the total cost of the salaries. Some of the taxes an employer has to pay for its employee in Boulder are the

Social Security tax, which is a 6.2%, Medicare, which is a 1.45%, and Federal Unemployment Tax (FUTA), which is 5.40% of the first $7,000 of each employee taxable income [41]. For the sake of simplicity, 13% will be used to calculate the total payroll taxes of the employer.

Table 5: Total cost of the human resources. *Source: own elaboration*

Role	Total amount of time (h)	Total cost ($)	Total cost with taxes ($)
IT Architect	672	29,722.56	33,586.49
Project Manager, IT	59.25	2,221.88	2,510.72
Senior Project Manager, IT	48	2,169.12	2,451.11
Software Tester	115	4,091.7	4,623.62
Technical Writer	80	2,308	2,608.04
TOTAL:	974.25	40,513.26	**45,779.98**

5.1.2 Generic costs

This section estimates the costs related to the hardware and software resources and to the general expenses.

Hardware resources:
Regarding the hardware resources, the cost of each element will have in this project has been calculated using the formula in figure 3.

$$Amortization = (Total\ Cost - Residual\ Value) * \frac{Time\ used}{Life\ expectancy}$$

Figure 3: Formula of the cost of an element used for a certain amount of time. *Source: own elaboration*

The residual value of the hardware resources is the value that they have after their life expectancy. In this project, it has been estimated that the residual value is the 20% of the original price.

The life expectancy of a hardware product is 4 years, which equals 48 months. This project has a duration of five months and one week, which is 5.25 months. Table 6 estimates the amortization of each hardware element required for this project.

Table 6: Total cost of the hardware resources. *Source: own elaboration*

Element	Total cost ($)	Residual Value ($)	Life expentancy (months)	Amortization
Toshiba Portege	1,019.62	203.924	48	89.22
Dell LCD 19” monitor	75	15	48	6.56
UNYKAch small keyboard	11	2.2	48	0.96
HP mouse	13	2.6	48	1.14
NetFPGA-SUME	6,995.00	1399	48	612.06
TOTAL:				**709.94**

Software resources:
The only software resource that requires purchasing a license is the Vivado Design Suite, which includes Vitis HLS. The license is Vivado ML Enterprise, and the version Node Locked, which is the cheapest one as it can only be used for one machine, has a cost of $2,995.00. It is permanent, it does not expire.

The cost of this resource is estimated by applying the same formula used with the hardware resources. However, the life expectancy of the hardware product is set to 3 years, which equals 36 months, and it will be used for about 3 months, for most of the Practical Part tasks (from tasks PP1 to PP7). Table 7 computes the total cost.

Table 7: Total cost of the software resources. *Source: own elaboration*

Element	Total cost ($)	Life expentancy (months)	Amortization
Vivado ML Enterprise	2,995.00	36	249.58
TOTAL:			**249.58**

Other costs:
Other expenses involved with the project are the following, computed in table 8:

- **Work space:** The PropertyShark website [42] has been used to determine the cost. The average shared office space in the center of Boulder is about $430 per month. This price includes all the utilities. It also includes using a conference room that would be an ideal place to have the meetings without any additional cost. Multiplying this monthly rate to the project duration (5.25 months), the total cost would be $2,257.50.

- **Electricity:** It is included in the workspace cost.
- **Water:** It is included in the workspace cost.
- **Internet cost:** It is included in the workspace cost.
- **HLS Introductory Book:** The paperback version of the book that will be read as an introduction to HLS [4] has a cost of $53.

Table 8: Total cost of the general expenses. *Source: own elaboration*

Element	Total cost ($)
Office space	2,257.50
HLS Introductory Book	53
TOTAL:	**2,310.50**

Incidentals costs:
It is important to take into account the costs of the different risks explained in 3.1.4 Potential obstacles and risks. Every potential risk has an estimated human cost (i.e. the number of hours multiplied by the hourly wage of an IT Architect), an estimated hardware cost and software cost (i.e. using more days the software license) and a probability to occur. The cost is obtained by multiplying the addition of the different expenses with the probability. Note that the amount of time used is the worst-case scenario described in 4.4 Risk management.

Table 9 computes the total cost.

Table 9: Incidentals costs. *Source: own elaboration*

Risk	Human resources cost ($)	Hardware cost ($)	Software cost ($)	Total cost ($)	Probability	Final cost ($)
Deadlocks (40h)	1,769.20		20.80	1,790.00	15%	268.50
Load FPGA module			62.4	62.4	15%	9.36
Inexperience (20h)	884.60		10.40	895.00	30%	268.50
Bugs (16h)	707.68			707.68	5%	35.38

Continued on the next page

Table 9 – continued from the previous page

Risk	Human resources cost ($)	Hardware cost ($)	Software cost ($)	Total cost ($)	Probability	Final cost ($)
Hardware incidents (8h)	353.84	8,113.62		8,467.46	5%	423.37
TOTAL:			11,922.53			**1005.12**

5.1.3 Total costs

In order to compute the total costs, a margin known as contingencies will be added. This project will use a 10% contingency rate.

Table 10 adds all the previously estimated costs to obtain the total cost, which goes up to **$55,060.63**.

Table 10: Total cost of the project. *Source: own elaboration*

Cost type	Original cost ($)	Contingency	Final cost ($)
Human resources	45,779.98	10%	50,357.98
Hardware resources	709.94	10%	780.94
Software resources	249.58	10%	274.54
Other resources	2,310.50	10%	2,541.55
Incidental costs	1005.12	10%	1,105.63
TOTAL:	50,055.12	10%	**55,060.63**

5.2 Management control

Some mechanisms will be used to control the potential budget deviations. Basically, they will be used to compute the difference between the estimated cost and the real cost. This way, it will be easy to quantify the deviations from the original budget.

- **Human resources deviation:** This metric has the goal of quantifying the deviation from the cost per hour. It will analyze each task from the Gantt diagram. It is divided into two parameters: the cost deviation, which evaluates the cost per hour, and the time deviation, which evaluates the total amount of time needed. Both factors are taken into account as they can greatly affect the final expense. The formula is in figure 4.

$$Human\ Res.\ cost\ deviation = (Estimated\ cost\ per\ hour\ - Real\ cost\ per\ hour) * Total\ number\ of\ hours$$

$$Human\ Res.\ time\ deviation = (Estimated\ number\ of\ hours\ - Real\ number\ of\ hours) * Real\ cost\ per\ hour$$

$$Human\ Res.\ total\ deviation = Human\ Res.\ cost\ deviation\ + Human\ Res.\ time\ deviation$$

Figure 4: Human resources deviation formulas. *Source: GEP slides*

- **Amortization deviation:** This metric quantifies the difference between the real and the estimated cost of the amortizations. The formula is in figure 5.

$$Amortization\ deviation = (Estimated\ hours\ of\ usage\ - Real\ hours\ of\ usage) * Cost\ per\ hour$$

Figure 5: Amortization deviation formula. *Source: GEP slides*

- **Incidental cost deviation:** This deviation is about the estimated cost of the potential risks. The formula is in figure 6.

$$Incidental\ deviation = (Estimated\ incidental\ hours\ - Real\ incidental\ hours) * Cost\ per\ hour$$

Figure 6: Incidental cost deviation formula. *Source: GEP slides*

- **Total cost deviation:** This is the general metric that adds all the previous deviations to obtain the deviation for the whole project. The formula is in figure 7.

$$Total\ deviation = Human\ Res.\ total\ deviation\ + Amortization\ deviation\ + Incidental\ deviation$$

Figure 7: Total cost deviation formula. *Source: GEP slides*

- **Proportion of the total cost deviation:** This last metric is the proportion of the total deviation. This way, one can easily quantify the global deviation by comparing it proportionally with the original budget. The formula is in figure 8.

$$Total\ proportion\ deviation = \frac{Total\ deviation}{Original\ budget}$$

Figure 8: Proportion of the total cost deviation formula. *Source: GEP slides*

5.3 Impact of the modifications from the original work plan

The changes from the original time planning explained in 4.5 Modifications from the original work plan have affected the budget of the project. Tasks ME3 and PP7 have been deleted, task PP6.2 has been added and tasks PP9 and FD1 have required more hours. The computation of the cost deviation will first be split between the tasks and it will finally be combined to obtain the total deviation. The metrics used are some of the ones explained in 5.2 Management control.

The amortization deviations have not been taken into account as no new hardware or software devices have been acquired and the ones used were sized for the 5.25 months that the project has lasted.

Task ME3:
This task did not require 2 hours from an IT Architect and 2 hours from a Senior Project Manager. The deviation is calculated in figure 9.

$$Human\ Res.\ time\ deviation\ (IT\ Architect) = \ 2h\ *\ \$44.23/h = \$88.46$$
$$Human\ Res.\ time\ deviation\ (Senior\ Project\ Manager) = \ 2h\ *\ \$45.19/h = \$90.38$$

Figure 9: Task ME3 deviation. *Source: own elaboration*

Task PP6.2:
This task requires 10 additional hours from an IT Architect and 10 more hours from a Software Tester. The deviation is calculated in figure 10.

$$Human\ Res.\ time\ deviation\ (IT\ Architect) = \ (0\ -\ 10h)\ *\ \$44.23/h = -\$442.3$$
$$Human\ Res.\ time\ deviation\ (Software\ Tester) = \ (0\ -\ 10h)\ *\ \$35.58/h = -\$355.8$$

Figure 10: Task PP6.2 deviation. *Source: own elaboration*

Task PP7:
This task did not require 60 hours from an IT Architect and 30 hours from a Software Tester. The deviation is calculated in figure 11.

$$Human\ Res.\ time\ deviation\ (IT\ Architect) = \ 60h\ *\ \$44.23/h = \$2,653.8$$
$$Human\ Res.\ time\ deviation\ (Software\ Tester) = \ 30h\ *\ \$35.58/h = \$1,067.4$$

Figure 11: Task PP7 deviation. *Source: own elaboration*

Task PP9:
This task requires 45 additional hours from an IT Architect. The deviation is calculated in figure 12.

$$Human\ Res.\ time\ deviation\ (IT\ Architect) = \ (0\ -\ 45h)\ *\ \$44.23/h = -\$1,990.35$$

Figure 12: Task PP9 deviation. *Source: own elaboration*

Task FD1:
This task requires 25 additional hours from a Technical Writer. The deviation is calculated in figure 13.

$$Human\ Res.\ time\ deviation\ (Technical\ Writer) = \ (0\ -\ 25h)\ *\ \$28.85/h = -\$721.25$$

Figure 13: Task FD1 deviation. *Source: own elaboration*

Total deviation:
The total budget deviation is **390.34**, which represents a saving of a **0.71%** from the originally planned budget. It is calculated in figure 14.

$$\begin{aligned} \textit{Total deviation} &= \$88.46 \ + \ \$90.38 \ - \ \$442.3 \ - \ \$355.8 \ + \ \$2,653.8 \ + \ \$1,067.4 \ - \\ &- \ \$1,990.35 \ - \ \$721.25 \ = \ \$390.34 \\ \textit{Total proportion deviation} &= \frac{\$390.34}{\$55,060.63} \ = \ 0.00708927594 \ \approx \ 0.71\% \end{aligned}$$

Figure 14: Total task deviation. *Source: own elaboration*

Final cost:
The final cost of the project is **$54,670.29**. It is calculated in figure 15.

$$\textit{Final cost} = \textit{Original cost} \ - \ \textit{Total deviation} \ = \ \$55,060.63 \ - \ \$390.34 \ = \ \$54,670.29$$

Figure 15: Final cost of the project. *Source: own elaboration*

6 Background

6.1 FPGA

An FPGA is a device in which its circuit can be redefined multiple times after manufacturing. FPGA stands for Field-Programmable Gate Array. The term field-programmable means that the hardware can be reconfigured even after it has been installed in the field [43]. The term gate array indicates that an FPGA is a matrix (two-dimension array) of logic gates, which are typically implemented using Lookup Tables, defined next.

The main use of an FPGA board is designing circuits to solve specific problems, also known as accelerators. This way, the board can achieve better performance at executing the specific function it has been designed for. This is called hardware acceleration.

Some of the basic elements that form an FPGA and that are used to evaluate the area usage of a design are the following:

- LUT: Lookup Tables (LUT) are used to implement all the combinational logic (AND, OR, etc) of the FPGA as truth tables [8]. It is basically a memory that computes a boolean function based on a truth table.
- FF: Flip-Flops (FF) are single-bit memory cells that latch the value they have on their input port and they hold it constant until the next rising clock edge. They are the basic memory element of an FPGA.
- DSP: Digital Signal Processing (DSP) are dedicated blocks focused on arithmetic and logic operations. These operations would require complex algorithms for implementation if a DSP slice was not available [29].
- SLICE: Slices are the basic unit of a CLB, which is explained later on. There are two different types of slices, referred to as SLICEM and SLICEL [44]. They are made of LUT and FF.
- CLB: Configurable Logic Blocks (CLB) are the basic unit of the FPGA, and they are made of slices, which are a group of LUT and FF. The FPGA is a matrix of CLB connected between them.
- BRAM: Block RAM (BRAM) are used to implement large memory structures. Its initial value can be set, so it can act as a ROM. It supports different clock periods on each of its ports.

- URAM: Ultra RAM (URAM) are also used to implement large memory structures. Its default value is always 0 and it cannot be changed. Every one of its ports requires the same clock period. It usually has a bigger size than the BRAM.
- LATCH: Latches are storage elements similar to FF, but they can change their output value anytime, not necessarily when the clock edge rises.
- SRL: Shift Register LUT (SRL) are an alternative for the LUT. Using them can improve performance and lead to cost savings [45].

FPGAs are configured using HDLs, which define a circuit that the FPGA will implement. This code is synthesized and converted into a gate-level design. After, the design is placed in the FPGA, which means that it is chosen where the gates will be mapped into the board. The following phase is the routing, which consists of interconnecting the already placed gates between them. The final step is generating the file that configures and is loaded to the FPGA, which is known as bitfile. A bitfile is specific for a board, as it depends on its architecture.

A common concept that appears when programming FPGAs are the Intellectual Property (IP) blocks. They are functions of the FPGA that have already been built and optimized for a specific FPGA. For example, clock generators, PCIe drivers, etc. They make interconnecting different blocks/functionalities of an FPGA design easier as they offer a layer of abstraction.

6.1.1 NetFPGA-SUME

The NetFPGA project defines itself the following way: "The NetFPGA project provides software, hardware and community as a basic infrastructure to simplify design, simulation and testing, all around an open-source high-speed networking platform" [16].

NetFPGA-SUME is a subset of the NetFPGA project. It includes an FPGA-based PCIe board with I/O capabilities.

In other words, the NetFPGA-SUME is the combination of the Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA, an x8 gen3 PCIe adapter card, 4 different 10-Gigabit Ethernet ports, 8 GB of DDR3 DRAM (though it supports up to 32 GB) and some storage devices. Figure 16 is an image of the board.

Figure 16: The NetFPGA-SUME board. *Source: [46]*

The NetFPGA-SUME project [36] offers a set of different open-source working designs, like a NIC or an IPv4 router, documentation, tutorials and online forums. It is supported by the community, which also shares other projects developed independently.

6.2 eBPF and XDP

In 1992, a network packet filter for the Unix kernel was developed. It was named **Berkeley Packet Filter (BPF)** [47]. Its purpose was to monitor and manage the network traffic. It run in a virtual machine environment in kernel space.

However, BPF is now an obsolete technology. It was replaced in 2014 by the **extended Berkeley Packet Filter (eBPF)** [13], which updates and optimizes the instruction set for modern hardware. It also adds many functionalities that are explained next. The original version is known as classic BPF, and it can still be executed as the bytecode is transformed into eBPF. The Linux kernel incorporates eBPF since version 3.18.

The **eXpress Data Path (XDP)** [10], [48] is a programmable packet processing technology included in the Linux kernel since version 4.8. Before XDP, programmable packet processing was typically implemented using kernel bypass techniques, where userspace applications would perform this task all on their own to avoid context switches between kernel and user space (from interacting with the networking hardware) and thus obtaining better performance. However, running applications on userspace that have such control of the underlying hardware is not as secure as executing them in kernel space.

XDP provides a safe execution environment for eBPF programs. Programs run in kernel space, and they are statically analyzed by a kernel verifier before loading them to ensure that they do

not have malicious intentions. The XDP program is executed when a network packet reaches the driver hook, which is located in the network device driver. The program is executed directly in that context before the packet goes through the kernel network stack.

XDP is not explained in more detail in this document as deepening into this technology is not relevant to this project. However, other practical aspects of this technology may be mentioned, like its influence on the eBPF programming style.

Back to eBPF, it is a part of the Linux kernel that adds functionality to the kernel but does not require to recompile it. The two main parts of the kernel that use eBPF programs are tc and XDP. As it has already been mentioned, XDP runs eBPF programs in a low-level environment (device-driver level). On the other hand, tc executes eBPF programs on a latter point of the Linux network stack, where the network packet has already been through some processing, so the eBPF program has access to more metadata. This project focuses on XDP, as it fits better an FPGA because of the low-level packet processing, but XDP eBPF programs can be converted to tc eBPF programs.

This project uses eBPF programs for networking, but they can also be used for tracing. Basically, the fact that eBPF programs access kernel space and can interact with user space makes them capable of analyzing and giving information about the low-level behavior of the system.

Some elements of eBPF programs are the following:

- **Virtual machine:** eBPF programs run in a virtual machine and are just-in-time-compiled (compiled after the program has started) to obtain a better performance. This type of compilation is faster as it converts the bytecode of the program in the CPU native instruction set and it has access to dynamic runtime information that is used to make better optimizations. For example, inlining functions that are often used.
- **Maps:** eBPF maps (i.e. key/value data structure) are the only way to keep the state of a program between different executions. They are also used to communicate with user-space applications.
- **Verifier:** It statically verifies the program to make sure that it is not malicious. However, it is limited, as programs have to prove that they are safe or are discarded otherwise, instead of the verifier doing an intensive job to find if they are a potential threat.
- **Helper functions:** Even though eBPF programs run in kernel space, some of its functionalities are achieved using "syscalls". The concept of helper functions is a way to execute core kernel functions with the security of doing it through kernel interfaces. For example, maps are accessed in eBPF using helper functions, and they are accessed in userspace programs using syscalls.

The verifier is very restrictive, and some of the rules it uses to determine whether to discard a program influence the style eBPF programs follow [49]:

- **Control loops:** Loops are not allowed in eBPF programs, as the program could get stuck in an infinite loop. The only work around this is to use loops with boundaries and make the compiler unroll them using `#pragma unroll`.
- **Program length:** The maximum number of instructions an eBPF program has been typically limited to is 4096. The only way to work around this is using tail calls, which enable eBPF programs to jump to another eBPF program, thus being able to execute 4096 instructions in every eBPF program it jumps into. However, after kernel 5.1, the limit of instructions has been increased to 1 million [12].
- **Unreachable instructions:** The verifier does not allow functions or instructions that can never be reached to prevent occupying kernel space with useless code.
- **Boundaries:** eBPF programs cannot jump outside its bounds. One consequence is that every function that is not the main function has to be inlined. Tail calls technically jump outside the bounds of a program, but they are performed using “syscalls”, so the kernel manages the process.

The main influence of XDP in the programming style of eBPF programs are the following restrictions:

- **Program argument:** The input and output of XDP eBPF programs is a pointer of a `struct xdp_md`. Its main elements are the memory address of the beginning of the packet and the address of the end of the packet.
- **Return value:** The return code determines the action that XDP applies to the packet. It has to be one of the following:
  1. XDP_ABORTED: The packet is discarded and an exception is raised, which is usually caught for monitoring purposes.
  2. XDP_DROP: The packet is discarded.
  3. XDP_PASS: The packet passes to the network stack, following the path it originally had.
  4. XDP_TX: The packet goes back to the exact same interface it came from.
  5. XDP_REDIRECT: The packet is redirected to an interface different from the one it was originally going to.

The programming language used in eBPF is restricted C. The *restricted* term comes from the limitations already explained. The code is compiled using LLVM and Clang, which convert it to

eBPF bytecode that is stored in an ELF object file. This bytecode is later loaded to the kernel using a loader, which in this project is XDP. The methodology used in the xdp-tutorial [38] is using a userspace program, which is called `xdp_prog_user.c`, that performs some XDP syscalls to load the eBPF bytecode. The eBPF program is called `xdp_prog_kern.c` in the tutorial. The reader should note that the name of the XDP loader includes the concept *user* while the eBPF program includes *kern*, as one is a userspace program and the other is a kernel space program.

Some of the use cases of XDP eBPF programs are security, as an eBPF program can implement firewalls, load-balancing, processing, monitoring, etc. The programmability that eBPF offers makes this technology suitable for implementing many different functionalities.

In order to get a better idea of what an eBPF program looks like, an example is commented in Appendix A. Example of an eBPF program.

6.3 Hardware Modeling

The objective of hardware modeling is to make an abstract representation of a function or algorithm. Synthesizing a model means converting it to actual wire. In the scope of this project, synthesizing has the objective of converting an eBPF program to loadable code of the NetFPGA-SUME.

There are different ways - known as levels - to model Hardware. In this project, the ones that have been explored are:

1. **Functional-Level (FL) and Cycle-Level (CL):** Both models are used to start the design, but without getting deep into the details of the implementation. They are also useful to build test benches. However, they are not synthesizable. In this project, they have been used to build a first idea of the resulting hardware.

2. **Register-Transfer-Level (RTL):** This model is the most accurate one with respect to the hardware. It is used to verify and synthesize a specific hardware implementation. It is the most relevant one for this project as it is can be converted into real wiring than can be loaded to an FPGA.

Both FL and CL have been exclusively used in PyMTL3. The RTL model has been used in PyMTL and in Vitis.

The goal of custom-designed hardware is to use hardware acceleration. Basically, it consists of running an algorithm or a function in hardware that was designed to execute that specific problem in an efficient way. It contrasts with a general-purpose CPU, as a CPU can run

different types of code at the cost of worse performance. In FPGA, hardware acceleration is used to offload computing tasks from the main CPU and solve them in a more efficient manner.

Using specific hardware minimizes oversizing the resources of the FPGA. It also affects the power usage and the performance because smaller resources tend to consume less power and be faster. For example, a 24-bit multiplication can be performed using a 24-bit hardware multiplier. It is more efficient than using a 32 or 64-bits CPU multiplier.

6.4 PyMTL3

In this section, PyMTL3 is explored in order to determine whether it is the best option to synthesize eBPF programs.

PyMTL3 is an open-source hardware modeling framework. Instead of coding with a Hardware Description Language, Python is used to describe hardware. This tool converts the Python code to Verilog.

PyMTL3 can be combined with different other tools, like the **pytest** framework [50] to test the different implementations and **GTKWave** [51] for viewing the waveforms of the designed model. During the learning phase of PyMTL3, both tools were used to debug the implementation.

An introductory tutorial [35] was followed. It was developed by Professor Christopher Batten, the creator of this tool. It first starts by explaining the different types of Hardware Modeling - the previously explained FL, CL and RTL levels. Then, the different data types are commented. After, there are three examples of hardware design models. A partial portion of the code is provided, but there are missing chunks on purpose so that they are completed by the reader. All three examples are modeled at FL, CL and RTL levels. Every level is verified using pytest and GTKWave, is executed using a simulator and the RTL model is translated to Verilog. The three examples are the following:

1. **Registered Incrementer:** This first example consists of developing a simple eight-bit register incrementer.

2. **Sort Unit:** This second example is a unit that sorts the different values that receive as input in ascending order.

3. **Greatest Common Divisor Unit:** This is the last example, and it is more complex than the previous ones. It is a unit that computes the greatest common divisor of two values. The code is divided into a datapath section, where the arithmetic operators, registers and muxes are located, and a control unit section, where signals are sent to the datapath to determine the behavior of the system.

The tutorial was a very good starting point to learn about PyMTL3. Once it was completed, it was time to synthesize a first trivial eBPF function to obtain a detailed and completed idea of the potential of this tool to model eBPF programs.

To give the reader an idea of what PyMTL3 code looks like, figure 17 provides a piece of code that creates a 16-bit multiplexer with two input ports, and then connects all the ports to other elements.

```
s.a_mux_sub = m = Mux( Bits16, 2 )
m.sel //= s.a_lt_b.out
m.in_[A_MUX_SWAP_SUB_A] //= s.a_reg.out
m.in_[A_MUX_SWAP_SUB_B] //= s.b_reg.out
m.out //= s.a_zero.in_
```

Figure 17: PyMTL3 code that creates and configures a multiplexer. *Source: own elaboration*

The eBPF program that was chosen was one that parsed IPv6 headers. It was coded just for this task, as one short and simple program was needed. Before starting to model it, there were some doubts about the tool that had not been solved by the tutorial, mainly about memory access. All eBPF programs perform multiple memory accesses, so it is a very important matter. After checking out the PyMTL3 GitHub repository [14], some memory structures were found.

After spending some time modeling the program at the RTL level with the PyMTL3 framework, the memory access issue had not been solved. It is possible to interact with memory, but it is a long process. I assumed that this tool is not suitable to model eBPF programs because of this reason. It is possible to do so, but it requires a lot of work. I would be willing to spend this huge amount of time modeling eBPF programs with PyMTL3, but this project is not about me. The outcome of this research is meant to be used afterward, and PyMTL3 would not be viable to be used because of the time resources it requires.

In order to be completely certain that PyMTL3 was not a suitable option, the creator of this tool, Professor Christopher Batten, was contacted by email to know his view regarding this topic. He replied that he considered that PyMTL3 was not the best tool to do this job. He strongly believes that PyMTL3 is not the right tool to model eBPF programs.

After discussing it with the technical advisors of the project, the option of using PyMTL3 for this project was discarded.

6.5 Vitis HLS

6.5.1 Xilinx

Xilinx is a tech company founded in 1984 that is focused on high-performance and adaptive computing. Xilinx invented the FPGAs. On October 27, 2020, AMD [52] announced that it would acquire Xilinx. However, the acquisition was not completed until February 14, 2022. According to the press release [53], AMD pretends to offer the industry's strongest portfolio of leadership CPUs, GPUs, FPGAs and Adaptive SoCs to address an approximately $135 billion market opportunity.

Apart from hardware, Xilinx is also known for its software innovations. They currently have four different platforms:

- Vitis™ Unified Software Platform: it is designed for hardware-aware software developers, using programming languages like C, C++, Python or MATLAB.
- Vitis™ AI Development Environment: it is thought to be used by data scientists looking to accelerate AI inference using frameworks like PyTorch or TensorFlow.
- Vitis™ Model Composer: it is designed to be used by system developers to perform Model-Based Designs.
- Vivado® ML: it is used by hardware designers to program in languages like VHDL, Verilog and TCL for an FPGA.

This project is going to use the Vitis™ Unified Software Platform. More specifically, Vitis HLS. This tool is intended to synthesize accelerated hardware from a C/C++ code. The reason this has been the chosen option is that the programs that are going to be synthesized are usually written in restricted C. Most eBPF programs are in this language.

6.5.2 The tool

As it is stated in the official Vitis User Guide [24], Vitis HLS is a high-level synthesis tool that allows C, C++, and OpenCL™ functions to become hardwired onto the device logic fabric. The Vitis HLS design flow is the following:

1. Write the C/C++ program.
2. Analyze and optimize the resulting code.
3. Synthesize the C program into an RTL design.

4. Verify the RTL implementation using C/RTL co-simulation.
5. Export the RTL implementation.

The working environment installed the Xilinx Unified Installer 2021.2, which was the newest at the moment the thesis started. Vitis HLS is a subset of the Vivado Design Suite. This project has used Vivado's Enterprise model license.

One requirement of this tool is using a machine with a 64 bits x86 architecture. Installing Vitis in the 32 bits version or in another architecture is not possible.

The only difference between the standard and the enterprise version is the device support they have, which can be appreciated in Appendix B. Device support in Vivado in different licenses. The board that this project uses is the Xilinx Virtex-7 690T. As well as every Virtex-7 board, it is only supported in the Enterprise Edition, and this project uses this version.

6.5.3 Synthesize process

One of the most important parts of the tool is the Flow Navigator. It is the section of the GUI where the actions of simulating, synthesizing and exporting the code are performed.

In order to synthesize C code, Vitis requires the following input:

- The C function that will be synthesized.
- The Vitis constraints. They are required, and some examples are the clock period, the uncertainty, etc.
- The directives. They are optional, but they are very important as they are used to influence the resulting hardware design.
- The C test bench that will be used to check that the model behaves as expected. Note that the return value of the test bench is the only way that Vitis makes sure that the code is correct. If the return value is different from 0, the simulation fails.
- The associated files, like the libraries that the C functions include in its code.

With that input, Vitis can produce an RTL implementation described in either VHDL or Verilog. It also produces reports to analyze the performance and the behavior of the implementation.

The actions on the Flow Navigator are the following:

1. **C simulation:** It verifies that the C code implements the program with the expected behavior. The C code is tested using the testing files. This phase is named pre-synthesis verification as it does not interact with synthesized code.

2. **C synthesis:** The C code is synthesized into an RTL design. It also produces a report of the synthesized code. However, the report is based on an arbitrary input, so the estimations can be inaccurate.

3. **C/RTL Cosimulation:** It verifies the RTL implementation using the C test files. It is known as post-synthesis verification. It uses the synthesized code of the previous phase, so any modifications to the source code require going through the synthesis again. It generates a report, which is more interesting than the previous one because it is a time estimation using the user-defined testing files as the input.

4. **Implementation:** The RTL implementation can be exported in different formats in this part. It also produces a report, which is the most accurate out of the three previous ones, as the others are projections. The RTL model may be exported and run in RTL Synthesis or in RTL Synthesis, Place & Route. The second option takes into account the overhead to place and connect the different cells of the design in the FPGA board.

6.5.4 Metrics

As it has been mentioned in the previous section, some actions of the Flow Navigator generate reports. Every main report's characteristics and metrics are presented following its Flow Navigator order of appearance:

1. **C simulation:** It does not generate a report, as it only runs C code. The output produced analyses of whether the C code passes the test benchmarks or not.

2. **C synthesis:** The report is not based on the C test files, so the output estimation goes through almost every line of code, which basically means that it shows the worst possible case scenario. Programs with a lot of conditionals tend to get worse results than in the C/RTL Cosimulation report. The most useful metrics for this project are:

   - Performance and resource estimation: The most relevant metrics of this part are the latency, which is the number of clock cycles to compute the output values given an input value, and the initiation interval (II), which is the number of clock cycles between two consecutive inputs. If pipeline is not explicitly enabled, the II should be the latency plus one.
   - Timing estimate: It includes the target clock period that has been defined by the programmer, which is the sum between the estimated clock period and the clock

uncertainty. The clock uncertainty is a default 12.5% margin that Vitis stores to avoid surpassing the defined clock period. The reason is that there can be delays due to the RTL logic synthesis, place and route.

- Software I/O Information: The C function arguments association with the hardware port names of the RTL implementation are shown.

3. **C/RTL Cosimulation:** The report estimates the behavior of the C test file with the RTL implementation. The main metrics are the latency and the II. Note that the II value should only be considered as valid if the test file simulates the design on multiple occasions (i.e. if the RTL function is invoked more than once).

4. **Implementation:** The report produced in this part is the most accurate one. Some of the metrics that are used are the following:

   - Resource usage: It shows a table summarizing the resources required to run the implementation.
   - Final timing: Clock period is referred as timing. The timing required and the timing acquired are shown. Vitis tries different clock periods to use the lowest one, which is the one named `CP achieved post-synthesis`.
   - Resources: There is a table that shows the different resource usage, in more detail than in the Resource usage part.
   - Fail Fast: It is a useful way to make sure that everything is alright, as some parameters are analyzed and the status of each one of them is verified. Any status with the value `REVIEW` should be improved.
   - Timing paths: The critical paths that affect the timing of the design the most are shown in this metric.

Some reports analyze the resource usage, and they use certain specific metrics, which have already been defined in 6.1 FPGA.

6.5.5 Bug

It was not possible to export the RTL implementation the first time that Vitis was installed during the development of this project. The error is illustrated in figure 18.

```
ERROR: '2205111913' is an invalid argument. Please specify an
integer value.
    while executing
"rdi::set_property core_revision 2205111913 {component component_1}"
    invoked from within
"set_property core_revision $Revision $core"
    (file "run_ippack.tcl" line 3679)
ERROR: [IMPL 213-28] Failed to generate IP.
```

Figure 18: Error that Vitis generated when trying to export the RTL Implementation. *Source: own elaboration*

After doing some research, some threads[2] about this issue were found on the official Xilinx support website. It appears that there is a patch that solves this issue. However, it also looks like it does not always work fine.

Someone commented in those threads that moving back the time of the computer that is executing Vitis into any day of the year 2021 solves this problem. The solution that was taken for this project is this one. The package `faketime` makes it very easy, as it can execute any application with a calendar time different from the one of the machine. In order to make it more comfortable to use, the `alias` shown in figure 19 was created and stuck in the `.bashrc` file.

```
alias vitis_hls_faketime='faketime -f '-1y' vitis_hls '
```

Figure 19: Alias to execute Vitis HLS using another time. *Source: own elaboration*

This way, typing `vitis_hls_faketime` in the command line executes `vitis_hls` making the program think that the current time is one year before the one in the machine.

6.5.6 Pragmas and directives

Pragmas and directives are used to configure synthesized hardware. Both are used as synonyms in this document, as they provide the same functionality. However, they are technically two different things.

[2] https://support.xilinx.com/s/question/0D52E00006ux2SrSAI/i-cant-export-rtl-with-error?language=en_US and https://support.xilinx.com/s/question/0D52E00006uxy49SAA/vivado-fails-to-export-ips-with-the-error-message-bad-lexical-cast-source-type-value-could-not-be-interpreted-as-target?language=en_US

Directives are added in a separate file. A project can have different solutions, and each one of them can have a unique directives file. They are a great option to explore different alternatives. A project that uses directives is made of the source code and the directives file.

Pragmas are added to the source code. Every solution of the project shares the same pragmas. A project that uses pragmas only has one file, the source code.

This project has only used pragmas. The reasons are that they are easier to export, as no other file is required. Furthermore, the code is easier to follow when pragmas are used, as they are placed in the part of the code that uses them. In the following sections, only the pragmas will be explained. However, every pragma has a directive that offers the same functionality, so they can be swapped for an equivalent directive.

7 Design and implementation

7.1 Code rewriting to make it synthesizable

The first step of the practical part of the research consists of synthesizing some Restricted C eBPF programs and transforming the code if needed.

7.1.1 Vitis unsupported C/C++ Constructs

The first thing was reviewing the unsupported constructs in the Vitis High-Level Synthesis User Guide [24]. Vitis supports C and C++ pretty well. However, there are a few exceptions.

The user guide states that, to be synthesized:

- The function must contain the entire functionality of the design.
- None of the functionality can be performed by system calls to the operating system.
- The C/C++ constructs must be of a fixed or bounded size.
- The implementation of those constructs must be unambiguous.

Luckily enough, most of these exceptions apply to eBPF programs. The most relevant ones are the following:

System calls:
System calls cannot be synthesized because they perform tasks with the operating system where the program runs. There is no way to avoid this restriction. However, Vitis has the option of running C code without synthesizing it. In this case, syscalls can be performed. They just need the special tag shown in figure 20.

```
#ifndef __SYNTHESIS__
    syscall
#endif
```

Figure 20: Tag to remove syscalls from the synthesis process. *Source: own elaboration*

This tag indicates that the code inside of it will not be synthesized.

Most eBPF programs use the maps data structure to communicate from kernel space to user space. The only way to interact with maps is using system calls. This restriction implies that eBPF programs that interact with maps cannot be synthesized without transforming the code.

Dynamic Memory Usage:
As it has previously been mentioned, the C/C++ constructs must be of bounded size. The compiler has to know the size of each data structure.

The way an eBPF program reads the raw network packet from memory is by using pointers. The size of the packet is defined by subtracting the `data` pointer to the `data_end`. The difference is the size, in Bytes, of the packet. But the packet does not have a fixed size, and that is a problem.

One option to overcome this restriction is by setting unbounded data structures with a size that it will never surpass. However, it is ineffective, as there could be a lot of unused assigned memory.

Pointer Limitations:
Vitis only supports pointer casting between native C/C++ types. It is a huge problem as eBPF programs continuously cast pointers between C non-native types.

The piece of code in figure 21 shows an example of common eBPF program pointer casting, between a pointer to an ethernet header and the pointer to the beginning of the raw packet.

```
int  xdp_parser_func(struct xdp_md *ctx) {
        void *data = (void *)(long)ctx->data;
        struct ethhdr *eth = data;
        ...
}
```

Figure 21: Example of typical pointer casting in eBPF programs. *Source: own elaboration*

The main reason that eBPF programs cast between pointers is to avoid copying the data structures. So, a workaround could be to just copy them. It may produce a poorer performance, but it would solve this issue.

7.1.2 Required transformations of the code

A list of the minimal modifications required to synthesize a Restricted C eBPF program is the following:

- **SEC tag:** Every eBPF has a SEC tag that is used to identify it by the loader (i.e. XDP). But the Vitis compiler does not recognize it, so it raises an error that prevents the compilation. It has to be removed.
- **Main function arguments:** This is probably the major change, and it is discussed next.
- **Syscalls:** Every syscall has to be removed (or put inside a non-synthesize tag). In this case, eBPF programs cannot perform any kind of interaction with maps. Another popular type of syscalls are the ones used to debug the code, like `bpf_printk`, and it has to also be removed. But it can be replaced with `printf` to maintain the functionality, even if it can only be used in the C Simulation. An elegant way to use this syscall only on the C Simulation is to define a macro as shown in figure 22.

```
#ifndef __SYNTHESIS__
#   define PRINT_DEBUG(str, args...) printf(str, ##args)
#else
#   define PRINT_DEBUG(str, args...)
#endif
```

Figure 22: Defined function to use printf only in C simulation. *Source: own elaboration*

This way, using `PRINT_DEBUG("...", args);`, the `printf` syscall is only used in the C simulation.

- **Pointers:** Every pointer cast between non-native C data types has to be replaced. A way to solve this is by copying the data structure instead of using a pointer.
- **Loop unrolling:** Loops are not allowed in eBPF programs, so it is necessary to unroll them using the tag `#pragma unroll`. However, the Vitis compiler does not use this syntax to unroll loops. The easy solution is to remove it. This change will be enhanced in the optimization section.
- **Modifying packet size:** Some eBPF may require shrinking or enlarging the packet size. As it is quite a complex operation, it is explained in more detail next.

- **Endianness:** Raw network packets usually have a different endianness than the machine that runs the eBPF program. This is the reason why it is quite common for eBPF programs to use functions like bpf_htons and bpf_ntohs. This functions are defined at #include <bpf/bpf_endian.h>. Vitis does not support this library, so the include has to be removed. One could think that copying the code from that include and defining a function with the same name could do the trick, but the compiler does not allow it. That library uses functions like __builtin_bswapXX that are not supported by the Vitis compiler. A way to solve it is using the Linux endian library (#include <endian.h>), which provides functions like htobeXX and htoleXX that work perfectly fine.
- **Union hack:** The union hack is the name that has been given, in this project, to the special use of the C union data structure to easily parse data with Vitis. It is explained in more detail next.

The three most complex modifications are explained in more detail:

Main function arguments:

The main function argument of an eBPF program is a struct xdp_md *ctx. The parameters data and data_end are data structures of type __u32, which is equivalent to unsigned int. They contain the memory address where the raw packet starts and where it ends, respectively. They technically are not pointers, as they are not defined as so, but they act as if they were. Because of the fact that Vitis is very restrictive about pointer casting and about array boundaries, this input parameter becomes troublesome.

Both data and data_end **cannot be accessed freely and cannot be treated as arrays**. Moreover, they cannot be easily cast to a pointer. However, even if they were converted to a pointer, the compiler would not allow any kind of array-like access as the pointer has no boundaries.

For these reasons, it was decided to modify the arguments of eBPF programs in order to make them synthesizable. The new arguments are uint8_t data[ETH_TOTAL_LEN] and int size. The combination of both substitute data and data_end. The array is the network packet itself. Its type is uint8_t, which is an 8-bit unsigned integer. It is defined at #include <stdint.h>. This way, byte access is simulated using integers (instead of other data types like char). The size of the data array is fixed, and it is the sum of ETH_FRAME_LEN and ETH_FCS_LEN. Both values are defined in the Ethernet library, #include <linux/if_ether.h>. The first one is the maximum size of the Ethernet frame and the second one is the size of the octets in the FCS. The second argument, int size, is the number of bytes (i.e. positions of the array) that the network packet occupies. It is used to be aware of the real size of the data. It can never surpass ETH_TOTAL_LEN.

The other parameters of `struct xdp_md *ctx` could also be added as more function arguments. In this project, they will be ignored as they will not be likely used in any program offloaded to an FPGA.

An alternative that was considered was using a pointer to access the array `data`. The main goal of doing that was to be able to modify its size at the front and at the back. However, this option was discarded as the simulation of the RTL implementation failed. It was hard to find the problem as the C simulation behaved as expected while the RTL implementation did not. The reason was that passing the arguments of the main function from C code (the testing file) to the RTL function (the eBPF program) was different than how it normally is. **By using a pointer to the array, the RTL implementation could only access the first element of the array.** Trying to access other elements of the array would return random values from memory. The lesson learned is that one has to be very careful about memory management in RTL implementation, as they do not necessarily follow the same logic as one can expect. Furthermore, this experience also proved that the behavior of a C simulation and the behavior of the RTL implementation of the same program can be different, so additional care has to always be taken.

Modifying packet size:
Some eBPF programs resize the length of the packet. For example, by pushing or popping a VLAN header. It is an interesting functionality that is implemented using function helpers. This helpers are basically the syscalls `bpf_xdp_adjust_head` and `bpf_xdp_adjust_tail`. They both receive as arguments the context `struct xdp_md *ctx` and the number of bytes that should be moved, `int delta`. If `delta` is greater than zero, the size of the packet is shrunk. A negative value for `delta` enlarges it.

The difference between the two syscalls is that adjusting the tail means modifying the end part of the packet (i.e. the highest memory addresses) while adjusting the head focuses on the beginning[3].

The key aspect of these syscalls is that the driver saves some extra space before and after the packet to ensure that there is some margin to modify its length. However, this strategy is not a good option for the HLS implementation that has been designed in this project.

The eBPF main function data argument is a fixed size array, so its size cannot be modified without heavily affecting the performance of the program. If the argument were a pointer, this original option could be more viable.

The solution that has been designed consists of enlarging or shrinking the packet by its tail.

[3] A packet begins with an Ethernet header

This way, the size of the `data` array is never modified but the variable that stores the size of its real data is. Consequently, only the `bpf_xdp_adjust_tail` helper function functionality can be performed, as it is not possible to readjust the head of the packet.

Union hack:
One of the limitations of Vitis is pointer casting. As it has already been mentioned in this report, eBPF programs tend to cast between different non-native C pointers data types. Vitis cannot synthesize programs that perform these operations.

One way to overcome this limitation is by using what this project has called Union hack. It allows the casting between different data types in an easy way.

A `union` is a data type that stores different data members, like a `struct`, but its size in memory is the one of its biggest members. This way, all the different members are mapped on the exact same memory region. All the members share the same values, partially if they have different sizes.

One potential usage of a `union` is reinterpreting the exact same raw data into different data types. This hack consists of defining a `union` with different members, all of them of the same size. One of the members is used to store a value from a certain data type (the same as this member), and then this value can be cast to every other data type that has a member in the union.

A good way to better understand this hack is through two examples:

1. **Ethernet header from a `uint8_t` array:** The main argument of the synthesized version of eBPF programs is the `uint8_t data` array. Converting it to an Ethernet header of type `struct ethhdr` is not trivial. With this hack, one can declare a `union`, having two members:
   - `struct ethhdr`: It will allow access to the Ethernet header members in an easy and intuitive way.
   - `uint8_t bytes[ETH_HLEN]`: The size of this array is the same as the one of the Ethernet header. It will be used to cast data from the `data` array to the Ethernet header data structure.

   An example of casting data between the `uint8_t data` array and an Ethernet header is the one in figure 23.

```
union ethhdr_byte_data {
    struct ethhdr eth;
    uint8_t bytes[ETH_HLEN];
};
uint8_t data[ETH_TOTAL_LEN] = {...};
union ethhdr_byte_data ethhdr;
for (i = 0; i < ETH_HLEN; i++)
    ethhdr->bytes[i] = data[i];
```

Figure 23: Example of casting between different data types using unions. *Source: own elaboration*

After copying the data, the different members of the `struct ethhdr` can be accessed directly.

2. **Four eight `bit` array values to a `long` attribute:** As the `data` array has 8-bit values, converting 4 of its elements to a 64-bit element is not trivial. It is still not difficult to do so[4], but writing a value is a bit harder. Definitely, it is possible, but this hack makes it easier and, as these operations are performed multiple times in eBPF programs, using this method is really convenient. Furthermore, Vitis does not always allow casting directly between pointers, but it allows this method.
A piece of code to exemplify this is in figure 24.

```
union pos_val {
    uint8_t *pos_8;
    unsigned long *pos_64;
};
uint8_t data[ETH_TOTAL_LEN] = {...};
uint8_t *ppos = data;
unsigned long new_val = 35000;
union pos_val value;
value.pos_8 = &ppos[4];
*(value.pos_64) = new_val;
```

Figure 24: Example of casting between 8-bit and 64-bit data types. *Source: own elaboration*

Basically, `data[4]`, `data[5]`, `data[6]` and `data[7]` have been overwritten by `new_val`.

[4]It can be done by shifting the first element 48 times to the left (the $\ll$ operator), the second element 32 times, the third element 16 times and then adding the resulting three elements with the fourth one.

7.2 Code arrangement

The code of this project is uploaded to GitHub so that the technical advisors can check it out whenever they feel like doing so. Furthermore, GitHub acts as a backup in the Cloud. It can also boost the productivity of the project as it helps control the different versions of the code.

In order to effectively upload the code to GitHub, a decision had to be made regarding the arrangement of the code. Vitis creates a lot of files for every different project, and most of them are not even human-readable. Being rigorous and following a certain structure makes the task of selecting which files to upload to GitHub easier and it also facilitates the navigation through the file system, both for the developer but also for any independent individual that is not familiar with the project. Moreover, having the code arranged makes the project more presentable and look more professional.

The file system starts with the main directory that has different sub directories. Each one of them is a different Vitis Project (i.e. a different eBPF program). Each directory has the following sub directories:

1. **Vitis auxiliary files:** This directory is created by default by Vitis, and it stores all the files that the program creates and needs.
2. **src:** It contains the source Restricted C code and headers of the eBPF program.
3. **test:** This directory has the C test source code and its headers. It is the test of the `src` programs.
4. **def:** It is an optional directory, and it contains a C (or Restricted C) header file with data structure definitions.

The first one is never uploaded to GitHub as it usually takes up a lot of storage and its content is only used by the Vitis tool. The other three directories are always uploaded as they contain the code of the project.

7.3 Vitis HLS interfaces

The main function arguments are synthesized into interfaces and ports. They define how the communication of the design with external components is. The interfaces can be configured using pragmas or directives.

Vitis supports two different flows:

- **Vivado IP flow:** It is the default one and it is used to develop RTL IP designs, which means that the project is exported as an IP and then imported to Vivado to integrate it into a design. IP are the blocks used in an FPGA, and they are an essential element of design reuse [5]. The complete process is explained with more detail in section 7.5.1 Vivado.
- **Vitis Kernel flow:** It is used to develop kernels, and it is more restrictive than the other one. Basically, the design is exported as a compiled object that can later be imported into another Vitis project. It is not compatible with Vivado.

This project uses the Vivado IP flow, as it offers more flexibility and there is no need to use the Vitis Kernel flow as no kernels are being developed. This document focuses on that option, so the different interfaces that are discussed can be used with this flow but some of them may not be compatible with the Vitis Kernel flow.

The Vivado IP flow supports three paradigms: memory, streams and registers.

7.3.1 Memory

Data is stored in a memory block, which can be DDR, BRAM, URAM, etc. Vitis implements the memory blocks as single or dual-port and chooses the one that gives the best performance to the design. Each memory block can only be accessed once (single-port) or twice (dual-port) each cycle, thus being a potential bottleneck. On the other hand, this paradigm is the only one that uses memory addresses, so it can be useful to share data between different designs.

The interfaces that can be used are `ap_memory` and `m_axi`. The difference between them are that `m_axi` can do burst accesses (up to 4KB) and `ap_memory` is a lightweight protocol.

The interface `m_axi` belongs to the AXI4 group, as it is AXI4 Memory Mapped. AXI4 stands for Advanced eXtensible Interface 4 and it is an interface specification original from ARM that Xilinx has standardized and uses in its FPGA boards to send data between different IP blocks [1]. Every other paradigm also has an interface from this group. The main characteristic of all the AXI4 interfaces is that they have the ability to do burst accesses, thus improving the throughput of the design. However, the AXI4 interfaces are the only ones supported in Vitis Kernel flow as the kernel has certain requirements that the interfaces have to fulfill [54].

7.3.2 Streams

Data is streamed into the design or out of it. The access to a stream has to be sequential and has to start at the first position of the array. Moreover, each element cannot be read more than

once. The data access can be a bottleneck, as elements cannot be read or written concurrently. Streams do not require memory resources, as data is not stored in memory by default. The access is faster than with a memory interface as streams do not use memory addresses, thus avoiding the added overhead of dealing with them. Streams can be read or written, but can never have both actions performed to the same data structure.

The interfaces that can be used are `ap_fifo` and `axis` (AXI4-Stream). The user guide strongly recommends using the stream datatype `hls::stream<...>`, which then can be configured to be either an `ap_fifo` or an `axis`. However, it is exclusive to C++ programs. The main differences between these protocols are the already explained characteristics of AXI4 interfaces.

Programming streams require a certain coding style. The ideal structure of the program should start reading the input stream in sequential order, then process the information using local variables and finally write the results in the output stream also in sequential order. Random accesses are not allowed. Not following these guidelines can result in a design that passes the C simulation but fails de C/RTL Cosimulation. Moreover, the designs that unexpectedly fail the C/RTL Cosimulation are hard to debug.

7.3.3 Registers

Data is stored in registers. The access is the fastest out of the three paradigms. The interfaces that can be used are `ap_none`, `ap_hs`, `ap_ack`, `ap_ovld`, `ap_vld` and `s_axilite` (AXI4-Lite adapter).

The simplest one is `ap_none`, which does not implement a protocol. It consists of just a wire with the constant value of the register. The most complex one is `ap_hs`, as it includes `ap_ack`, `ap_ovld` and `ap_vld`. The protocol `ap_ack` provides a signal that indicates when data is consumed, `ap_vld` provides a signal that indicates that the data is valid and hence can be read, and `ap_ovld` behaves the same way `ap_vld` does, except that input data is treated like `ap_none` does. The `s_axilite` protocol is a slave interface used to communicate between a processor and a kernel.

7.3.4 Default interfaces

Vitis implements every argument using a specific interface. Table 11 shows the default interfaces for each C argument type.

Table 11: Default interfaces in Vitis HLS. *Source: Vitis User Guide [24]*

C-Argument Type	Supported Paradigms	Default Paradigm	Default Interface Protocol		
			Input	**Output**	**Inout**
Scalar variable (pass by value)	Register	Register	ap_none	N/A	N/A
Array	Memory, Stream	Memory	ap_memory	ap_memory	ap_memory
Pointer	Memory, Stream, Register	Register	ap_none	ap_vld	ap_ovld
Reference	Register	Register	ap_none	ap_vld	ap_vld
hls::stream	Stream	Stream	ap_fifo	ap_fifo	N/A

7.3.5 The interface pragma

The interface pragma has a long syntax that offers many different options as listed in figure 25.

```
#pragma HLS interface mode=<mode> port=<name> bundle=<string>
register register_mode=<mode> depth=<int> offset=<string>
latency=<value> clock=<string> name=<string> storage_type=<value>
num_read_outstanding=<int> num_write_outstanding=<int>
max_read_burst_length=<int> max_write_burst_length=<int>
```

Figure 25: The interface pragma. *Source: own elaboration*

Most of the options are exclusive for AXI4 Memory Mapped interfaces, which have not been used. Further details on that decision can be found at 7.3.6 Best interfaces for eBPF programs. So, for the sake of simplicity, only the options used during the project are going to be explained.

- **mode:** The mode indicates the type of interface that is going to be used. For example, `ap_memory`, `ap_fifo` and `ap_none`.
- **port:** The port is the name of the variable that will have the pragma applied.
- **depth:** The depth is used to indicate the maximum size of the FIFO. When the FIFO is bounded (i.e. an array with a fixed size), by default the depth is the size of the data structure. However, in cases where pointers are used, this option has to be set or the FIFO will have size 1.

7.3.6 Best interfaces for eBPF programs

The best interfaces for eBPF programs depend on how the program is integrated into the final design. There are two different options:

1. **Independent IP:** If the eBPF program is an IP block, its top-function arguments will be interfaces. The type of interface is very flexible. However, using streams is the most intuitive way to deal with the input and output parameters as network packets tend to be streamed instead of being stored in memory. Regarding the type of stream, both `ap_fifo` or an `axis` are valid options. The best option is to match the protocols that the other IP blocks have. Regarding the input and output size arguments, `ap_hs` is the best option if the value can be modified, as it forces synchronization. If the value is treated like a constant, `ap_none` is the best choice.

2. **Subset of another IP:** If the eBPF program is imported as a function inside a larger IP block, the best option is to not apply any interface pragma, as the values will not be interfaces. They are arguments of a function, so they should not be treated as interfaces.

7.4 Vitis HLS optimizations

Vitis offers many optimizations directives, but the optimizations that are discussed in this section are the ones that have been used in this project. The optimizations that have been used seek to obtain the best possible throughput over the area of the design.

The two optimizations directives that have been used are `pipeline` and `array_partition`. A third one, `unroll` is also discussed even though it is not necessarily used as it is implicitly included in another one. The code has been transformed in order to make the most out of them. The directives are explained next. The impact of these optimizations is evaluated in 8.2 Impact of the optimizations.

7.4.1 Loop unrolling

In a rolled loop, each iteration uses the same hardware. That is, every iteration is performed in different clock cycles. Unrolling loops consists of starting more than one iteration in the same clock cycle. The area of the design is `N` times bigger, where `N` is the `factor` (i.e. how many iterations are performed per cycle). On the other hand, the throughput can increase up to `n` times faster. Loops are kept rolled in Vitis by default.

A loop can be partially or completely unrolled. Specifying the `factor` partially unrolls a loop, so a number of `factor` iterations could be executed in parallel. Completely unrolling a loop means that the design could be capable of performing the original loop in just one cycle. However, the bottleneck is usually the memory access, so other optimizations (i.e. `array_partition`) may also be applied to improve the performance.

Data dependencies between iterations could prevent loop unrolling. Loops with variable bounds cannot be completely unrolled. Transforming the code can be required in order to use these optimizations. More details and examples about these are given after.

As it has previously been mentioned, this directive is not always required. The reason behind that is that the `pipeline` directive completely unrolls all the loops by default. However, it is explained to better understand the directive.

The syntax of the pragma of this directive is the one listed in figure 26.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Figure 26: The unroll pragma. *Source: own elaboration*

The options, which are all optional, are the following:

- **factor:** The factor is used to indicate how many iterations have the potential to be performed every clock cycle. The value has to be an integer greater than zero. Not specifying this option completely unrolls the loop.
- **region:** The region is used to only unroll the loops that are hierarchically below the pragma. In other words, if the pragma is located inside a loop, the directive is not applied to this loop but it is to the other ones inside it.
- **skip_exit_check:** This option is only applied if the `factor` is specified. The bound of the loop does not necessarily have to be a multiple of the `factor`, so there is always an exit condition check to make sure that no extra iterations are performed. For example, a loop that performs 5 iterations and that is partially unrolled with a `factor` of 2 would do 3 iterations, but the last one should not be completed, as the bound of the loop is 5 and not 6. Using the `skip_exit_check` option prevents Vitis to add the exit check condition. It only makes sense to use it if the loop has a variable bound, otherwise, Vitis removes the exit condition check at compile time if it detects that the bound is a multiple of the `factor`.

7.4.2 Pipelining

Pipelining consists of executing tasks and instructions before the previous one has been completed. That is, overlapping its execution to be able to run parts (or all) of them in parallel. The result of correctly applying the `pipeline` directive is a lower II, which positively affects the throughput of the design.

Pipelining can be used with functions and loops. Function pipelining is applied when the `pipeline` directive is located in a region where multiple functions are invoked. This way, some functions can be potentially executed in parallel. On the other hand, loop pipelining makes the execution of loop iterations overlapped possible. However, loop pipelining is not applied to different executions of the whole loop by default. The difference between loop pipelining and loop unrolling is that pipelining does not replicate the underlying hardware of each loop iteration, as it focuses on executing different tasks of different iterations at the same time, while loop unrolling executes the same tasks of different iterations at the same time by replicating the hardware. Loop unrolling generally offers potentially better performance at the cost of using more area. However, `pipelining` unrolls loops by default, so both directives are combined to obtain the best possible performance.

The main obstacle to pipelining is dealing with data dependencies and loops with variable bounds. On the one hand, function pipelining requires functions to not read or write to the same data structure (generally with arrays that are stored in memory blocks). They have to follow the idea of reading some data and writing some other data so that the one that is read is not read by any other function and the data written is a part of the input of only one other function. On the other hand, loops are unrolled by default, and a loop with variable bounds cannot be unrolled, which results in Vitis disabling the pipelining. The solution to these restrictions is transforming the code, and it is explained in section 7.4.4 Transformation of the code.

The syntax of the pragma of this directive is the one listed in figure 27.

```
#pragma HLS pipeline II=<int> off rewind style=<value>
```

Figure 27: The pipeline pragma. *Source: own elaboration*

The options, which are all optional, are the following:

- **II:** The `II` option is used to indicate the desired II. Viti tries to meet it and will issue a warning if it cannot because of data dependencies. Not using this option indicates Vitis

to obtain the best possible II (i.e. it would be the same as using `II=1`).

- **off:** It is used to turn off pipelining for a specific region of the code.
- **rewind:** This option is used in loop pipelining to overlap different executions of the same loop.
- **style:** The `style` option is used to flush the pipeline. By default, the pipeline continuously reads the input data or it stalls for new data. Flushing the pipeline shuts down every pipeline stage when there is no input data available. It is useful to avoid deadlock in certain circumstances. No more detail is given about this option and about flushing pipelining in general as it is not relevant for the project.

7.4.3 Array partitioning

Vitis synthesizes arrays that are stored in memory into single or double-port RAM blocks. It chooses the one that offers the best performance. This means that no more than one (single-port) or two (dual-port) accesses can be made every cycle to the same array. This restriction usually makes the memory access a bottleneck. The only way to avoid it is by partitioning the array, which means that the data is stored in different smaller memories or in registers instead of a larger memory.

An array that is a stream can also be partitioned. Instead of creating one stream, Vitis creates multiple smaller streams at compile time.

There are three different ways to partition an array:

- **Block:** The array is divided into blocks with consecutive elements.
- **Cyclic:** The array is divided into blocks so that consecutive elements are located in different blocks.
- **Complete:** Every element of the array is stored in registers.

The best options for eBPF programs are the `cyclic` and the `complete` partitioning as the network packet is accessed sequentially. It would not be possible to access the data more than twice every cycle using the `block`, but it would be using any of the other two. The option that offers the best possible performance is always the `complete`, but at the cost of using more resources.

The syntax of the pragma of this directive is the one listed in figure 28.

```
#pragma HLS array_partition variable=<name> type=<type>
factor=<int> dim=<int>
```

Figure 28: The array_partition pragma. *Source: own elaboration*

The options are the following:

- **variable:** This required option is used to indicate the name of the array this pragma will be applied.
- **type:** This option is required, and it indicates how the array is going to be partitioned. The options are the ones already explained, `block`, `cyclic` and `complete`.
- **factor:** This option is not used in `complete` partitioning but is required in the other two types. It indicates the number of smaller arrays where the main array will be partitioned. The size of the array does not necessarily need to be a multiple of the `factor`. However, using a factor that is not a power of two adds an overhead to the access of the data, as deciding which subarray has the data requires a division instead of shifting arithmetic.
- **dim:** This optional argument specifies in which dimension the array should be partitioned in a multi-dimensional data structure.

7.4.4 Transformation of the code

The optimizations can be directly applied to the code without modifying it. However, the performance will probably be very limited. Some code transformations and adjustments between directives have to be applied in order to make the most of them.

Variable loop bounds:
Variable loop bounds disable pipelining, as they cannot be unrolled. The only way to transform a variable bound to a fixed bound is by defining a maximum value and setting it as the bound of the loop. In the case of eBPF programs, the network packets may have different sizes, but a maximum packet size can be defined.

For example, a loop that reads two arrays, from position `pos` to `size`, and where both variables are not constants, can be found in figure 29.

```
for (i = pos; i < size; i++) {
        data_out[i] = data[i];
}
```

Figure 29: Example of a loop with a variable bound. *Source: own elaboration*

This loop would disable the ability to pipeline the whole design. The only way to unroll this loop to be able to pipeline the design would be to define a constant, `MAX_PACKET_LEN`, and also define the value of the number of iterations, `i`, as it is shown in figure 30.

```
for (i = 0; i < MAX_PACKET_LEN; i++) {
        if (i >= pos && i < size) {
                data_out[i] = data[i];
        }
}
```

Figure 30: Modified loop to enable pipelining the design. *Source: own elaboration*

Even if the transformation makes the code appear to be slower, it can potentially improve the throughput a lot, as the loop is completely unrolled and the design is able to pipeline.

Pipelining failing due to read after read:
Task pipelining has a restriction, which is that the different tasks cannot read or write to the same data structure. Accessing different elements of an array in different tasks should be allowed, as no elements are read by two different tasks. However, it is not possible to do it in Vitis [55]. This means that different tasks that parse different headers of a network packet cannot be pipelined between them. Trying to trick Vitis by casting the array and using pointers does not work at all.

Furthermore, it also affects the writing functions, as some functions may only parse the network packet up to a certain header, and the remaining data of the packet could be read by a writing function. Disabling pipelining between the parsing functions and the writing functions would affect the resulting performance a lot. The solution is to first read all of the data, which is a stream, and write it to different arrays. This way, every function receives a different array as input, and the design can pipeline.

Resulting pipelining schema:
The code has been divided into three different tasks. First, the network packet is copied into different local arrays. After, the data is parsed. Finally, the data is written into the output stream.

This way, the input of the first task is the input array, and the output is the different arrays that are a copy of the original. The input of the second task is one (or some) of those arrays, and the output is the data structures that store the different headers. The input of the third task is some of the arrays generated by the first task (they can never be the same as the ones read by the second task) and the headers' data structures. The output of this third task is the output packet.

Options of array partitioning:
The value of the `factor` option that offers the best potential performance is either `complete` or the closest higher number to the maximum packet size which is a power of two. This way, the access to the array does not become the bottleneck of the program.

For example, if `MAX_PACKET_LEN` is 130, the best possible performance would be obtained with `factor` being `complete` or 256. However, partitioning the stream in 128 would not probably affect the performance that much. It would be up to the programmer to test both options and decide which one is more convenient.

On the other hand, the partitioning should always be `cyclic` or `complete`. As it has already been mentioned in a previous section, partitioning in a `block` style does not improve the performance of the design.

Conditional tasks:
Tasks that are executed only if a certain condition is met are not suitable to be synthesized. First of all, a conditional task disables the pipelining of a design. Secondly, based on the multiple programs that have been synthesized during the development of this project, the performance estimate that Vitis offers after synthesizing is usually worse when there are conditional tasks. Even if a conditional task is not a pipelined task but a subset of one, Vitis is not able to apply many compiler optimizations as it does with a more sequential code.

The way to avoid conditional tasks is by executing them every time and adding a flag to decide when the task has to do the work and when it does not. For example, a function that parses a VLAN header only if it finds one could look like figure 31.

```
if (nh_type == htobe16(ETH_P_8021Q) ||
    nh_type == htobe16(ETH_P_8021AD)) {
        nh_type = parse_vlanhdr(&pos, data, &vlan1, size);
}
```

Figure 31: Example of a conditional task. *Source: own elaboration*

To make the task (i.e. the parse_vlanhdr function) unconditional, the last argument `size`, which is the size of the network packet, can be set to the real size if there is a VLAN header or to -1 if there is not. This way, the function is always executed, and it is decided there if it has to parse the VLAN based on the value of the last argument. The resulting code would be the one in figure 32.

```
if (nh_type == htobe16(ETH_P_8021Q) ||
    nh_type == htobe16(ETH_P_8021AD)) {
        size_cond = size;
} else {
        size_cond = -1;
}
int nh_type_vlan = parse_vlanhdr(&pos, data, vlan1, size_cond);
```

Figure 32: Conditional task converted to unconditional. *Source: own elaboration*

Order of the array accesses:
Applying the directive `array_partition` in a `cyclic` mode adds an overhead to the data access as it has to be determined which partition of the array has the data. As it has been previously discussed, it is very important to partition the array in a number that is a power of two. Knowing which positions of the array are going to be accessed at compile time makes the design faster, as it is not necessary to add circuitry to calculate it. Rewriting the code to ensure this is also an optimization. However, it is not always possible to do so because of the nature of some programs.

For example, a program that writes the IPv4 header, if there is one, to an output stream, and then writes the remaining of the packet, is shown in figure 33.

```
int pos = ETH_HLEN;
if (eth->eth.h_proto == htobe16(ETH_P_IP)) {
    pos += IPv4_BASE_LEN;
    write_IPv4_hdr(pos, data_out, iphdr);
}
write_remaining_packet(pos, data, data_out, size);
```

Figure 33: Example of a program that writes an IPv4 header if there is one. *Source: own elaboration*

Note that the function `write_IPv4_hdr` requires the position of the end of the header, while `write_remaining_packet` takes the position of the first byte to be copied. This is the reason why both functions take the same value for `pos`, which is the position of the partitioned array that is read.

The previous example can be enhanced by trying to hardcode the value of `pos`, so that Vitis knows at compile time which subarray is accessed. The resulting code is the one in figure 34.

```
if (eth->eth.h_proto == htobe16(ETH_P_IP)) {
        write_IPv4_hdr(ETH_HLEN+IPv4_BASE_LEN, data_out, iphdr);
        write_remaining_packet(ETH_HLEN+IPv4_BASE_LEN, data,
                               data_out, size);
} else {
        write_remaining_packet(ETH_HLEN, data, data_out, size);
}
```

Figure 34: Example of hardcoding the access to an array. *Source: own elaboration*

Reduce memory accesses:
In general, memory accesses have to be minimized as much as possible as they tend to be slow. Using temporal variables is a great way to accomplish it.

For example, figure 35 provides a piece of code that reads a map and only increases its value it is not zero.

```
if (rxcnt[0]) rxcnt[0]++;
```

Figure 35: Reading a map and increasing its value if it is diferent from zero. *Source: own elaboration*

That line makes three memory accesses: a read to evaluate the condition of the `if`, another read and a write to increase its value. In total, three memory accesses.

However, the code in figure 36 reduces that to only two by using a temporal variable.

```
__u64 icmp_count = rxcnt[0];
if (icmp_count) icmp_count++;
rxcnt[0] = icmp_count;
```

Figure 36: Reducing the access to a map by using a local variable. *Source: own elaboration*

One could think that the first example offers a better performance in the best-case scenario, as it only reads the value and does not perform the second read and the write if the value is equal to 0. However, the II takes the worst-case scenario, so it is always slower than the transformed version.

Type qualifiers:
There are some qualifiers that affect the synthesis process. They are mentioned as they can affect the performance of the final design.

- **volatile:** This attribute is used to indicate the compiler to not apply any optimization to a variable as it will be accessed multiple times. The developer has to be aware that Vitis never applies any optimization to a variable with this qualifier.
- **static:** A `static` variable keeps its value between functions calls. Vitis implements them as registers, while other variables can also be stored in memory. This means that every access of a `static` variable takes one cycle. Accessing multiple times a `static` variable can become a bottleneck. The solution is using temporal variables, as it has already been explained before.
- **const:** This type qualifier indicates that the variable will never be updated. Variables are treated as constants in the RTL design, and arrays are mapped to ROM. It is a good practice to use this qualifier with function arguments that are only read, as then Vitis has more information to apply any optimizations it considers appropriate.

- **Global variables:** This type qualifier is not exposed as an RTL port by default, but instead it is treated as memory accesses.

C simulation:
This optimization is not useful in the final design. However, it makes the C simulation faster. It consists of using dynamic memory to store large data structures. However, dynamic memory is not permitted during synthesis, so a special check has to be performed to ensure it. It is always a good practice to free the memory after using the data structures. The example in figure 37 stores a large array using `malloc`.

```
#ifndef __SYNTHESIS__
        uint8_t *data_tmp = (uint8_t *) malloc(MAX_PACKET_LEN);
#else
        uint8_t data_tmp[MAX_PACKET_LEN];
#endif

.....

#ifndef __SYNTHESIS__
        free(data_tmp);
#endif
```

Figure 37: Storing an array using malloc only in the C simulation. *Source: own elaboration*

Note that **the synthesized code is different from the used in the C simulation**. It is usually not recommended to do so, but it is not a problem in cases like this one.

During the development of this project, some programs would fail the C simulation as they overflowed the stack because of allocating large structures there. This trick solves these types of problems.

7.5 Loading eBPF programs in the NetFPGA-SUME

7.5.1 Vivado

The Vivado Design Suite is a tool developed by Xilinx that is used to place and route designs onto device resources, within the logical, physical, and timing constraints of the board [56].

In other words, Vivado is used to integrate RTL designs, Netlist designs and IP-centric design flows in order to create the file that is loaded and configures the FPGA - the bitfile.

The goal of using Vitis HLS in this project is to synthesize eBPF programs, which means converting them into an RTL design. This design is exported from Vitis, and imported to Vivado as an IP block. IP blocks can be integrated with the other blocks of the design. Some of those blocks may already be implemented by Vivado, so they can be directly added.

Once the design is completed, it can be synthesized and implemented in Vivado. Finally, the bitfile can be exported and loaded to the FPGA.

A more detailed guide on the path to obtaining a bitfile from a Vitis project can be found in Appendix C. Creation of a Bitfile from a Vitis project.

7.5.2 Base design

The design that has been developed expands the design of the reference NIC project from the NetFPGA-SUME Project. It is an infrastructure that forwards packets through its interfaces. Figure 38 is a diagram of the design. The chosen method to load eBPF programs is adding them as modules in this working design.

The reference NIC design starts when a packet arrives from one of the four Ethernet interfaces or from the DMA module, which is connected to the PCIe port. Then, it goes to the **input arbiter**, which is a FIFO that rotates between all the interfaces in a round-robin manner, and moves the packet to the next module, the **output port lookup**. This module decides which port the packet goes to, and writes this information in a side channel. After, the packet arrives at the **output queues**, which is a module that has a FIFO for each output interface, and that determines which FIFO the packet should go to based on the metadata. Finally, when the packet reaches the end of the FIFO queue it is in, it is sent to the output interface.

The communication protocol used to transport the packet is an AXI4-Stream adapter with side channels. It is 256-bit wide and it includes a side channel, TUSER, that is 128-bit wide. As it will be discussed later on, integrating a new module into this design means that the module has to be adapted to receive this stream as input and generate a similar one as output.

The first decision that had to be made was placing the eBPF logic in the design. The chosen location was between the input arbiter and the output port lookup. The reason is that the packet is not much processed but it has already been converted to a stream. Adding the module before the input arbiter does not make much sense, as the task that performs the module - reading from all the physical ports - is strictly necessary. On the other hand, the output port lookup spends

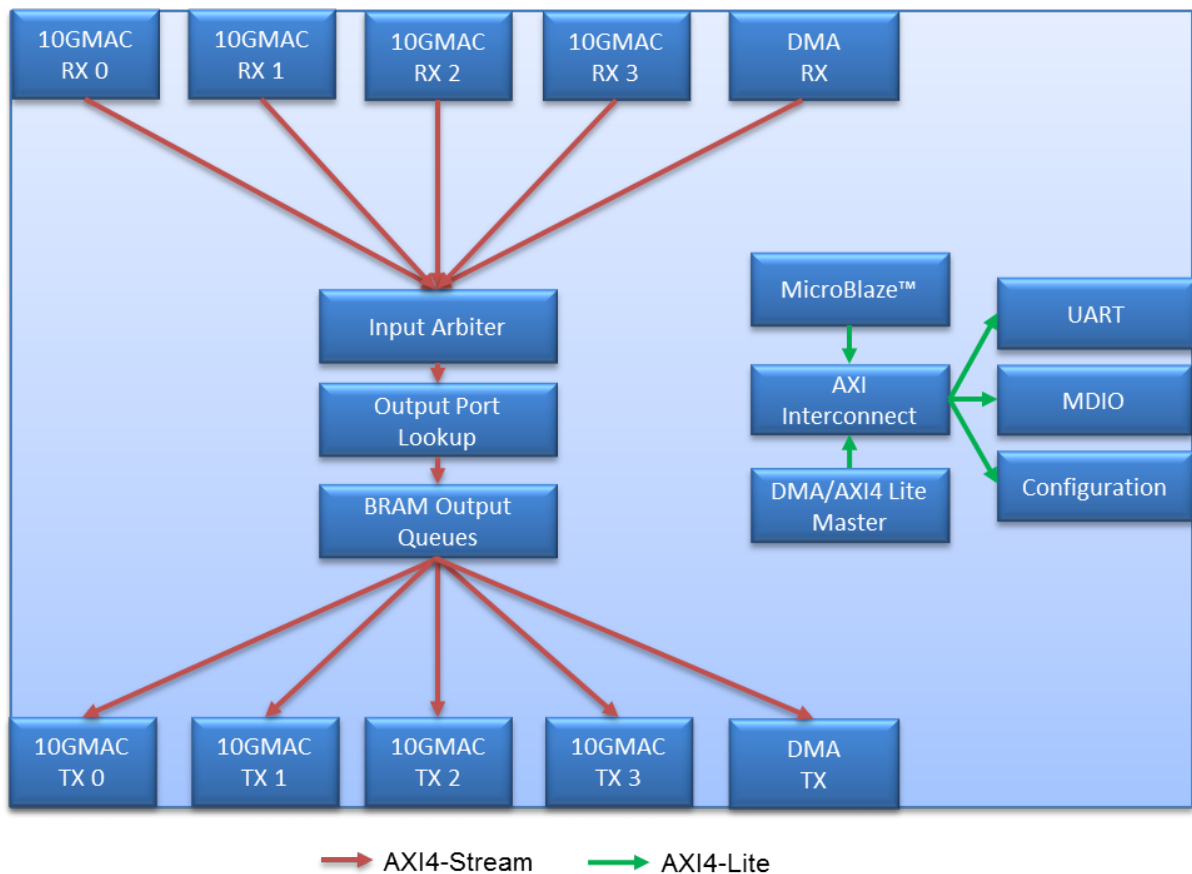

Figure 38: Reference NIC block diagram. *Source: [36]*

resources writing metadata to the packet, and the fact that eBPF programs can either discard or redirect (i.e. modify the destination) a packet would make that module's work inefficient.

7.5.3 Adapters for eBPF programs

The input arguments of synthesized eBPF programs are `uint8_t` arrays and the real size of those arrays, as they are oversized with a size limit. Those arguments do not match an AXI4-Stream, so it is necessary to have an input adapter and an output adapter that convert the AXI4-Stream to the eBPF arguments and vice versa.

The two potential ways to incorporate eBPF programs into the design are by either adding the program and the adapters as different IP blocks and interconnecting them or by adding only one IP block that incorporates both the adapters and the eBPF program. The chosen solution is the second one. Both are valid options, but the first one requires more work interconnecting the IP blocks in Vivado, while the second one requires more work in Vitis incorporating the C

code and the functions. As this project has spent more time using Vitis, the developer felt more comfortable with the second option. The first option may be more suitable for someone more familiar with the different Vivado IP blocks, as connecting the adapters and the eBPF program requires using additional modules (like a FIFO Generator) and adding synchronization between some of the parameters.

Figure 39 is the resulting IP block of the module. It has the following ports:

- **ap_ctrl:** It manages when the IP block starts and how it is synchronized with the other blocks.
- **data_in:** It is the input 256-bit wide AXI4-Stream with side channels.
- **ap_clk:** It is the clock of the block.
- **ap_rst_n:** It is the reset of the block.
- **data_out:** It is the output 256-bit wide AXI4-Stream with side channels.
- **ap_local_block and ap_local_deadlock:** They are two optional signals used to debug the block.

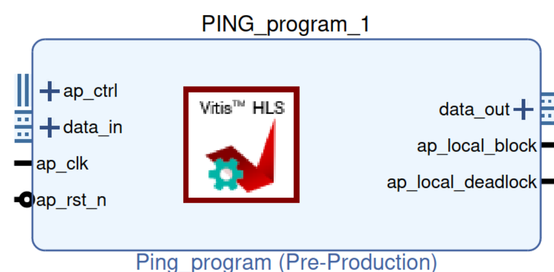

Figure 39: IP block of the loaded eBPF program. *Source: own elaboration*

The final design is a C++ file, while the eBPF program is a C file, as it has been previously discussed in this report. The reason is that the only way to treat streams with side channels in Vitis is using a specific datatype, `hls::stream`, which is only available in C++. However, it is possible to import a C function in a C++ file, so invoking the eBPF program in the file is possible.

The program starts with the **input adapter** reading the AXI4-Stream, stores the side channels - both TUSER and TKEEP - into a local buffer, and converts the 256-bit chunk of data to 32 8-bit sets of data to create a `uint8_t` array, and also counts the real size of the stream.

Then, the **eBPF program** is invoked as a function, using the created array and its real size as the input arguments.

When the eBPF program finishes, the **output adapter** reads the output arguments of the function and converts the 8-bit array to 256-bit streams. It incorporates the previously buffered side channels. Not doing so would result in the design failing. If the packet is dropped by the eBPF program, the size that the function outputs is zero, so the output adapter does not create the stream and it results in effectively dropping the packet at the earliest possible point of the design.

Figure 40 is the resulting IP block of the eBPF program (and the adapters) connected with the input arbiter (at the left) and the output lookup port (at the right). Both the clock and reset ports are connected to the same one the other two blocks are wired to, `data_in` and `data_out` are connected to the output stream and input stream of the input arbiter, and the output lookup port respectively, the `ap_ctrl` is always with `ap_start` set to '1' to make the block always run and the other two ports remain unconnected as they are only used to debug.

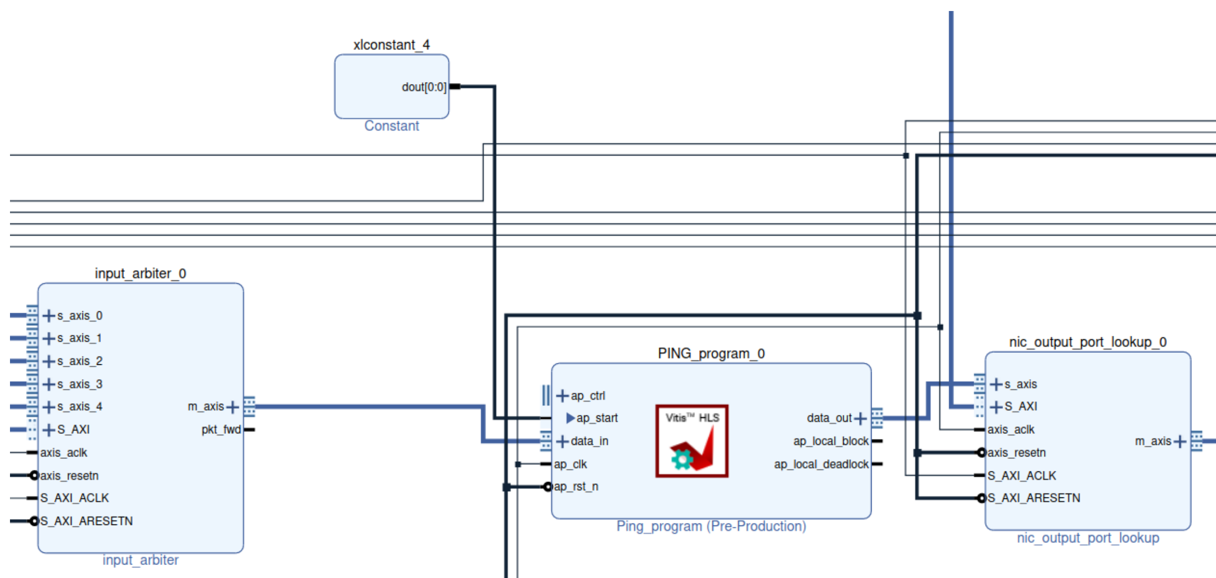

Figure 40: Integration of the IP block of the eBPF program in the design. *Source: own elaboration*

7.5.4 Validating the design

The testing environment of the board is complex. Because the project did not have access to an extra NIC, the testing had to be done only with the FPGA. This testing environment was mainly developed by other students of the supervisors of this project.

The solution consists of creating two namespaces in the machine. The routing table has been modified to force going through the board in order to communicate both namespaces. The

only tool that can be used in this testing environment is the ping, which is executed from one namespace and has as a destination the other. More details about this testing environment, including the exact steps to replicate it can be found in Appendix D. Loading and testing a Bitfile in the NetFPGA-SUME.

The testing environment does not offer many alternatives to test the design. The only input that it can produce is ICMPv4 packets. So a new eBPF program was developed, which parses the ICMPv4 packet and it takes two actions depending on the value of the sequence number. If it is odd, the packet continues to the output port lookup. But if it is even, the packet is discarded. This way, it was easy yet reliable to test the design, which worked.

8 Performance evaluation

8.1 Methodology

The performance of this project is evaluated using a set of eBPF programs. They have all been transformed to successfully synthesize them, following the guidance given in section 7.1 Code rewriting to make it synthesizable. Furthermore, the optimizations explained in 7.4 Vitis HLS optimizations have also been applied afterward to compare their impact.

The programs that have been tested have also been used for the same purpose on hXDP. This way, they can be directly compared. There is another program that has been evaluated only in this project (i.e. not in hXDP), which is the one that has been loaded to the NetFPGA-SUME. Table 12 describes the different programs that have been tested.

Table 12: Programs used to evaluate the performance of the project. *Source: own elaboration*

Program	Description	Source
XDP1	Parses the packet up to the IPv4 or IPv6 header and then drops it.	`https://github.com/torvalds/linux/blob/master/samples/bpf/xdp1_kern.c`
XDP2	Parses the packet up to the IPv4 or IPv6 header and then forwards it.	`https://github.com/torvalds/linux/blob/master/samples/bpf/xdp2_kern.c`
XDP_ADJUST_TAIL	Parses the packet up to the IPv4 header and then shrinks its size if it is bigger than a certain value.	`https://github.com/torvalds/linux/blob/master/samples/bpf/xdp_adjust_tail_kern.c`
Adapter_PING	Parses the packet up to the ICMPv4 header and drops it if its sequence number is even, otherwise it lets it pass. It includes adapters specifics for the NetFPGA-SUME.	Own elaboration

The tool used for this testing evaluation is the built-in Vitis simulator. The reasons why the performance has been tested in the simulator instead of the FPGA board are the following:

- **Many functionalities have not been implemented in the FPGA design:** Most of the tested eBPF programs use maps, which have not been implemented and tested in the FPGA. There is one program that modifies the packet size, and this functionality has also not been checked yet.

- **Testing environment:** The environment used in the NetFPGA-SUME is very restrictive, as it can only use the tool `ping`, which means that the only input for these programs is ICMP packets. Testing the programs with such few varieties of input would produce inaccurate results.
- **Accuracy of the performance evaluation:** Because of the already mentioned restrictive testing environment, it is not easy and liable to test the performance on the board. There are some overheads that could corrupt the results, especially when dealing with the two namespaces created to test the board without using an external NIC.
- **The design is in its first version:** The conclusion would be that the design is still in development as it still has room for improvement. A proper performance evaluation could be developed once more support for eBPF functionalities is added and the testing environment is enhanced.

However, in order to replicate the results, some parameters of the Vitis simulator have to be set in a certain way. The board used is the Xilinx Virtex-7 XC7V690T FFG1761-3 (the FPGA of the NetFPGA-SUME), whose part is `xc7vx690tffg1761-3`. The design is clocked at 156.25 MHz (6.4 ns), which is the frequency that the board works at. The board used in hXDP is the same, so the results are clocked identically. It is relevant to mention the fact that the results obtained using the simulator are independent of the computer used to run them. The simulator options used are the C/RTL Cosimulation to obtain the throughput and the RTL Implementation for the resource usage. The selection of the cosimulation is due to the fact that it provides more precise results than other faster options such as the synthesis. The version of Vitis HLS used is the 2021.2, which was the newest at the time the project started.

Another important matter is the fact that the maximum packet size has a bound, `MAX_PACKET_LEN`, whose value can be relevant. But it does not affect the overall performance of the main logic of the optimized design as the network packet is partitioned in a `complete` mode (i.e. the network packet is stored in registers), so the throughput is independent of the length of the packet. In other words, if the packet is longer, the area of the design will be larger, but the performance will be the same. However, it does affect the results of the non-optimized version. For the sake of simplicity, the maximum packet length used in both versions has been kept the same. The speedup acquired from the optimizations is a little bit lower than the real one because of that, hence making the impact of the optimizations look inferior. However, the results are good enough even with this inconvenience. The most relevant values, which are the performance of the optimized versions as they are compared with hXDP, are surely precise.

The value of `MAX_PACKET_LEN` is set to 300 for all the tests. The machine used to run the simulator is old, so it can take up to a couple of hours to perform every test. Setting the value higher could make the test last days. An alternative to work around this was to use a large

server owned by one of the directors of the project, but the architecture of that server is ARM and Vitis is only compatible with 64 bits x86 machines.

The simulator does not offer the throughput as a metric, so it has to be manually calculated. Given the II, which is indeed given by the simulator, the throughput `Y` measured in Mpps - Millions of packets per second - is obtained using the following formula in figure 41, where `X` is the II.

$$Throughput = \frac{1\ packet}{X\ cycles} * \frac{1\ cycle}{6.4\ ns} * \frac{1\ Mp}{10^6\ packet} * \frac{10^9\ ns}{1\ s} = Y\ Mpps$$

Figure 41: Formula to calculate the throughput given the II. *Source: own elaboration*

The reason why II is the metric used to obtain the throughput instead of the latency is that it ends up being more precise as every packet takes II cycles to be processed except for the first one, which takes the whole latency.

The Speedup formula, which is also used during the comparisons, is shown in figure 42.

$$Speedup = \frac{Non - Optimized\ version\ II}{Optimized\ version\ II} = \frac{Optimized\ version\ throughput}{Non - Optimized\ version\ throughput}$$

Figure 42: Speedup formula. *Source: own elaboration*

Regarding the area usage, the RTL Implementation offers a precise estimation. In the following sections, the percentage of the area will be the percentage of usage of the highest used element. For example, if there is a 2% of FF and a 1% of LUT, the area usage is considered 2%. The reader should note that the area usage depends on the length of the packet, so the results shown are for 300 Bytes packets.

The source code of the programs is provided at the public GitHub repository of this project[5] so that the results can be replicated.

8.2 Impact of the optimizations

This section analyses the impact of the optimizations by comparing eBPF programs before and after optimizing them. In order to easily reproduce this experiment, there is a defined

[5] `https://github.com/aviba2000/eBPF_NetFPGA-SUME`

value, `OPTIMIZATION`, in the file `DataTypes.h` located in `00_Common_Libraries/def/`. Manually modifying the value to 0 makes the code execute the non-optimized version, and modifying it to 1 invokes the optimized functions.

The first program tested is the XDP1. Table 13 compares both versions of the program. The optimized version achieves a speedup of 18x.

Table 13: Performance of the XDP1 program. *Source: own elaboration*

Version	Latency	II	Throughput	Area usage
XDP1 - non-optimized	56 cycles	54 cycles	2.89 Mpps	0.09 %
XDP1 - optimized	4 cycles	3 cycles	52.08 Mpps	0.05 %

The second program tested is the XDP2. Table 14 compares both versions of the program. The optimized version achieves a speedup of 17.3x.

Table 14: Performance of the XDP2 program. *Source: own elaboration*

Version	Latency	II	Throughput	Area usage
XDP2 - non-optimized	109 cycles	104 cycles	1.5 Mpps	0.41 %
XDP2 - optimized	9 cycles	6 cycles	26.04 Mpps	1.71 %

The third and last program tested in this section is the XDP_ADJUST_TAIL. Table 15 compares both versions of the program. The optimized version achieves a speedup of 81.38x.

Table 15: Performance of the XDP_ADJUST_TAIL program. *Source: own elaboration*

Version	Latency	II	Throughput	Area usage
XDP_ADJUST_TAIL - non-optimized	241 cycles	242 cycles	0.64 Mpps	0.49 %
XDP_ADJUST_TAIL - optimized	6 cycles	3 cycles	52.08 Mpps	2.05 %

Finally, table 16 offers a summary of the performance achieved by the three optimized programs and the speedup obtained because of the optimizations.

Table 16: Summary of the performance of the tested programs optimized version. *Source: own elaboration*

Program	Throughput	Speedup of the optimizations	Area usage
XDP1	52.08 Mpps	18x	0.06 %
XDP2	26.04 Mpps	17.3x	1.71 %
XDP_AJUST_TAIL	52.08 Mpps	81.38x	2.05 %

The best performance has been achieved by the programs XDP1 and XDP_ADJUST_TAIL. The first has the advantage of dropping every packet it receives, so it does not write any output packet. Usually, writing packets takes more time than parsing them. The writing phase is less predictable as it depends on which headers have been previously parsed, and conditional tasks tend to make the design slower. The second has few conditional tasks. It always writes the whole packet directly, or it writes a shrunk version of it. These two defined actions make the design very straightforward, as the packet will either take one path or another. However, this performance comes at the cost of area usage, as the design has an infrastructure for both potential actions (while a normal program would only have the infrastructure for one action), and the area used is the greatest of the three programs. The slowest program is XDP2 as it has many conditional tasks, especially in the writing phase. An example of a task that has conditional and unpredictable behavior is writing to an output stream without knowing at compile time the positions of the stream that are written.

The speedup of the optimizations of XDP1 and XDP2 is similar and lower than the one in XDP_ADJUST_TAIL. The main reason why XDP_ADJUST_TAIL's optimizations improve the design much more is that the program is more complex. It performs larger operations - like computing the IPv4 and ICMP checksum - which benefit a lot from the optimizations. Overall, **the optimizations have proven to significantly improve the performance of the design**.

8.3 Comparison with hXDP

The throughput of the programs evaluated in the previous section is also tested in the hXDP paper [2]. Figure 43 provides the results that appear there. An estimation has to be made to convert the figure into numbers in order to compare them with the results of the previous section. Table 17 provides an estimation.

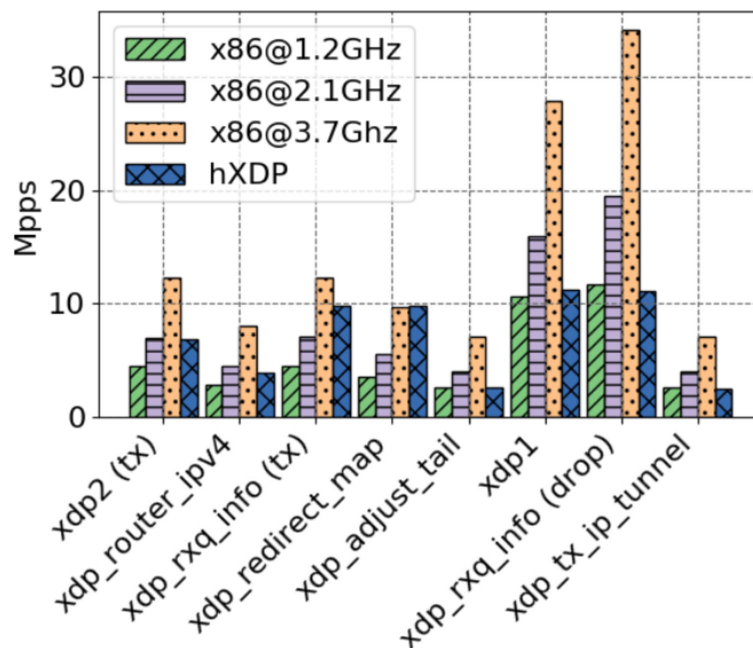

Figure 43: Throughput of Linux's XDP programs in hXDP. *Source: [2]*

Table 17: Estimation of the throughput of Linux's programs in hXDP. *Source: own elaboration*

Program	Estimated throughput
XDP1	11 Mpps
XDP2	6-7 Mpps
XDP_ADJUST_TAIL	2-3 Mpps

Table 18 compares the three Linux example programs evaluated in the previous section with the performance they have in hXDP. **The speedup ranges between 3.72x and 26.04x**, and the area usage of these programs is lower than in hXDP, as the whole hXDP soft-processor uses about 15% of the resources of the board.

Table 18: Comparison of the performance of the optimized programs with hXDP. *Source: own elaboration*

Program	hXDP throughput	Optimized version throughput	Speedup	Area usage
XDP1	11 Mpps	52.08 Mpps	4.73x	0.06 %
XDP2	6-7 Mpps	26.04 Mpps	3.72x - 4.34x	1.71 %
XDP_ADJUST_TAIL	2-3 Mpps	52.08 Mpps	17.36x - 26.04x	2.05 %

The program with the best throughput is XDP1, as hXDP's design benefits from programs that live entirely in the NIC (i.e. the packets are always dropped). Following is XDP2, which fits inside the category of programs that forward the packets, and that acquire the average performance. The slowest programs are those that are considered "long". XDP_ADJUST_TAIL belongs to this category.

This classification between different types of programs has been obtained from the hXDP paper, and it is interesting to analyze whether the synthesized programs in this project follow the same trend or not.

The programs that have been tested in the previous section fit into two categories: predictable or not predictable. Both XDP1 and XDP_ADJUST_TAIL are considered predictable as they have few conditional tasks. On the other hand, XDP2 has more conditional tasks. Regarding the third category of hXDP, "long" programs appear to achieve great performance at the cost of using more resources.

8.4 Performance of the loaded IP

The value of `MAX_PACKET_LEN` has been set to 256 for this test. The reason why is that the AXI4-Streams are 256-bit wide, which means that the size has to be a multiple of 32. As 300 does not fulfill this restriction, 256 has been chosen even though any other multiple of 32 would work.

The Linux example's programs can be switched between its non-optimized and optimized version by just modifying the constant `OPTIMIZATION`. However, this program does not support such functionality as the tricks to make this work without negatively affecting the performance cannot be applied to this code because of the usage of AXI4-Streams. The proposed solution to that is manually modifying the name of the non-optimized/optimized function to match the top-function name, `PING_program`.

Table 19 compares the non-optimized with the optimized version. The achieved speedup is 33.38x.

Table 19: Performance of the loaded IP block. *Source: own elaboration*

Version	Latency	II	Throughput	Area usage
Adapter_PING - non-optimized	239 cycles	267 cycles	0.59 Mpps	1.51 %
Adapter_PING - optimized	17 cycles	8 cycles	19.53 Mpps	1.51 %

The main bottleneck of this program is the adapters. The main eBPF logic has an II of only 1 cycle. However, the access to read or write a top-function stream argument is slow. It takes 1 cycle to complete it, and each stream cannot be accessed more than once every cycle. Taking into account that the streams are 256-bit wide, a packet with a length of 256 Bytes will always take 8 cycles to process.

The conclusion from loading an eBPF program to an FPGA is that **the performance really depends on the board and on the design itself**. In this case, the design streams 256-bit wide AXI4-Streams. Another board could use other protocols. Furthermore, other designs also from the NetFPGA-SUME could use other interfaces, like AXI4 Memory-mapped arguments, or streams with different wide, which would definitely affect the throughput of the design. The design that has been loaded has still room from improvement.

9 Sustainability and social commitment

9.1 Environmental dimension

Consumption of the design:

- **Have you quantified the environmental impact of undertaking the project? What measures have you taken to reduce the impact? Have you quantified this reduction?**

  The main elements that have impacted the energy consumption are the PC, the workstation (that includes the NetFPGA-SUME) and the monitor.

  The total amount of time dedicated to this project has been 922.25 hours, as it is explained in 4.5 Modifications from the original work plan, and those hours have been spread out over a total of 115 working days.

  The PC consumes 48.52W under normal use, 7.38W in idle mode and 0.25W when it is off[6]. The consumption is calculated in figure 44.

$$PC\ consumption = 922.25h * (0.8 * 48.52W + 0.2 * 7.38W) + 115\ days * 16h/day * \\ * 0.25W = 37.6kWh$$

Figure 44: PC consumption. *Source: own elaboration*

The workstation consumes up to 850W when it performs stressing tasks, 8.346W in idle and 0.142W when it is off[7]. The total use of the Workstation has been 105 hours (task PP9), in a total of 14 days. The consumption is calculated in figure 45.

$$Workstation\ consumption = 105h * (0.8 * 850W + 0.2 * 8.346W) + 115\ days * 16h/day * \\ * 0.142W = 71.8kWh$$

Figure 45: Workstation consumption. *Source: own elaboration*

The monitor consumes an average of 35W when it is being used[8]. The consumption is calculated in figure 46.

[6]Source: `https://www.cnet.com/reviews/toshiba-portege-r835-review/`
[7]Source: `https://www.tomshardware.com/reviews/evga-supernova-850-g3-psu,4930-5.html`
[8]Source: `https://www.it.northwestern.edu/hardware/eco/stats.html`

$$Monitor\ consumption = \ 922.25h\ *\ 35W\ =\ 32.3kWh$$

Figure 46: Monitor consumption. *Source: own elaboration*

The total consumption is the addition of the previous consumption. The consumption is calculated in figure 47.

$$Total\ consumption = \ 37.6kWh\ +\ 71.8kWh\ +\ 32.3kWh\ =\ 141.7kWh$$

Figure 47: Total consumption of the project. *Source: own elaboration*

The consumption of the project has been estimated at **141.7kWh**.

In order to minimize the impact, the developer of the project has followed good practices regarding the usage of electronics. For example, the computers have only been on when they have been used and the workstation has only been used when it was necessary, so it has not been used before and after task PP9. The developer has also minimized printing and using paper to the point of going without using physical paper. Instead, information has been stored electronically.

It is hard to quantify these reductions, but they have already been taken into account in the calculation of the consumption, as the workstation has only been used for 14 days and all the computers have been off for 16 hours a day.

- **If you carried out the project again, could you use fewer resources?**

  The project could reduce the impact it has had by using devices that consume less energy. Both the workstation and the personal computer consume more than other models, so replacing them with newer low-consumption models would reduce the energy used. On the other hand, using only one machine instead of two could also reduce the impact. However, using only one machine means that it would require support for the NetFPGA-SUME, so that machine would probably be a desktop computer, which tends to consume more than laptops.

Ecological footprint:

- **What resources do you estimate will be used during the useful life of the project? What will be the environmental impact of these resources?**

  The main resource that will be used is the workstation, as it has the NetFPGA-SUME. The environmental impact is the one already discussed, 850W when it is performing stressing

tasks, 8.346W on idle and 0.142W when the machine is off. However, the board itself can consume up to 184.4W [46], so using another machine that consumes less as a whole is possible.

- **Will the project enable a reduction in the use of other resources? Overall, does the use of the project improve or worsen the ecological footprint?**

  This project enables a reduction in CPU usage as tasks can be offloaded to the FPGA. A high-end CPU can have a peak consumption of 165W[9]. Though it is slightly lower than the one of the NetFPGA-SUME, the FPGA offers a scalable solution to processing packets, as only the board is needed to be able to receive more traffic. Otherwise, acquiring a whole new system would mean requiring more resources. Overall, using an FPGA benefits the ecological footprint.

Environmental risks:

- **Could situations occur that could increase the project's ecological footprint?**

  The main risk of the project is the possibility that the FPGA board stopped working, and that a new one should be acquired. In this case, the ecological footprint would worsen as the resources required to manufacture a new FPGA would be needed.

9.2 Economic dimension

Invoice:

- **Have you quantified the cost (human and material resources) of undertaking the project? What decisions have you taken to reduce the cost? Have you quantified these savings?**

  The final cost of the project, updated after the modifications of the budget, has been calculated in section 5.3 Impact of the modifications from the original work plan. The estimated final cost of the project is $54,670.29. The project has not really taken many decisions to reduce the cost, as the budget has been tight since the beginning and there has not been any margin to have savings.

- **Is the expected cost similar to the final cost? Have you justified any differences (lessons learnt)?**

  The original budget is higher than the final one, as 0.71% has been saved. No decisions oriented at reducing the costs have been taken. Instead, the savings come because of

[9] `https://www.intel.com/content/www/us/en/products/sku/198017/intel-core-i910980xe-extreme-edition-processor-24-75m-cache-3-00-ghz/specifications.html`

the obstacles that have forced to modify the original planning and that has already been discussed in this report. It is not an unexpected result, as the main risks of the project have been identified since the beginning, and this outcome has always been considered as a potential result of the project.

Viability plan:

- **What cost do you estimate the project will have during its useful life? Could this cost be reduced to increase viability?**

  The main cost that this project will have during its useful life is the consumption of power and the FPGA board, as it may need to be replaced after its life expectancy. Furthermore, minimal maintenance of the system may be needed. These costs are not easy to reduce as they are minimal.

- **Have you considered the cost of adaptations/updates/repairs during the useful life of the project?**

  During the useful life of the project, the FPGA may need some maintenance. However, it could be considered a negligible cost as it would consist of a single worker occasionally checking that the board is working fine. But if newer functionalities are added to the board (i.e. different eBPF programs), the development of such would have a considerable cost. In that case, it would consist of the hourly wage of the developer. The amount of time needed would depend on the complexity of the program.

Economic risks:

- **Could situations occur that are detrimental to the project's viability?**

  Some situations that are out of the scope of this project could occur that could limit the viability of this project.

  On the one hand, the project depends on Xilinx's software platform to synthesize programs and load them to the FPGA. In the unlikely scenario that Xilinx decided to stop developing them, the project would be very limited. It is an extreme situation that may never happen, keeping in mind the recent acquisition of Xilinx from AMD.

  On the other hand, the NetFPGA-SUME is an expensive device, so an improper use that could break the board would mean buying another one. However, it is assumed that this risk is up to following good practices with the device.

9.3 Social dimension

Personal impact:

- **Has undertaking this project led to meaningful reflections at the personal, professional or ethical level among the people involved?**

  This project has introduced me to research. I consider it an area of innovation, and this project has proposed a solution in a way that no one had before. Some competencies that I have acquired during this project are learning new things by myself. Every technology and tool used - eBPF, XDP, PyMTL3, HLS, FPGA - was new for me. It has been a hard and complex project, that has faced some obstacles. The biggest lesson that I have learned is to keep trying even if nothing appears to go well. At the end of the day, every attempt and effort is important.

Social impact:

- **Who will benefit from the use of the project? Could any group be adversely affected by the project? To what extent?**

  This project benefits data centers that have to handle big quantities of network traffic. It also benefits the research community, as the project is open to anyone interested in learning more about it.

  No risks are considered that could affect any part of the society, as this project offers a solution to improve a very unique problem that does not directly replace any specific job. Conversely, this project positively affects a specific population segment, which is the workers of data centers, as it presents a solution to improve the performance of managing network traffic.

- **To what extent does the project solve the problem that was established initially?**

  The project has completed each and every objective that it originally had. To summarize, this project has investigated a need and its current state of the art. It has developed a methodology to synthesize and optimize eBPF programs, and it has loaded that design into the NetFPGA-SUME. Furthermore, it provides an open-source library to help anyone that is willing to use it.

Social risks:

- **Could situations occur in which the project adversely affects a specific population segment?**

  As it has already been mentioned, it is very unlikely that this project could affect a specific fragment of society. No risks are considered.

- **Could the project create any kind of dependency that puts users in a weak position?**

The project has a dependency on Vitis and Vivado. Both tools are property of Xilinx and the project requires using them. That puts the user in a weak position, especially with Vivado, as it is needed to create bitfiles for the board.

10 Conclusions

This project presents the design and implementation of eBPF programs offloaded to an FPGA board. It also introduces a prototype design of an eBPF program that has been loaded to the NetFPGA-SUME.

The performance obtained is 3.72 to 26.04 times better than the closest project to this one, hXDP. The reason is that hXDP acts as a processor instead of taking full advantage of hardware acceleration. The differential fact that leads to the remarkable improvement in performance comes at the cost of the effort of both transforming the eBPF program to make it synthesize and afterward applying the optimizations to make it faster. To make this process faster, an open-source library with optimized common eBPF functions is provided[10].

The concept of a trade-off between area and performance has appeared in this project. This idea is linked with what a specific circuit is. It has been very interesting to realize the limitations and differences between a general-purpose and a specific design. And that is the key to understanding why this project has obtained better performance than hXDP.

This project has developed a successful design to load an eBPF program to an FPGA board, which has proven that the guidance given about synthesizing eBPF programs is feasible. However, this part of the project is completely dependent on the FPGA, so it is applicable to the NetFPGA-SUME but may not be compatible with other FPGAs. On the other hand, eBPF programs can be exported as functions inside a large IP, hence making them - and the library provided - independent of the board.

Some of the benefits of offloading eBPF programs are scalability and security. A big portion of a CPU can be dedicated to managing data from the network, so an increase in the amount of traffic could be difficult to deal with as CPUs are not easy to expand. Conversely, as long as there are free PCIe ports, multiple FPGAs can be used at the same time. Another convenience of offloading these programs to an FPGA is the inherent security that isolating tasks from the kernel offers.

Even though this project has faced some obstacles that were identified - loading the modules to the NetFPGA - and some that were unexpected - getting COVID -, all of the objectives that were set at the beginning of the research have been accomplished. Furthermore, a new objective was added during the development of the research - comparing the performance with hXDP -, which has also been completed.

On a personal level, developing this project has greatly contributed to my development. On the

[10] `https://github.com/aviba2000/eBPF_NetFPGA-SUME/tree/main/00_Common_Libraries`

one hand, doing research has been a fulfilling experience. On the other hand, getting to know the American culture has been amazing.

10.1 Future work

The project can be expanded in two different ways: synthesizing eBPF programs or loading eBPF programs to an FPGA. It is curious the fact that those parts even use different tools: Vitis HLS and Vivado respectively.

Regarding the synthesis of eBPF programs, there is not much work left, as every potential optimization has been analyzed to decide whether it should be applied. However, it would be interesting to offer more eBPF function helpers like `bpf_csum_diff` or `bpf_tail_call`. They could be implemented as independent functions of the library.

There are lots of opportunities in the second part. The design that has been loaded to the FPGA can have more functionalities added and the testing environment of the FPGA can improve. Some interesting ideas worth exploring are the following:

- **Maps:** eBPF maps have not been tested in the FPGA, and the adapter does not support them yet. A potential solution would be to use AXI4 Memory-mapped blocks to create a static memory region shared by all the loaded eBPF programs that could be also accessed by user space, hence simulating eBPF maps.
- **Different designs:** The loaded eBPF program uses the design from `reference_nic`, and using other designs or modifying this one - like using other protocols instead of AXI4-Stream - could improve the design.
- **Larger testing environment:** The testing environment for the NetFPGA is very limited, as the only tool that it has is `ping`. Adding new functionalities to this environment will enhance the process of debugging, testing and evaluating the board.

Finally, an extension of this project would consist of creating a tool that automatically transforms and synthesizes eBPF programs. The objective would be to load those programs to the NetFPGA-SUME without requiring any manual modifications of the code. This tool would act like hXDP - in the way that it would also receive unmodified eBPF programs - but the output would be an IP block with custom hardware for that specific eBPF program.

10.2 Completion of technical competencies

This project has developed the following technical competencies from the IT specialization:

CTI1:

To define, plan and manage the installation of the ICT infrastructure of the organization.

- **[CTI1.2]** To select, design, deploy, integrate and manage communication networks and infrastructures in an organization (*Enough*):
  The combination of technologies such as XDP and eBPF has the potential to create an infrastructure to manage data that comes from the network. Offloading those functionalities to the NetFPGA-SUME has built a system to manage a communication network offering better performance than the most similar existing solution.

CTI2:

To guarantee that the ICT systems of an organization operate adequately, are secure and adequately installed, documented, personalized, maintained, updated and substituted, and the people of the organization receive a correct ICT support.

- **[CTI2.3]** To demonstrate comprehension, apply and manage the reliability and security of the computer systems (CEI C6)(*A little bit*):
  Offloading tasks does not only benefit in terms of performance and freeing the main system from spending resources on executing those tasks, but it also adds a layer of security. As it has already been mentioned in this project, eBPF programs run in kernel space, which is very delicate. Exploiting a vulnerability could cause massive damage to the system. Running those programs in the FPGA prevents that from happening, which results in higher security. Furthermore, eBPF programs are widely used for security, as they inspect packets and can discard them based on rules (i.e. a complex firewall).

CTI3:

To design solutions which integrate hardware, software and communication technologies (and capacity to develop specific solutions of systems software) for distributed systems and ubiquitous computation devices.

- **[CTI3.3]** To design, establish and configure networks and services (*In depth*):
  This project has revolved around designing a solution to offload networking processing tasks to an FPGA. Loading them to the board has required designing and configuring a specific solution for the NetFPGA-SUME. This is the competence that defines this research better as the main goal of this project is to offer a design to manage network traffic.

CTI4:

To use methodologies centered on the user and the organization to develop, evaluate and manage applications and systems based on the information technologies which ensure the accessibility, ergonomics and usability of the systems (A little bit).

During this project, an open-source library of common eBPF functions has been developed. The goal of doing it has always been to provide a faster and easier way to effectively transform eBPF programs and make them synthesizable. Some examples of eBPF programs are given to illustrate how to use the library, and there are testing files that ensure that the functions behave as expected.

Bibliography

References

[1] Xilinx. *Vivado Design Suite: AXI Reference Guide*. [Online; Retrieved on June 16, 2022]. 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[2] Marco Spaziani Brunella et al. *hXDP: Efficient Software Packet Processing on FPGA NICs*. [Online; Retrieved on February 8, 2022]. Nov. 2020. URL: https://www.usenix.org/conference/osdi20/presentation/brunella.

[3] IETF. *RFC 792 - Internet Control Message Protocol*. [Online; Retrieved on June 16, 2022]. 1981. URL: https://datatracker.ietf.org/doc/html/rfc792.

[4] M. Fingeroff. *High-level Synthesis: Blue Book*. Xlibris Corporation, 2010. ISBN: 9781450097246.

[5] TechTarget. *What is IP core (intellectual property core)?* [Online; Retrieved on June 14, 2022]. 2011. URL: https://www.techtarget.com/whatis/definition/IP-core-intellectual-property-core.

[6] IETF. *RFC 791 - Internet Protocol*. [Online; Retrieved on June 16, 2022]. 1981. URL: https://datatracker.ietf.org/doc/html/rfc791.

[7] IETF. *RFC 2460 - Internet Protocol*. [Online; Retrieved on June 16, 2022]. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2460.

[8] ni. *FPGA Fundamentals - NI*. [Online; Retrieved on March 24, 2022]. 2021. URL: https://www.ni.com/en-us/innovations/white-papers/08/fpga-fundamentals.html.

[9] IEEE. *IEEE 802.1Q-2018*. [Online; Retrieved on June 16, 2022]. 1998. URL: https://standards.ieee.org/ieee/802.1Q/6844/.

[10] Toke Höiland-Jörgensen et al. *The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel*. [Online; Retrieved on January 20, 2022]. Heraklion, Greece, 2018. DOI: 10.1145/3281411.3281443. URL: https://doi.org/10.1145/3281411.3281443.

[11] Alan Mauldin. *Global Internet Traffic and Capacity Return to Regularly Scheduled Programming*. [Online; Retrieved on February 28, 2022]. 2021. URL: https://blog.telegeography.com/internet-traffic-and-capacity-return-to-their-regularly-scheduled-programming.

[12] Cilium. *BPF and XDP Reference Guide*. [Online; Retrieved on January 25, 2022]. URL: https://docs.cilium.io/en/latest/bpf/.

[13] *eBPF - Introduction, Tutorials & Community Resources*. [Online; Retrieved on June 13, 2022]. URL: https://ebpf.io/.

[14] *Pymtl 3 (Mamba), an open-source Python-based hardware generation, simulation, and verification framework*. [Online; Retrieved on February 7, 2022]. URL: https://github.com/pymtl/pymtl3.

[15] *EuropeColorado Program — International Programs — University of Colorado Boulder*. [Online; Retrieved on March 30, 2022]. URL: https://www.colorado.edu/engineering-international/europe-colorado-program.

[16] Noa Zilberman et al. *NetFPGA SUME: Toward 100 Gbps as Research Commodity*. [Online; Retrieved on February 4, 2022]. 2014. DOI: 10.1109/MM.2014.61. URL: https://ieeexplore.ieee.org/document/6866035?arnumber=6866035.

[17] *hBPF = eBPF in hardware*. [Online; Retrieved on February 9, 2022]. URL: https://github.com/rprinz08/hBPF.

[18] Netronome. *Agilio SmartNIC and Software*. [Online; Retrieved on February 6, 2022]. URL: https://www.netronome.com/.

[19] Jakub Kicinski and Nicolaas Viljoen. *eBPF Hardware Offload to SmartNICs: cls bpf and XDP*. [Online; Retrieved on February 9, 2022]. URL: https://legacy.netdevconf.info/1.2/papers/eBPF_HW_OFFLOAD.pdf.

[20] Jakub Kicinski and Nicolaas Viljoen. *XDP Hardware Offload: Current Work, Debugging and Edge Cases*. [Online; Retrieved on February 9, 2022]. URL: https://legacy.netdevconf.info/2.2/papers/viljoen-xdpoffload-talk.pdf.

[21] Quentin Monnet. *The Challenges of XDP Hardware Offload*. [Online; Retrieved on April 19, 2022]. 2018. URL: https://archive.fosdem.org/2018/schedule/event/xdp/attachments/slides/2220/export/events/attachments/xdp/slides/2220/fosdem18_SdN_NFV_qmonnet_XDPoffload.pdf.

[22] Razvan Nane et al. *A Survey and Evaluation of FPGA High-Level Synthesis Tools*. [Online; Retrieved on March 15, 2022]. 2016. DOI: 10.1109/TCAD.2015.2513673. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7368920.

[23] *Xilinx - Adaptable. Intelligent.* [Online; Retrieved on March 1, 2022]. 2021. URL: https://www.xilinx.com/.

[24] Xilinx. *Vitis High-Level Synthesis User Guide*. [Online; Retrieved on March 23, 2022]. 2021. URL: https://docs.xilinx.com/api/khub/maps/oK7qoHuV~Mn874fOMSk49Q/attachments/LmULqnFrteyQU~na0xHevQ/content?Ft-Calling-App=ft%5C%2Fturnkey-portal&Ft-Calling-App-Version=3.11.15&download=true.

[25] *Basic examples for Vitis HLS.* [Online; Retrieved on March 1, 2022]. 2021. URL: https://github.com/Xilinx/Vitis-HLS-Introductory-Examples.

[26] Xilinx. *Tutorials and Examples.* [Online; Retrieved on March 1, 2022]. 2021. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Tutorials-and-Examples.

[27] so-logic. *Basic HLS Tutorial.* [Online; Retrieved on March 1, 2022]. 2017. URL: https://www.so-logic.net/documents/upload/Basic_HLS_Tutorial.pdf.

[28] imperix. *Xilinx Vitis HLS introduction - imperix.* [Online; Retrieved on March 5, 2022]. 2017. URL: https://imperix.com/doc/help/xilinx-vitis-hls.

[29] C. Unsalan and B. Tar. *Digital System Design with FPGA: Implementation Using Verilog and VHDL.* McGraw-Hill Education, 2017. ISBN: 9781259837913.

[30] *NetFPGA.* [Online; Retrieved on February 4, 2022]. URL: https://netfpga.org/NetFPGA-SUME.html.

[31] ProjectManager. *Waterfall Methodology : The Ultimate Guide to the Waterfall Model.* [Online; Retrieved on February 22, 2022]. URL: https://www.projectmanager.com/waterfall-methodology.

[32] Mary Lotz. *Waterfall vs. Agile: Which is the Right Development Methodology for Your Project?* [Online; Retrieved on February 22, 2022]. 2018. URL: https://www.seguetech.com/waterfall-vs-agile-methodology/.

[33] *NetFPGA.* [Online; Retrieved on June 10, 2022]. URL: https://netfpga.org/_pages/1G-License.html.

[34] *Linux kernel source tree.* [Online; Retrieved on February 7, 2022]. URL: https://github.com/torvalds/linux/.

[35] School of Electrical and Cornell University Computer Engineering. *ECE 5745 Complex Digital ASIC Design, Tutorial 3: PyMTL3 Hardware Modeling Framework.* [Online; Retrieved on February 7, 2022]. 2021. URL: https://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-tut3-pymtl.pdf.

[36] *NetFPGA-SUME public repository.* [Online; Retrieved on June 14, 2022]. URL: https://github.com/NetFPGA/NetFPGA-SUME-public.

[37] Quentin Monnet. *Dive into BPF: a list of reading material.* [Online; Retrieved on February 3, 2022]. 2016. URL: https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/.

[38] *XDP Tutorial.* [Online; Retrieved on January 20, 2022]. URL: https://github.com/xdp-project/xdp-tutorial.

[39] *Payscale - Salary Comparison, Salary Survey, Search Wages.* [Online; Retrieved on March 14, 2022]. URL: https://www.payscale.com/.

[40] *Amazon.com - Spend less. Smile more.* [Online; Retrieved on March 14, 2022]. URL: https://www.amazon.com/.

[41] *Free Colorado Payroll Calculator — 2020 CO Taxes Rate — OnPay.* [Online; Retrieved on March 14, 2022]. URL: https://onpay.com/payroll/calculator-tax-rates/colorado/.

[42] *Boulder, CO Office Space for Lease & Rent — PropertyShark.* [Online; Retrieved on March 15, 2022]. URL: https://www.propertyshark.com/cre/office/us/co/boulder/.

[43] Andrew Moore. *FPGAs For Dummies.* [Online; Retrieved on June 8, 2022]. 2017. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/misc/fpgas-for-dummies-ebook.pdf.

[44] Nate Eastland. *FPGA – Configurable Logic Block – Digilent Blog.* [Online; Retrieved on March 24, 2022]. 2021. URL: https://digilent.com/blog/fpga-configurable-logic-block/.

[45] Xilinx. *Xilinx XAPP465 Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs application note - xapp465.pdf.* [Online; Retrieved on March 24, 2022]. 2005. URL: https://www.xilinx.com/support/documentation/application_notes/xapp465.pdf.

[46] Digilent. *NetFPGA-SUME Reference Manual.* [Online; Retrieved on June 8, 2022]. 2016. URL: https://reference.digilentinc.com/_media/sume:netfpga-sume_rm.pdf.

[47] Steven McCanne and Van Jacobson. *The BSD Packet Filter: A New Architecture for User-Level Packet Capture.* San Diego, California, 1993. URL: https://dl.acm.org/doi/10.5555/1267303.1267305.

[48] Toke Höiland-Jörgensen and Jesper Dangaard Brouer. *XDP – challenges and future work.* [Online; Retrieved on January 20, 2022]. 2018. URL: https://people.netfilter.org/hawk/presentations/LinuxPlumbers2018/lpc18-xdp-future.pdf.

[49] D. Calavera and L. Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking.* 2019.

[50] *pytest · PyPI.* [Online; Retrieved on March 22, 2022]. URL: https://pypi.org/project/pytest/.

[51] *GTKWave.* [Online; Retrieved on March 22, 2022]. URL: http://gtkwave.sourceforge.net/.

[52] *AMD — High Performance & Adaptative Computing.* [Online; Retrieved on March 1, 2022]. URL: https://www.amd.com/en.

[53] AMD. *AMD Completes Acquisition of Xilinx*. [Online; Retrieved on March 1, 2022]. 2022. URL: https://www.amd.com/en/press-releases/2022-02-14-amd-completes-acquisition-xilinx.

[54] Xilinx. *Kernel Interfaces*. [Online; Retrieved on May 12, 2022]. 2022. URL: https://docs.xilinx.com/r/2020.2-English/ug1393-vitis-application-acceleration/Kernel-Interfaces?tocId=7mIj9uvWncupVBowsHoM3Q.

[55] *Vivado HLS Multidimensional Array Question*. [Online; Retrieved on May 16, 2022]. 2020. URL: https://support.xilinx.com/s/question/0D52E00006hpLx6SAE/vivado-hls-multidimensional-array-question?language=en_US.

[56] Xilinx. *Vivado Design Suite User Guide: Implementation (UG904)*. [Online; Retrieved on June 14, 2022]. 2022. URL: https://docs.xilinx.com/r/en-US/ug904-vivado-implementation.

Other sources

[57] Jason Wang. *Accelerating VM networking through XDP*. [Online; Retrieved on January 22, 2022]. 2017. URL: https://events19.linuxfoundation.cn/wp-content/uploads/2017/11/Accelerating-VM-Networking-through-XDP_Jason-Wang.pdf.

[58] Magnus Karlsson. *AF_XDP Sockets: High Performance Networking for Cloud-Native Networking Technology Guide*. [Online; Retrieved on January 27, 2022]. URL: https://builders.intel.com/docs/networkbuilders/af-xdp-sockets-high-performance-networking-for-cloud-native-networking-technology-guide.pdf.

[59] The Linux Kernel. *AF_XDP*. [Online; Retrieved on January 27, 2022]. URL: https://www.kernel.org/doc/html/v5.0/networking/af_xdp.html.

[60] Marcelo De Branches. *Transparent Network Acceleration with XDP*. [Presentation; Assisted on February 11, 2022]. 2022.

[61] Sebastiano Miano et al. *Creating Complex Network Services with eBPF: Experience and Lessons Learned*. [Online; Retrieved on January 25, 2022]. 2018. DOI: 10.1109/HPSR.2018.8850758. URL: https://ieeexplore.ieee.org/document/8850758.

[62] Matt Fleming. *A thorough introduction to eBPF*. [Online; Retrieved on January 24 8, 2022]. 2017. URL: https://lwn.net/Articles/740157/.

[63] Lavanya Chockalingam. *What Is eBPF and Why Does It Matter for Observability?* [Online; Retrieved on January 28, 2022]. 2021. URL: https://newrelic.com/blog/best-practices/what-is-ebpf.

[64] Brendan Gregg. *Linux Extended BPF (eBPF) Tracing Tools.* [Online; Retrieved on January 31, 2022]. URL: https://www.brendangregg.com/ebpf.html.

[65] Michael Bolin. *How I ended up writing opensnoop in pure C using eBPF.* [Online; Retrieved on February 2, 2022]. 2018. URL: https://bolinfest.github.io/opensnoop-native/.

[66] *BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more.* [Online; Retrieved on February 1, 2022]. URL: https://github.com/iovisor/bcc.

[67] Greg Marsden. *BPF: A Tour of Program Types.* [Online; Retrieved on February 2, 2022]. 2019. URL: https://blogs.oracle.com/linux/post/bpf-a-tour-of-program-types.

[68] *Full introduction EBPF-Concept.* [Online; Retrieved on February 3, 2022]. URL: https://www.programmerall.com/article/35711068253/.

[69] Brandon Cook. *Bottom-up eBPF.* [Online; Retrieved on February 4, 2022]. 2019. URL: https://medium.com/@phylake/bottom-up-ebpf-d7ca9cbe8321.

[70] *Getting Started with eBPF and Go.* [Online; Retrieved on February 2, 2022]. 2021. URL: https://networkop.co.uk/post/2021-03-ebpf-intro/.

[71] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. *Linux Socket Filtering aka Berkeley Packet Filter (BPF).* [Online; Retrieved on February 5, 2022]. URL: https://www.kernel.org/doc/Documentation/networking/filter.txt.

[72] *eBPF XDP: The Basics and a Quick Tutorial.* [Online; Retrieved on February 5, 2022]. URL: https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/.

[73] Marcos A. M. Vieira. *Fast Packet Processing using eBPF and XDP.* [Online; Retrieved on February 6, 2022]. URL: http://evcomp.dcc.ufmg.br/wp-content/uploads/eBPF-XDP.pdf.

[74] Jakub MACKOVIČ. *Packet Filtering Using XDP.* [Online; Retrieved on February 5, 2022]. 2019. URL: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=197340.

[75] *eBPF and XDP samples.* [Online; Retrieved on February 9, 2022]. 2019. URL: https://github.com/netoptimizer/prototype-kernel/tree/master/kernel/samples/bpf.

[76] *Welcome to PyMTL3 documentation!* [Online; Retrieved on February 7, 2022]. URL: https://pymtl3.readthedocs.io/en/latest/index.html.

[77] *PyMTL3 Memory, Project repo for cache/memory related components implemented in pymtl3.* [Online; Retrieved on February 7, 2022]. 2020. URL: https://github.com/cornell-brg/pymtl3-mem.

[78] *Introduction to NetFPGA And OpenFlow.* [Online; Retrieved on February 5, 2022]. URL: https://www.cct.lsu.edu/~xuelin/openflow/IntroductionToOpenFlowNetFPGA.pdf.

[79] Digilent. *NetFPGA SUME Reference Manual.* [Online; Retrieved on February 5, 2022]. URL: https://digilent.com/reference/sume:refmanual.

[80] *A bcc-based Python eBPF (Extended-Berkeley-Packet-Filter) wrapper.* [Online; Retrieved on February 8, 2022]. URL: https://github.com/dany74q/pyebpf.

[81] *Userspace eBPF VM.* [Online; Retrieved on February 8, 2022]. URL: https://github.com/iovisor/ubpf.

[82] Netronome. *eBPF Offload Getting Started Guide, Netronome CX SmartNIC.* [Online; Retrieved on February 10, 2022]. 2018. URL: http://www.netronome.com/documents/305/eBPF-Getting_Started_Guide.pdf.

[83] Daniel Firestone et al. *Azure Accelerated Networking: SmartNICs in the Public Cloud.* [Online; Retrieved on February 6, 2022]. Renton, WA, USA, 2018. URL: https://dl.acm.org/doi/10.5555/3307441.3307446.

[84] Netronome. *Agilio eBPF Software.* [Online; Retrieved on February 9, 2022]. URL: https://www.netronome.com/products/agilio-software/agilio-ebpf-software/.

[85] Rishabh Poddar et al. *SafeBricks: Shielding Network Functions in the Cloud.* [Online; Retrieved on February 3, 2022]. Renton, WA, Apr. 2018. URL: https://www.usenix.org/conference/nsdi18/presentation/poddar.

[86] Stewart Grant et al. *SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud.* [Online; Retrieved on February 4, 2022]. Virtual Event, USA, 2020. DOI: 10.1145/3387514.3405895. URL: https://doi.org/10.1145/3387514.3405895.

[87] *Hardware Enclaves & Intel SGX.* [Online; Retrieved on February 5, 2022]. URL: https://inst.eecs.berkeley.edu/~cs261/fa18/slides/Hardware_Enclaves.pdf.

[88] Stephen Checkoway and Hovav Shacham. *Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface.* [Online; Retrieved on February 6, 2022]. Houston, Texas, USA, 2013. DOI: 10.1145/2451116.2451145. URL: https://doi.org/10.1145/2451116.2451145.

[89] Mohammad J. Hashemi, Greg Cusack, and Eric Keller. *Towards Evaluation of NIDSs in Adversarial Setting.* [Online; Retrieved on February 8, 2022]. Orlando, FL, USA, 2019. URL: https://eric-keller.github.io/papers/2019/adversarial_NIDS_BigDAMA2019.pdf.

[90] Benjamin C. Nilsen. *Fuzzing the Berkeley Packet Filter.* English. [Online; Retrieved on February 8, 2022]. 2020. URL: https://www.proquest.com/docview/2438697711.

[91] Arseniy Zaostrovnykh et al. *Verifying Software Network Functions with No Verification Expertise*. [Online; Retrieved on February 7, 2022]. Huntsville, Ontario, Canada, 2019. DOI: 10.1145/3341301.3359647. URL: https://doi.org/10.1145/3341301.3359647.

[92] Aimee Coughlin et al. *Breaking the Trust Dependence on Third Party Processes for Reconfigurable Secure Hardware*. [Online; Retrieved on February 7, 2022]. 2019. URL: https://eric-keller.github.io/papers/2019/sdsh_fpga2019.pdf.

[93] David Horn. *What's different between Vivado and Vitis?* [Online; Retrieved on March 1, 2022]. 2022. URL: https://digilent.com/blog/whats-different-between-vivado-and-vitis/.

[94] Ash Bhalgat. *Choosing the Best SmartNIC*. [Online; Retrieved on February 2, 2022]. 2021. URL: https://developer.nvidia.com/blog/choosing-the-best-dpu-based-smartnic/.

[95] Sergey Bratus. *The Journey of a Packet Through the Linux Network Stack*. [Online; Retrieved on February 4, 2022]. URL: https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Lab9_modified.pdf.

[96] A.M.Sekhar Reddy. *Linux Networking And Useful Tips for Real-Time Applications*. [Online; Retrieved on February 4, 2022]. 2014. URL: http://amsekharkernel.blogspot.com/2014/08/what-is-skb-in-linux-kernel-what-are.html.

Appendix A. Example of an eBPF program

This appendix defines and explains a typical eBPF program. The program has been elaborated in this project, but it follows the style of the xdp-tutorial [38], and some functions are originally from the tutorial.

```
1  SEC("xdp_vlan_modify")
2  int xdp_vlan_swap_func(struct xdp_md *ctx) {
3      void *data_end = (void *)(long)ctx->data_end;
4      void *data = (void *)(long)ctx->data;
5
6      struct hdr_cursor nh;
7      int nh_type;
8      nh.pos = data;
9
10     struct ethhdr *eth;
11     nh_type = parse_ethhdr(&nh, data_end, &eth);
12     if (nh_type < 0)
13         return XDP_DROP;
14
15     if (proto_is_vlan(eth->h_proto))
16         vlan_tag_pop(ctx, eth);
17     else
18         vlan_tag_push(ctx, eth, 1);
19
20     return XDP_PASS;
21 }
```

Figure 48: Example of an eBPF program that modifies the packet length by adding or removing a VLAN header. *Source: own elaboration*

Figure 48 provides an example of an eBPF program that parses the Ethernet header and adds a VLAN header if there is none, and removes a VLAN header if there is one.

Line 1 declares the name of the eBPF program. XDP will load this program using the name in `SEC`. Line 2 declares the function, that has the argument XDP eBPF programs have, a `struct xdp_md`. Line 3 and 4 create two local variables, `data` and `data_end`, that are used to easily access the beginning and the end of the packet. Line 6 defines a `hdr_cursor`, which is basically a data structure that only has one element, `void *pos`. It is used as a pointer to keep track of the position of the packet the program is at. The reason to make a `struct` is to make it easier to access it and modify its contents.

The main logic of the program starts at line 11, where a function that parses the Ethernet and VLAN headers (if there are any) is invoked. This function is analyzed later. Line 12 and 13 drop the packet if the function that parsed the Ethernet header returns an error code. Lines 15 to 18 check if there is a VLAN header in function `proto_is_vlan`, and remove it if there is one or add one if there is not. The three functions are analyzed next. Finally, the program returns `XDP_PASS` so that the packet goes to the network stack.

```
1  static __always_inline int parse_ethhdr(struct hdr_cursor *nh, void
       *data_end, struct ethhdr **ethhdr) {
2      struct ethhdr *eth = nh->pos;
3      int hdrsize = sizeof(*eth);
4
5      if (nh->pos + hdrsize > data_end)
6              return -1;
7
8      nh->pos += hdrsize;
9      *ethhdr = eth;
10
11     struct vlan_hdr *vhdr = nh->pos;
12     int vhdrsize = sizeof(*vhdr);
13     int i;
14     __u16 h_proto = eth->h_proto;
15
16     #pragma unroll
17     for (i = 0; i < VLAN_MAX_DEPTH; i++) {
18         vhdr = nh->pos;
19         if (nh->pos + vhdrsize <= data_end && proto_is_vlan(h_proto)
               ) {
20             nh->pos += vhdrsize;
21             h_proto = vhdr->h_vlan_encapsulated_proto;
22         } else {
23             break;
24         }
25     }
26     return h_proto;
27 }
```

Figure 49: Function that parses an Ethernet and VLAN headers. *Source: own elaboration*

Figure 49 is the code of the function that parses the Ethernet and VLAN headers. Line 1 declares the function, which has to be inlined. Otherwise, the eBPF verifier would not load the eBPF program. Line 2 casts the current position of the packet into an Ethernet header. Lines 5 and 6 verify that the following memory accesses are inside the boundary of the packet or return

an error code. Without that check, the verifier discards the program. Even if the check were redundant, the verifier would not load the program if it had been removed. Line 8 updates the position variable to point to the end of the Ethernet header. Line 9 updates the pointer to the Ethernet header. Lines 11 to 14 cast the position to a VLAN header data structure. Line 16 has a compiler directive to make sure the loop is unrolled. Otherwise, the program will not be loaded. Lines 17 to 25 check if there are any VLAN headers, and parse every one there is. The value of `VLAN_MAX_DEPTH` is constant and is previously defined. Otherwise, the loop would not be able to be unrolled. Line 26 returns the protocol of the next header to parse. It is common for the parsing functions to return this value.

```
1 static __always_inline int proto_is_vlan(__u16 h_proto) {
2     return !!(h_proto == bpf_htons(ETH_P_8021Q) || h_proto ==
          bpf_htons(ETH_P_8021AD));
3 }
```

Figure 50: Function that evaluates if the protocol of the next header is VLAN. *Source: xdp-tutorial [38]*

Figure 50 is a function that only compares a value to two other ones, which are the potential values a VLAN protocol is identified with. The reader should note that the values from a raw packet do not necessarily follow the endianness of the machine that is executing the eBPF program, so the function `bpf_htons` adapts the value.

Figure 51 is a function that pops the outermost VLAN header. Lines 1 to 6 declare the functions and some local variables. Lines 8 and 9 check if there is a VLAN header - an error code is returned if there is not one. Line 11 casts a VLAN header based on the position it should be in (just after the Ethernet header). Lines 13 and 14 check that the VLAN header actually fits in the boundaries of the packet. Lines 16 and 17 save the VLAN ID and the next header's protocol number respectively. Line 19 copies the whole Ethernet header to a local variable. Lines 21 and 22 modify the head of the packet using the helper function `bpf_xdp_adjust_head`. This function basically shrinks the packet, by the beginning of it, the number of bytes of a VLAN header. If it fails, it returns an error code. Lines 24 to 27 update the position where the Ethernet header should now be, update the `data_end` variable and check that an Ethernet header fits. It is a redundant comparison but not doing it would make the verifier discard the program. Line 29 moves the previously copied Ethernet header to its new position. Line 30 updates the Ethernet next protocol that has been previously copied. Line 32 returns the ID of the removed VLAN header.

Figure 52 is a function that adds a VLAN header. Lines 1 to 4 declare the functions and

```
1 static __always_inline int vlan_tag_pop(struct xdp_md *ctx, struct
      ethhdr *eth) {
2     void *data_end = (void *)(long)ctx->data_end;
3     struct ethhdr eth_cpy;
4     struct vlan_hdr *vlh;
5     __be16 h_proto;
6     int vlid;
7
8     if (!proto_is_vlan(eth->h_proto))
9         return -1;
10
11    vlh = (void *)(eth + 1);
12
13    if (vlh + 1 > data_end)
14        return -1;
15
16    vlid = bpf_ntohs(vlh->h_vlan_TCI);
17    h_proto = vlh->h_vlan_encapsulated_proto;
18
19    __builtin_memcpy(&eth_cpy, eth, sizeof(eth_cpy));
20
21    if (bpf_xdp_adjust_head(ctx, (int)sizeof(*vlh)))
22        return -1;
23
24    eth = (void *)(long)ctx->data;
25    data_end = (void *)(long)ctx->data_end;
26    if (eth + 1 > data_end)
27        return -1;
28
29    __builtin_memcpy(eth, &eth_cpy, sizeof(*eth));
30    eth->h_proto = h_proto;
31
32    return vlid;
33 }
```

Figure 51: Function that removes the outermost VLAN header. *Source: xdp-tutorial [38]*

some local variables. Line 6 copies the whole Ethernet header to a local variable. Lines 8 and 9 modify the head of the packet using the helper function `bpf_xdp_adjust_head`. Using a negative value as the size to be shrunk, the packet is expanded. In this case, the packet is made bigger to fit a new VLAN header. If the function fails, it returns an error code. Lines 11 and 12 update the `data_end` variable and update the position where the Ethernet header should now be. Lines 14 and 15 check that an Ethernet header fits in the packet boundaries. Line 17 moves the previously copied Ethernet header to its new position. Lines 18 to 21 make a VLAN header data structure point to the position it has to be in, and then check that a VLAN header actually fits in the packet. Lines 23 and 24 write the values of the new header. Line 25 updates the next protocol header value of the Ethernet header with the code of the VLAN header. Line 26 returns zero as the function has successfully ended.

```
1  static __always_inline int vlan_tag_push(struct xdp_md *ctx, struct
       ethhdr *eth, int vlid) {
2      void *data_end = (void *)(long)ctx->data_end;
3      struct ethhdr eth_cpy;
4      struct vlan_hdr *vlh;
5
6      __builtin_memcpy(&eth_cpy, eth, sizeof(eth_cpy));
7
8      if (bpf_xdp_adjust_head(ctx, 0 - (int)sizeof(*vlh)))
9          return -1;
10
11     data_end = (void *)(long)ctx->data_end;
12     eth = (void *)(long)ctx->data;
13
14     if (eth + 1 > data_end)
15         return -1;
16
17     __builtin_memcpy(eth, &eth_cpy, sizeof(*eth));
18     vlh = (void *)(eth + 1);
19
20     if (vlh + 1 > data_end)
21         return -1;
22
23     vlh->h_vlan_TCI = bpf_htons(vlid);
24     vlh->h_vlan_encapsulated_proto = eth->h_proto;
25     eth->h_proto = bpf_htons(ETH_P_8021Q);
26     return 0;
27 }
```

Figure 52: Function that adds a VLAN header. *Source: xdp-tutorial [38]*

Appendix B. Device support in Vivado in different licenses

Table 20 shows the device support that Vivado offers in the Standard and in the Enterprise license. Note that the FPGA in NetFPGA-SUME belongs to the group "Virtex-7 FPGA", which is only supported in the Enterprise version.

Table 20: Device support in Vivado ML Standard and in Vivado ML Enterprise. *Source:* *https://www.xilinx.com/products/design-tools/vivado/vivado-ml.html#architecture*

Device	Vivado ML Standard Edition	Vivado ML Enterprise Edition
Zynq®	Zynq-7000 SoC Device: -XC7Z010, XC7Z015, XC7Z020, XC7Z030, XC7Z007S, XC7Z012S, XC7Z014S	Zynq-7000 SoC Device: -All
Zynq® UltraScale+™ MPSoC	UltraScale+ MPSoC: -XCZU2EG, XCZU2CG, XCZU3EG, XCZU3CG, XCZU4EG, XCZU4CG, XCZU4EV, XCZU5EG, XCZU5CG, XCZU5EV, XCZU7EV, XCZU7EG, XCZU7CG	UltraScale+ MPSoC: -All
Zynq UltraScale+ RFSoC	UltraScale+ RFSoC: -None	UltraScale+ RFSoC: -All
Alveo	Alveo: -All	Alveo: -All
Kria	Kria: -All	Kria: -All
Versal	N/A	AI Core Series: -VC1902, VC1802 Prime Series, VM1802

Continued on the next page

Table 20 – continued from the previous page

Device	Vivado ML Standard Edition	Vivado ML Enterprise Edition
Virtex FPGA	**Virtex-7 FPGA:** **-None** Virtex UltraScale FPGA: -None	**Virtex-7 FPGA:** **-All** Virtex UltraScale FPGA: -All Virtex UltraScale+ FPGA: -All Virtex UltraScale+ HBM: -All Virtex UltraScale+ 58G: -All
Kintex FPGA	Kintex®-7 FPGA: -XC7K70T, XC7K160T Kintex UltraScale FPGA: -XCKU025, XCKU035 Kintex UltraScale+ FPGA: XCKU3P, XCKU5P	Kintex®-7 FPGA: -All Kintex UltraScale FPGA: -All Kintex UltraScale+: -All
Artix FPGA	Artix-7 FPGA: -XC7A12T, XC7A15T, XC7A25T, XC7A35T, XC7A50T, XC7A75T, XC7A100T, XC7A200T	Artix-7 FPGA: -All
Atrix UltraScale	Atrix UltraScale+: -XCAU25P, XCU20P	Artix UltraScale+ -All
Spartan-7	Spartan-7: -XC7S6, XC7S15, XC7S25, XC7S50, XC7S75, XC7S100	Spartan-7: -All

Appendix C. Creation of a Bitfile from a Vitis project

This appendix explains the process of creating a bitfile given a Vitis project. It has been documented as the developer of this project did not find much information related to this, and has considered that documenting it is worth it.

The first step is to write the program in Vitis, simulate it and debug it (either with C simulation or C/RTL Cosimulation) to make sure that it behaves as desired. After, the program has to be synthesized. Then, it can be exported as an RTL. However, the developer of this project always uses the option Run Implementation, as it exports the RTL and it also produces a report with the final timing and the resource usage.

The second step uses Vivado. In order to import an IP block, one has to go to Settings, IP, Repository. There is a search menu to add new sources. Selecting the folder of the Vitis project is enough, as it automatically detects any IP in subdirectories.

The next step is developing the design. In the flow navigator, in the IP Integrator section, selecting Create/Open Block Design opens a window to manage the design. Clicking Add IP or just Ctrl+I opens a search menu and writing `hls` there will automatically show all the IP imported from Vitis. Once the design is finished, the next action should be going to Sources, selecting the design, and first Generate Output Product and then Generate HDL Wrapper.

The last steps consist of Run Synthesis, Run Implementation and finally, Generate Bitstream. All of these actions are in the flow navigator.

Even though it has not been mentioned, the machine that is running Vivado has to always have a proper license.

Figure 53 summarises all the steps that have to be taken to obtain a bitfile given an eBPF program.

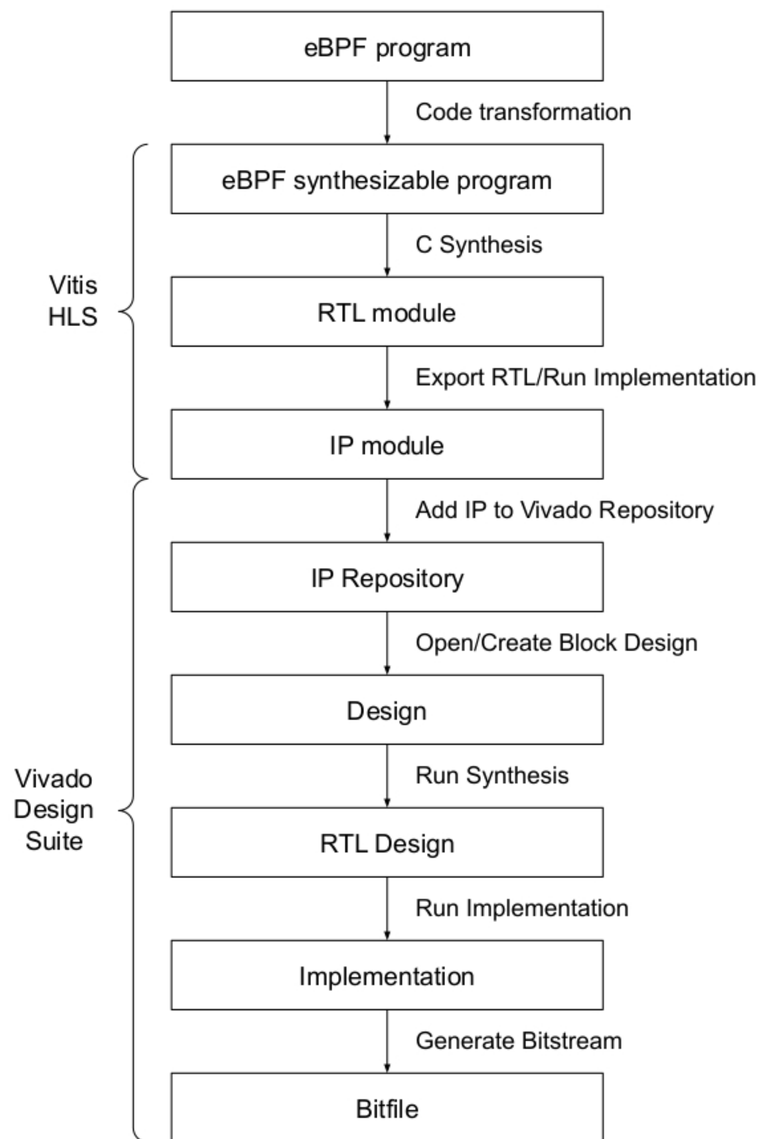

Figure 53: Diagram of the steps to "convert" an eBPF program to a bitfile. *Source: own elaboration*

Appendix D. Loading and testing a Bitfile in the NetFPGA-SUME

This appendix explains the process of loading a bitfile to the NetFPGA and how to test it. It has been added to the report to make the process of replicating the project easier.

Before starting, the NetFPGA-SUME project should be cloned and its locations should be saved in `$NetFPGA-SUME-live`.

The first step consists of writing the bitfile to the FPGA. Executing `xsct` in terminal invokes the Xilinx Software Command-Line Tool. After, `connect` establishes the connection with the board and `fpga -f bitfile.bit` loads the bitfile (named bitfile.bit). Finally, the machine has to be rebooted.

The second step is building the drivers for the NetFPGA-SUME. Figure 54 contains the commands that initialize the drivers. They can be saved to a script to speed up the process. The variable `$DRIVER_FOLDER` is `$NetFPGA-SUME-live/lib/sw/std/driver/sume_riffa_v1_0_0`.

```
cd $DRIVER_FOLDER
make all
make install
modprobe sume_riffa
```

Figure 54: Initialization of the drivers. *Source: [36]*

The third step is to create two namespaces and modify the routing table to force them to communicate through the board. Figure 55 contains the code to do it.

```
ip netns add ns1 > /dev/null 2>&1
ip link set enp3s0f0 netns ns1
ip netns exec ns1 ip addr add 192.168.0.40/24 dev enp3s0f0
ip netns exec ns1 ip link set dev enp3s0f0 up
ip netns exec ns1 ip route add 192.168.1.0/24 via 192.168.0.27
ip netns exec ns1 ip n add 192.168.0.27 dev enp3s0f0 lladdr
    02:53:55:4d:45:00

ip netns add ns2 > /dev/null 2>&1
ip link set enp3s0f1 netns ns2
ip netns exec ns2 ip addr add 192.168.1.40/24 dev enp3s0f1
ip netns exec ns2 ip link set dev enp3s0f1 up
sudo ip netns exec ns2 ip route add 192.168.0.0/24 via 192.168.1.27
sudo ip netns exec ns2 ip n add 192.168.1.27 dev enp3s0f1 lladdr
    02:53:55:4d:45:01

ip netns exec ns1 ip link set dev lo up
ip netns exec ns2 ip link set dev lo up
ip netns exec ns1 ip link set enp3s0f0 up
ip netns exec ns2 ip link set enp3s0f1 up
ip addr add 192.168.0.27/24 dev nf0
ip addr add 192.168.1.27/24 dev nf1
ip n add 192.168.0.40 dev nf0 lladdr 90:e2:ba:84:46:20
ip n add 192.168.1.40 dev nf1 lladdr 90:e2:ba:84:46:21
ip route add 192.168.0.0/24 via 192.168.0.27
ip route add 192.168.1.0/24 via 192.168.1.27
```

Figure 55: Initialization of the testing environment. *Author: Marcelo De Abranches*

Finally, `ping` can be used to test the design. Adding some extra options - `-c` to control the number of packets sent, and `-i` to decide the interval between sending the packets - make the tool behave faster. Figure 56 contains the full command.

```
ip netns exec ns1 ping 192.168.1.27 -c 10 -i 0
```

Figure 56: Ping command to test the design of the board. *Source: own elaboration*