UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PABLO FELIPE ROSA RODRIGUES

Enabling Programmable Data Planes with C++ and High-Level Synthesis for Custom Packet Forwarding

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. José Rodrigo Furlanetto Azambuja
Coadvisor: Prof. Dr. Weverton Luis da Costa Cordeiro

Porto Alegre
February 2024

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGEMENTS

The journey until here was not easy. Besides all the challenges faced, I am completely grateful for the lessons learned, and for all the amazing people I met and have the opportunity to exchange ideas and lessons. It seems more than fair to me the first words of this work be dedicated to thanks to those who in some way have to do with the completion of it. I highlight that these few words are just a portion of my immense gratitude to you.

I owe more to my mother Rose and my sister Ianca than anyone else. Thank you for all support in life and for believing me even when I doubt of myself. You are my basis.

I would like to give a special thank to Alessandra, my partner in life and who all these years together have always supported and encouraged me to achieve my goals. You are the person I can rely in good times and bad times. Every single day you inspire me and give me strength to be better. You are the loviest one. I love you!

To two big friends the life brings to me during my graduation, Mateus Saquetti and Leonardo Gobatto, a special thanks for all the time we spent discussing about this work and for helping me with that. You are references to me. Definitely I am in debt with both of you. I think that barbecue we have talking about can solve that, agreed?

Thanks to all my friends, whose near or far always encouraged me.

Finally, a special thank to my advisor José Azambuja and my co-advisor Weverton Cordeiro for all the guidance and dedication during my academic life and for the collaboration in this work. Thank you for open the laboratory doors years ago for that guy whose just was curious about doing science and wanted learn something new.

ABSTRACT

New networking technologies for accelerating packet processing with programmable hardware have enabled great innovation in the networking field, from domain-specific languages to programmable forwarding planes. Despite the research efforts, current approaches to describe forwarding devices behavior provide limited expressiveness and in most cases a lower supportive, mature, and transparent workflow to translate these descriptions to a suitable format for running in a hardware platform. In this work, we present a proof-of-concept for enabling high flexibility and programmability at the network forwarding plane. The programming methodology and the workflow proposed in this work allow describing the behavior of a network forwarding device using the C++ language and submitting the code to a high-level synthesis process in order to obtain a synthetizable RTL description suitable for running in a FPGA board. We measure performance, resource utilization, and perform a behavioral analysis to validate the proposal. The results show that it is feasible to describe forwarding devices using the proposed workflow with no impact on resource utilization at the same time it can achieve around 14% more throughput than the P4-NetFPGA canonical reference design.

Keywords: Field-Programmable Gate Array. High-Level Synthesis. C++. NetFPGA Architecture. P4. Programmable Data Planes.

RESUMO

Novas tecnologias para acelerar o processamento de pacotes utilizando hardware programáveis tem possibilitado grande inovação na área de redes de computadores, desde linguagens de domínio específico até plano de dados programáveis. Apesar dos esforços, as soluções atuais para descrever o comportamento de um dispositivo de encaminhamento fornecem limitada expressividade e, na maioria dos casos, um baixo suporte, uma baixa maturidade, e um processo pouco transaparente para transformar essas descrições em um formato adequado para ser executado em uma plataforma de hardware. Neste trabalho, nós apresentamos uma prova de conceito para fornecer alta flexibilidade e programabilidade no plano de encaminhamento de redes. A metodologia de programação e o processo proposto neste trabalho permitem descrever o comportamento de um dispositivo de encaminhamento usando a linguagem de programação C++ e submeter o código escrito a um processo de síntese de alto nível para gerar uma descrição em hardware adequada para executar em uma placa FPGA. Nós medimos desempenho, utilização de recursos e realizamos uma análise comportamental para validar a proposta. Os resultados mostram que é possível realizar a descrição de dispositivos de encaminhamento utilizando o processo proposto sem causar impactos na utilização de recursos, ao mesmo tempo que a solução pode fornecer uma largura de banda de até 14% a mais do que a arquitetura canônica de referência P4-NetFPGA.

Palavras-chave: Vetor de Porta Programável em Campo. Síntese de Alto Nível. C++. Arquitetura NetFPGA. P4. Planos de Dados Programáveis.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CLB	Configurable Logic Blocks
CPU	Central Processing Unit
DSL	Domain-Specific Language
DMA	Direct Memory Access
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLS	High-Level Synthesis
IO	Input/Output
IP	Internet Protocol
LAN	Local Area Network
L2	Layer 2
MAC	Media Access Control
OPL	Output Port Lookup
PDP	Programmable Data Plane
POF	Protocol Oblivious Forwarding
P4	Programming Protocol-Independent Packet Processors
RX	Receive
SDN	Software-Defined Networking
SDNet	Software Defined Specification Environment for Networking

TX	Transmit
UDP	User Datagram Protocol
WAN	Wide Area Network

LIST OF FIGURES

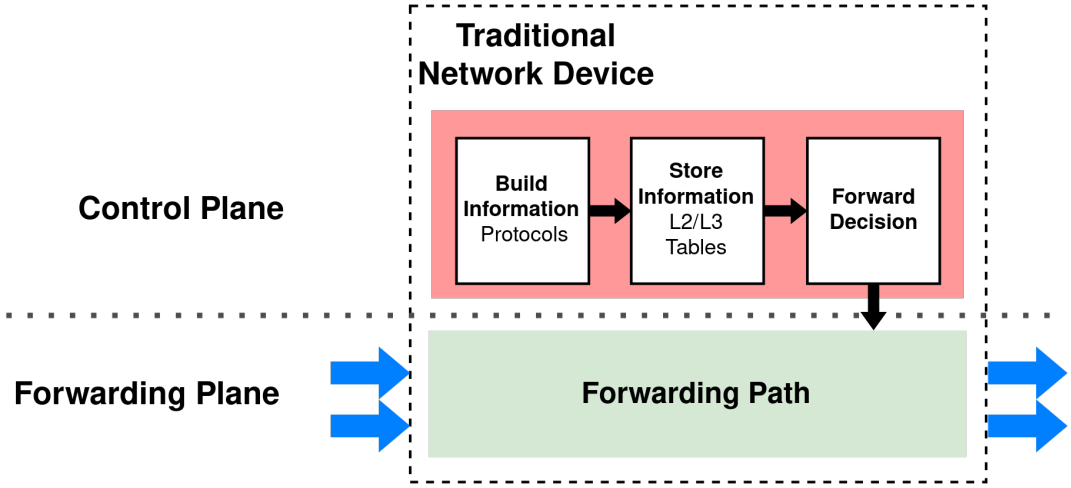
LIST OF TABLES

CONTENTS

1 INTRODUCTION

Traditionally, network devices are designed mainly following a fixed and frozen model based on Application Specific Integrated Circuits (ASICs). It means that the behavior of the main functions of a device, functionally divided into the control plane and forwarding plane (also commonly referred to as the data plane), and the engines running on that are defined at the moment the devices come out of the vendor's facility to the shelves. These engines are the core of a network device. It is in the control plane that the control engine lies on, being responsible for the device management tasks and for keeping the forwarding policies up-to-date, whereas the forwarding engine in the forwarding plane is responsible for processing the incoming packet traffic and forwarding it to the right destination with good performance. Fig. 1.1 shows a generic view of a traditional network device architecture.

Figure 1.1 – Traditional network device architecture.



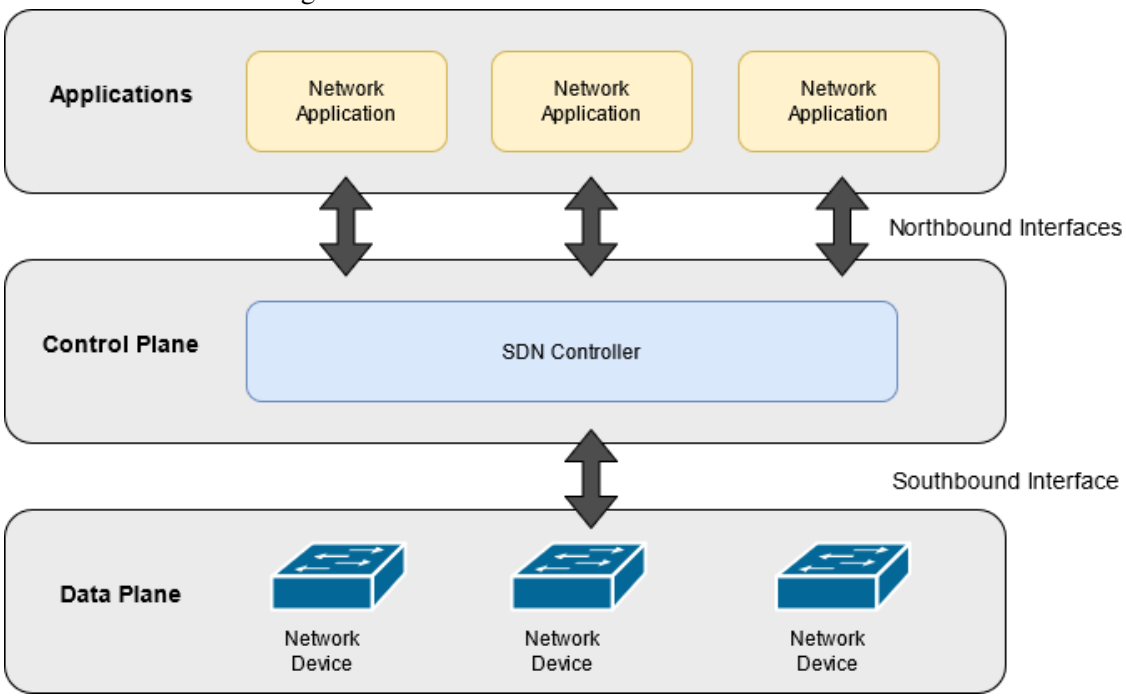
Source: The Author

The problem with that vision is that since the device continues with a fixed-function set and its management is hanged into it, the devices are managed in a decentralized model, becoming the management a very plastered process, and every novelty imposes the industry and big players to redesign a device in order to support the new functionalities.

The concept of Software-Defined Networking (SDN) comes as an attempt to solve those issues. It has reshaped the way forwarding devices are designed (KREUTZ et al., 2015). SDN instigated researchers and network operators to think about a more programmable network. With SDN, the network intelligence is more flexible, taking the

control plane logic out of the network devices and letting the devices only responsible for the forwarding roles. Fig. 1.2 shows an overview of an SDN architecture. Network applications are at the higher level, communicating with the control plane to set the device management policies and forward rules through northbound interfaces. A centralized SDN controller receives and propagates these changes through a southbound interface to the devices operating on the data plane. This added flexibility and brings the possibility to manage the network in a centralized way. In turn, this allows network operators to control and improve their network in software.

Figure 1.2 – An overview of the SDN architecture.



Source: The Author

Despite SDN's efforts to solve some networking problems, the data plane remains fixed in hardware. It is justified in part since to process intensive packet traffic promptly, the forwarding tasks must be made on silicon. However, in that way, networking still faces limited devices, disallowing the feasibility of programming the device when a newer protocol initially not supported by the network device is enabled to be used in networks and modify its forwarding algorithms, still forcing operators to operate network devices based on the functionalities predefined by manufacturers.

The advent of Programmable Data Planes (PDPs) (MICHEL et al., 2021) provided a new approach for flexible programming of network devices, enabling the provisioning of novel networking protocols and services through the use of Domain-Specific Languages (DSLs) like POF (SONG, 2013), P4 (BOSSHART et al., 2014), and Lyra (GAO et al.,

2020). These DSLs helped deliver the feasibility of a programmable forwarding plane by providing a higher-level and vendor-agnostic API. This API contrasts with the ones generally specified on fixed-function chips of traditional devices by manufacturers, making the PDP more flexible and breaking the "network ossification."

Since reconfigurability is the main key for supporting the new concept of programmable devices built using DSLs, Field-Programmable Gate Arrays (FPGAs) turn out to be one of the main targets for building the devices, offering flexibility and high performance, a middle ground between generic Central Processing Units (CPUs) and ASICs. Particularly, P4 was the DSL that has drawn the attention of the networking community since its first release, and efforts were made to support the language and spread its use. Xilinx Software Defined Specification Environment for Networking (SDNet) (MAXFIELD, 2014) (Xilinx Inc., 2018) is the main and more mature tool used for translating a P4 program into a Hardware Description Language (HDL) module through a High-Level Synthesis (HLS) workflow in order to simulate in an FPGA board the behavior described by a designer. Approaches using P4 and SDNet to provide a programmable forwarding plane were proposed in the literature like the P4-NetFPGA (IBANEZ et al., 2019) project, which allows users to program a P4 code, transforms it in HDL module using the SDNet tool, then running it in the NetFPGA SUME board.

Despite the advantages shown by the use of P4 and SDNet, the workflow using these technologies for generating a HDL description lacks important aspects that must be considered when developing a network device. P4 is a DSL, meaning its power of expressiveness is limited to its general scope of use. Optimizations related to the packet processing flow for developing more complex devices must be made using externs, functions written using HDLs, that will be called by the P4 program, therefore imposing hardware description knowledge in order to develop more optimized devices. Similarly, SDNet imposes license requirements for its use, provides limited support, and does not provide an accessible documentation.

As a first step to provide a transparent workflow and a manner to designers describe network devices using a sufficiently flexible high-level language for enabling programmable forwarding planes, we propose a proof of concept and a methodology for programming the devices. In our proposal, we adapt the P4-NetFPGA workflow to allow designers to describe the device behavior at a high level using the C++ language, then generate a hardware description through an HLS workflow in order to simulate the desired behavior on the NetFPGA SUME board. We use our testbed to verify the feasibil-

ity of our proposal for building an L2-switch networking device using the C++ language and integrate the generated HDL module into the NetFPGA canonical architecture. We measure resource utilization and performance to observe the occupation of the device and compare the performance results with an L2-switch device described by a P4 program targeting the same FPGA platform. Our evaluation provides results showing our approach's feasibility without incurring any resource utilization overhead. Yet, it presents an increase in performance compared to the NetFPGA design.

The remainder of this work is organized as follows. We present the theoretical foundations and background that support this work in Chapter 2. We discuss our proposal, the P4-NetFPGA project modification, and our methodology to build devices in Chapter 3. In Chapter 4, we present the results of our evaluation. We close this work in Chapter 5 and make some remarks about the future work in Chapter 6.

2 BACKGROUND

This chapter discusses the background topics that underlie this work, including FPGA architectures, High-Level Synthesis, the P4 language, the SDNet tool, the P4-NetFPGA workflow, and the NetFPGA reference design.

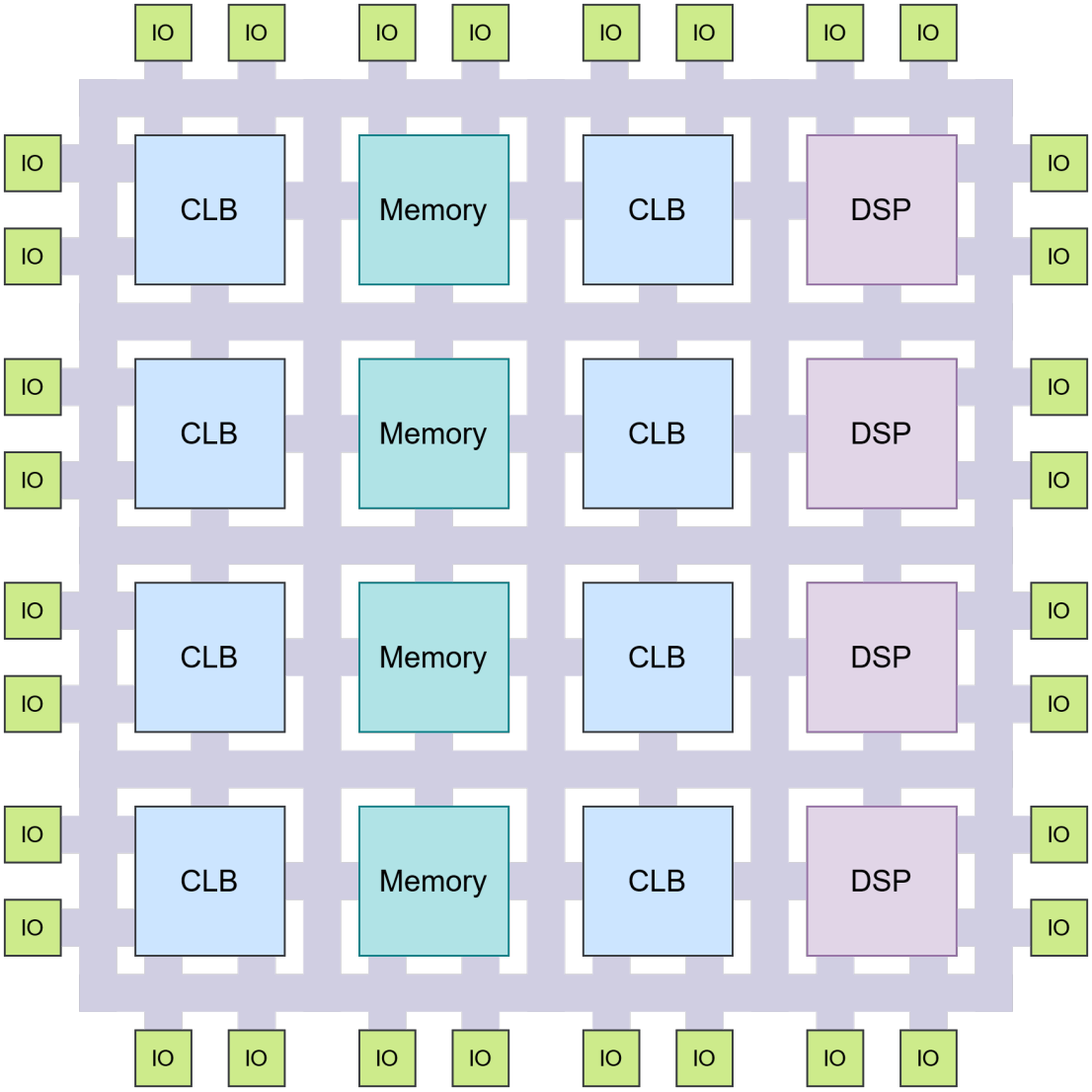
2.1 Field-Programmable Gate Arrays

The use of hardware for accelerating applications and systems is a recent approach. Network applications would be the same. Forwarding devices are the main core in networks that enable network traffic forwarding in Local Area Networks (LANs) and routing packets through Wide Area Networks (WANs). Typically, vendors use ASICs and FPGAs to improve the data processing in line-rate (LOCKWOOD et al., 2007). Although high performance is what distinguishes ASIC-based systems, FPGAs encompass and provide more flexibility in design and cost, and at the same time, they show an acceptable performance improvement when compared to systems built using general-purpose CPUs.

Fundamentally, an FPGA is a silicon chip that can be modified to virtually behave as a digital circuit wielding some specific functionality (KUON et al., 2008). The Fig. 2.1 shows the basic internal organization of FPGAs. FPGAs are composed of Configurable Logic Blocks (CLBs) that contain logic elements that can be configured to wield several different functions in a combinational or sequential manner using logic gates. Also, as part of FPGA organization, there are configurable memory blocks used for storing data, Digital Signal Processing(DSP) components used to accelerate and optimize typical digital signal processing tasks like multipliers and fast Fourier transforms (FFTs), and Input/Output (IO) blocks, that allows the FPGA to communicate with external devices. FPGAs also possess programmable interconnect matrixes that provide routing paths between the organization entities to the FPGA, which plays the expected behavior. The FPGA boards are programmed using HDLs like VHDL and Verilog. This allows a hardware designer to understand a given circuit's high-level description.

As already mentioned, FPGAs show several advantages when compared to the ASICs. The time-to-market of a system designed in FPGA is usually faster and simpler than the ones designed on ASICs. FPGAs can be programmed and tested in a few steps, unlike ASICs, which can take a long time and cost millions of dollars for a prototype. FPGAs are a clear choice for building new high-performance hardware products in a

Figure 2.1 – Basic internal organization of FPGAs.



Source: The Author

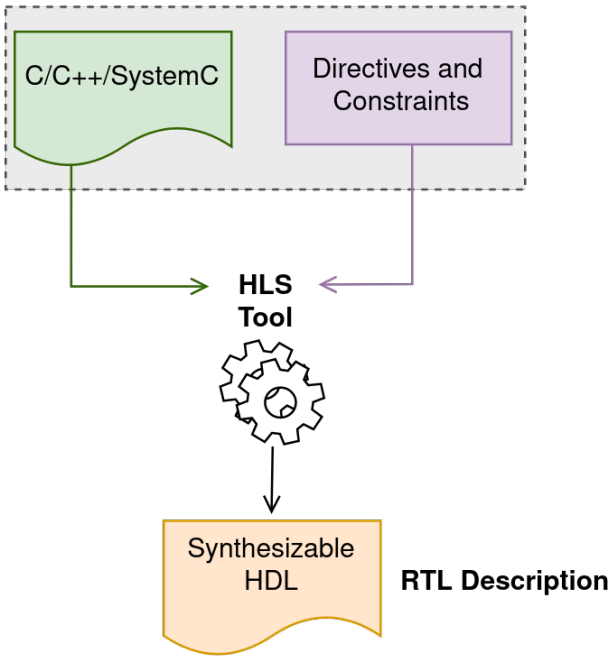
fast-growing environment. Regarding innovative designs on the market, FPGAs allow designers to build a system according to the customer's needs. With the advent of the notion of data plane programmability and DSLs, using an FPGA, the development of new networking products, protocols, and standards can be done more quickly than in the traditional costly and time-consuming way of waiting for ASICs that enable a new stack of fixed protocols (SIVARAMAN et al., 2015). However, in contrast to the flexible capacity of FPGAs, their use presents a significant cost in area, delay, and energy consumption (KUON; ROSE, 2007). Therefore, its use must always consider all these characteristics to choose the best target that suits the purpose planned by the customer's needs.

2.2 High-Level Synthesis

As discussed in the prior section, FPGAs are an attractive choice for programmable data plane applications due to their reconfiguration capability and the faster drafting for building new solutions. However, the design effort and the learning curve needed for building FPGA-based solutions using HDL remains more complex than the effort demanded in the use of high-level languages. One of the technologies that aims to address and solve this gap between software and hardware systems descriptions is High-Level Synthesis (HLS). Using HLS, designers are allowed to work at higher abstraction levels while still benefiting their systems with the hardware acceleration advantages (COUSSY et al., 2009).

HLS is a powerful process that aims to simplify the conception of digital hardware development. Initially, it transforms a program written in a programming language like C, C++, or SystemC into a hardware description language like Verilog or VHDL (CONG et al., 2011). Since HLS abstracts many of the hardware design concerns, designers can focus on expressing their ideas using a higher-level language. Commonly, to concept a hardware system, a team of hardware designers with an in-depth understanding of hardware architecture, HDLs, and its aspects is needed, showing a large entry barrier to those who want to build a system with the performance a hardware acceleration provides. HLS provides this opportunity. HLS offers a productivity boost by allowing designers to work at a higher level of abstraction and empowers developers to contribute to hardware systems design using their existing programming skills. Also, it enables fast prototyping, so designers evaluate and test their designs more efficiently, reduces the verification time of the RTL description since it is generated from a fully verified high-level source, and can apply a set of optimizations for the generated code. Therefore, users can focus on the system functionality instead of the architecture-specific optimizations. And last, but equally important, the learning curve is reduced. Fig. 2.2 shows an overview of the HLS process. Typically, it involves specifying a circuit behavior in a higher-level programming language, submitting the description to the HLS tool, which inspects the source code and applies optimizations based on the HLS directives and the constraints imposed by the specified target, then transforms the high-level description in a synthesizable HDL representing the system behavior to be executed in the FPGA.

Figure 2.2 – High-level synthesis workflow.



Source: The Author

2.3 Programming Protocol-Independent Packet Processors

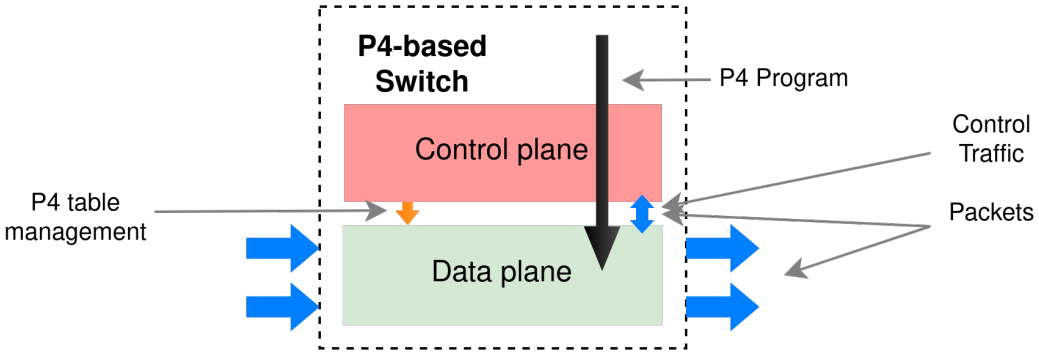
Programming Protocol-independent Packet Processors (P4) (BOSSHART et al., 2014) is a high-level declarative language used to configure programmable data planes and specify how switches must process and forward packets.

The main goals of the P4 language consist of reconfigurability, protocol independence, and target independence. Reconfigurability defines a network operator's ability to redefine the parsing and processing of packets. P4 language can offer protocol independence by extracting non-standard header fields of packets specifying the packet parser and a set of tables and actions responsible for handling incoming packets of the network device. The target independence focuses on the network operators' ability to describe a device's functionality without requiring extra information about the physical substrate.

Generally, a traditional switch is built on top of a "bottom-up" approach, where the switch data plane functionality is predefined by manufacturers. In contrast, a P4-based switch does not have a fixed data plane, thus offering a "top-down" approach and providing a network operator with the possibility to program the data plane writing a P4 program with a set of functionalities and then load it in a programmable switch. Fig. 2.3 shows the new approach.

Essentially, a vendor must provide an architecture model specifying the compo-

Figure 2.3 – P4-based switch.



Source: The Author

nents a network operator can define and program and a P4 compiler for the packet processing target device.

A P4 program is composed of five basic elements: (i) headers, formatted data that specifies network protocols; (ii) parser, implemented as a finite state machine to extract and analyze headers and fields of an incoming packet; (iii) match-action processing block, which uses tables and actions to analyze fields within a packet to try to perform a match with the key metadata, resulting in a certain action; (iv) deparser, a control block that reassembles headers of an outgoing packet, and (v) metadata, structures containing packet information. In Figs. 2.4, 2.5, and 2.6, an example of an L2-switch device (forwards Ethernet packets) is shown.

In Fig. 2.4, the `start` state submits every incoming packet to the `parse_ethernet` state to extract the Ethernet header and then accepts the packet for processing it in the switch processing block.

```
parser TopParser(packet_in pkt_in,
                 out Parsed_packet hdr) {
    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        pkt_in.extract(hdr.ethernet);
        transition accept;
    }
}
```

Figure 2.4 – Parser of an L2-switch described in the P4 language.

Table II shows a MAC address table with populated entries. In Fig. 2.5, the L2 switch match- action processing block receives a parsed packet (`hdr`) and a metadata

```
control TopPipe(inout Parsed_packet hdr,
                inout sume_metadata_t sume_metadata) {

    action forward(bit<8> port) {
        sume_metadata.dst_port = port;
    }
    table mac {
        actions = {
            forward;
        }
        key = {
            hdr.ethernet.dst_addr: exact;
        }
        size = 64;
    }
    apply {
        mac.apply();
    }
}
```

Figure 2.5 – Processing block of an L2-switch described in the P4 language.

Table 2.1 – Example of a MAC address table.

Key	Action	Port
08:11:11:11:11:08	forward	0
08:22:22:22:22:08	forward	1

struct (`sume_metadata`). In the case of the destination MAC address field in the ethernet header (`hdr.ethernet.dst_addr`) of the parsed packet is matched with an entry on the MAC address table, the `forward` action is activated, resulting in the packet being forwarded to the destination MAC address port specified in `mac` table.

As shown in Fig. 2.6, after processing the packet, the parsed data are then reassembled and emitted to the output packet in order to forward it to the switch output.

```
control TopDeparser(packet_out pkt_out,
                    in Parsed_packet hdr) {
    apply {
        pkt_out.emit(hdr.ethernet);
    }
}
```

Figure 2.6 – Deparser of an L2-switch described in the P4 language.

2.4 Software Defined Specification Environment for Networking

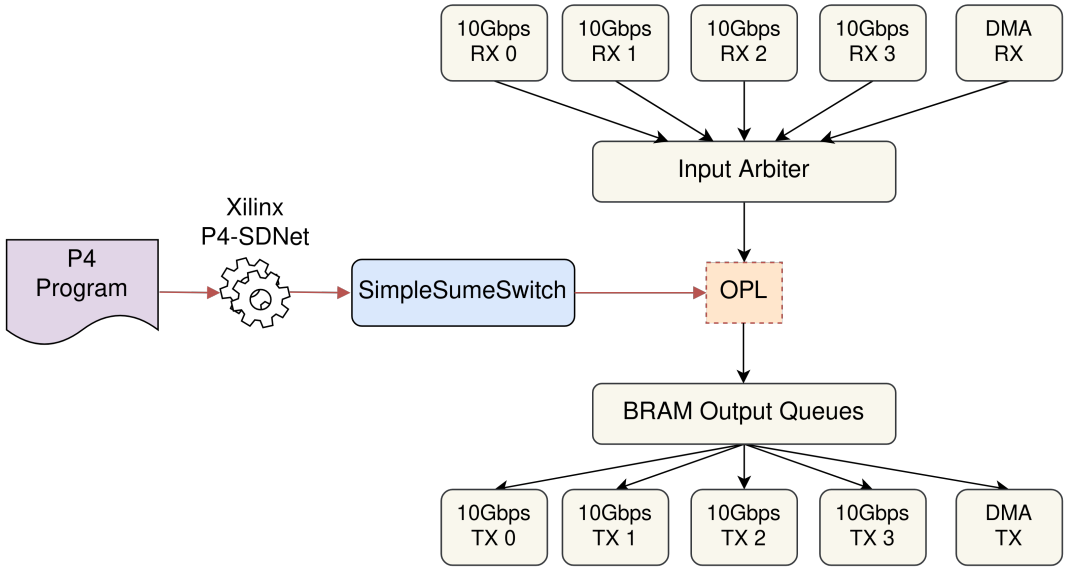
As briefly presented in Chapter 1, the Xilinx Software Defined Specification Environment for Networking (SDNet) is the more mature environment responsible for transforming P4 code to HDL and mapping the P4 logic elements to the SDNet engines. Moreover, SDNet is a powerful tool that enables prototyping, development, and deployment of packet processing functions in Xilinx FPGA boards. As a result of the process, SDNet delivers a programmed data plane with concise communication with the device control plane. In this manner, service providers are able to dynamically reconfigure the switch forwarding properties like tables without interrupting the service running on the board. P4 is essentially a target-independent language, so SDNet becomes a main provider in terms of basis to build network devices in FPGAs targets.

SDNet owns a set of engines that allows network operators and designers to specify a packet processing flow. Some are very similar to the P4 entities, like parser and processing blocks. In order to implement a P4 specification, the SDNet compiler for P4 (Xilinx Inc., 2018) maps the control onto a custom data plane architecture built upon the SDNet engines. As mentioned above, P4 is a target-independent language, meaning that the programming language must generically accommodate different target platforms. When considering architecture-specific details (e.g., lookup tables implementation and external functions), certain design concerns are solved. Each target platform must have a compiler back-end that can define its level of support in these specific cases. p4c-sdnet (Xilinx Inc., 2018) is the compiler which translates P4 language to an intermediate language called PX (Xilinx Inc., 2017). Files containing information about P4 to SDNet objects nomenclature and other mapping information are generated in this process. The designers do not have to understand anything about the underlying FPGA devices and their internal architectures. They have only to define what they want the device to do. The SDNet specification is then passed to the SDNet compiler. The output from the SDNet compiler is then placed into the Xilinx Vivado tool (Xilinx Inc., 2023), which handles the implementation details. Despite the advantages showed SDNet, Xilinx requires a license requirement to obtain access to its use, and the documentation access and the support to it is limited. That is an indication that, despite the emergence of DSLs, the aim to engage the software community to collaborate in hardware designs is limited by engines that provide the bridge between the high-level description program and the generated description to program an FPGA board.

2.5 P4-NetFPGA architecture

The P4-NetFPGA architecture is based on the canonical NetFPGA reference design and in the Simple Sume Switch (SSS) architecture model. The device logic is composed of a parser, a match-action pipeline, and a deparser (IBANEZ et al., 2019). Fig. 2.7 shows the canonical NetFPGA reference design, which consists of four TX/RX and DMA TX/RX ports, Input Arbiter (IA), Output Port Lookup (OPL), and BRAM Output Queues (BOQ). The Input Arbiter admits packets arriving from input ports and forwards them towards the Output Port Lookup, where the main packet processing occurs and the output port is selected. The OPL is a placeholder responsible for hosting a device instance. In the P4-NetFPGA, a P4 program describing a networking device is compiled using the SDNet tool and transformed into an HDL instance in order to replace the OPL module. The BRAM Output Queues buffers packets while they wait to be sent to output ports. At last, the Management Interface allows the network operator to communicate with the switch.

Figure 2.7 – P4-NetFPGA workflow built upon the canonical NetFPGA design.

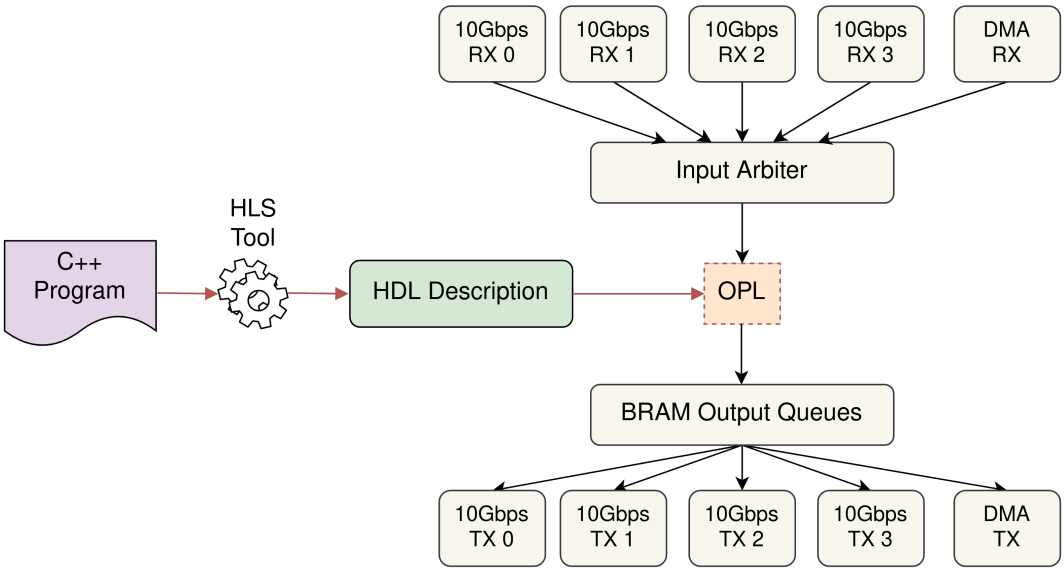


Source: The Author

3 PROPOSED ARCHITECTURE

The proposed workflow and programming methodology discussed in this work is a proof of concept that seeks to enable the programmability of network devices with more flexibility and transparency using the widely known C++ programming language to describe network devices' behavior and submit the written code to an HLS workflow to generate a HDL module to run in a FPGA board. Unlike the restricted use and support of SDNet, there are already commonly used tools like HLS (AMD Xilinx Inc., 2018) and Vitis (AMD Xilinx Inc., 2020) to generate an HDL module from a C++ source code, and a large community around the language, in addition to its extensive support. Since C++ is a general-purpose language, there are no restrictions about the domain where the language is used, resulting in a code that is more adaptive to the needs of a network operator or device designer. The proposal is built upon the P4-NetFPGA workflow, replacing a P4 instance with one described in the C++ language. Fig. 3.1 shows an overview of the newly adapted architecture, which now allows the OPL module to receive an HDL instance generated from a C++ code that will be generated using HLS. Our adapted workflow consists of seven steps: (i) write a C++ program; (ii) submit it to the HLS process to generate the HDL instance; (iii) run C++ simulation (optional); (iv) integrate the HDL instance into the NetFPGA design architecture; (v) write a python script to generate test data for simulations; (vi) run HDL simulations; and (vii), build bitstream for FPGA.

Figure 3.1 – Proposed workflow, generating an RTL description from a C++ program.

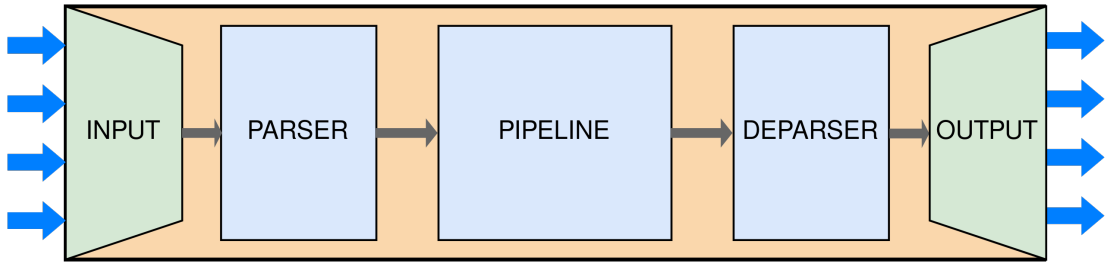


Source: The Author

3.1 C++ Programming Methodology

The proposal provides a simple but concise distribution of an architectural model into different files, where each one plays a role within the system to perform different tasks. The architectural model of the work is similar to the SimpleSumeSwitch architecture originally used for the development of P4 instances in the P4-NetFPGA project. In this work, the developed model has three main suggestive entities that the designer will program to describe the desired behavior of the device: Parser, Pipeline, and Deparser. The designer must use the basic templates provided by the workflow and create a new file in C++ defining the behavior of the top entities. In short, each entity is mapped into a C++ function. Fig. 3.2 shows the proposed architectural model.

Figure 3.2 – Proposed architectural model design.



Source: The Author

3.1.1 Parser

The parser is the entity responsible for extracting the package headers. Fig. 3.3 shows the entity template that the designer must program. The entity receives a state (`state`) that is carried transversally between the proposed entities model so that its value informs whether any treatment should be made on the input packet (`packet_in`), which is an array of bits with size equal to the NetFPGA architecture bus size (256). When identifying the receipt of a new packet, the parser will receive the status equal to `START` and must check it to begin its operation. It is up to the designer to write the rest of the Parser logic. The benefit of it is that the logic to parsing headers in that case will be must in part defined by the designer criteria, according to its preferences. A point to observe is that, unlike the high learning curve to write a hardware description for a device using the proposed templates, a designer must only learn a few details about C++-defined types specific to HLS. Particularly, the proposed model only exposes a unique defined type that

uses a template parameter and is specific to the HLS context: `ap_uint`. As mentioned above, it is an array of bits with the size of the array defined by the parameter passed in the variable declaration. The learning is just about how to extract a set of bits and how to write in a set of bits.

If starting processing a packet is mandatory, the designer must describe a behavior that leads to a state of `ADMIT` or `REJECT`. The subsequent entity will check for one of these states to process or discard a packet. `parsed_packet` is an object that will carry information extracted from the input packet to the next entities. The `parsed_packet` is organized according to structures (e.g., headers) that logically divide a packet. Examples of structures include the Ethernet, IP header, and UDP header.

```
void Parser(State &state,
            ap_uint<DATA_BUS_SIZE> &packet_in,
            ParsedPacket_t &parsed_packet)
{
#pragma INLINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=packet_in
#pragma HLS INTERFACE ap_none port=parsed_packet

  switch (state) {
   case START:
      /*****************************
      * Parser behavior goes here
      *****************************/
      break;

   default:
      state = REJECT;
      break;
   }
}
```

Figure 3.3 – Definition of the Parser entity template.

3.1.2 Pipeline

The pipeline is the processing block of the device. In this entity, actions are taken according to the desired behavior of the device. In the pipeline stage, forwarding policies are applied through table queries and actions (functions) performed on the desired data. Each action is built from functions and types defined previously by the user in a separate file that must be included using the `#include` preprocessor directive in the file defining

the device behavior. Fig. 3.4 shows the template of the Pipeline entity. Just like the Parser entity, the pipeline stage uses the state query (`state`) to guarantee admission for data processing and decides whether the data should be rejected or passed through the stage. It is critical that prior entities are responsible for setting the state properly to be processed by the next stage right after it. Furthermore, data extracted during the Parser stage can be carried through `parsed_packet`. It is based on the data stored in the `parsed_packet` that match and actions will be made in this stage. The `sume_metadata` corresponds to the `tuser` bus in the SUME reference design and is from it that information such as packet size and source and destination ports are carried into the NetFPGA architecture and into the device stages, consequently (IBANEZ, 2019). Through this variable and the use of tables of actions, packet metadata can be changed to serve as information for the next modules of the architecture, such as output queues, which will be responsible for forwarding the packet to the specified output port.

```
void Pipeline(State &state,
              ParsedPacket_t &parsed_packet,
              ap_uint<METADATA_BUS_SIZE> &sume_metadata)
{
#pragma INLINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=state
#pragma HLS INTERFACE ap_none port=parsed_packet
#pragma HLS INTERFACE ap_none port=sume_metadata

   switch (state) {
   case ADMIT:
      /*****************************
      * Pipeline behavior goes here
      *****************************/
      break;

   default:
      state = REJECT;
      break;
   }
}
```

Figure 3.4 – Definition of the Pipeline entity template.

3.1.3 Deparser

Finally, the Deparser entity is responsible for emitting the output packet. Fig 3.5 shows the entity template. At this stage, the entity uses the content extracted from the input packet and stored in the `parsed_packet`. At this stage, the necessary information will be transferred to the output packet (`packet_out`).

```
void Deparser(State &state,
              ap_uint<DATA_BUS_SIZE> &packet_out,
              ap_uint<DATA_BUS_SIZE> &packet_in,
              ParsedPacket_t &parsed_packet)
{
#pragma INLINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=parsed_packet
#pragma HLS INTERFACE ap_none port=packet_out
#pragma HLS INTERFACE ap_none port=packet_in
   /*****************************
   * Deparser behavior goes here
   *****************************/
}
```

Figure 3.5 – Definition of the Deparser entity template.

Figs. 3.6, 3.7, 3.8, 3.9, and 3.10 show an example of a table and the entities description for an L2-switch using the proposed methodology.

Auxiliary.hpp

```
#include Core.hpp // pre-defined file: provides access to
   ↪ namespace 'port' to get the NetFPGA SUME port mapping and
   ↪  basic definitions like PORT_BITS_SIZE, DATA_BUS_SIZE,
   ↪ and METADATA_BUS_SIZE.
/**************
* Structures
**************/
typedef ap_uint<48> EthAddr_t;

struct Ethernet_h {
   EthAddr_t dstAddr;
   EthAddr_t srcAddr;
   ap_uint<16> etherType;
};

struct ParsedPacket_t {
   Ethernet_h ethernet;
};

/***********************
* Functions and Tables
***********************/
static const ap_uint<48> table_mac[] =
{ // MAC Address -> Port
   0x081111111108, // 08:11:11:11:11:08 -> nf0
   0x082222222208, // 08:22:22:22:22:08 -> nf1
   0x083333333308, // 08:33:33:33:33:08 -> nf2
   0x084444444408  // 08:44:44:44:44:08 -> nf3
};

static void getPort(ap_uint<48> &mac,
                    ap_uint<PORT_BITS_SIZE> &port)
{
   if (mac == table_mac[0])
      port = port::nf0;
   else if (mac == table_mac[1])
      port = port::nf1;
   else if (mac == table_mac[2])
      port = port::nf2;
   else if (mac == table_mac[3])
      port = port::nf3;
   else
      port = port::drop;
};
...
```

Figure 3.6 – Auxiliary file containing the L2-switch tables and actions pt. 1.

Auxiliary.hpp

```
...
static void forward(ap_uint<METADATA_BUS_SIZE> &metadata,
                    ap_uint<PORT_BITS_SIZE> &dst_port)
{
   metadata.range(31, 24) = dst_port;
}
```

Figure 3.7 – Auxiliary file containing the L2-switch tables and actions pt. 2.

L2Switch.cpp

```
void Parser(State &state,
            ap_uint<DATA_BUS_SIZE> &packet_in,
            ParsedPacket_t &parsed_packet)
{
#pragma INLINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=packet_in
#pragma HLS INTERFACE ap_none port=parsed_packet

  switch (state) {
   case START:
      parsed_packet.ethernet.dstAddr = packet_in.range(47,0);
      parsed_packet.ethernet.srcAddr = packet_in.range(95,48);
      parsed_packet.ethernet.etherType = packet_in.range(111,
         ↪ 96);

      switch (parsed_packet.ethernet.etherType){
      case IPV4_TYPE:
         state = State::ADMIT;
         break;
      default:
         state = State::REJECT;
         break;
      }
      break;

   default:
      state = REJECT;
      break;
   }
}
```

Figure 3.8 – Parser of a L2-switch described in the C++ language.

L2Switch.cpp

```
void Pipeline(State &state,
              ParsedPacket_t &parsed_packet,
              ap_uint<METADATA_BUS_SIZE> &sume_metadata)
{
#pragma INLINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=state
#pragma HLS INTERFACE ap_none port=parsed_packet
#pragma HLS INTERFACE ap_none port=sume_metadata

   switch (state) {
   case ADMIT:
      ap_uint<8> dst_port;

      // Lookup table
      getPort(parsed_packet.ethernet.dstAddr, dst_port);

      // Action
      forward(sume_metadata, dst_port);
      break;

   default:
      state = REJECT;
      break;
   }
}
```

Figure 3.9 – Processing block of a L2-switch described in the C++ language.

3.2 Integration into the NetFPGA design

In order to perform the integration into the NetFPGA architecture, the starting point was to question how to adapt the C++ code to be compatible with the NetFPGA wrapper (OPL) interfaces/signals to connect the generated HDL module with the rest of the architecture. The NetFPGA project was conceived using the Xilinx Vivado 2018.2 tool, so organically, the HLS tool chosen to generate HDL code from the C++ code described in this work was Xilinx HLS 2018.2. The first interesting aspect is that other tools can generate HLS in a way that is decoupled from the project flow.

Through HLS for C/C++/SystemC, the generic behavior to generate hardware description is that high-level functions are mapped to modules described in HDL, and their set of parameters become inputs and outputs of the generated HDL module. Moreover, as an HLS workflow requirement, a function defined in some of the files included in the

L2Switch.cpp

```
void Deparser(State &state,
              ap_uint<DATA_BUS_SIZE> &packet_out,
              ap_uint<DATA_BUS_SIZE> &packet_in,
              ParsedPacket_t &parsed_packet)
{
#pragma INLINE
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=parsed_packet
#pragma HLS INTERFACE ap_none port=packet_out
#pragma HLS INTERFACE ap_none port=packet_in
   // emit (copy the parsed data in the packet out)
   packet_out = packet_in;
   if (state != TRANSPASS) {
      packet_out.range(47,0) = parsed_packet.ethernet.dstAddr;
      packet_out.range(95,48) = parsed_packet.ethernet.srcAddr;
      packet_out.range(111,96) = parsed_packet.ethernet.
         ↪ etherType;
   }
}
```

Figure 3.10 – Deparser of a L2-switch described in the C++ language.

HLS project must be set as its top function. During synthesis, the HLS tool will generate an HDL module corresponding to the top function. A set of other modules will be generated for each function defined in the project. So, if one function is called by the top function, it will be translated to an internal instantiation in the HDL module generated from the top function. With that in mind, a main function was created with its parameters corresponding to the sizes and roles of the connectivity signals of the OPL wrapper.

Although HLS provides libraries for types like AXI4-Stream or pragmas like `pragma INTERFACE axis`, in order to indicate that a given C++ parameter will use the AXI4-Stream protocol (Xilinx Inc., 2022), several attempts were made to use it. Still, the tools change bus sizes, defining extra signals during the synthesis as showing an unexpected behavior during the simulation, thus turning out the generated module incompatible with the rest of the architecture signals and incorrect. It was necessary to look for alternatives. To try and get around this problem, we created a structure (struct) in C++ that contained variables that controlled the data bus as if it were an AXI4-Stream. Therefore, it was up to the HLS project's top function to carry out the interim control of the signals in terms of communication and signaling with the modules that communicate with the generated HDL module. Fig. 3.11 shows the declaration of the top function in C++, which will become an HDL module, along with the description of the new AXI4-Stream

structure.

```
template<unsigned D>
struct axi_stream_u {
   ap_uint<D> TDATA;
   ap_uint<D/8> TKEEP;
   ap_uint<1> TLAST;
   ap_uint<1> TVALID;
   ap_uint<1> TREADY;
};

void DeviceTop(axi_stream_u<256> &s_axis,
               axi_stream_u<256> &m_axis,
               ap_uint<1> &enable_processing,
               ap_uint<1> &tuple_in_sume_metadata_VALID,
               ap_uint<128> &tuple_in_sume_metadata_DATA,
               ap_uint<1> &tuple_out_sume_metadata_VALID,
               ap_uint<128> &tuple_out_sume_metadata_DATA,
               ap_uint<1> &tuple_out_digest_data_VALID,
               ap_uint<256> &tuple_out_digest_data_DATA,
               ap_uint<9> &control_S_AXI_AWADDR,
               ap_uint<1> &clk_lookup,
               ap_uint<1> &clk_lookup_rst,
               ap_uint<1> &internal_rst_done)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=s_axis
#pragma HLS INTERFACE ap_none port=m_axis
#pragma HLS INTERFACE ap_none port=enable_processing
#pragma HLS INTERFACE ap_none port=tuple_in_sume_metadata_VALID
#pragma HLS INTERFACE ap_none port=tuple_in_sume_metadata_DATA
#pragma HLS INTERFACE ap_none port=
   ↪ tuple_out_sume_metadata_VALID
#pragma HLS INTERFACE ap_none port=tuple_out_sume_metadata_DATA
#pragma HLS INTERFACE ap_none port=tuple_out_digest_data_VALID
#pragma HLS INTERFACE ap_none port=tuple_out_digest_data_DATA
#pragma HLS INTERFACE s_axilite port=control_S_AXI_AWADDR clock
   ↪ =clk_control
#pragma HLS INTERFACE ap_none port=clk_lookup
#pragma HLS INTERFACE ap_none port=clk_lookup_rst
#pragma HLS INTERFACE ap_none port=internal_rst_done
...
}
```

Figure 3.11 – Declaration of the HLS design top function in the C++ language.

Fig. 3.12 and 3.13 show a possible implementation for controlling the AXI4-Stream signals. One of the aims of our proof of concept is maintaining high-level programming. Therefore, all the lower-level controls, including the AXI4-Stream signals and state signaling passed to the entities discussed in Section 3.1, were wrapped into the

top function. A designer does not touch the code written in the top main function. Also, as shown in Fig. 3.12 and 3.13, it is the responsibility of the top function to make the calls to the entity functions at the right time. The top function file includes a header file containing the entity's function declarations. During the HLS synthesis, the compiler is informed that the functions will be declared through a forward declaration mechanism, so then the designer must respect the function's signatures and define them in a separate file in order for the definitions to be linked to its declarations during the synthesis process.

In HLS, it is not possible to specify which parameters are input and which are output. The tool uses an inference method based on the use of the parameter within the function to infer the direction of the generated signal. In simple terms, parameters that receive assignments (left-hand side) become output signals, and parameters that are read (right-hand side) become input signals.

After programming the top function, the C++ source code was submitted to the HLS workflow, running synthesis and generating the HDL module. Fig. 3.14 shows the signals of the module generated in Verilog (`DeviceTop.v`) from the `DeviceTop` function in C++.

Once the module was generated, the work was to integrate it into the NetFPGA design, change the P4-NetFPGA scripts and project flow to no longer instantiate P4 modules, and then integrate the C++ module into its HDL project construction flow.

```
if (s_axis.TVALID == 1) { // Data received
   m_axis.TVALID = 1;

   ap_uint<256> packet_in = s_axis.TDATA;
   ParsedPacket_t parsed_packet;
   DigestData_t digest_data;
   if (prior_TVALID == 0) {
      // Detects rising edges to start processing a new packet
      state = State::START;

      /*****************************************
                  PARSER
       *****************************************/
      Parser(state, packet_in, parsed_packet);

      // Not used, just repass
      tuple_out_digest_data_VALID = digest_data.vld_signal;
      tuple_out_digest_data_DATA = digest_data.data;

      tuple_out_sume_metadata_DATA =
         ↪ tuple_in_sume_metadata_DATA;
      /*****************************************
      *    PIPELINE
      *****************************************/
      Pipeline(state, parsed_packet,
         ↪ tuple_out_sume_metadata_DATA);
      tuple_out_sume_metadata_VALID = 1;
   }
   else {
      // Runs the data through the stages without processing it
      state = State::TRANSPASS;
   }

   if (m_axis.TREADY) {
      // Transfers the data to the output port (emits the
         ↪ outgoing packet)

      /*****************************************
      *    DEPARSER
      *****************************************/
      Deparser(state, m_axis.TDATA,
               s_axis.TDATA,parsed_packet);
      s_axis.TREADY = 1;
      state = State::IDLE;
   }
   ...
```

Figure 3.12 – AXI4-Stream signals controlling implementation pt 1.

```
    ...
    else {
        // Blocks a new data transfer until the current data is
            ↪ ready to be delivered
        s_axis.TREADY = 0;
    }
}
else {
    // Signals the module is ready to receive new data
    m_axis.TVALID = 0;
    s_axis.TREADY = 1;
}

prior_TVALID = s_axis.TVALID;
```

Figure 3.13 – AXI4-Stream signals controlling implementation pt 2.

```
module DeviceTop (
      ...
);
input [255:0] s_axis_TDATA_V;
input [31:0] s_axis_TKEEP_V;
input [0:0] s_axis_TLAST_V;
input [0:0] s_axis_TVALID_V;
output [0:0] s_axis_TREADY_V;
output [255:0] m_axis_TDATA_V;
output [31:0] m_axis_TKEEP_V;
output [0:0] m_axis_TLAST_V;
output [0:0] m_axis_TVALID_V;
input [0:0] m_axis_TREADY_V;
input [0:0] enable_processing_V;
input [0:0] tuple_in_sume_metadata_VALID_V;
input [127:0] tuple_in_sume_metadata_DATA_V;
output [0:0] tuple_out_sume_metadata_VALID_V;
output [127:0] tuple_out_sume_metadata_DATA_V;
output [0:0] tuple_out_digest_data_VALID_V;
output [255:0] tuple_out_digest_data_DATA_V;
input [0:0] clk_lookup_V;
input [0:0] clk_lookup_rst_V;
output [0:0] internal_rst_done_V;
input s_axi_AXILiteS_AWVALID;
output s_axi_AXILiteS_AWREADY;
input [C_S_AXI_AXILITES_ADDR_WIDTH - 1:0] s_axi_AXILiteS_AWADDR
   ↪ ;
input s_axi_AXILiteS_WVALID;
output s_axi_AXILiteS_WREADY;
input [C_S_AXI_AXILITES_DATA_WIDTH - 1:0] s_axi_AXILiteS_WDATA;
input [C_S_AXI_AXILITES_WSTRB_WIDTH - 1:0] s_axi_AXILiteS_WSTRB
   ↪ ;
input s_axi_AXILiteS_ARVALID;
output s_axi_AXILiteS_ARREADY;
input [C_S_AXI_AXILITES_ADDR_WIDTH - 1:0] s_axi_AXILiteS_ARADDR
   ↪ ;
output s_axi_AXILiteS_RVALID;
input s_axi_AXILiteS_RREADY;
output [C_S_AXI_AXILITES_DATA_WIDTH - 1:0] s_axi_AXILiteS_RDATA
   ↪ ;
output [1:0] s_axi_AXILiteS_RRESP;
output s_axi_AXILiteS_BVALID;
input s_axi_AXILiteS_BREADY;
output [1:0] s_axi_AXILiteS_BRESP;
input ap_clk;
input ap_rst_n;
input clk_control;
input ap_rst_n_clk_control;
...
```

Figure 3.14 – Generated Verilog signals of the HLS design top function.

4 EVALUATION

Our evaluation aims to verify the integration of the generated HDL module, device behavior, resource utilization, and performance. We built our proposed workflow targeting the NetFPGA-SUME Virtex-7, shown in Fig. 4.1. In the experiments, the L2-switch device description written the C++ language shown in Section 3.1 was used. The C++11 was the standard adopted to write the device code. Finally, we synthesize the L2-switch source code using the Xilinx HLS 2018.2 tool to generate the device HDL module in Verilog. We gathered the P4-NetFPGA resource utilization and performance data from the literature to compare with our proposal (SAQUETTI, 2020).

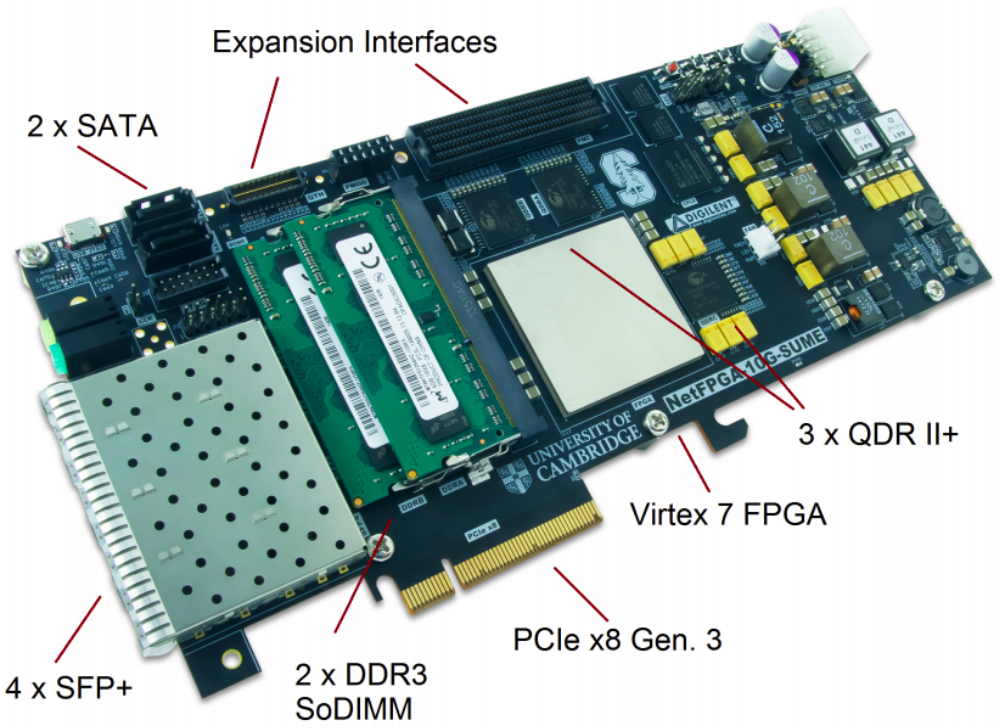


Figure 4.1 – NetFPGA-SUME FPGA Development Board (Digilent Inc., 2019).

4.1 Integration and Behavior

To validate the AXI4-Stream controlling, the device behavior, and the integration of the HDL module into the NetFPGA-SUME architecture, firstly, the P4-NetFPGA original Python script that generates test packets used to apply stimulus in the design built in the FPGA was modified to simulate a single incoming 64-byte Ethernet packet flow from the source MAC address `08:11:11:11:11:08` to the destination MAC address

`08:22:22:22:22:08`, which maps to the network interface 1 of the NetFPGA-SUME board according to the written L2-switch `forward` table.

Figs. 4.2 and 4.3 show the AXI4-Stream signals of the module operating correctly. In Fig. 4.2, the rising edge is detected when the module receives the incoming packet by `TDATA`, setting `TVALID` to 1. In Fig. 4.3, after receiving all the packet bits successfully, indicated by `TKEEP` equals to `ffffffff`, the module informs the packet is ended by setting to 1 the `TLAST` signal.

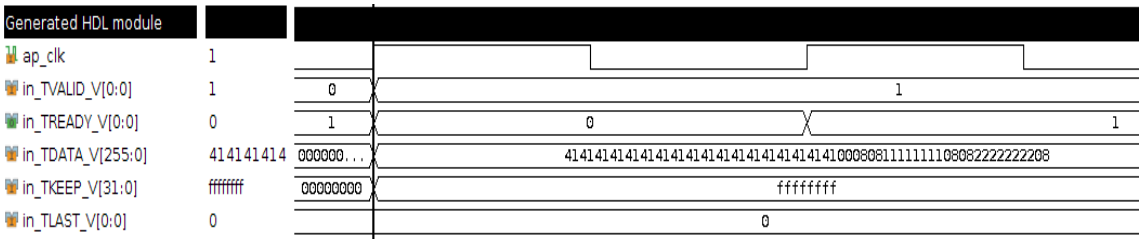


Figure 4.2 – Valid incoming packet signaling.

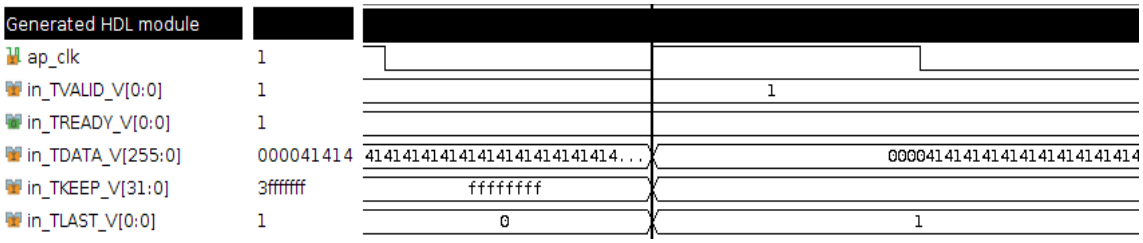


Figure 4.3 – End of packet signaling.

Fig. 4.4 shows the packet source port(`in_src_port`) equal to 1 and the destination port(`in_dst_port`) equal to 0 in the SUME metadata bus before the data enters the L2-switch. The packet is processed after the L2-switch receives the packet and its metadata. Since the physical interface 1 corresponds to the third bit from left to right using the one-hot encoding in the destination port field of the SUME metadata tuple (IBANEZ, 2019), it is set to 1, resulting in a `0x04` hexadecimal value, as shown in Fig. 4.5.

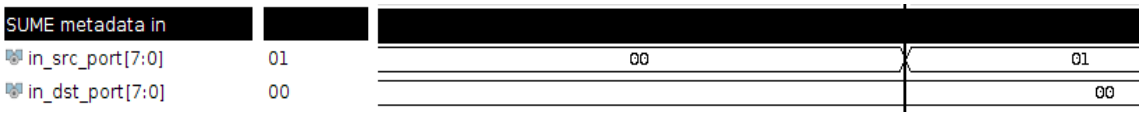


Figure 4.4 – SUME input metadata before the data enters the L2-switch.

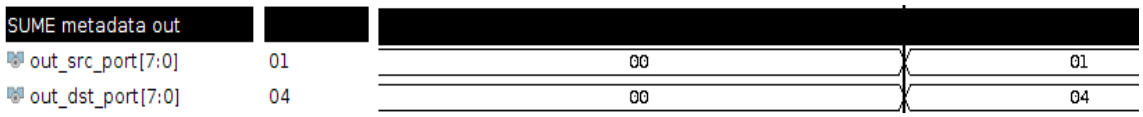


Figure 4.5 – SUME output metadata after the data is processed by the L2-switch.

Fig. 4.6 shows the packet being propagated to the module output signal after

it was processed. Finally, Fig 4.7 shows the output packet outgoing by the interface 1 output signal of the architecture.

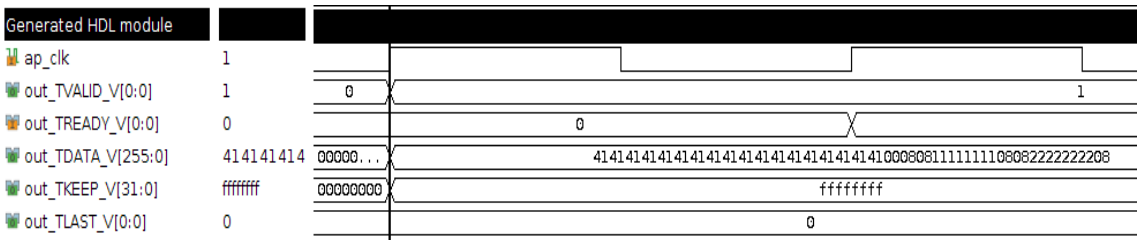


Figure 4.6 – Valid outgoing packet signaling.

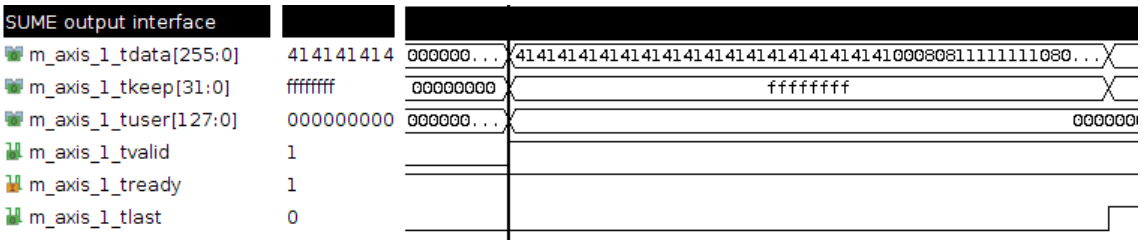


Figure 4.7 – SUME output interface 1.

4.2 Resource utilization

Our implementation resulted in a maximum operating frequency of 547.6 MHz, the same maximum operating frequency obtained with the P4-NetFPGA design. This value is directly related to the fixed part of both architectures, where the critical path resides. Fig. 4.8 shows the implemented design in the NetFPGA-SUME board mesh.

Table 4.1 shows the occupation area, measured in terms of Lookup Tables (LUTs) and the use of registers for allocating our proposal and P4-NetFPGA without the resources occupied for the HDL instances in both designs, and the resource utilization of the L2-switch instances described using the two approaches, our proposed C++ and HLS workflow and the P4-SDNet workflow used by the P4-NetFPGA.

Both architectures, running a single module instance, have similar resource usage. It shows that our proposal can add more flexibility to the programmable devices and does not require more hardware resources compared to P4-NetFPGA architecture. Our L2-switch instance occupies less than 0.01% to implement its logic. The lower spent resources compared to an L2-switch device written in P4 are due to the fact in our proposal the forwarding tables are not dynamic, therefore it does not use hardware resources needed for storing the table information in memory since forwarding table are configured

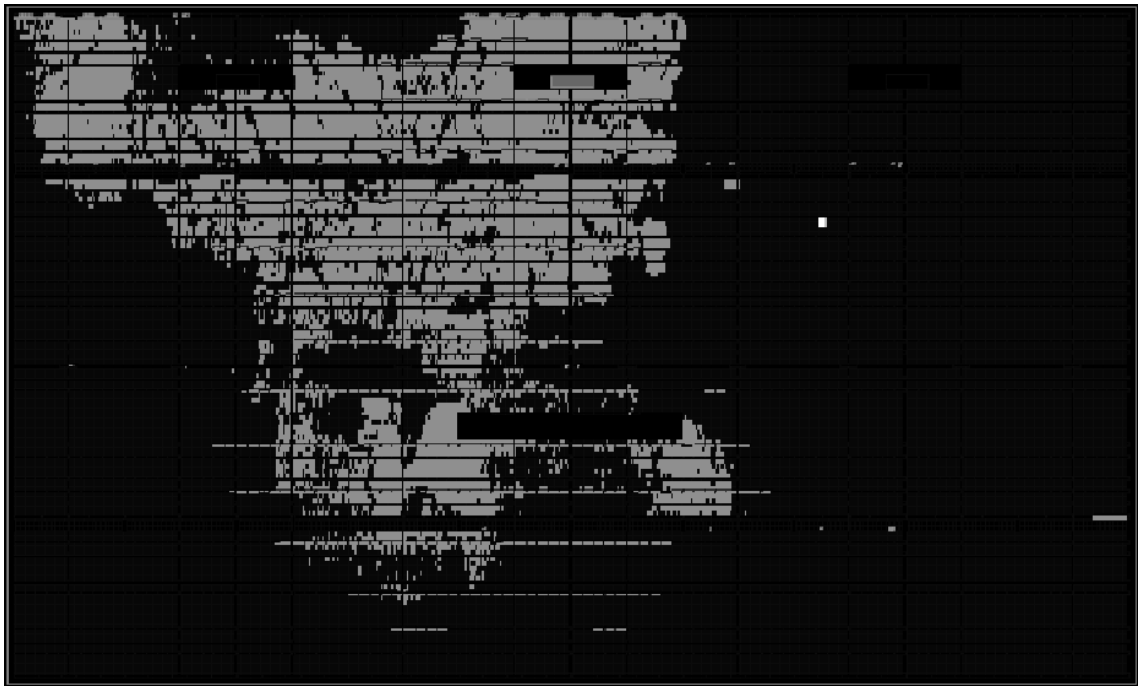


Figure 4.8 – Implemented design in the NetFPGA-SUME FPGA board mesh.

statically when the C++ code is written and its data is converted into constants in the HDL module generated by the HLS flow.

Table 4.1 – NetFPGA SUME resource usage.

Module	LUTs		Registers	
	#	%	#	%
Proposal	44495	(10.3%)	67877	(7.8%)
L2-switch (C++ and HLS)	15	(0.003%)	30	(0.003%)
P4-NetFPGA (IBANEZ et al., 2019)	44447	(10.3%)	67926	(7.8%)
L2-switch (P4 and SDNet)	28096	(6.500%)	40506	(4.700%)

4.3 Performance

We simulated our proposal running at the theoretical maximum operating frequency of 547.6 MHz rather than the clock frequency of the NetFPGA-SUME, which is 200 MHz. To measure throughput and latency, we injected 512 × 256-byte packets and 2 × 64-byte packets, respectively, to simulate an incoming packet flow. The network packets are generated using the Python `Scapy` library (Scapy Community., 2008) and a script for storing the packets content into an output file. The simulation flow scripts read the packet data from the output files produced by the script and use it as input data for the design under test simulation.

For both packet flows, we measured the number of clock cycles between the arrival of the first packet and the completion of the last on the Xilinx Vivado Simulator.

Table 4.2 shows the achieved throughput, where our proposal achieved around 140 Gbps, which translates to 50 Gbps at the deployed 200 MHz operating frequency. When comparing, the proposal throughput is around 14 times higher than P4-NetFPGA.

Table 4.2 – Throughput ($Gbps$)

Network Device	P4-NetFPGA (IBANEZ et al., 2019)	Proposal
L2-switch	121.8	139.5

Table 4.3 shows latency, which was around 0.041 µs, showing a decrease of 92% when compared to P4-NetFPGA.

Table 4.3 – Latency (μs)

Network Device	P4-NetFPGA (IBANEZ et al., 2019)	Proposal
L2-switch	0.520	0.041

5 CONCLUSION

The ability to achieve data plane programmability has offered network operators and designers several opportunities to define their network behavior. High-level description languages and FPGAs have enabled new ways of thinking about packet forwarding at line rate and abstracting most of the decisions inherent to the target. However, the lower flexibility of the languages and the lack of support for using tools that must provide the bridge between the high-level description and hardware description are still issues to be faced.

In this work, we take a first step into a more flexible data plane and more transparent workflow direction, presenting our proposal, a workflow and methodology for building network devices using the C++ language, and the High-Level Synthesis process to generate HDL instances to run into an FPGA board. The proposal integrates its main module into the NetFPGA architecture and allows designers to program a device in a high-level abstraction while providing more prototyping flexibility. Although the abstraction is smaller than in P4, it provides a great advantage compared to direct description in HDL. An L2-switch was implemented to validate our proof-of-concept. The experimental results show the proposed workflow does not require more hardware resources to be implemented when compared to the P4-NetFPGA project; at the same time, it can offer good performance latency and throughput for networking devices implemented in hardware.

With this work, we seek to foment the community with solutions and alternatives for new ways of thinking about developing such devices using higher programming abstractions and FPGA boards.

6 FUTURE WORK

The choice at the first moment for not implementing some functionalities that the P4 language, our main DSL for comparison, offers was made to eliminate barriers that could increase the complexity in the process of conceiving our proof-of-concept. Therefore, improvements can still be applied to better support common functionalities a programmable network device requires. For future work, it is intended to achieve the following goals:

6.1 Packet Processing

The proposal can only implement forwarding devices that process a maximum of data contained in the first data transfer received by the NetFPGA architecture data bus, which uses 32 bytes. Although the design can support packets bigger than the bus size, the design is not capable of aggregating data transfer if needed. It can happen in situations where the packet headers are bigger than the bus size, thus making it necessary to wait for a subsequent data transfer to assemble the packet headers and then submit them to the parser stage. The current top function written in C++ only gets packet data coming in subsequent transfers. It passes it from the input to the output of the module, transversing the stages. One solution would be to implement a packet aggregation and disaggregation mechanism where the package is aggregated, processed, and finally disaggregated into successive transfers to the next module. As it involves hardware processing, the study of possible stalls in the pipeline must be considered in this case.

6.2 Dynamic Forwarding Tables

An approach to be adopted in order to be able to access and configuring the tables at runtime is to decouple the table configuring from the program writing flow, by implementing a memory mechanism whose tables are preconfigured during the project build workflow and then accessed by the C++ code by the addressing of memory regions configured for the tables. Then, during a device operation, a network operator can change the table entries through the management interface, and changes will be reflected in the network device behavior.

6.3 HLS and Implementation Workflow Integration

Currently, the process for generating an HDL module is being separated from the workflow responsible for simulating the NetFPGA architecture using the test data scripts and implementing the design using the Xilinx Vivado tool. A future work will be to integrate the flows.

REFERENCES

AMD Xilinx Inc. **Vivado Design Suite User Guide: High-Level Synthesis**. [S.l.], 2018. Available from Internet: <https://docs.xilinx.com/v/u/2018.2-English/ug902-vivado-high-level-synthesis>.

AMD Xilinx Inc. **Vitis High-Level Synthesis User Guide**. [S.l.], 2020. Available from Internet: <https://docs.xilinx.com/v/u/2020.1-English/ug1399-vitis-hls>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833.

CONG, J. et al. High-level synthesis for fpgas: From prototyping to deployment. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 30, n. 4, p. 473–491, 2011.

COUSSY, P. et al. An introduction to high-level synthesis. **IEEE Design & Test of Computers**, v. 26, n. 4, p. 8–17, 2009.

Digilent Inc. **NetFPGA-SUME Reference Manual**. [S.l.], 2019. Available from Internet: <https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/reference-manual>.

GAO, J. et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 435–450.

IBANEZ, S. **Workflow Overview**. [S.l.], 2019. Available from Internet: <https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview#simplesumeswitch-architecture>.

IBANEZ, S. et al. The p4-netfpga workflow for line-rate packet processing. In: **ACM/SIGDA Int'l Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2019. (FPGA '19), p. 1–9. ISBN 978-1-4503-6137-8.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, 2015.

KUON, I.; ROSE, J. Measuring the gap between fpgas and asics. **IEEE Transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 26, n. 2, p. 203–215, 2007.

KUON, I. et al. Fpga architecture: Survey and challenges. **Foundations and Trends® in Electronic Design Automation**, Now Publishers, Inc., v. 2, n. 2, p. 135–253, 2008.

LOCKWOOD, J. W. et al. Netfpga–an open platform for gigabit-rate network switching and routing. In: IEEE. **2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)**. [S.l.], 2007. p. 160–161.

MAXFIELD, E. M. Xilinx introduces sdnet & 'softly' defined networks. 2014. Available from Internet: <https://www.eetimes.com/xilinx-introduces-sdnet-softly-defined-networks/>.

MICHEL, O. et al. The programmable data plane: Abstractions, architectures, algorithms, and applications. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 4, may 2021. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3447868>.

SAQUETTI, M. **Programmable virtual switches : design and implementation of the forwarding engine and supporting features**. 2020. Available from Internet: <https://lume.ufrgs.br/handle/10183/211315>.

Scapy Community. **Scapy's Documentation**. [S.l.], 2008. Available from Internet: <https://scapy.readthedocs.io/en/latest/>.

SIVARAMAN, A. et al. Dc. p4: Programming the forwarding plane of a data-center switch. In: **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. [S.l.: s.n.], 2015. p. 1–8.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 127–132. ISBN 978-1-4503-2178-5.

Xilinx Inc. **SDNet PX Programming Language Reference Manual**. [S.l.], 2017. Available from Internet: <https://docs.xilinx.com/v/u/2017.3-English/ug1016-px-programming>.

Xilinx Inc. **P4-SDNet User Guide**. [S.l.], 2018. Available from Internet: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1252-p4-sdnet-translator.pdf>.

Xilinx Inc. **AXI4-Stream Interface**. [S.l.], 2022. Available from Internet: <https://docs.xilinx.com/r/en-US/pg256-sdfec-integrated-block/AXI4-Stream-Interface>.

Xilinx Inc. **Vivado Design Suite User Guide: Getting Started**. [S.l.], 2023. Available from Internet: <https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started>.