UNIVERSITY OF THE AEGEAN
SCHOOL OF ENGINEERING

DEPARTMENT OF INFORMATION AND COMMUNICATION SYSTEMS ENGINEERING

MASTER OF SCIENCE PROGRAMME

INTERNET OF THINGS

Intelligent Environments in Next-Generation Networks

A comparative study of programmable switches on modern networking hardware

Master Thesis

of

Panagiotis Famelis

Supervisor: Kalligeros Emmanouil, Assistant Professor

Members of the thesis committee: Kambourakis Georgios, Professor
Karybali Irene, Ph.D., Laboratory Teaching Staff Member

Samos, July 2023

This page intentionally left blank.

Acknowledgements

First, I would like to thank Assistant Professor Manolis Kalligeros for the trust he put in me in exploring a relatively new field. Even though the particular subject we worked on was not in his main research interests, his continuous support was fundamental in the completion of this thesis. The same holds true for Dr. Georgios Katsikas, who went above and beyond his duties as a colleague and supported this thesis from conception to completion. Without their guidance and support, this thesis would have been just an idea.

I would also like to thank Dr. Dimitrios Klonidis and the whole NSIT team, with whom I have worked in parallel to this thesis in Ubitech, for their understanding, assistance, and resources they provided. Of course, these acknowledgements would not be complete without the people we walk together in life; my ever-growing family (too many to list them!), my always true friends (D.D., G.D, D.F., M.Z., C.M., G.K. and Clutch) and Despoina who bears with me through life for all these years.

© 2023

Panagiotis Famelis

Department of Information and Communication Systems Engineering

University of the Aegean

"O great table, without whom we are as naught."

-Most probably no one

Table of Contents

1 Introduction **1**

1.1 Modern Computer Networks 1

1.2 Thesis Topic 2

1.3 Structure of the Thesis 3

2 Theoretical Background **5**

2.1 Software Defined Networks (SDNs) 5

2.1.1 The Road to SDN *5*

2.1.2 SDN Architecture *6*

2.1.3 Controllers *8*

2.1.4 OpenFlow *11*

2.1.5 P4 ..... *13*

2.2 Programmable SmartNICs 23

2.2.1 The Need for SmartNICs *23*

2.2.2 Network Function Virtualization *23*

2.2.3 Programmability of SmartNICs *24*

2.3 State of the Art 26

3 Experiments **29**

3.1 Common Setup 29

3.1.1 Servers *29*

3.1.2 Metrics *30*

3.1.3 Network Performance Framework *31*

3.1.4 Click Modular Router *34*

3.1.5 Test Cases *34*

3.2 Open Virtual Switch (OVS) 34

3.2.1 OVS and OVS-DPDK *34*

3.2.2 Experimental Setup *36*

3.3 P4 37

3.3.1 Bmv2 ..... *37*

3.3.2 P4-DPDK *37*

3.3.3 Experimental Setup *37*

3.4 Xilinx SN1000 FPGA 38

3.4.1 Vitis Networking P4 ..... *38*

3.4.2 SN1000 Plugins *39*
3.4.3 P4 Programs *40*
3.4.4 Experimental Setup *41*

4 Results **43**

4.1 Bmv2 43
4.2 OVS-vanilla 45
4.3 OVS-DPDK 49
4.4 P4-DPDK 53
4.5 Xilinx SN1000 FPGA 57
4.6 Remarks on the Comparisons 58

5 Conclusions **61**

5.1 Closing Remarks 61
5.2 Future Work 62

Bibliography **63**

Annex A: P4 program used for Software Switches **67**

Annex B: P4 program used for Xilinx SN1000's FPGA **71**

List of Figures

Figure 1: SDN Architecture according to RFC 7426 7
Figure 2: ONOS Architecture 9
Figure 3: TeraFlowSDN Release 2 architecture 11
Figure 4: PISA (Protocol-Independent Switch Architecture) 14
Figure 5: V1Model Architecture 15
Figure 6: PSA Architecture 15
Figure 7: PNA Architecture 16
Figure 8: P4Runtime Reference Architecture 21
Figure 9: P4 complete compilation procedure 22
Figure 10: High-level NFV framework 24
Figure 11: Server setup 30
Figure 12: NPF testie flow 33
Figure 13: OVS components and interfaces 35
Figure 14: OVS and OVS-DPDK architecture 36
Figure 15: Testing flow of a P4 RTL 39
Figure 16: OpenNIC Architecture 40
Figure 17: Loss Ratio graph for bmv2 over different frame sizes 43
Figure 18: Throughput graph for bmv2 over different frame sizes 44
Figure 19: Latency graph for bmv2 over different frame sizes 45
Figure 20: Loss Ratio graph for OVS-vanilla over different frame sizes 46
Figure 21: Throughput graph for OVS-vanilla over different frame sizes 47
Figure 22: Latency graph for OVS-vanilla over different frame sizes 49
Figure 23: Loss Ratio graph for OVS-DPDK over different frame sizes 50
Figure 24: Throughput graph for OVS-DPDK over different frame sizes 51
Figure 25: Latency graph for OVS-DPDK over different frame sizes 53
Figure 26: Loss Ratio graph for P4-DPDK over different frame sizes 54
Figure 27: Throughput graph for P4-DPDK over different frame sizes 55
Figure 28: Latency graph for P4-DPDK over different frame sizes 57
Figure 29: Comparative results for packet loss 59
Figure 30: Comparative results for throughput 60

List of Tables

Table 1: Server characteristic 30
Table 2: Metrics measured or calculated in the experiments 31
Table 3: OVS-DPDK parameters 36
Table 4: P4-DPDK EAL parameters 38
Table 5: bmv2 measurements 44
Table 6: OVS-vanilla measurements 48
Table 7: OVS-DPDK measurements 52
Table 8: P4-DPDK measurements 56

List of Snippets

Snippet 1: Ethernet Header 17
Snippet 2: IPv4 Header 17
Snippet 3: Parser for Ethernet and IPv4 18
Snippet 4: Change MAC address example 19
Snippet 5: Forward table example 19
Snippet 6: Control Block example 20
Snippet 7: Conditional apply 20
Snippet 8: Deparser for Ethernet and IPv4 20
Snippet 9: OpenFlow rule used for OVS 36

Abbreviations

AI	Artificial Intelligence
AMD	Advanced Micro Devices Inc.
API	Application Programming Interface
ARP	Adress Resolution Protocol
ASIC	Application-Specific Integrated Circuit
BMV2	Behavioral Model Version 2
CDN	Content Delivery Network
CPU	Central Processing Unit
CX5	NVIDIA ConnectX 5 SmartNIC
DPDK	Data Plane Development Kit
DPU	Data Processing Unit
DSL	Domain Specific Language
DUT	Device Under Test
EAL	Environment Abstraction Layer
IP	Intellectual Property
RTL	Register-Transfer Level
ETSI	European Telecommunications Standards Institute
FPGA	Field-Programmable Gate Array
gRPC	gRPC Remote Procedure Calls
HDL	Hardware Description Language
HLS	High Level Synthesis
HTTP	Hyper Text Transfer Protocol
HW	Hardware
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IO	Input Output
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communications
IPSec	Internet Protocol Security
IRTF	Internet Research Task Force
ISP	Internet Service Provider
L2 / L3	Layer 2 / 3
LPM	Longest Prefix Match

LUT	Look-Up Table
MAC	Media Access Control
ML	Machine Learning
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NB	Northbound
NFV	Network Function Virtualization
NIC	Network Interface Card
NOS	Network Operating System
NPF	Network Performance Framework
NVMe	Non-Volatile Memory Express
ONF	Open Networking Foundation
ONOS	Open Networking Operating System
OS	Operating System
OVS	Open Virtual Switch
PC	Personal Computer
PCAP	Packet Capture
PISA	Protocol-Independent Switch Architecture
PMD	Poll Mode Driver
PNA	Portable NIC Architecture
PNF	Physical Virtual Function
PSA	Portable Switch Architecture
QoS	Quality of Service
RAM	Random Access Memory
RFC	Request for Comments
RX	Receive
SB	Southbound
SDN	Software Defined Networking
SN1000	Xilinx Alveo SN1000 SmartNIC
SNMP	Simple Network Management Protocol
SR-IOV	Single Root – IO Virtualization
STDIO	Standard IO
SW	Software
TCP	Transmission Control Protocol
TX	Transmit

UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNF	Virtual Network Function
YANG	Yet Another Next Generation

Περίληψη

Τα Δίκτυα Καθοριζόμενα από Λογισμικό (Software Defined Networks - SDNs) έχουν αλλάξει ριζικά τον τρόπο με τον οποίο σχεδιάζουμε και διαχειριζόμαστε τα δίκτυα δεδομένων. Με τον διαχωρισμό του επιπέδου ελέγχου (control plane) από το επίπεδο δεδομένων (data plane), πληρέστερα μοντέλα δικτύων μπορούν να αναπτυχθούν. Ενώ έχουν και δυνατότητα προγραμματισμού. Σε αυτό το πλαίσιο εμφανίστηκε και η P4, μια πολλά υποσχόμενη γλώσσα στον τομέα των δικτύων, για προγραμματισμό του επιπέδου δεδομένων. Με τη χρήση της P4 διευκολύνεται η ανάπτυξη και υλοποίηση συγκεκριμένων λύσεων και πρωτοκόλλων προσαρμοσμένων σε ιδιαίτερες ανάγκες και περιπτώσεις χρήσης.

Ταυτόχρονα, η Εικονικοποίηση Δικτυακών Λειτουργιών (Network Function Virtualization - NFV) με την σειρά της άλλαξε τον τρόπο με τον οποίο διαχειριζόμαστε τις δικτυακές λειτουργίες, με μεταφορά πόρων από τους φυσικούς σε εικονικούς. Με αυτόν τον τρόπο επιτυγχάνεται μεγαλύτερη ευελιξία και αποδοτικότητα. Σε αυτό το καινούργιο πλαίσιο, οι μεταγωγείς (switch) υλοποιημένοι σε λογισμικό παίζουν καθοριστικό ρόλο στη δικτύωση των σύγχρονων κέντρων δεδομένων (data centers). Τα switches υλοποιημένα σε λογισμικό μπορούν να τρέχουν είτε στον επεξεργαστή κάποιου εξυπηρετητή (server) ή σε ειδικό υλικό, όπως οι Έξυπνες Κάρτες Δικτύου (smartNICs), οι οποίες είναι βελτιστοποιημένες για να αναλαμβάνουν την εκτέλεση δικτυακών εργασιών.

Για τον λόγο αυτόν, η συγκριτική ανάλυση διαφορετικών switches και smartNICs είναι σημαντική, έτσι ώστε να μπορούμε να ελέγξουμε και να επιλέξουμε τη λύση που ταιριάζει καλύτερα στις ανάγκες κάθε διαφορετικής περίπτωσης. Στο πλαίσιο αυτής της διπλωματικής, συγκρίνουμε τέσσερα διαφορετικά switches, κλασσικού SDN και με P4 (bmv2, OVS, OVS-DPDK, P4-DPDK), σε δύο διαφορετικές κάρτες δικτύου (NVIDIA ConnectX 5 και Xilinx Alveo SN1000) υψηλής ρυθμαπόδοσης (100 Gbps). Ταυτόχρονα, ερευνούμε τον τρόπο με τον οποίο τα προγραμματιζόμενα, σε επίπεδο υλικού, ολοκληρωμένα (Field Programmable Gate Arrays – FPGAs) μπορούν να χρησιμοποιηθούν για να υλοποιήσουν δικτυακές λειτουργίες. Για τον σκοπό αυτό, γράψαμε ένα ειδικό πρόγραμμα σε P4, το οποίο μετατράπηκε σε σχεδιασμό για FPGA μέσα από τα εργαλεία της εταιρείας Xilinx και αξιολογήθηκε.

Λέξεις Κλειδιά: *SDN, NFV, smartNIC, P4, OVS, DPDK, FPGA*

Abstract

Software Defined Networks (SDN) have revolutionized the way we design and manage networks. By separating the control plane and the data plane, they allow for better abstractions over the network and enable programmability to allow custom solution. In this context, P4 has emerged as a highly promising, domain specific language for data plane programming, facilitating development and implementation of tailor-made protocols addressing specific needs and use cases.

Simultaneously, Network Function Virtualization (NFV) has transformed network function management, by transitioning from physical to virtual resources, providing flexibility and efficiency. This dynamic landscape has made software switches an integral component of modern data centers and their networks. Software switches can run on the host server's CPU, or on specialized hardware such as Smart Network Interface Cards (smartNICs), optimized for offloading networking tasks.

In this regard, comparisons of different switches and smartNICs is important in order to validate where each solution fits best and pinpoint any problems. In this thesis, we compare four different switches, considering both traditional SDN and P4-enabled SDN solutions (bmv2, OVS, OVS-pdk and P4-DPDK) on two different smartNICs (NVIDIA ConnectX 5 and Xilinx Alveo SN1000) at input rates up to 100 Gbps. Additionally, we investigate how Field Programmable Gate Arrays (FPGAs) can be used to host networking functions. To this end, a special P4 program was also created, converted into an FPGA design using the provided toolchain by Xilinx, and evaluated.

Keywords: *SDN, NFV, smartNIC, P4, OVS, DPDK, FPGA*

1

Introduction

1.1 Modern Computer Networks

Communication networks existed long before the first computers were conceived. From the ancient systems of phryctoriae, pyrseia and hydraulic telegraphs to the more recent electrical telegraphs, people needed a way to communicate across long distances. Modern networks are far more complex and sophisticated than older ones, but the main components are still there: a) a channel- phryctoriae towers had to be visible to their neighbor towers, b) a code- pyrseia operators had to have a common coding system and c) synchronization - hydraulic telegraph operators had to pull the plugs simultaneously. Of course, different physical layers would allow for different and more complex functionalities. This is why for example, the electrical telegraph was able to dominate the world and herald the coming of modern communication networks.

Computer networks, however, allowed for something more. Basically, computer networks merged two branches together, that of communication networks and computer science. By moving from analogue to digital communication, higher abstractions were created. The underlying layer is now bits, an abstraction of the physical world. Due to this, better and more specialized protocols were developed, serving highly specialized purposes, allowing for richer communication and in turn expanding the networks. Eventually, this constant development gave us today's internet.

Yet, richer networks also mean further complexity, in terms of protocols, the ever-expanding network size, the involved hardware, i.e., routers, switches, NICs (Network Interface Cards), etc., and the types of communication, i.e., wired or radio. The innovations in PCs (Personal Computers), laptops and smartphones, allowed more people to connect to the internet. At the same time, the concept of IoT (Internet of Things) emerged, which envisioned the internet as a cyber-physical system. IoT describes the network of various physical entities, called things, which are connected

to the internet. These things collect measurements from their environment and communicate with other things or servers, allowing, in the end, the automation of physical tasks and self-configuration [1]. The increasing number of connected devices and data shared among them proved to be a difficult problem.

To handle this complexity, various management tools and protocols were created, including specific solutions tailored to vendor-locked cases. Also open protocols emerged, like IETF's (Internet Engineering Task Force) SNMP (Simple Network Management Protocol) [2], which became the most famous and de facto management standard for many years. With those tools, operators can connect to entities in an IP (Internet Protocol) network, like routers, monitor their status and configure them. However, that did not change the fact that routers started to become bloated elements, due to their design that needed to support all the various protocols of the internet. At the same time, end-users were becoming more specialized, with specific needs in terms of communication. This meant that on the one hand, not all features of a router were needed, but on the other hand more specialized protocols (like IoT protocols) were developed, which led to more specialized hardware too [3].

The solution can be found not only in creating better management protocols but also in changing the architecture of networks, focusing on alternate ways of controlling and managing the various entities. This gave birth to the notion of SDN (Software Defined Networking), which is a new way of developing and thinking about networks. In SDN, as we will see later in this thesis, networks have a centralized control plane that is programmable with conventional programming techniques and is able to configure the data plane, where the actual switching and routing takes place. Going one step further in this direction, programmable data planes were also introduced, i.e., white-box switching elements that operators are able to program with specific functionalities, allowing tailor-made solutions for their own networks and circumventing the aforementioned bloated software stacks of routers.

More importantly, programmable switches bring a paradigm shift in how we design and implement networks, even more than SDN. P4, the de facto data plane programming language, brings such abstractions that allows a developer to easily create intricate protocols and implement complex functionalities. Moreover, the nature of data plane programmability is interdisciplinary, standing between the areas of networks, systems programming, and hardware development. As we will see later, tools have been developed that allow P4 to serve as an HDL (Hardware Description Language) and produce hardware design for FPGAs (Field-Programmable Gate Array). However, programmable data plane is also a new area, with a lot of innovation and potential use cases, but limited testing and actual implementations. While it is fairly easy to run simulations or emulations of software white-boxes, actual hardware programmable switches and smartNICs are still much more difficult to handle, in terms of price, availability and configuration.

1.2 Thesis Topic

The introduction of virtualization technologies migrated the physical servers to VMs (virtual machines), which means that parts of the networking operation also migrated to serve VMs. This led to the introduction of software switches. Networking has also evolved, introducing new

paradigms like programmable data planes. This means that virtual networking should also evolve to reap the benefits of such evolutions. At the same time, the introduction of smartNICs to the modern data center has allowed for increased performance by offloading the networking operations to specialized and programmable hardware.

In this master thesis we engage with exactly the question of measuring the performance of different software switches using different smartNICs as the physical networking elements. We want to compare similar smartNICs in terms of their performance and evaluate whether they would serve the modern needs of data centers and virtualization. For that reason, an NVIDIA ConnectX 5 and a Xilinx Alveo SN1000 smartNICs were used. Both of these cards have the same speed rate of 100Gbps. However, the Xilinx card also includes an FPGA to which hardware can be implemented to quickly serve highly specialized functions.

Simultaneously, we want to check how emerging technologies like P4 programmable software switches stand next to tested and proven solutions. As such we test four different software switches, from simple proof-of-concept ones like bmv2, to production grade switches like OVS and OVS-DPDK, and experimental new switches like P4-DPDK. We test these four switches with a varying number of packet lengths and check their performance in terms of latency and throughput. Additionally, we try to check the FPGA programmability of the Xilinx card using P4 as an HDL, both in terms of performance and ease of use.

In this endeavor, we hope to achieve a dual objective. On the one hand, to come in grip with modern networking concepts and see the methods and technologies involved in designing, building, and managing modern networks. On the other hand, we would like to test certain promising software switches that, to the best of our knowledge, have not been thoroughly tested like P4-DPDK.

1.3 Structure of the Thesis

This thesis is organized in five chapters. In Chapter 2 we provide a thorough introduction of the underlying concepts that are required to understand the needs that have arisen in modern networks. In Chapter 3, we detail the experimental setup where we ran our experiments, along with the tools and methods used. In Chapter 4, the results of the experiments are presented along with a discussion on them. Finally in Chapter 5 a recapitulation of our findings from the performed experiments is presented together with some thoughts on future work that this thesis has inspired.

More specifically in Chapter 2, we conduct a comprehensive review of the various changes that happened in the history of modern computer networks. We introduce the notion of Software Defined Networks, their architecture and definitions, along with the most important protocols and controllers. We also review P4, which is, as mentioned the most common programming language used for programming data planes. Furthermore, we discuss the emergence of smartNICs, detailing their applications, programmability, and how they shape today’s data centers. We conclude Chapter 2 with a review of the relevant literature in the area, especially that regarding testing of switches and smartNICs.

In Chapter 3, the experimental procedure followed is described. In the first section we discuss about the common setup, and the servers used. Additionally, the common software tools that are present in all the experiments are described. Following, for each experiment we present the software switch under test and the parameters that were used to configure it. A little more detail is provided

regarding the Xilinx Alveo SN1000 card, its architecture and the procedure we followed to program its reconfigurable device.

Chapter 4 contains the results acquired from our experiments. The measurements obtained in multiple executions of the tests are presented in diagrams detailing the loss ratio, throughput, and latency for each switch and smartNIC. In this way, a comparative analysis is possible that highlights the differences in behavior among the various switches. We conclude this chapter by describing the difficulties presented in programming and testing the FPGA of the second card.

Finally, Chapter 5 concludes the thesis by recapitulating our findings. Additionally, as we consider this thesis the beginning in a series of work related to programmable networks, we highlight what we believe that possible future work could be and open issues that could be pursued after this.

2

Theoretical Background

2.1 Software Defined Networks (SDNs)

2.1.1 The Road to SDN

One could argue that SDN is one of the most important paradigm shifts that happened in the field of computer networks. Of course, there were earlier attempts to introduce programmability to computer networks. For example, Active Networking in the 1990s was one of the first research projects that paved the way to modern software defined networks. In Active Networks, the operator could send signals to network nodes, along with specific code that had to run on them, essentially changing the behavior of the network programmatically from afar [4].

With the expansion of the internet in terms of data transferred, that kind of in-network processing was starting to seem a good solution for the ISPs (Internet Service Providers), who wanted a way to be able to implement traffic engineering and route traffic in an efficient manner. Traditional distributed routing algorithms, while essential to the operation of any network, lack in terms of ensuring QoS (Quality of Service). However, this kind of programmability was still hard to implement on a large scale, mainly due to the slow adoption rates of innovation by the big telecommunication providers. The solution came with the introduction of OpenFlow Internet protocol in 2008. OpenFlow built on top of the existing TCP/IP (Transmission Control Protocol) suite of protocols, enabling fast adoption to the various operators, while also using slightly modified existing hardware [5]. OpenFlow also marks the first formal introduction of the SDN paradigm as such. For many years SDN was synonymous to OpenFlow, which obfuscated what SDN really is about: is it just some new protocols, is it a new architecture, is it a new way of developing software for networking?

2.1.1.1 Separation of Planes

In the heart of SDN lies the distinction between two separate planes of a networking element: the control plane and the data plane (or forwarding plane). The data plane has to do with the functionality of moving data. A switch for example, has the ability to receive data in port 1 and transmit it through another port 2, this is part of the data plane. The control plane on the other hand,

involves how or why this data must go through port 2. This is done through specific rules installed in the switch. Traditional switches and routers encompassed both control and data planes, with the various decisions being made locally using distributed protocols. In SDN, switches and routers are left only with the data plane, i.e., the functionality of transferring data, while the control plane is centralized in a single controller for all the elements in the network. The controller collects data from the switches to learn the state of the network and then makes decisions based on the whole picture. By using what is called a forwarding abstraction, the controller can pass any instructions to the SDN-enabled switches. Forwarding abstractions are well-defined interfaces between controller and the switches, which allow any switch vendor to implement independently and without any restrictions their data plane, provided it can expose and consume that interface. In that way, a controller can operate over a heterogeneous group of switches in a multi-vendor network. In the end, SDN enables not only more efficient routing and switching, but also the implementation of more complex functionalities, like traffic engineering, QoS, NAT (Network Address Translation), firewalls, etc. [6].

2.1.2 SDN Architecture

The separation of control plane and data plane is fundamental to the SDN concept. However, just that distinction is not enough to describe the full architecture of an SDN. Following the introduction of OpenFlow, a lot of projects and individual research have enhanced the definition and use cases of SDN. We presented above, the main distinction between data or forwarding plane and control plane. Network engineers started expanding this duality by introducing more layers, like the management layer which concerns the managerial aspects of the control plane (enable/disable ports, etc.), or the application layer, which encompasses all the complex algorithms that may run over the control plane.

"Software-Defined Networking (SDN): Layers and Architecture Terminology" (RFC[1] 7426) [7] is IRTF's (Internet Research Task Force) attempt to standardize SDN architecture through collecting and consolidating the various ideas that appeared until then. According to RFC 7426, SDN can be fully described by five different planes, going from the infrastructure to higher abstractions (Figure 1):

1. Forwarding Plane
2. Operational Plane
3. Control Plane
4. Management Plane
5. Application Plane

Mapping to the aforementioned data plane / control plane split, the above variation further splits those two planes according to whether their job involves the function of the network itself (Forwarding and Control planes), or the management of its entities (Operational and Management

[1] RFCs stands for "Request for Comments" and correspond to publications of Internet Engineering Task Force, regarding internet operation, protocols, and architecture.

planes). Additionally, it adds another layer, the application plane, which proves to be of great interest.

At the base of the architecture are the network elements, which encompass the forwarding and the operational plane. The forwarding plane contains the functionalities of the data plane, as they were described before, i.e., it is responsible for receiving packets and change, forward, or drop them according to some predetermined rules. The operational plane contains the functionalities that are related to the state of the device, i.e., it is responsible for informing upper planes about the device status (active/inactive), checking and changing if needed, the status of each port (enabled/disabled) and also collecting and exposing monitoring data (temperature, resources utilization, etc.).

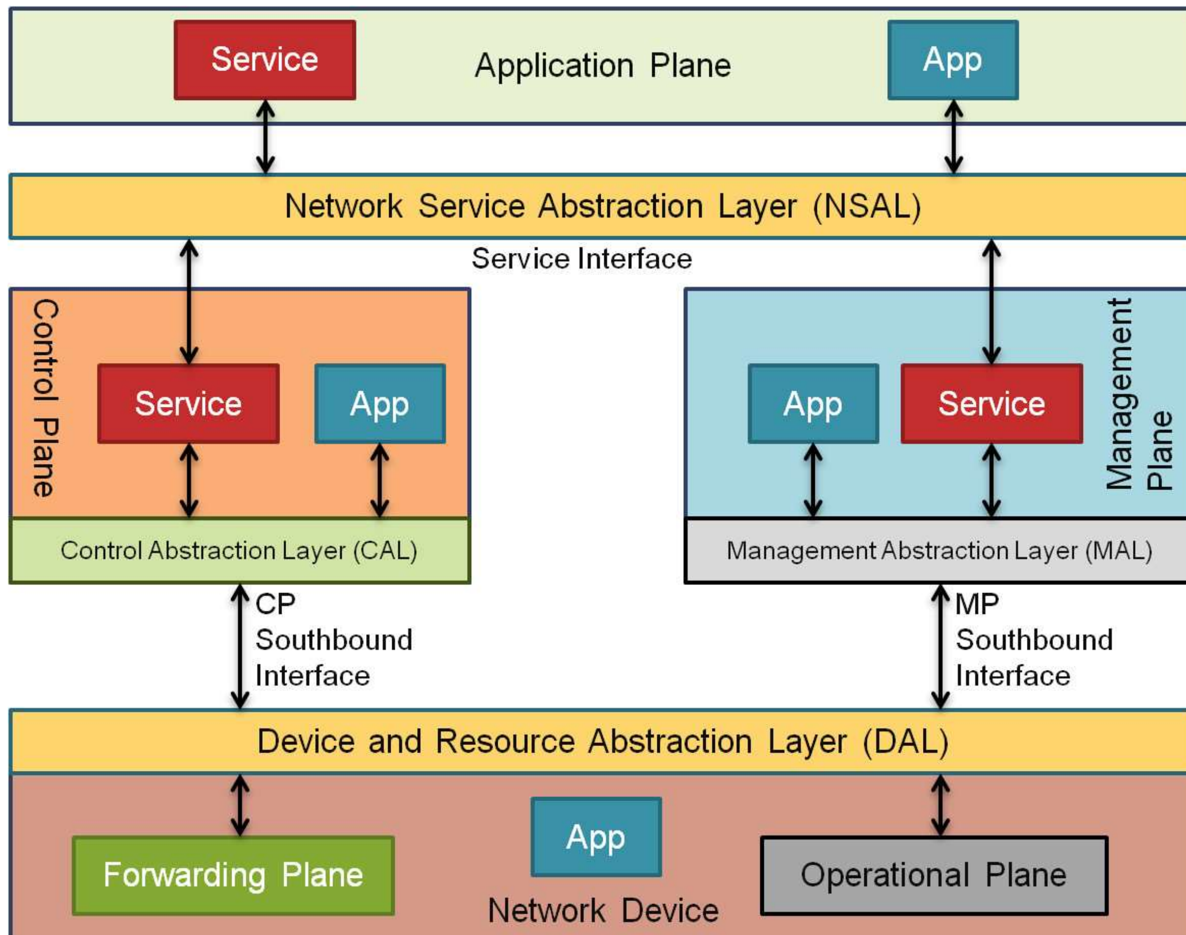

Figure 1: SDN Architecture according to RFC 7426[2]

This conceptual separation between network functionality and management of elements is also present in the differences between control plane and management plane. The control plane is responsible for deciding about how different flows should travel throughout the network, what paths they should take in the controlled network and pushing down such configuration to the forwarding plane. On the other hand, the management plane is responsible for monitoring the managed elements and configuring them in terms of their operations. For example, in a scenario of energy efficiency,

[2] https://sdn.ieee.org/newsletter/september-2017/overview-of-rfc7426-sdn-layers-and-architecture-terminology (accessed 03/07/2023)

the control plane is responsible for finding energy efficient routes and grooming the traffic to utilize the full bandwidth of a channel. Then the management plane is responsible for finding and turning off the switches or the ports that are not in use anymore, after the changes are pushed down by the control plane.

Before moving on, it is important to note that while forwarding and operational planes, and control and management planes are considered different, their exposed interfaces to the upper and lower levels are common. Essentially this means that the original control plane / data plane dichotomy still exists; control plane encompasses both control and management planes, while data plane encompasses both forwarding and operational planes. A Device and Resource Abstraction Layer (DAL) is considered southbound of the controller to manage and control the data plane, while a Network Service Abstraction Layer (NSAL) is defined northbound, to communicate with high-level applications.

These applications are part of the application plane, which contains all high-level services and algorithms that can be run on a network. These could include complex functionalities like policy and QoS enforcement, running through complex algorithms that could also include ML (Machine Learning) and AI (Artificial Intelligence). It should be mentioned that specific applications can also live in the control and management planes or even in the data plane. However, in that case, the applications should primarily support the basic operation of the network, such as the routing processes, or be involved in automations of the device itself, for example power management.

2.1.3 Controllers

This structured architecture is extremely useful as it allowed computer scientists to perceive SDNs like OS (Operating Systems) [8]. The data planes can be understood as computer devices. The control plane communicates with the network entities and provides higher abstractions to northbound applications. In turn, those applications can control the network entities, without having to rely on low level control. That is why the term NOS (Network OS) was coined to also describe the controller. Many controllers were developed over the years, from commercial solutions provided by big technology industries to smaller experimental ones, usually open source, that serve specific use cases. From the second category two solutions seem to be the most interesting in terms of how they influence the research community, in the past and the future: ONOS (Open Networking Operating System) by ONF (Open Networking Foundation) and TeraFlowSDN by ETSI (European Telecommunications Standards Institute).

2.1.3.1 ONOS

ONOS is one of the oldest open source SDN controllers, first released in 2014 and written in Java [9]. As an SDN controller it can configure and manage a variety of devices using network management protocols like Openflow, P4Runtime, NETCONF, etc. and provides a NorthBound API (Application Programming Interface) which allows the development of applications that serve a specific purpose. One of its main advantages is the ability to run as a distributed system across different servers.

ONOS follows the principles of Code Modularity and Configurability, which mean that the system is extensible with new subsystems or applications and the user has the ability to define which

subsystems are loaded each time. At the same time there is Separation of Concern, with each subsystem having a clear function and Protocol Agnosticism, in the sense that it provides internal abstractions that can handle any southbound protocol in a uniform way. These principles lead to the architecture of Figure 2. Essentially ONOS is a monolithic application comprising of three main components: the Distributed Core, the NB (Northbound) Core API and the SB (Southbound) Core API. The Distributed Core is responsible for storing the network state, providing information to the NB API and configuration to the SB API. The NB Core API exposes specific interfaces, to which applications can adhere and communicate with ONOS, providing higher-level functionalities. The SB Core API, exposes a specific interface, which is consumed by device providers and handles the translation of the abstract models of ONOS to the specific models of each different device (OVS, bmv2, stratum, etc.) or protocol (SNMP, OpenFlow, P4Runtime, NETCONF, etc.)

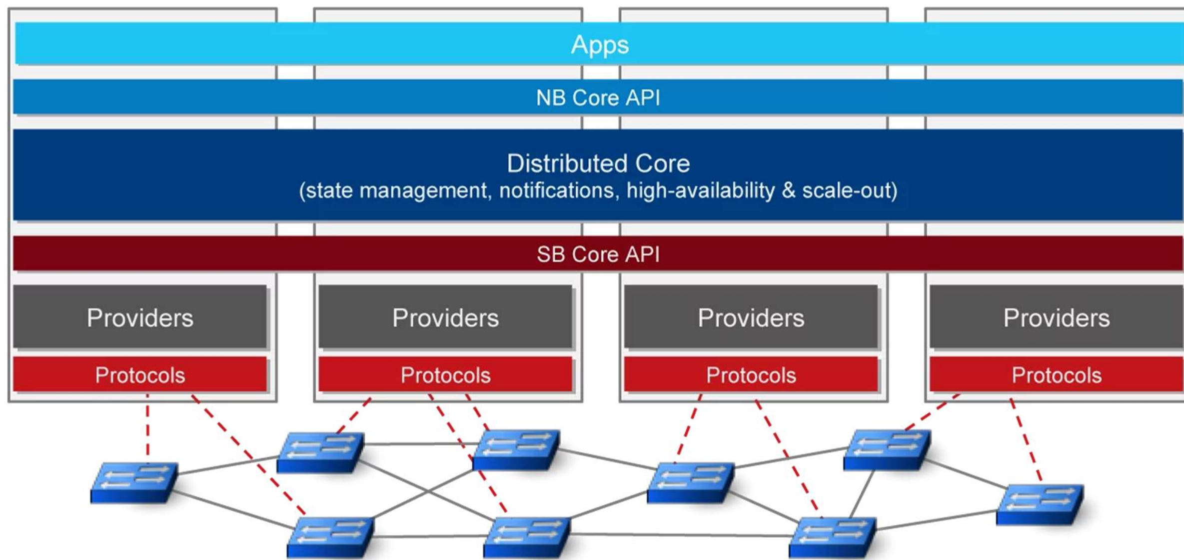

Figure 2: ONOS Architecture[3]

ONOS is currently used as the basis for most of ONF's projects. As such it has proven to be able to serve many use cases. For example, in SD-RAN[4], it is shown how ONOS can be used to manage radio resources, while in SD-FABRIC[5], ONOS is used to control programmable data planes in the edge cloud. All in all, ONOS is a robust, yet complex SDN controller, which has greatly benefitted the SDN community. Unfortunately, even though ONOS has many promising features, it seems that its development has stopped. The main project's github repository has been last updated in August 2022 and the last release (2.7) was in December 2021[6]. This roughly coincides with the acquisition of ONF's development team by Intel. At the same time there is an effort to transition from ONOS to μONOS, adapting the Distributed Core to microservices. However, the roadmap is

[3] https://opennetworking.org/onos/ (accessed 29/06/2023)

[4] https://docs.sd-ran.org/master/index.html (accessed 29/06/2023)

[5] https://opennetworking.org/sd-fabric/ (accessed 29/06/2023)

[6] https://github.com/opennetworkinglab/onos/tree/master (accessed 29/06/2023)

not clear, as there is still no major release for it and most of the information that can be found is from 2019, when the effort started.

2.1.3.2 TeraFlowSDN

TeraFlowSDN, which emerged from the Horizon2020 research project TeraFlow, is an open source cloud-native SDN controller build around microservices [10]. As such it is distributed by nature, able to auto-scale to serve a large number of flows and devices, and load balance across its different instances. It provides support for a variety of different networks and protocols, including Transport API, NETCONF, gNMI, P4Runtime, and native support for various YANG (Yet Another Next Generation) models, like IETF's microwave elements model.

Following the traditional SDN architecture, TeraFlowSDN comprises of two sets of microservices: the core microservices and overlay netapps. Core microservices include the functionalities that belong to the control and management plane of RFC 7426, while overlay netapps would belong to the application plane. In the heart of the core components is the Context microservice, which keeps all the necessary information for the state of the network, like topology, devices, services, etc.

TeraflowSDN's aim is to be an SDN controller for the whole spectrum of different kind of networks. From the 5G edge, to transport networks and to the IP networks inside a datacenter. For that reason, it follows a model with a single SB interface, the Device microservice, on which different plugins for different kinds of devices can be developed. Over Device, another abstraction is exposed by the Service microservice, which describes a service possibly spanning over different kinds of networks and devices. In that way an operator can describe high level policy requirements in the service level and TeraFlowSDN is able to handle all the necessary configurations at the device level [11].

Similarly, TeraFlowSDN exposes an NB interface to communicate with other systems. This can be used to combine different instances of TeraFlowSDN to form hierarchical controllers and orchestrators, in case, for example, that we need to have different controllers for different domains. Additionally, TeraFlowSDN can cooperate through the NB interface with other systems, like for example ETSI OSM-Open Source MANO (Management and Orchestration), which is able to manage NFV (Network Function Virtualization). The whole TeraFlowSDN architecture is presented in Figure 3.

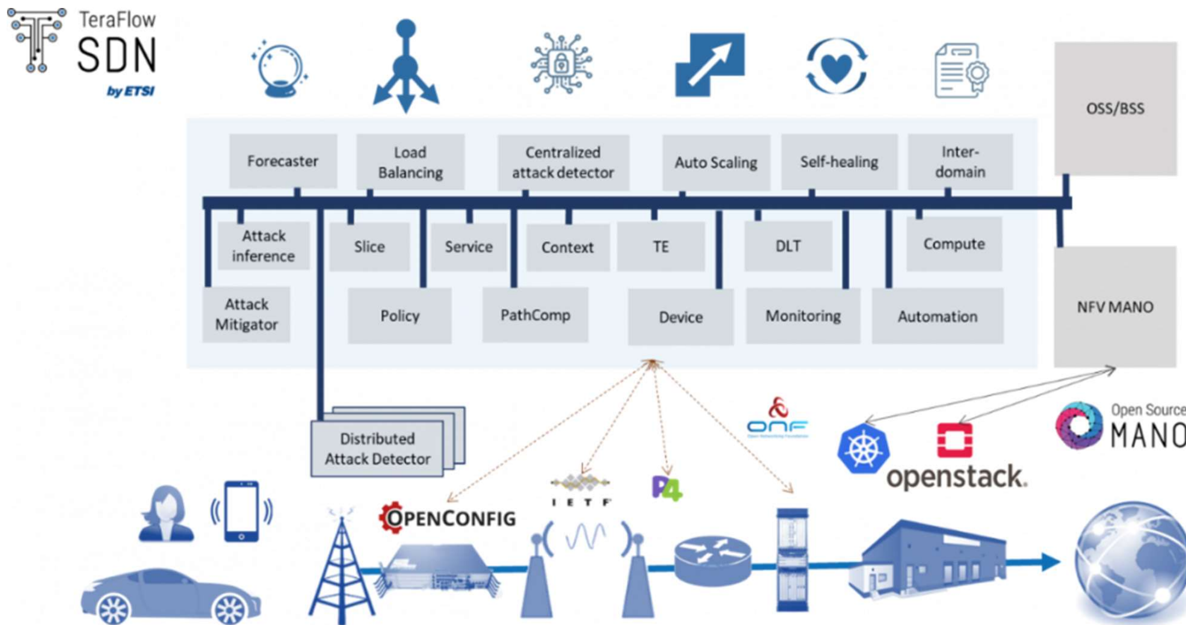

Figure 3: TeraFlowSDN Release 2 architecture[7]

2.1.4 *OpenFlow*

2.1.4.1 OpenFlow Operation

OpenFlow was for a long time the de facto forwarding abstraction, i.e., a protocol that enables the communication between a controller and SDN-enabled devices. There are two types of OpenFlow devices: OpenFlow-only and OpenFlow-hybrid. The first one means that the switch operates only based on the OpenFlow tables, while hybrid means that the switch also supports standard Ethernet operations, like L2 (Layer 2) switching, VLAN (Virtual Local Area Network), L3 (Layer 3) routing, etc. But in both cases, an OpenFlow device contains one or more tables (flow tables) which can contain specific rules (flow entries), installed by the controller. When a packet enters the switch, the switch checks if a flow entry matches with the parameters of the received packet. In this case, the switch executes the related instructions. If no rule matches, a default action can be set by the controller.

While OpenFlow has been discontinued in terms of further developments, it is useful to present how it works, because of its simplicity and how fundamental it is for other newer protocols. According to OpenFlow version 1.5 [12], flow entries consist of the following elements. Note that, except for the Counters field, all the rest of the elements are set by the controller.

- Match fields: specify the information to match the packet.
- Priority: specifies the priority of the specific flow entry. Only the highest priority flow entry that matches the packet will be selected.

[7] https://www.teraflow-h2020.eu/blog/upcoming-etsi-teraflowsdn-release-2-features (accessed 29/06/2023)

- Counters: mark how many packets have matched with the specific flow entry. It is automatically updated and not set by the controller.
- Instructions: specify what should be done with the matched packets.
- Timeouts: specify the amount of idle time before the flow is considered expired and removed by the switch.
- Cookie: is an arbitrary value that is chosen by the controller. Its function is to provide a means for the controller to associate extra information with flow rules.
- Flags: alter the way flow entries are managed. For example, the controller can set specific flags in order to also check another table if the flow entry is matched, or remove the flow entry after it matches, etc.

It is obvious that the two most important elements are the Match Fields and Instructions elements. In Match Fields, a controller can request to match the packet based on ingress port, Ethernet and IP types, source and destination MAC (Media Access Control), IPv4 and IPv6 addresses, and source and destination ports of TCP and UDP (User Datagram Protocol). These are the required fields recognized by OpenFlow from its first version. Following versions added more fields as optional, depending on the implementation, like ICMP (Internet Control Message Protocol) fields, ARP (Adress Resolution Protocol) fields, various metadata, etc. It should be noted that the match is not only exact. The controller can request in total four types of matching: exact, wildcard, prefix, and range matching.

Like Match Fields, the Instructions element also defines two different sets of possible actions. One is required to be implemented by any OpenFlow switch, while a second set is optional. The required set allows the selection of egress port, the ability to drop a packet, and the ability to flood it (through the use of groups). The optional set contains various actions mainly related to changes in VLAN and MPLS (Multiprotocol Label Switching) tagging, but also the ability to set a specific value to some field of the packet. That last functionality, while not required by OpenFlow is recommended to be considered as required. Those actions can also be set by the controller as the default action in case no flow entry is matched, as mentioned before.

2.1.4.2 OpenFlow Criticisms

OpenFlow was the first major implementation of SDN. As such, it became synonymous with it. Of course, some time was needed until SDN was standardized as something more than OpenFlow, as we will present in the next section. However, during that time, criticism of OpenFlow meant criticism of SDN and vice-versa. The most common criticism had to do with the misconception that if ultimately an SDN switch was to be without control plane, i.e., without the ability to decide where each packet should go, then that would mean that each packet (or at least the first of a flow) arriving on a switch should travel to the controller, which would decide the actions needed and install a new flow entry to the switch, specific for that packet or flow. Of course, this is not true. The controller can install flow entries preemptively and it preconfigure what is to be done with packets that do not match any currently installed flow entries. Ultimately, SDN does not mean that the network could not work without the controller. On the contrary, in many cases, the controller just intervenes when something is wrong, or some special case arises [4].

Another criticism that arose had to do with the centralized nature of SDN. The truth is that collecting the control plane of all the network elements in a centralized silo, the controller, could seem like a security risk. If the controller is lost, so does the whole network. The answer to this criticism has two parts. On the one hand, as mentioned before, the network can continue either by having hybrid switches, which can fall back to non-SDN operation, or by pre-installing correct rules that allow the network to operate with minimal control plane intervention. On the other hand, the fact that the SDN proposes a centralized architecture does not mean that the controller itself is centralized. The controller can be, and most probably is, distributed in nature [13]. Its functions are logically centralized, but the controller can be physically distributed, running across different servers. The exact relationship between the controller's various instances are up to each specific implementation. If for example they use a hierarchical or swarm model, but in a correctly built network, the controller should be able to always fall back to a live running instance.

2.1.5 P4

As the SDN concept was evolving, OpenFlow's development stumbled upon some questions: Can OpenFlow, as the de facto SDN implementation, keep up in terms of elements supported? Can we continue to have fixed tables and fields that flows can match to? Or, maybe, we need more flexibility on how we define both the functionality of the data plane and its management?

The above questions led to what came to be the new standard for SDN, the P4 programming language [14]. P4 stands for Programming Protocol-independent Packet Processors and is a DSL (Domain Specific Language), which can be used for data plane programming. The idea of having programmable switches was not new. For example Click modular router [15] had already been proposed, as a way to define specialized software switches. However, P4 was designed with the control plane in mind, encompassing the full SDN paradigm and with the aim of evolving into an OpenFlow 2.0. The programmer can use P4 to define the pipeline of a switch. This is done by defining parsers of protocols. The switch can understand and define specific match-action tables that it will use to act on each incoming packet. The match-action tables are defined in such a way that allow a control plane to control the switch functions. This is done through an API, called P4Runtime, that is generated at compile time and defines all the possible actions and instructions the controller can use to instruct the switch.

Before describing in detail the internals of the P4 language, it is important to note the significance of it. While SDN changed the way we see and design networks, P4 went one step further, unlocking the full potential of the paradigm shift that takes place. In the history of programming languages there are two main points that accelerated the evolution of the field. The introduction of assembly language, which allowed programmers to write readable code and the introduction of high-level languages, which allowed programmers to reason about code. The introduction of P4 bears the same significance. If protocols are like assembly, readable but cumbersome, leading easily to ossification, P4 is the high-level language for networks, allowing the development of tailor-made protocols for each network, with flexibility and agility in their deployment [16]. We are still in the midst of this revolution and cannot foresee how it will evolve, but P4 can act as the bridge between the fields of network and software engineering. This is not only beneficial in terms of what we can describe, or how we can offload parts of programming to the

network itself, but more importantly, it means that networks can be formally verified and not depend on the implementation of some specific protocol [17].

2.1.5.1 P4 Architectures

At the heart of P4 lies PISA (Protocol-Independent Switch Architecture), as shown in Figure 4. PISA comprises of three basic building blocks:

- Programmable Parsers
- Programmable Match-Action Pipeline
- Programmable Deparsers

Through the programmable parser, the developer can define the protocols that their data plane will recognize and use. Like in OpenFlow, packets pass through a series of tables, the programmable match-action pipeline, where the packets are altered, routed, dropped, or stored based on specific conditions. Finally, the developer is also able to declare the programmable deparser, defining how the output packets will look in the wire.

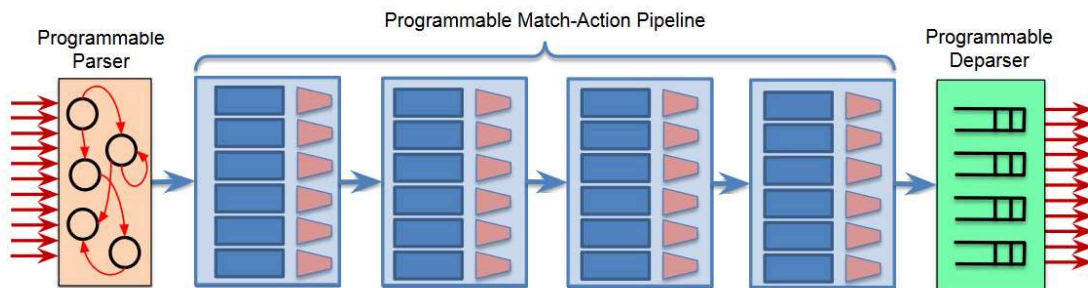

Figure 4: PISA (Protocol-Independent Switch Architecture)[8]

In its most simple architecture, PISA can look like Figure 4, i.e., an incoming packet would first go through a parser, then continue to one or more match-action tables and finally arrive at a deparser before going out the switch. However, one could think of different setups. First of all, special non-programmable blocks can also be included. These blocks are called externs in P4 and have some fixed function they are serving. Additionally, the order of the various blocks is not fixed. Most probably, a parser is needed at the beginning and a deparser at the end of the pipeline, but in the middle all four blocks can exist in various setups to better specific needs or different hardware designs. For example, one could argue for an architecture that would first parse some specific protocol, decide through a table on it and if the packet were to continue, then parse the inner protocols. That would mean that the architecture would have pairs of one parser and match-action tables in a row.

P4 has two main versions: $P4_{14}$[18] and $P4_{16}$[19]. In $P4_{14}$, the V1Model (Figure 5) was the standard architecture used. In it two different sections are defined; an Ingress, comprised of a parser and some match-action tables that are used, among other things, to verify the checksum of the incoming packet, and an Egress, comprised of some math-action tables, that among other things, can update the checksum of the packet to be sent out of the switch, and a deparser to serialize the

[8] https://opennetworking.org/wp-content/uploads/2020/12/P4_tutorial_01_basics.gslide.pdf (accessed 30/06/2023)

packet. In-between, the Traffic Manager extern is present that queues the packets and assigns priority if needed. V1Model was deprecated in $P4_{16}$, which introduced two new standard architectures: PSA (Portable Switch Architecture) [20] and PNA (Portable NIC Architecture) [21].

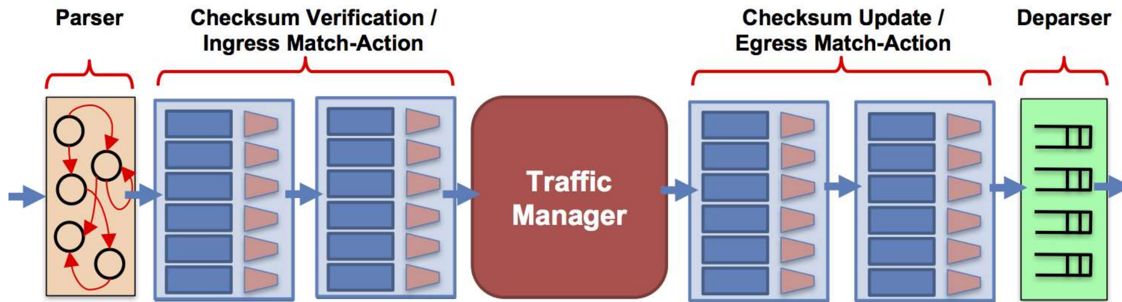

Figure 5: V1Model Architecture[9]

PSA [Figure 6(a)] is the evolution of the V1Model. Like previously, two sets of pipelines are defined, Ingress and Egress. Both have a parser, some match-action tables and a deparser. In-between a Packet Buffer and Replication Engine exists, which is responsible for queuing and replicating packets if needed (for example in the case of multicast). Also, at the end, a Buffer Queueing Engine is present that queues the packets and can also forward them to the CPU (Central Processing Unit). In Figure 6(b), the various paths a packet can take are presented. It can be seen that while PSA is similar to V1Model, it is much more flexible, allowing for better recirculation and handling of multicast packets.

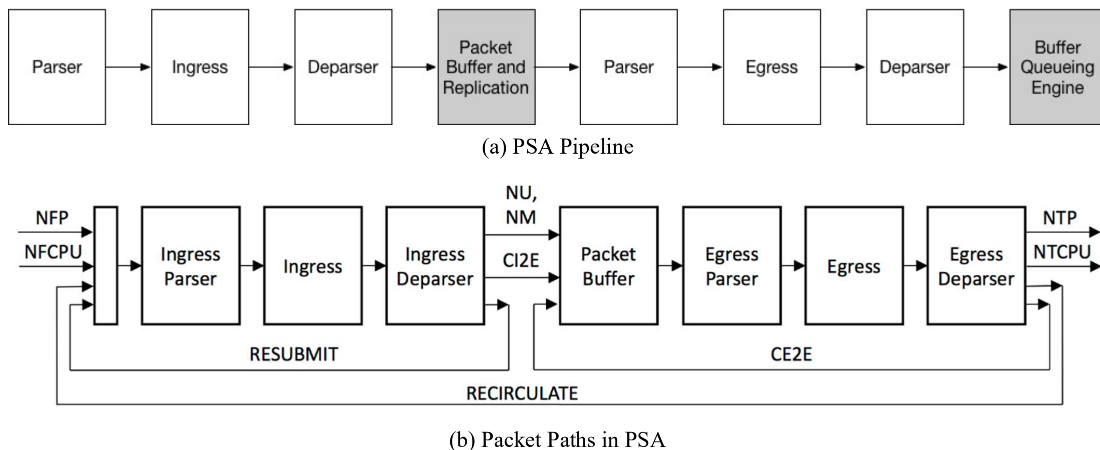

Figure 6: PSA Architecture [20]

Previous architectures were designed to be used mainly by switches. However, as we will see later, P4 can also be used to program other kinds of data planes, like smartNICs. This led to the design of another standard architecture. PNA (Figure 7) is the architecture of an abstract smartNIC, as such its design is more specific, taking into account functionalities that are usually found in NICs, like encryption/decryption. It should also be noted that it is still a work in progress, which is why planned extensions are present in Figure 7.

[9] https://forum.p4.org/t/p4-architecture/246/2 (accessed 01/07/2023)

PNA's, programmable parts are fewer than PSA, reminding the original PISA. It consists of a parser, two blocks of match-action tables (Pre control and Main Control) and a deparser. Additionally, other than the ports and the queuing systems, two main externs are defined: Net-to host and Host-to-net. These two are used mainly for functionalities that have to do with the network the host is attached to. For example, in the case of IPsec (IP Security), the authentication and decryption of the incoming packets would be the responsibility of the Net-to-host inline extern, while the signing and the encryption would be the responsibility of the Host-to-net inline extern. Because of this particularity arises the need for the Pre control programmable block, which is responsible for checking that the incoming packet is signed in a correct way and needs to be sent to the Net-to-host inline extern. In this case, the packet passes the programmable pipeline, going to the extern for authentication and decryption and then is recirculated again to the programmable pipeline. If the packet was not signed correctly, the pre control would drop it immediately, without circulating further. On the other hand, Main control, has the complete functionality the developer would want to implement on the smartNIC.

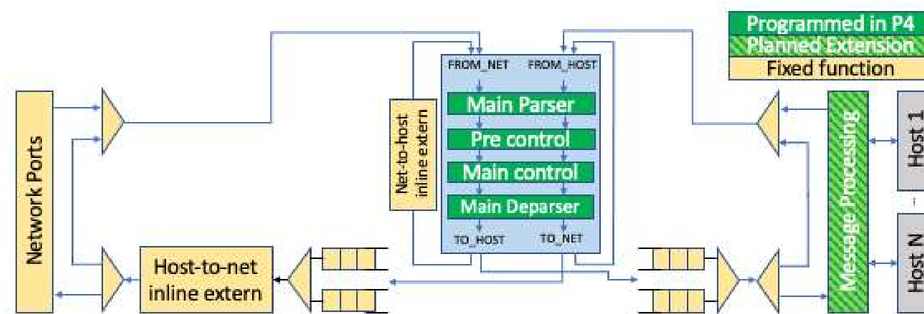

Figure 7: PNA Architecture [21]

PSA and PNA are the two standard architectures defined by the P4 consortium. However, vendors are free to conform to those two or define their own architectures that suit their needs. For example, Xilinx uses their own custom architecture, which is a modified V1Model, presented in chapter 3. Of course, the externs in different architectures could be different, which is why it is recommended to use the standardized externs in the P4 library, but the programmable parts of a P4 program are highly reusable. When writing a P4 program, the developer defines different blocks and at the end, matches those blocks to the architecture provided. This modularity in P4 programs enables greater code reuse, as a P4 program written for one architecture would need mostly rearrangements to comply to other architectures.

2.1.5.2 P4 Language

P4 is a programming language, specific to the domain of data plane programming. However, that does not mean that it is not a complete language, with its own grammar and syntax. Inspired by C, it looks like it, but it is a DSL and as such, it has specific statements, limited to the needs of network programming; for example, there is no loop (for or while) statement to be used in P4. The full formal description of the language can be found in the $P4_{16}$ language specification [19], from which the following short, informal description has been extracted.

A typical P4 program contains three main components:

1. Header definition
2. Parser logic
3. Control Blocks

2.1.5.2.1 Header

In the header definition, a programmer defines the structure of the header of the protocols that they want to use. For example, the Ethernet frame headers and IPv4 headers are shown in Snippet 1 and Snippet 2 respectively.

```
header Ethernet_h {
    bit<48>      dstAddr;
    bit<48>      srcAddr;
    bit<16>      etherType;
}
```

Snippet 1: Ethernet Header

```
header IPv4_h {
    bit<4>       version;
    bit<4>       ihl;
    bit<6>       dscp;
    bit<2>       ecn;
    bit<16>      totalLen;
    bit<16>      identification;
    bit<3>       flags;
    bit<13>      fragOffset;
    bit<8>       ttl;
    bit<8>       protocol;
    bit<16>      hdrChecksum;
    bit<32>      srcAddr;
    bit<32>      dstAddr;
}
```

Snippet 2: IPv4 Header[10]

Basically, a header definition is a traditional structure which describes all the fields of the protocol. The sequence of the fields is important, because the header is parsed sequentially and it is mapped based on the order of the definition. Also, the headers themselves, as structures, can be used in defining other structures, such as sets of packets to be parsed. In fact, it is needed to define what kind of header combinations are acceptable by the parsers. Again, the sequence is important.

[10] Note that the IPv4 headers defined here, do not have the Options field. This is omitted here for simplicity. The Options field has variable and not constant length. As such, for the definition, one would have to use the varbit keyword and parse the headers with care.

2.1.5.2.2 Parser Logic

After defining the protocols a P4 program uses, a programmer should define the parsers to be used. Parsers are basically state machines, with each state usually representing one protocol header and the transitions to other protocol headers in simple programs. In more complex programs, transitions could be used to implement loops or change between different options inside a protocol header.

In Snippet 3, a simple parser for the above Ethernet and IPv4 is shown. There should always be a start state, from which the state machine should begin. Then the header fields are extracted and stored in some header variables. It is possible to also extract only parts of the header. Based on the extracted values, transitions are defined that look similar to the switch statement in C. Like the special start state, two more special states are defined: accept and reject, which mark the parsed packet as ready to move on to the next block, or to be dropped. In case no transition is mentioned on some block, the reject transition is considered the default one. While it may seem like fields must be first fully parsed in order to be used, various functionalities are available that can be used to look ahead to some field, not yet fully parsed, etc.

```
parser MyParser(packet_in pkt, out accepted_packet hdr) {
    Checksum16() cksum;

    state start {
        pkt.extract(hdr.ethernet);
        transition select(pkt.ethernet.etherType) {
            // If the value of etherType is 0x0800
            // transition to parse_ipv4
            // else drop (implicit)
            0x0800: parse_ipv4;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ip);
        cksum.clear();
        cksum.update(hdr.ip);
        // check that checksum is 0, else raise an error
        verify(cksum.get() == 16w0, error.IPv4ChecksumError);
        transition accept;
    }
}
```

Snippet 3: Parser for Ethernet and IPv4

In Snippet 3, the keyword “out” is shown. P4 defines 3 kinds of directions that define whether a parameter can be written or not: `in`, `out`, `inout`. Read-only parameters are marked by `in` and are initialized by copying the value of the corresponding argument. `out` marks that the parameter is uninitialized and after the execution of the call, the value will be copied to the corresponding variable. Finally `inout` means that the initial value is copied from the corresponding argument and it will be updated after the execution ends. If a parameter is not marked by such a keyword, it means that it is set by the control plane, or it is known at compile-time, or it is set by another calling block, in which case it is implied as an `in` parameter.

2.1.5.2.3 Control Blocks

Control blocks are used for both Match-Action processing and deparsing. A control block has three basic parts: Actions, Tables and application. An action looks like an imperative function with no return value. It consists of some Action Code and some Action Data. The first is set by the P4 program, while the second is set by the control plane. For example, in Snippet 4, the mac addresses of the packet are changed. The new source address is taken from the old destination address and the new destination address is set by the controller. Action data is the new source address, while the rest is action code. Inside an action only statements and declarations are allowed, with no switch or if statements. What that means is that an action will just apply what is defined in it every time. All conditional checks should be done one level higher, on the level of tables.

```
action set_mac(bit<48> new_dst_addr) {
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = new_dst_addr;
}
```

Snippet 4: Change MAC address example

More important is the table part. As mentioned before, tables are the main mechanism in OpenFlow, through which the controller influences the data plane. The same structure has been kept in P4 too. Like in OpenFlow, a table should contain different entries. However, in P4, we describe the schema of the table and not the entries themselves. A table has two main properties: keys and actions.

A key is an expression that describes the parameters that are going to be used for look-up in the table. It consists of a definition of what kind of header fields can be defined and also the type of matching. P4 supports three kinds of matching: exact, ternary and lpm. Exact looks for the whole value in the key, ternary uses both a value and a mask for that value, while lpm (longest prefix match) is similar to ternary but checks only for the prefix of a value. Note that a key can be a combination of different fields. In Snippet 5, for example the key is set as the source mac address of the packet with exact matching.

The actions part of the table consists of a list of actions that can be used in this table. The action parameters are not set here, but by the controller, unless the action definition has some direction (*in*, *out*, *inout*). In that case, the action parameter should be bound when defining the table. Additionally, we can specify some default action in the table definition. All the packets that pass through the table will either hit some table entry or not. In case they miss, the default action is invoked. For example, in Snippet 5, we define that if there is no hit, the packet should be dropped.

```
table fwd {
        key = {
            hdr.ethernet.srcAddr: exact;
        }
        actions = {
            set_mac;
            my_drop;
        }
        default_action = my_drop;
    }
```

Snippet 5: Forward table example

Finally, after defining the actions and the associated tables, we can invoke tables in a control block by using the apply method, like in Snippet 6.

```
control ingress(inout Header_t h, inout Meta_t m,
                inout standard_metadata_t standard_meta) {

    action set_mac(bit<48> src_addr) {
     // See Snippet 4
    }

    table fwd {
     // See Snippet 5
    }

    apply {
        fwd.apply()
    }
```

Snippet 6: Control Block example

Do note that we can also use conditionals depending on whether the table returned a hit or miss, and which action was executed, like in Snippet 7

```
if (fwd.apply().hit) {
    // do something if there was a hit
} else {
    // do something if there was a miss
}

switch (fwd.apply().action_run) {
    set_mac: {
        // do something if set_mac was executed
    }
    my_drop: {
        // do something if my_drop was executed
    }

}
```

Snippet 7: Conditional apply

For deparsing, a control block is used like in Snippet 8. Basically, we just instruct P4 to emit the specified headers in the specified sequence we want.

```
control MyDeparser(inout accepted_packet p, packet_out pkt) {
    apply {
        pkt.emit(hdr.ethernet);
        pkt.emit(hdr.ip);
    }
}
```

Snippet 8: Deparser for Ethernet and IPv4

2.1.5.3 P4Runtime

P4 is the continuation of OpenFlow and, like its predecessor, control plane populates the tables that are defined by a P4 program. However, the protocol of how the control plane sends instructions has also changed. This happened for two reasons. First, OpenFlow works with static tables that are pre-defined and known to the control plane, by virtue of using the protocol alone. Contrary to that, P4

defines tables dynamically. Each P4 program can have totally different tables. As such, OpenFlow cannot be used as is and there needs to be a way to communicate what tables are defined by each program. Secondly, OpenFlow could use some improvements on how it works and how it instructs the switches to operate. For these reasons, P4 deemed useful to define a new protocol called P4Runtime [22] in place of OpenFlow. P4Runtime utilizes gRPC[11], which is a modern high-performance RPC framework that utilizes HTTP (Hyper Text Transfer Protocol), to enable communication between the control plane and the controllable switch. By using gRPC, P4Runtime achieves better communication times and better structure of communication than OpenFlow, while at the same time, it allows for more flexible architectures in how many controllers can control a single device. Additionally, an intermediate layer called P4Info is used to convey information about the available tables.

2.1.5.3.1 Architecture

Figure 8 depicts the reference architecture of P4Runtime's mode of operation. Other than the actual programmable switch, two more entities are of interest. First of all, in order to be able to communicate with the controllers, each switch must be equipped with a gRPC server. This server exposes a gRPC API that allows controllers to connect to it and send requests, by acting as gRPC clients. These requests vary from installing a specific program to the switch to installing flow entries to its tables.

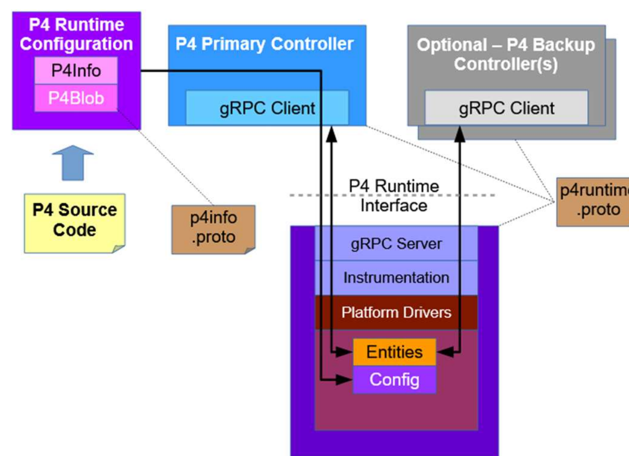

Figure 8: P4Runtime Reference Architecture [22]

P4Runtime reference architecture also takes into account controller-less architectures and architectures with more than one controller. In the first case, P4Runtime can be seen as an IPC (Inter-Process Communications) protocol, between the gRPC server and the tables of the switch. In case there are more than one controller, either two external controllers, or if we consider the gRPC server acting as an IPC controller, P4Runtime provides role-based access capabilities, so that each controller can write only to specific tables. Additionally, a high-availability setup is also supported,

[11] gRPC stands for "**g**RPC **R**emote **P**rocedural **C**all" according to the official site: https://grpc.io/docs/what-is-grpc/faq/ (accessed 02/07/2023)

with one controller and more backup controllers. P4Runtime, supports an arbitration scheme to allow only one instance every time to have write permission.

2.1.5.3.2 Compilation Procedure

As we mentioned earlier, due to the dynamic nature of P4 tables, there needs to be a way to communicate which tables are configurable and in what way. To achieve this, a representation of the tables, their keys and actions is provided through P4Info. When a P4 program is compiled, two different artifacts are produced, P4bin and P4Info (Figure 9).

P4bin is the binary programming of the switch that should be applied to the programmable data plane. As P4bin is target specific, its format is not standard and depends on the vendor-provided information. This information includes things like architecture and special externs that are not in the standard library, along with the format that is needed to program the switch(e.g. FPGA bitstream, json for SW switches, etc.).

P4Info includes the target-independent information regarding the tables that a controller can configure. In terms of gRPC, P4Info includes the protobuf files that describe what kind of communication is possible between the gRPC client and server. Note that for a given P4 program, P4bin can change depending on the target, however P4Info is always the same. As such a controller would need just one P4Info artifact if it was to control different kinds of switches running the same program. For example, for one SW switch and one HW switch, a P4 program would have to produce two P4bin files (one for each), but only one P4Info file.

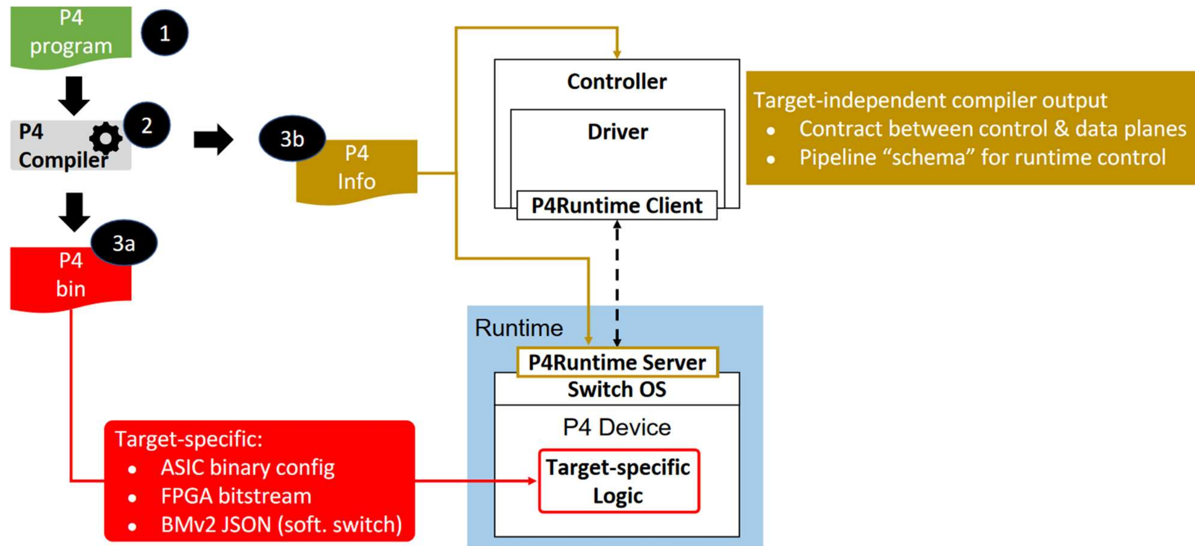

Figure 9: P4 complete compilation procedure[12]

[12] https://labs.etsi.org/rep/groups/tfs/-/wikis/uploads/addc67f7182d3e97e6742c145b7fc207/TFS_HF_2022_-_Session_3.pdf (accessed 08/07/2023)

2.2 Programmable SmartNICs

2.2.1 The Need for SmartNICs

The advancements in virtualization technology enabled the rise of the cloud. Bare-metal servers stopped being the main model of operation for most industries. Instead, servers were migrated to cloud servers. Either in the form of public cloud, private cloud, or utilizing edge and fog computing, virtualization is pervasive. A physical server can host a number of different VMs, limited by its computing and memory resources. A hypervisor takes care of the scheduling and memory management, but also the networking between the VMs themselves and the outside world.

Two problems arise from this approach. First, precious resources are committed to networking operations. Virtual switches are instantiated to allow rich networking between the VMs. In that way, VMs can be handled in the same way as a standard bare-metal machine. However virtual switches are VMs themselves and as such they consume CPU and RAM (Random Access Memory), that could otherwise be used to host "useful" VMs. Second, the network latency also takes a hit, as packets must travel through the hypervisor before disseminated to the appropriate virtual switches. These problems led to the design of smartNIC or DPU (Data Processing Units) specialized hardware, in which a server can offload networking operations. Additionally, with the advancement in technologies like SR-IOV (Single Root – Input Output Virtualization), VMs hosted on a server can share physical resources and access them directly without mediating through the hypervisor.

This essentially means that by employing and correctly configuring smartNICs in a data center, full utilization of resources can be achieved, only for "effective" VMs. At the same time, line rates can be achieved, as packets handled by smartNICs can be forwarded directly to the VM, bypassing the hypervisor. Most importantly, smartNICs are programmable by nature, which explains the "smart" in their name. As such one could offload more than just standard networking functionality. For example, VNFs (Virtualized Network Functions) can be offloaded, implementing specialized functionalities, like storage-related optimizations, or ML-powered algorithms.

2.2.2 Network Function Virtualization

The paradigm of understanding SDN controllers as similar to OS, led to another major breakthrough in network engineering, that of NFV (Network Function Virtualization). PNFs (Physical Network Functions) are the various network equipment that serve various functionalities in a network. These could include routers, CDN (Content Delivery Networks), firewalls, deep packet inspection equipment, etc. The idea of having applications that could run over a network, as first seen in the SDN architecture along with the progress in virtualization technology mentioned above led to the idea of creating VNFs. VNFs are software functions that serve the same utility as PNFs but can run on commodity off-the-self servers [23] as VMs. That can potentially reduce operation costs and allow for greater flexibility and innovation for computer networks. While VNFs can be run as VMs in a server, it is more prudent to offload these capabilities to smartNICs, especially if their functionalities have to work on line rates.

Comparing the NFV architecture (Figure 10) to the SDN architecture we presented earlier, we can find a lot of similarities. NFV Infrastructure corresponds to the data plane, NFV Management

and Orchestration corresponds to the control plane and VNFs corresponds to the application plane. Although the conceptual relationship between NFV and SDN is obvious, these two concepts are not the same and can exist independently. Actually, they are complementary in the effort to build modern networks [24].

SDN can be of use to NFV by providing an easy way to manage the network between VNFs and serving the VNF chains. On the other hand, NFV can be of use to SDN by providing a structured way to manage applications and also virtualize the controller itself [25]. However, the most important benefit comes from the combination of both SDN and NFV in a common framework [26]. In this way, the operator could treat VNFs as applications that run in the network but also expand the network elements that SDN can control and manage. By introducing virtualization, SDN expands from just switches to the data center. SDN nowadays is about controlling physical switches, as much as controlling VNFs that that serve VMs, like virtual switches. Thus SDN also extends to the management of specialized hardware that is gaining ground in the data centers, like smartNICs, where VNFs need to run to be able to match current network speeds [27].

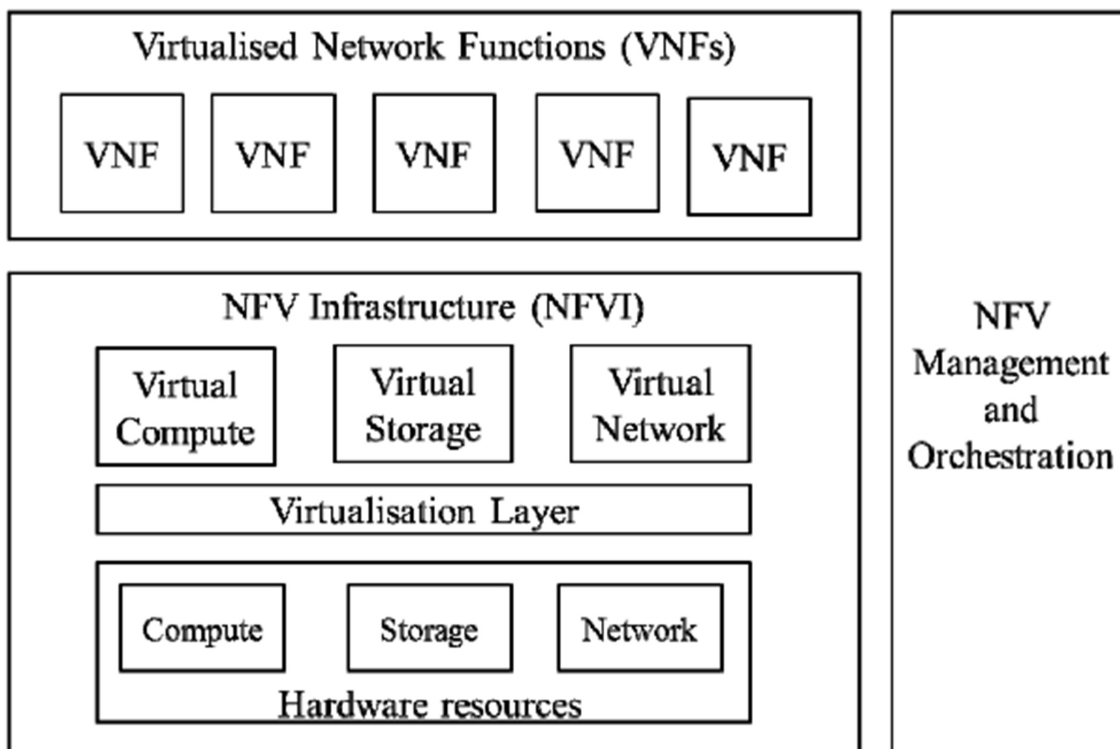

Figure 10: High-level NFV framework [28]

2.2.3 Programmability of SmartNICs

We have described above how one could use P4 as a high-level language to program a smartNIC. While P4 provides robust abstractions that allow flexibility on developing network functionalities, other lower-level techniques also exist, which have been developed in tandem. From systems programming the idea of kernel bypass arose, with DPDK (Data Plane Development Kit) at its spearhead, which allowed programs to run in userland while being able to control the hardware

elements directly, bypassing the kernel and reducing overhead. Additionally, coming from the field of hardware engineering, smartNIC programming seems to be a good use case for FPGA mapping, since pipeline architectures and P4 can be easily converted to hardware designs. This allows developers to write in high-level languages like P4, which can then be translated to FPGA bitstreams.

2.2.3.1 DPDK

DPDK[13] is an open-source software framework designed to enable high-performance packet processing in network data planes. Its core objective is to provide an optimized platform for applications that require rapid and efficient packet handling that cannot be achieved by the traditional kernel network stack. DPDK's functionality centers around the concept of kernel bypass, where applications directly interface with smartNICs without going through the kernel. This approach aims to mitigate the latency and overhead that comes with the traditional packet processing of the kernel network stack. To accomplish this, DPDK creates a user space environment in which applications communicate directly with the HW, accessing packet data from dedicated memory pools. In that way, fine-grained control over the packet processing and memory allocation can be achieved, allowing for optimized performance.

DPDK's architecture consists of several integral components that collectively contribute to its efficiency. The framework provides a suite of libraries and APIs, including the mempool and mbuf libraries, which concern the handling of memory and buffers respectively, and the PMD (Poll Mode Driver), which is responsible for configuring the devices and handling the TX and RX. PMD is critical to its operation, as it is analogous to traditional kernel drivers. Different devices come with different PMDs that enable applications to directly poll the smartNIC for packet reception and transmission, without using interrupts and context switches. Thus, packet processing efficiency is significantly improved, while concurrently latency is reduced.

Memory management is also a core aspect of DPDK's architecture. The framework implements memory optimization strategies through the use of memory pools dedicated to handling packet buffers. This design promotes efficient memory reuse, eliminating the need for repetitive memory allocation calls. Furthermore, DPDK supports multi-threaded execution enabling applications to leverage the capabilities of modern multi-core processors. This functionality is instrumental in concurrently processing packets across multiple threads, efficiently utilizing available system resources. It is worth noting that DPDK could control simpler NICs, that do not have processing power themselves. In that case, CPU and RAM resources should be allocated specifically for DPDK's application. As such, DPDK would allow for fine-grained control and reduced delays, in expense of resources, further advocating for the need of specialized hardware, like smartNICs.

2.2.3.2 FPGA

FPGAs are advanced integrated circuits that offer flexibility in hardware design and implementation. They are semiconductor devices capable of being configured and reconfigured to perform various digital logic functions. The conception and evolution of FPGAs stemmed from the

[13] https://doc.dpdk.org/guides/index.html (accessed 20/07/2023)

pursuit of customizable and adaptable digital hardware, addressing the limitations of ASICs (Application-Specific Integrated Circuits) that have fixed functionalities.

At its core, an FPGA is composed of a grid of programmable logic blocks and configurable interconnects. These logic blocks consist of LUTs (Look-Up Tables), flip-flops, and other components that can be programmed to emulate various logic functions, from basic gates to more complex circuits, like CPUs. The internal design allows these logic blocks to be interconnected and wired together in a manner that results in the desired functionality. During configuration, an HDL (Hardware Description Language) is used to define the design, which is then synthesized and compiled into a configuration bitstream. The bitstream contains the information required to program the logic blocks and interconnections, defining the final FPGA's behavior.

Recently a lot of FPGA-based smartNICs have been produced, trying to harness the programmability of reconfigurable devices. As can be derived from the architectures presented Section 2.1.5.1, smartNICs have relatively simple architectures that mainly consists of parsers and look-up tables. At the same time the flow of information follows a sequential order passing through various stages of a pipeline. This can be easily modelled in FPGAs that organize computation in a spatial manner [29]. Prime examples of this are the Xilinx Alveo FPGA-based smartNICs[14], of which the SN-1000 is also used in the current thesis. Additionally, some work has been done in the NetFPGA[15] and Open-NIC[16] projects, which provide an open-source FPGA-based NIC platform, with some user-programmable parts of the pipeline, allowing for custom-made solutions. The Open-NIC project provides both the hardware design (called OpenNIC shell) and the drivers for controlling the NIC, including DPDK drivers.

While the OpenNIC shell and the user programmable parts are in HDL, P4 can also be used as an HLS (High Level Synthesis) language [30]. HLS is the approach of using high level languages, like C or C++, to describe hardware. The high-level programs are then transpiled in some HDL before being synthesized into bitstreams. P4's specific structure seems to be easily translated to hardware designs. This allows network engineers to develop data plane programs in P4 which can then be used to program an FPGA, through the use of some source-to-source compiler. Efforts in this direction have been made that can take P4 programs and produce hardware description[31], [32]. Again, Xilinx seems to push towards this direction also with its own SDNet suite [33], which was later renamed to Vitis Networking P4[17].

2.3 State of the Art

The objective of this master thesis is to compare the performance of two different smartNICs with respect to different software switches. In this subsection, we provide a brief survey of similar work

[14] https://www.xilinx.com/products/boards-and-kits/alveo.html (accessed 20/07/2023)

[15] https://netfpga.org/ (accessed 20/07/2023)

[16] https://github.com/Xilinx/open-nic (accessed 20/07/2023)

[17] https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html (accessed 20/07/2023)

that has been undertaken by other engineers and scientists and published in relevant scientific journals.

In [34], Zhang et al. compare seven different software switches in four different setups: a physical to physical (similar to our case), a physical to virtual involving a VNF, a virtual to virtual, and a "loopback" scenario involving both two physical NICs and a VNF. Their methodology is thorough, including many different scenarios and cases. Their final results show that for their setups, no software switch prevailed in all cases, highlighting the need to choose the right setup depending on the requirements of each case.

Similarly in [35], Katsikas et al. stress test four different smartNICs and measure their performance in different categories. They test how the number of tables and installed rules impact the performance of each smartNIC and show that updates to the smartNIC classifier could severely impact the operation of the card. Based on the findings from the above test, they show how one could use DPDK to update rules by using in-memory update, optimizing the rule updates in the cards by 80%.

The discrepancy in the advancements of CPUs and smartNICs and the increasing benefits of correctly using smartNICs is shown in [36]. In that work Ghasemirahni et al. study the way CPUs handle the incoming packets and, more importantly, how they fail to handle packets from different flows that arrive scrambled in various ways. Their work involves the development of smartNIC functionalities that deliberately delay the handling of packets by the CPU, in order to allow for batch processing of similar flows. By coordinating the congestion control mechanisms and cache-based optimization they are able to triple a server's performance.

Building on the above, some of the authors of the previous work continued their research on VNF chaining in [37] and [27]. The Metron framework was presented, which is an NFV platform, able to manage and orchestrate VNFs. Moreover, Metron is able to correctly tag packet flows and deliver them to NFV chains in the most efficient order, effectively offloading the packet processing and CPU core dispatching to smartNICs. Their experiments include different setups, involving two servers connected back to back, with different VNFs serving different functions. Ultimately, they show that Metron can achieve speeds near the underlying hardware in an efficient manner.

While they do not test a specific hardware or software switch, of great interest is the work in [38] by Kundel et al., in which a novel approach for network testing and monitoring using programmable data planes is presented. They introduce the P4STA open-source framework that leverages the programmability of modern network switches, to perform highly accurate and purpose-independent testing of network elements. By interposing a P4 programmable data plane between the DUT (Device Under Test) and the load generators, they can stamp packets according to their needs, to monitor and check their behavior in the DUT in line rates. In their experiments they showed that the added latency is around 2 μs, which can be considered negligible.

The above can be considered as a form of in-network computing, which is a really interesting and disruptive innovation. Similar work can be found in [39] and [40], where the authors propose different extensions to the P4 language and the available architectures that allow for computation to happen inside the switches in line rates. In both publications robust models on the feasibility of such offloading are presented, but testing is limited.

Our work is more similar to the two publications presented first [34], [35]. While smaller in scope, we compare both software switches and different smartNICs. In this endeavor, we can observe how software switches behavior changes when running with different smartNICs, with the added value of examining P4-enabled software switches too. Additionally, we aspired to check how an FPGA based switch would compare to a software switch with kernel bypass and highlight the differences between them, in terms of development process and effort. Unfortunately, that last step came to no avail, due to problems in using the FPGA-enabled smartNIC we had in our possession at high data rates, as will be shown later in Chapter 4.

3

Experiments

3.1 Common Setup

Five experiments were conducted, each testing a different programmable switch that run on top of a smartNIC:

- bmv2
- OVS-vanilla
- OVS-DPDK
- P4-DPDK
- SN1000 fpga

Two different smartNICs were used: a Xilinx Alveo SN1000[18] and an NVIDIA ConnectX 5[19], both of which have a nominal link speed of 100 Gbps. All experiments, except the final one, run on both smartNICs. The final one run only in SN1000, as the CX5 does not have a similar functionality. As we will see later in this chapter, bmv2 (behavioral model version 2) serves just as a functional test, as its measurements are not comparable with the other cases. The purpose of these experiments is to compare the behavior of the different switches but also to compare the performance of the two cards. Before moving on to the description of the various components involved in each experiment, we first describe the common tools and setups that were used.

3.1.1 Servers

The experiments that are described below, ran in the same setup. Two identical servers, named *goku* and *vegeta*, were used. Their main characteristics are shown in Table 1. In terms of network hardware both servers are equipped with two smartNICs: a Xilinx Alveo SN1000 and an NVIDIA ConnectX 5. However, for the experiments, *goku* uses only the CX5.

[18] referred to SN1000 subsequently

[19] referred to CX5 subsequently

Table 1: Server characteristic

CPU					RAM			Storage	
Model	Sockets	Cores	Threads	Frequency	Size	Type	Frequency	Type	Size
AMD EPYC 7763	1	64	128	2.45 GHz	256 GB	DDR4	3.2 GHz	NVMe	1.92 TB

The setup of the servers is shown inFigure 11. The two servers are connected back-to-back. The CX5 of *goku* has two ports, p1 and p2. The first one is connected to p1 of sn100 in *vegeta* and the second one is connected to p1 of CX5 in *vegeta*. *Goku* takes the role of the client, which sends packets (Tx) to *vegeta* and receives them back (Rx). *Vegeta* is considered the DUT and implements the forwarding switch (FWD). Based on the received packets, *goku* can measure or compute specific metrics that we can use to evaluate the behavior of the implemented switch.

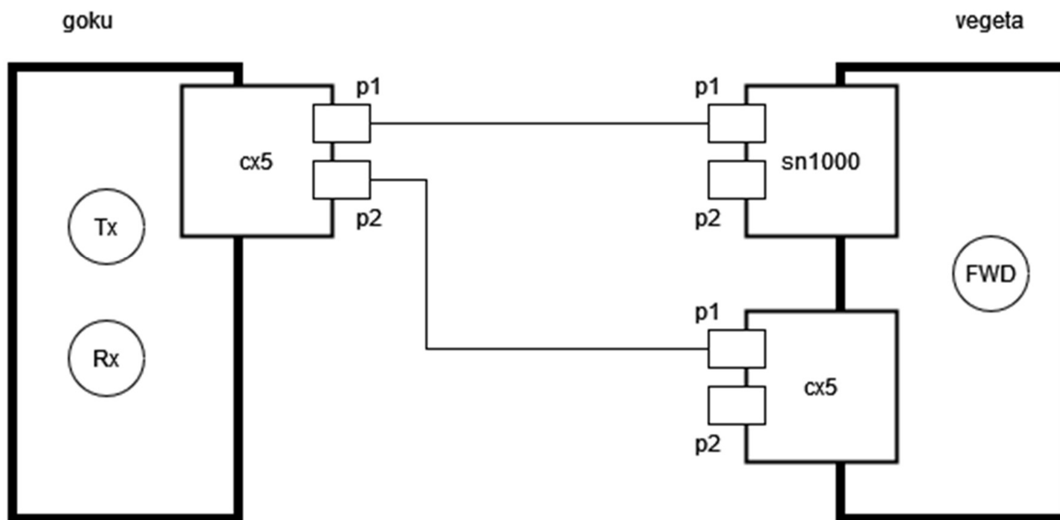

Figure 11: Server setup

3.1.2 Metrics

The metrics that were measured or calculated are the same for each experiment and are shown in Table 2. The final results are shown in the respective diagrams of Chapter 4. Not all metrics were eventually used, either because of the similarities with other metrics (for example Byte Rate and Packet Rate), or due to their more supportive nature. For example, the Dropped packets due to bad MAC Address metric was used as an extra check before actually running the experiment, to make sure that correct programming and configuration was loaded to the switch.

All experiments were run 6 different times in total and the results that are shown in the tables of Chapter 4 are the average for each metric. An exception is latency, where the results were gathered as characteristics of its distribution, as will be shown in the respective diagrams.

Table 2: Metrics measured or calculated in the experiments

Metric	Measuring Point	Description
Bit Rate	*goku* (Tx), *goku* (Rx)	The bit rate in Tx and Rx in Giga bits per second (Gbps)
Packet Rate	*goku* (Tx), *goku* (Rx)	The packet rate in Tx and Rx in millions packets per second (Mpps)
Count	*goku* (Tx), *goku* (Rx)	The number of packets in Tx and in Rx
Packet Loss	*goku* (Tx / Rx)	The packet loss as a percentage of the Rx/Tx packets measured by the receiver.
Dropped packets due to bad MAC Address	*goku* (Rx)	The number of packets received that had bad MAC address
Test time	*goku*	The time it took for the test to conclude in seconds
Latency	*goku* (Tx / Rx)	The round-trip time of each packet (from the Tx to the FWD server and back to the Rx). It is measured in microseconds by the receiver. Characteristics of the distribution are captured (average, standard deviation, percentiles, minimum and maximum latency)

3.1.3 Network Performance Framework

In order to orchestrate and automate the experiments we are using the Network Performance Framework (NPF)[20]. NPF allows for the definition of experiments in terms of bash scripts that should be run in specific machines. The machines involved in the experiments are defined as a cluster, with each machine containing specific parameters, including its network interfaces. NPF is responsible for connecting to the involved machines and running the appropriate bash scripts according to some flow control that dictates the order. To control the flow two mechanisms are used: events, which are raised by each script informing other scripts that they can be initiated and tags that enable or disable specific scripts.

[20] https://github.com/tbarbette/npf (accessed 27/07/2023)

All these are outlined in a testie file[21] which describes the used variables, the bash scripts (either inline or referenced as files), along with their corresponding tags and finally the metrics that should be collected. While NPF does not collect measurements in itself, it can listen for events in the stdio (standard input-output), where scripts print measurements. When such an event is triggered, NPF will store the specified measurement under the appropriate category. Additionally, NPF can automatically produce simple diagrams of the collected measurements. However, the diagrams presented in Chapter 4 were not produced in this way, as they are a little more complex and needed customization to be able to show different results next to each other. Finally, NPF is also practical in its ability to run many instances of one experiment and collecting all the measurements in csv files.

In our case, NPF is very useful because we need a way to coordinate the execution of the transmitter and receiver in *goku* and correctly configure the forwarding switch in *vegeta*. Furthermore, we need a simple way to run consecutive experiments with different parameters, e.g., packet size, different switches, etc. For that reason, a testie file was created that contains the various configurations of our experiments.

A flow diagram of our testie file is presented in Figure 12. We use the following notation: a circle at the end of an arrow symbolizes that an event is emitted, while an open semicircle denotes that the arrow needs the event that it comes from to trigger. On the right side, the flow of the client, in our case *goku*, is presented. A trace generation script runs at first, which makes sure that the traces are present and if not, it creates them. At the end it emits a “TRACE_FINISHED” event. The flow of the DUT, in our case *vegeta* is presented on the left side. Together with the trace generation the appropriate init script is executed, based on the input. The four switches have similar paths, with distinct scripts for each one of them. The only difference is in OVS, which require the extra step of setup, depending on whether we use DPDK or not. After the execution of initialization script, a “BR_READY” event is emitted, which triggers the installation of the forwarding rules in each data plane. When this is done, a “DUT_READY” event is emitted. It is important to note that the emission of the events happen in the NPF space, which means that they are not constraint to the machine they were emitted. We use that feature for coordination between *goku* and *vegeta*, so the “DUT_READY” and the “TRACE_FINISHED” event together trigger the initiation of the traffic generator on *goku*. A “GEN_FINISHED” event means that the generation has finished and signals the DUT to cleanup. Finally, once everything is cleaned up, an “EXIT” event is emitted, which signals both machines to stop execution and NPF to stop.

[21] .testie is the name of the extension of the files that are used to configure NPF

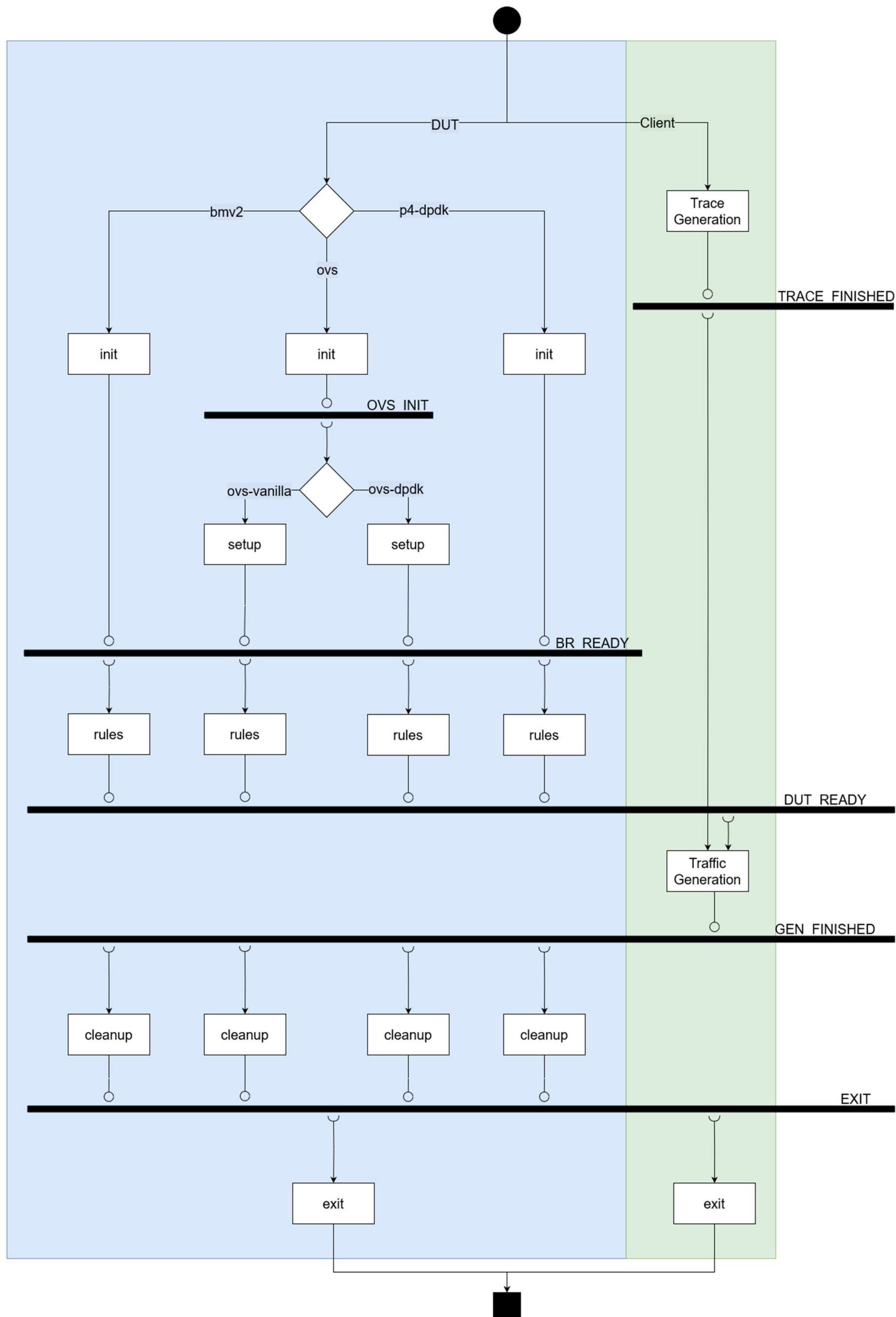

Figure 12: NPF testie flow

3.1.4 Click Modular Router

The various switches tested are described below in their own sections. However, it is necessary to also mention the Click modular router [15], which was used in *goku*, as the transmitter and receiver and also for the creation of the packets that served as test cases. Click is a pioneering software framework designed to streamline the complexities of network packet processing. It deconstructs network tasks into discrete building blocks known as elements. These modular elements can be interconnected to effectively create and program highly customizable network functions. In our case we used it as an easy way to first create network traffic and then replay it and capture the response.

3.1.5 Test Cases

For testing the software switches, we created pcap (Packet Capture) files which correspond to specific traffic. We tried both TCP and UDP flows, however in the end there weren't many differences in terms of performance between the two and opted to use only TCP for our final measurements. The final traces used, included 10000 clients, with 1 flow per client, each comprising of 128 packets. In total each trace consists of 1280000 packets. Additionally, different packet sizes were used. Specifically, we considered the minimum and maximum sizes for a regular ethernet packet and all the powers of two in between:

- 64 bytes (minimum size)
- 128 bytes
- 256 bytes
- 512 bytes
- 1024 bytes
- 1500 bytes (maximum sizes)

3.2 Open Virtual Switch (OVS)

3.2.1 OVS and OVS-DPDK

OVS[22] constitutes a pivotal component within contemporary networking paradigms, particularly in the context of network virtualization and SDN. As an open-source multilayer virtual switch, OVS enables the operator to manage network tasks better and more effectively, enhancing networks with enhanced flexibility, scalability, and manageability. OVS offers a range of functionalities of network orchestration and administration. Central among these is the ability to instantiate and manage virtual switches, enabling the configuration of intricate network architectures within virtualized domains. At the same time, its operation is similar to HW switches, thus enabling the seamless management of hybrid network, spanning across the physical and the virtual domain. In parallel, OVS is OpenFlow enabled, which means that it encompasses the SDN architecture with centralized control and dynamic adaptation of network flows. Its open-source nature allowed it to

[22] https://www.openvswitch.org/ (accessed 27/07/2023)

flourish to a production grade software switch, which is widely used in important virtualization projects like OpenStack[23].

OVS is comprised of many different components. The two major ones are the ovs-vswitchd, a userspace daemon which acts as the interface of the OVS to the user and the controller and the kernel datapath, which is the interface of the OVS to the kernel and the devices it handles (Figure 13).

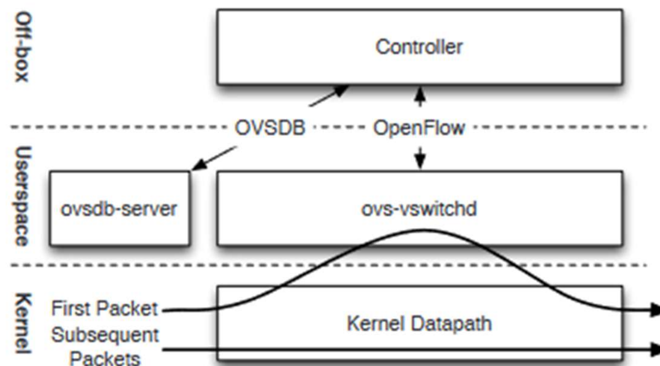

Figure 13: OVS components and interfaces[41]

Recently OVS-DPDK was developed, which allowed OVS to leverage the kernel-bypass features of DPDK. This means that the OVS datapath no longer runs in the kernel, but in the userland instead (Figure 14), allowing for higher programmability and reduced overhead times. Additionally, there were efforts to create modules[24,25] for OVS that would enable P4 programmability [42], [43]. However these modules neither seem to be stable nor are they actively developed.

In our experiments we used both OVS-vanilla, which exploits the kernel driver and OVS-DPDK. While we knew beforehand that OVS-DPDK would surely surpass OVS-vanilla in performance, we also included OVS-vanilla as both a base case and also as a means to further compare the two cards: CX5 and SN1000.

[23] https://www.openstack.org/ (accessed 27/07/2023)

[24] https://github.com/osinstom/P4-OvS (accessed 27/07/2023)

[25] https://github.com/Orange-OpenSource/p4rt-ovs (accessed 27/07/2023)

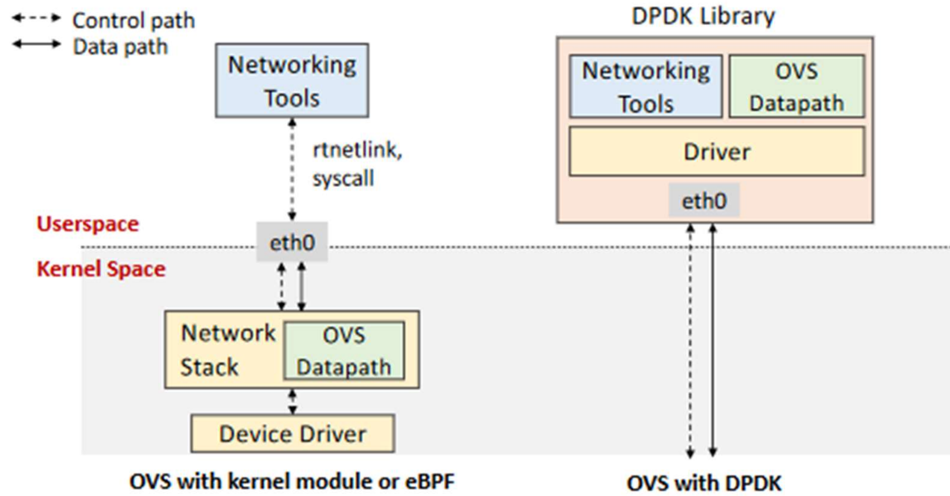

Figure 14: OVS and OVS-DPDK architecture [44]

3.2.2 Experimental Setup

In the experiments, we used the rule shown in Snippet 9 for both switches, which swaps the source and destination MAC addresses and sends the packet to the port it came from.

```
ovs-ofctl add-flow ${OVS_BR_NAME} "table0, priority=0, in_port=${DUT_PORT},
eth_src=${SRC_MAC}, actions=mod_dl_src:${DST_MAC}, mod_dl_dst:${SRC_MAC},
output:in_port"
```

Snippet 9: OpenFlow rule used for OVS

Additionally, for OVS-DPDK, we used the DPDK parameters shown inTable 3, which were the equivalent of the parameters that were also set for P4-DPDK, as will be shown later in Table 4.

Table 3: OVS-DPDK parameters

Parameter	Value	Description
dpdk-lcore-mask	0x1	Specifies the CPU cores which DPDK lcore threads will use
n_rxq	16	Number of rx queues
dpdk-socket-mem	4096	Specifies the memory to pre-allocate from hugepages[26] on specific sockets

[26] Hugepages are contiguous blocks of memory that help improve system performance by reducing memory fragmentation and improving memory access efficiency for high-speed data processing.

3.3 P4

3.3.1 Bmv2

As described before, in 2.1.5, P4 is a DSL that enables us to program specific network behaviors. Bmv2[27] is the second version of the P4 reference software switch that implements the whole specification of $P4_{16}$. Bmv2 is mainly used for development purposes and it is by no means a production grade switch. It does not support any kernel-bypass features or other similar technologies. In that sense, we include it in our experiments more as a proof of concept and check for the P4 program, rather than something that we can use in comparison with other switches.

3.3.2 P4-DPDK

To use DPDK with P4 there are a couple of solutions. P4-DPDK[28] is the one that we test in this thesis. P4-DPDK runs a switch written in C using the DPDK library and exposes a P4Runtime interface through which a controller can send a P4 program, set rules, etc. There is also T4P4S[29], which is not exactly a switch, but rather a compiler that takes P4 code and produces C code, using the DPDK C library. T4P4S uses DPDK version 18.08, while P4-DPDK uses version 22.03. Due to the constraints that were imposed by the SN1000 driver and the specific DPDK versions that were needed, we weren't able to use T4P4S to compare it to P4-DPDK. However, this is something that is in our future plans.

Unfortunately, P4-DPDK is currently in alpha version (0.1.0) and, as such, a lot of functionalities are missing or not easily configured. More importantly, there has been no documentation for in many areas, which leads to a program that is arcane in many areas and needs manual testing in order to configure correctly. Even the simplest functionalities, like correctly configuring the switch to include a specific physical port, are not fully documented. It should be noted that in our endeavor to set up P4-DPDK, a lot of help came from sources outside the project[30]. As such P4-DPDK cannot be considered production grade and it has much lower performance than OVS-DPDK, as we will see in Chapter 4.

3.3.3 Experimental Setup

The P4 program that we used for both bmv2 and P4-DPDK is a simple one that switches the source and destination MAC addresses of packets coming in a port and sends them back through the same port. The source code can be found in "

[27] https://github.com/p4lang/behavioral-model (accessed 27/07/2023)

[28] https://github.com/p4lang/p4-dpdk-target (accessed 27/07/2023)

[29] https://github.com/P4ELTE/t4p4s (accessed 27/07/2023)

[30] https://github.com/Yi-Tseng/p4-dpdk-target-notes (accessed 27/07/2023)

Annex A: P4 program used". To set the control plane rules, the P4Runtime shell[31] was used, which is based in IPython[32] and allows the user to send control messages through the use of python commands.

Regarding bmv2, there are no options that one could use to modify any parameters regarding the smartNIC that it attaches to, as it employs the regular kernel driver of the card. In the case of P4-DPDK, the configuration is mainly done through the EAL (Environment Abstraction Layer) parameters of DPDK, when binding the card. The ones that were used are shown in Table 4 and are equivalent t the parameters OVS-DPDK parameters described above.

Table 4: P4-DPDK EAL parameters

Parameter	Value	Description
-l	0-1	List of cores to run on
-n	16	Set the number of memory channels to use
-m	4096	Amount of memory to pre-allocate at startup.

3.4 Xilinx SN1000 FPGA

Xilinx SN1000 comes with an AMD XCU26 FPGA, which allows for developing specialized hardware designs to be used in a pipelined manner inside the smartNIC. In this section we describe in more detail the basic information needed to understand how we can use P4 to program the FPGA of SN1000.

3.4.1 Vitis Networking P4

As mentioned above, Xilinx has developed an extension to the Vitis suite that allows to use P4 as an HLS, supporting a variety of target FPGA devices. The procedure is simple and revolves around a custom IP (Intellectual Property) provided by Xilinx. This IP is customizable; you can give it a complete P4 program and it will compile it and create the appropriate RTL (Register-Transfer Level) design that corresponds to the functionality defined by the P4 program. It will also create the appropriate tables that are needed based on the kind of tables defined.

Furthermore, a behavioral model is provided to validate the functionality of the produced RTL design during simulation. Figure 15 shows the corresponding testing process in which that works. The developer creates some input packet data files. These inputs are given to both the behavioral model and the simulated RTL, along with any relevant commands, like installed table rules. In this way, the developer can check if the produced RTL behaves in the same way as the behavioral model, which is considered the proof of concept.

[31] https://github.com/p4lang/p4runtime-shell (accessed 27/07/2023)

[32] https://ipython.org/ (accessed 27/07/2023)

Essentially Vitis Networking P4 is a tool that uses P4 code as HLS and produces kernels. Kernels refer to the computational units or algorithms that are implemented on hardware for acceleration purposes and represent a specific computational task or function to be executed on the FPGA. In the case of P4, this would translate to a kernel, which basically implements a pipeline that takes packets as input and transforms them, according to specific rules, at its output.

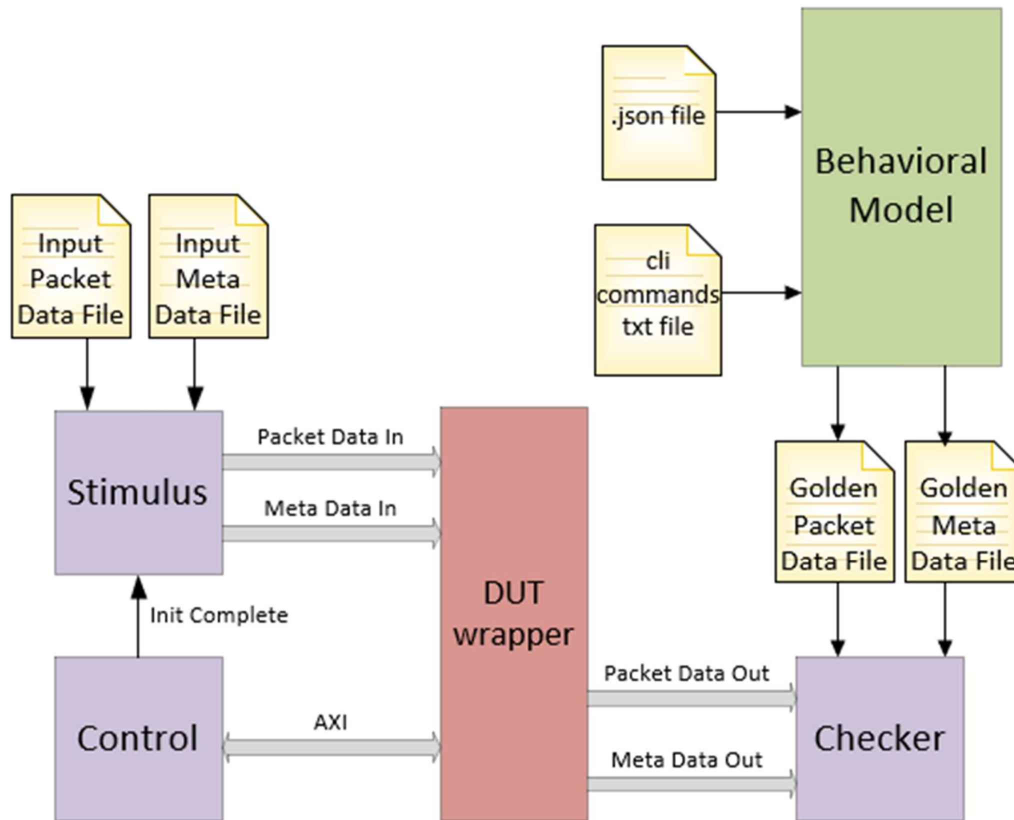

Figure 15: Testing flow of a P4 RTL[33]

3.4.2 SN1000 Plugins

SN1000 is not just an FPGA board, but is a complete smartNIC, with all the functionality one would expect from a NIC. Its architecture is similar to the OpenNIC project's architecture (Figure 16), in the sense that there are specific parts of it that are user programmable, while most of the functionalities are in system blocks. This means that SN1000 uses plugins to configure its programmable blocks, which are kernels wrapped around a specific interface that is able to communicate in some way with the external blocks.

These plugins are written in some HDL or HLS, including P4 and are synthesized and packaged in an .xclbin file, which contains the bitstream with the actual FPGA programming of the various LUTs and their connections. Additionally, it contains specific metadata and runtime information

[33] https://docs.xilinx.com/r/en-US/ug1308-vitis-p4-user-guide (accessed 03/06/2023)

that is used by the host software to interact with the FPGA at runtime like memory allocation details, kernel invocation information, and other runtime-specific settings.

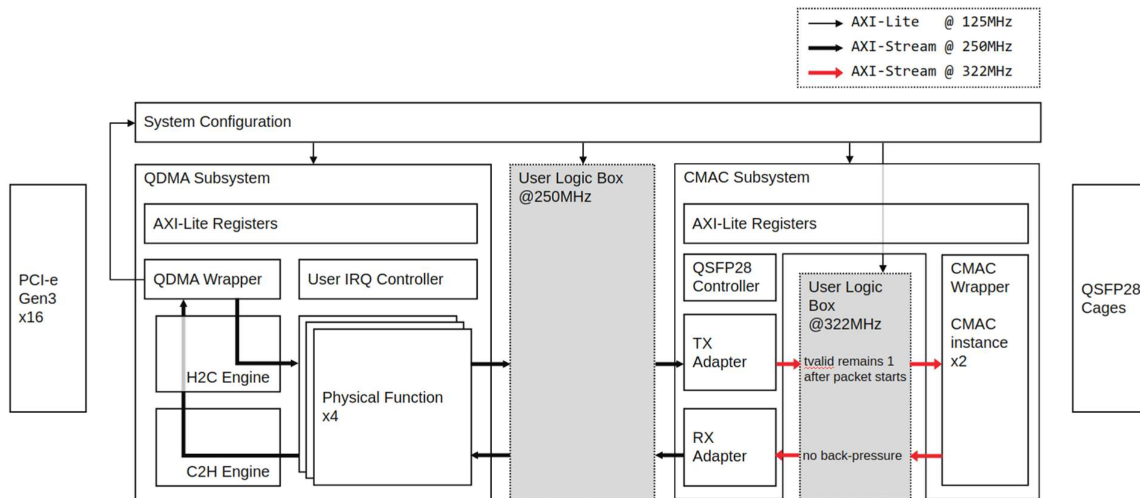

Figure 16: OpenNIC Architecture[34]

3.4.3 P4 Programs

However, in the case of SN1000, the P4 program itself should act in some specific metadata, named capsule headers. These headers are inserted to the beginning of the packet when received and contain internal metadata that are used by the NIC to route and manipulate the packet. Very important are the metadata that route each packet to the programmable regions in the NIC, to the Tx or to the CPU. A valid P4 program should be able to parse the above capsule headers and manipulate them in a meaningful way. These capsules are 224 bits in size and of different format depending on where they came from. As such, when manipulating them, each field must be handled with care.

Furthermore, there are some more changes that have to be made to the P4 programs used in SN1000. All in all, the resulting P4 programs are mostly valid according to the standard $P4_{16}$, but there are some changes, which lead to some idiomatic way of doing things.

First of all, only direct counters can be defined, in an implicit manner by defining a table. Every definition of a table in a P4 program, leads also to the definition of a matching counter. This means that to create a counter, a dummy table must be defined that when it matches, takes no action. Additionally, the tables defined seem to be capable of having only one action. No multi-action tables are allowed. So, in the control plane instead of defining a matching field, a table and an action, you just define a matching field and a table. Such a rule, implies that the only action defined in the table should be executed.

Finally, the architecture is simple, similar to the V1Model (Figure 5). It consists of a parser, a processing block and a deparser. However, the produced RTL design seems to have double the tables defined, one set for Rx and one set for Tx. This indicates that, in the end, the pipeline is replicated and we end up with an architecture that looks more like PSA (Figure 6) than the V1Model.

[34] https://github.com/Xilinx/open-nic (accessed 03/06/2023)

3.4.4 Experimental Setup

In order to prepare the SN1000 FPGA experiment, we began with the same P4 program that was used for the rest of the experiments. First, we passed it to Vitis Networking P4 and created an RTL design from it. On that design we run some simulated tests using the behavioral model described above. To do that we prepared specific test cases, comprising of input packets that resembled the packets we will eventually use in the final tests. These input packets were first passed through the behavioral model to acquire the golden packet data file, which indicated how the P4 program should behave. Using the golden packet data file, we simulated the hardware design and made sure that the output was the same. In that way we made sure the P4 program we used, implemented the functionality we wanted to. The simulations were run following the same standard procedures used for any case in the Vivado tool.

Following that, we had to prepare the plugin specifically for SN1000, based on the existing examples that were given by Xilinx. These examples comprised of some *make* scripts that took as input a program in some HLS language (in our case P4) and produced a plugin encoded in *.xclbin* format. Studying the examples that used P4, we found out that there were extra sections specifically for the capsule headers. As such, we also had to enrich our original P4 program with this functionality. Unfortunately, example packets that had the full capsule headers were not provided and creating a full set of them ourselves was not that easy.

By studying SN1000's documentation, we were able to grasp the modelling of the capsule metadata protocol and developed functions inside our P4 program to correctly parse and manipulate the capsule headers. Additionally, a small number of sample packets were created that also included a capsule header, which were used in new simulations to test the P4 program functionality. After making sure that the capsule headers were manipulated correctly, based on the documentation, we were ready to create our plugin.

The final program is shown in "Annex B: P4 program used for Xilinx SN1000's FPGA". The most important difference with the program used in the previous examples, is the correct parsing and modification of the capsule header fields that dictate the path the packet will take. What we want is for the packet to go directly to the Tx, without having to go through the CPU first. However, to do that we must also change other fields, as the capsule carries different metadata, depending on the route it is sent. What we eventually want to achieve is to offload the whole P4 pipeline to the smartNIC hardware, without involving the server's CPU at all. It's worth mentioning that the examples given by Xilinx did not include a similar plugin. All the examples implemented some new functionality, i.e., packet counters, timestamping (RFC 781), etc. but, in the end, the packets would be handled by the CPU. Thus, what we wanted to achieve, a real hardware switch, was not provided as an example.

After the synthesis was completed, we had the final plugin in the form of a *.xclbin* file. Using the tools provided by Xilinx (*xnpradmin*), we were able to upload the plugin to the smartNIC and enable it. After testing with simple packets to check the functionality, we moved on to tests utilizing the framework we used for the rest of the experiments.

4

Results

4.1 Bmv2

As discussed above, bmv2 is used mostly as a proof of concept, rather than an actual switch under test. This can be seen in the following diagrams of packet loss (Figure 17) and throughput (Figure 18), on the y-axis of which logarithmic scale is used. As seen in Table 5, Packet loss ratio is in all cases over 99.5%, with slight deviations, but with CX5 always bringing a better score than SN1000. In the same motive, throughput is too low, under 0.5 Gbps, again with CX5 scoring better than SN1000 in all cases.

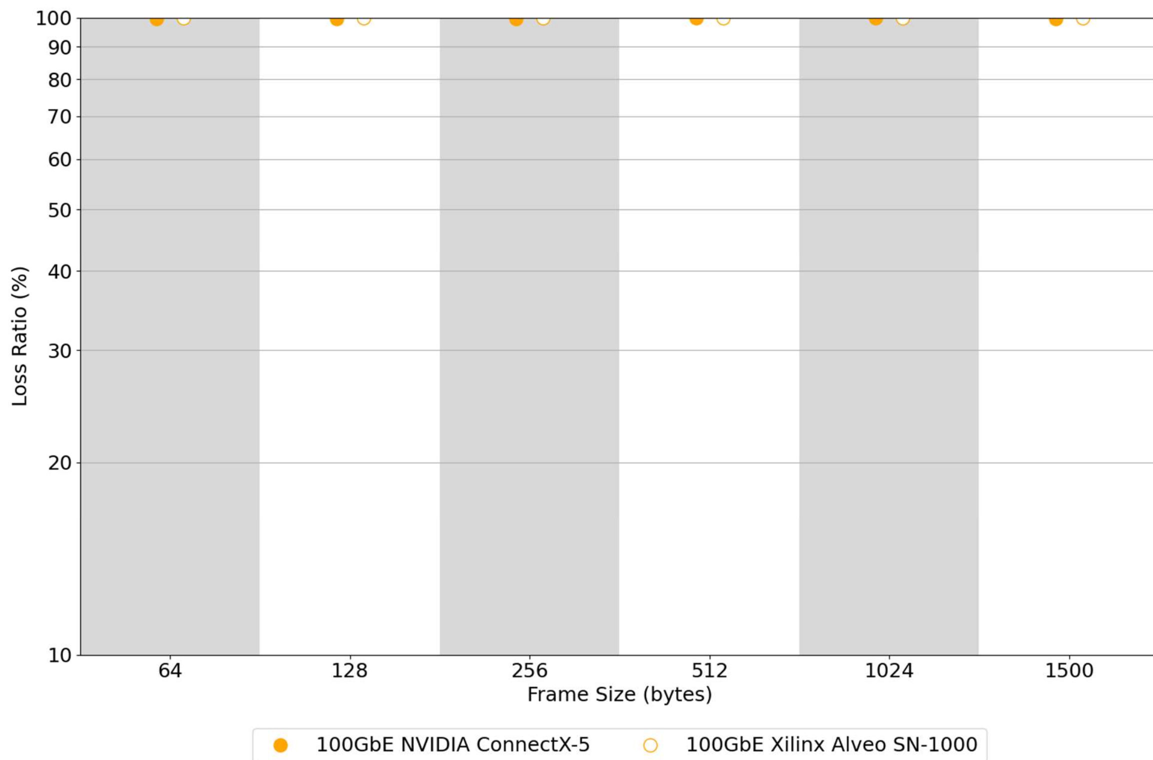

Figure 17: Loss Ratio graph for bmv2 over different frame sizes

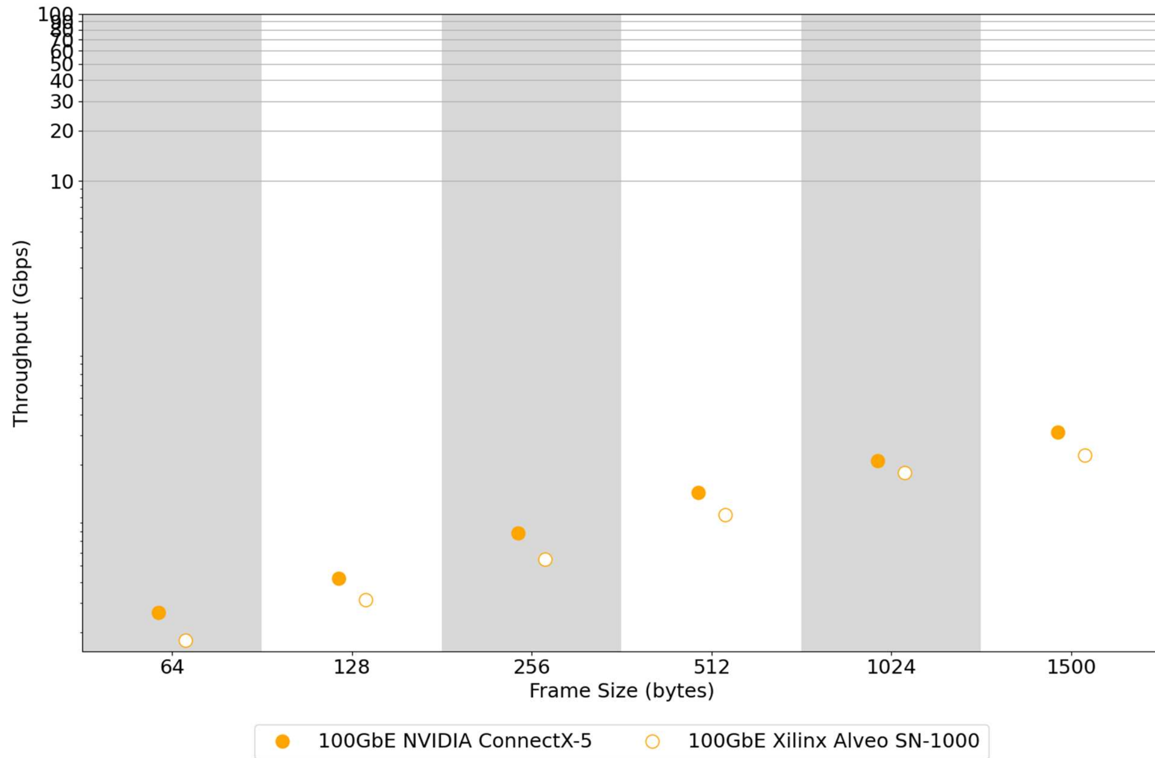

Figure 18: Throughput graph for bmv2 over different frame sizes

Table 5: bmv2 measurements

Frame Size / Metric	64		128		256		512		1024		1500	
	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000
Loss Ratio (%)	99.66	99.76	99.70	99.76	99.69	99.77	99.72	99.78	99.75	99.77	99.65	99.73
Throughput (Gbps)	0.03	0.02	0.04	0.03	0.08	0.05	0.14	0.10	0.21	0.18	0.32	0.23

Figure 19 shows the latency measurements for bmv2, for both cards, over different frame sizes. Again, the performance is not great, but the medians of both cards are more or less the same. The big difference between the cards is the deviation, with CX5 showing a more robust behavior than SN1000.

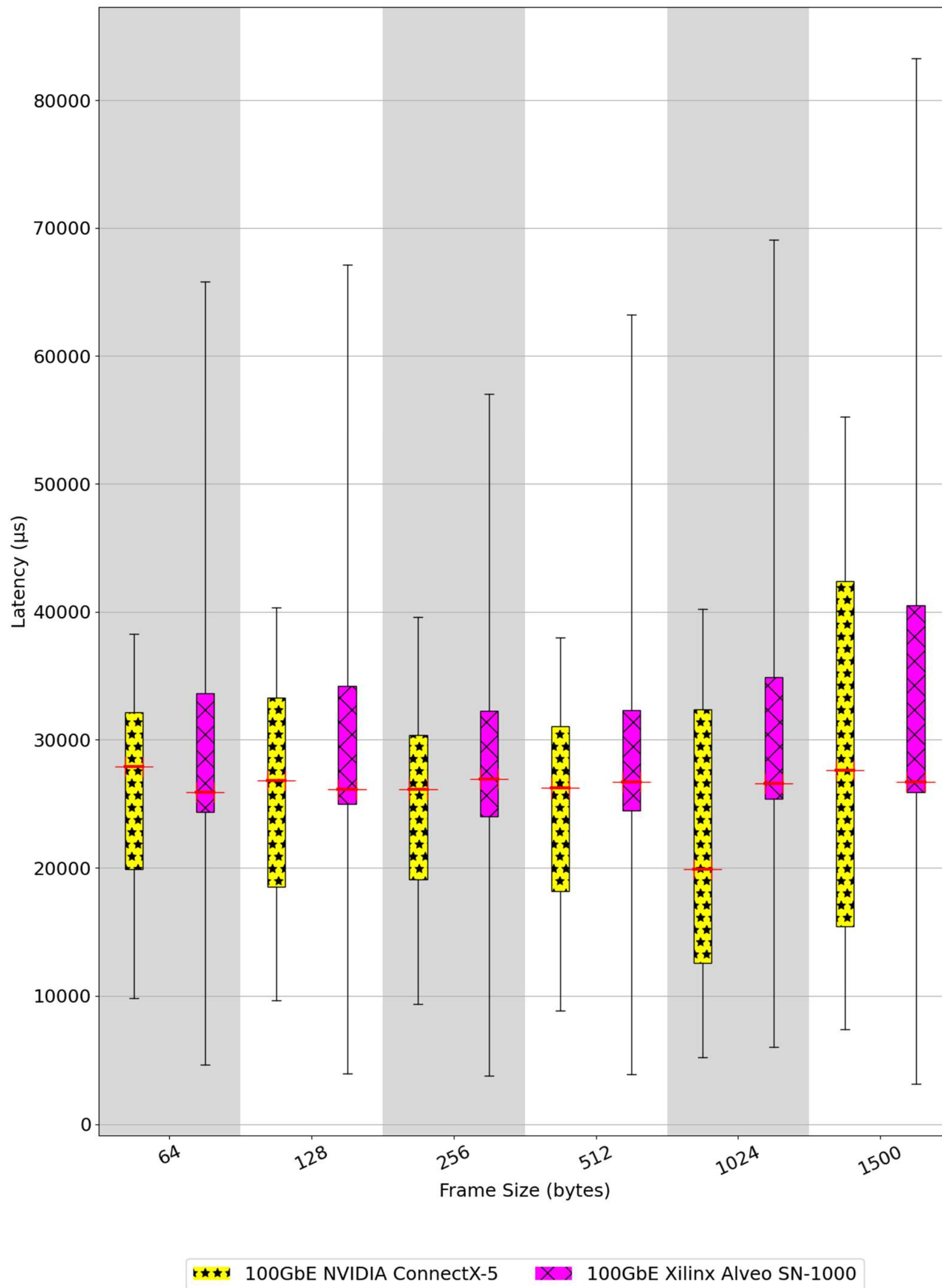

Figure 19: Latency graph for bmv2 over different frame sizes

4.2 OVS-vanilla

Ovs-vanilla is the first switch that the measurements are meaningful. The packet loss and throughput are shown in Figure 20 and Figure 21 respectively in logarithmic scale. While the performance is still low, especially if we compare it to the nominal speed of 100 Gbps, the attained rates are of

interest, especially in larger frame sizes, taking into account that OVS-vanilla uses the kernel driver for each card. What is more interesting is the discrepancy between the two cards (Table 6). Cx5 attains rates near the nominal one (~ 80Gbps for 1500 bytes frame size) and surpasses SN1000's best case (~18Gbps for 1500 bytes frame size) even with much lower frame sizes (~23Gbps for 256 bytes frame size). It is clear that NVIDIA took into account OVS while developing the drivers for CX5, which is sensible, as OVS is the de facto virtual switch used in many data centers. On the contrary, Xilinx didn't seem to consider any optimizations regarding OVS and the results for SN1000 are disappointing. The above can be seen more clearly on the loss ratio, where SN1000 never drops below the 80% mark, while CX5 worst result is at ~19%, as can be seen in Table 6.

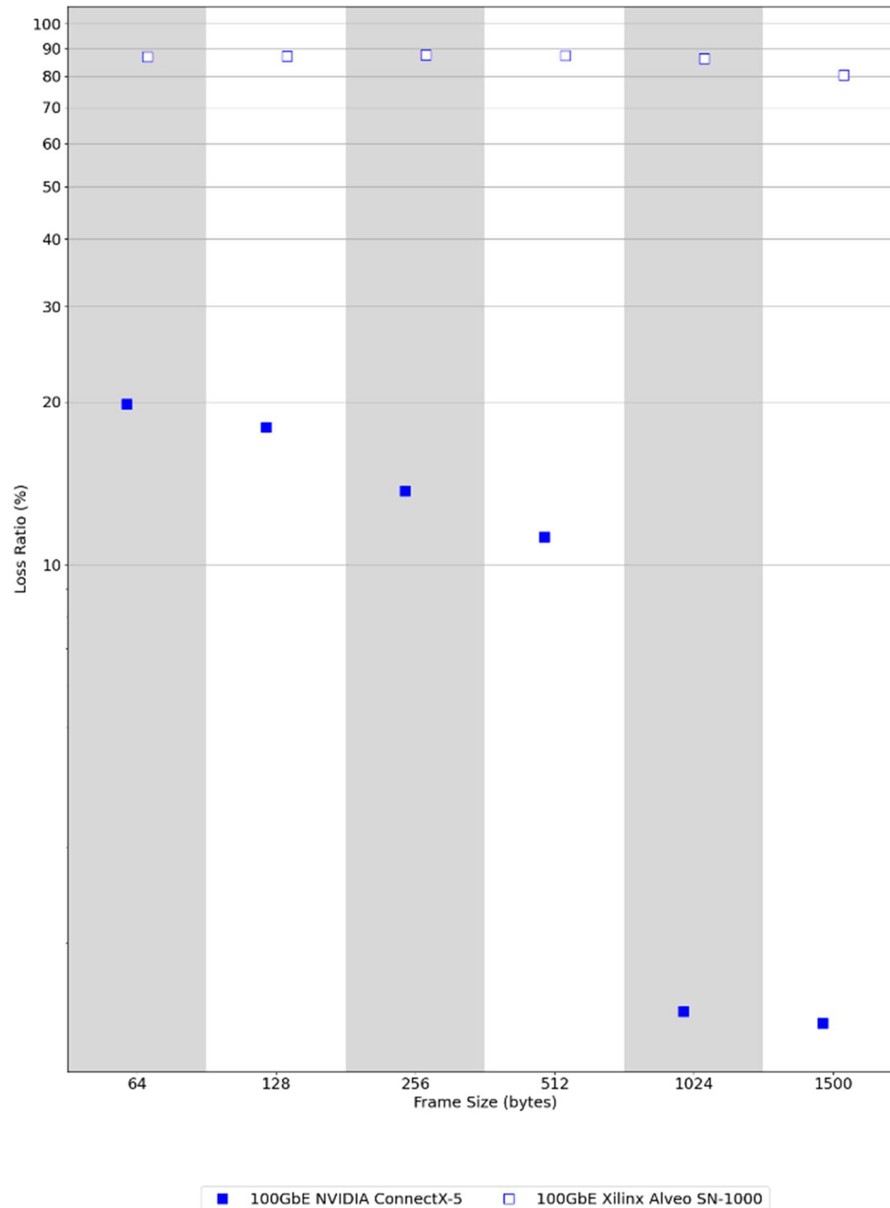

Figure 20: Loss Ratio graph for OVS-vanilla over different frame sizes

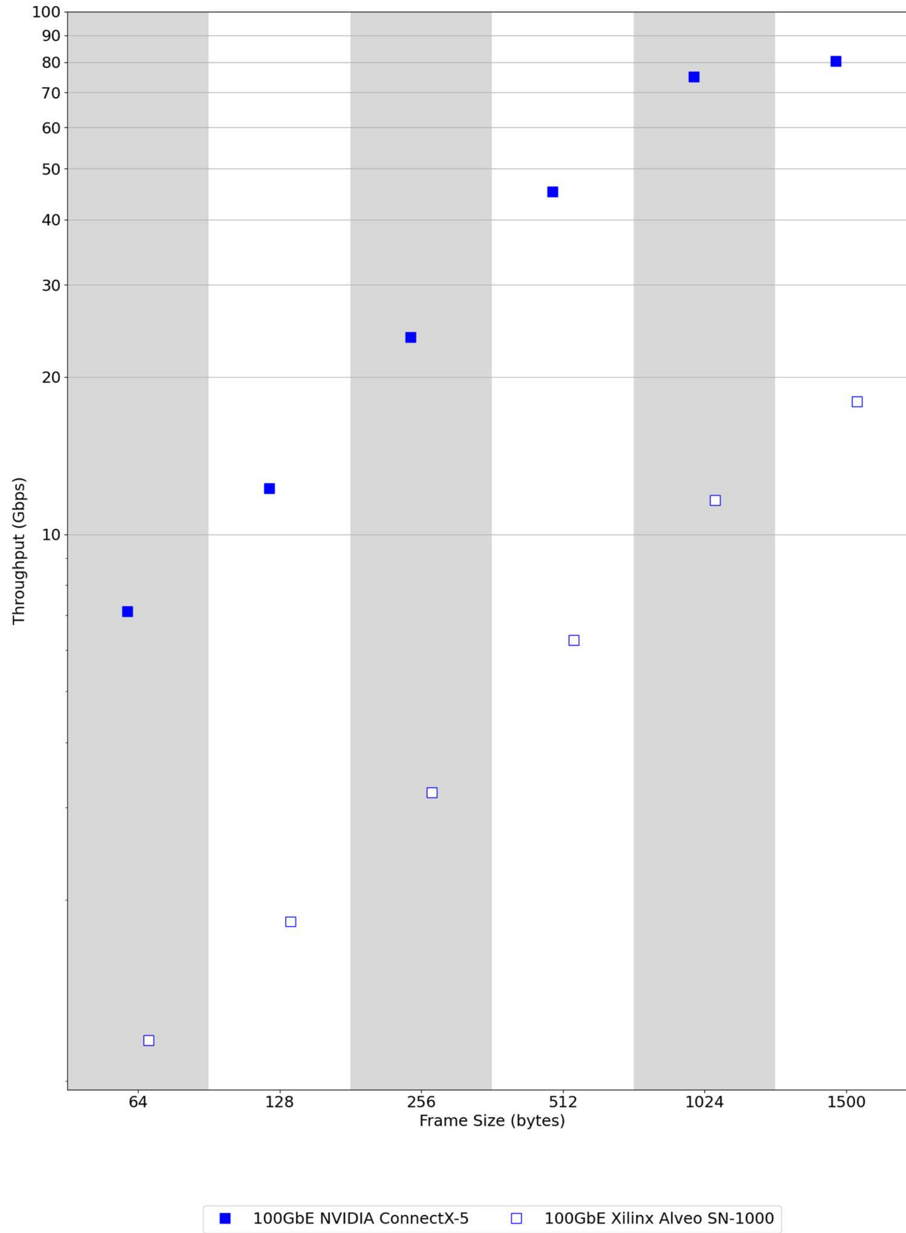

Figure 21: Throughput graph for OVS-vanilla over different frame sizes

Table 6: OVS-vanilla measurements

Frame Size / Metric	64		128		256		512		1024		1500	
	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000
Loss Ratio (%)	19.85	86.88	17.95	87.16	13.69	87.61	11.24	87.36	1.49	86.19	1.42	80.27
Throughput (Gbps)	7.11	1.08	12.24	1.82	23.79	3.20	45.25	6.27	74.96	11.63	80.41	17.92

Figure 22 shows the latency measurements for OVS-vanilla, for both cards, over different frame sizes. The pattern established from the previous diagrams continues here. Cx5 outperforms SN1000, with much lower and robust latency measurements for small frames. It is interesting however that for large frames (1024 and 1500 bytes), the behavior of both cards is similar.

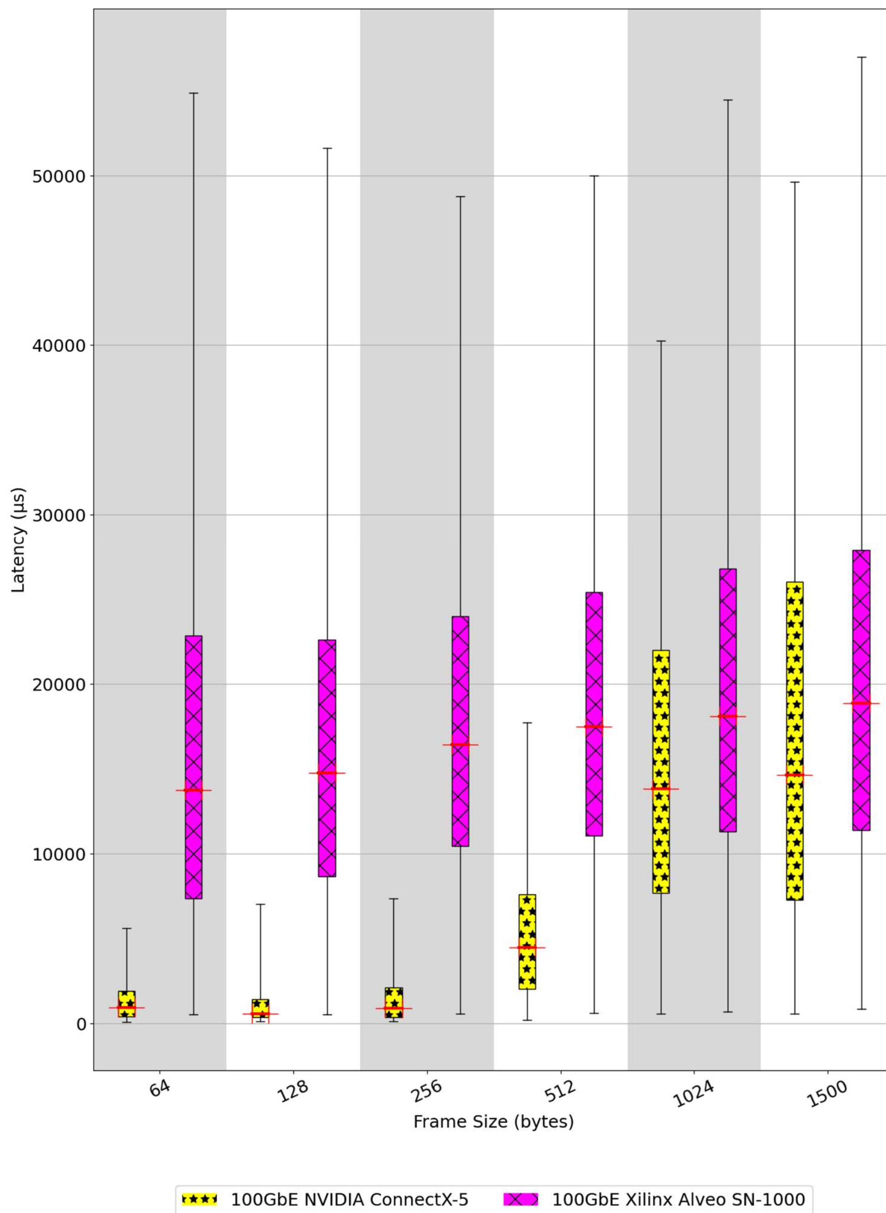

Figure 22: Latency graph for OVS-vanilla over different frame sizes

4.3 OVS-DPDK

OVS-DPDK measurements are really good for the two cards, both in terms of packet loss and throughput (Figure 23 and Figure 24, respectively). The results of both cards are comparable in every case, other than packet loss for 1024 bytes frame size, which is clearly higher for SN1000. In general, both came really close to attaining maximum speed with 92.71 Gbps for CX5 and 92.99 Gbps for SN1000, as shown in Table 7. However, the pattern of CX5 outperforming the SN1000 is

present here too for all cases other than the 1500 bytes frame size, where the SN1000 has slightly better results in both packet loss and throughput.

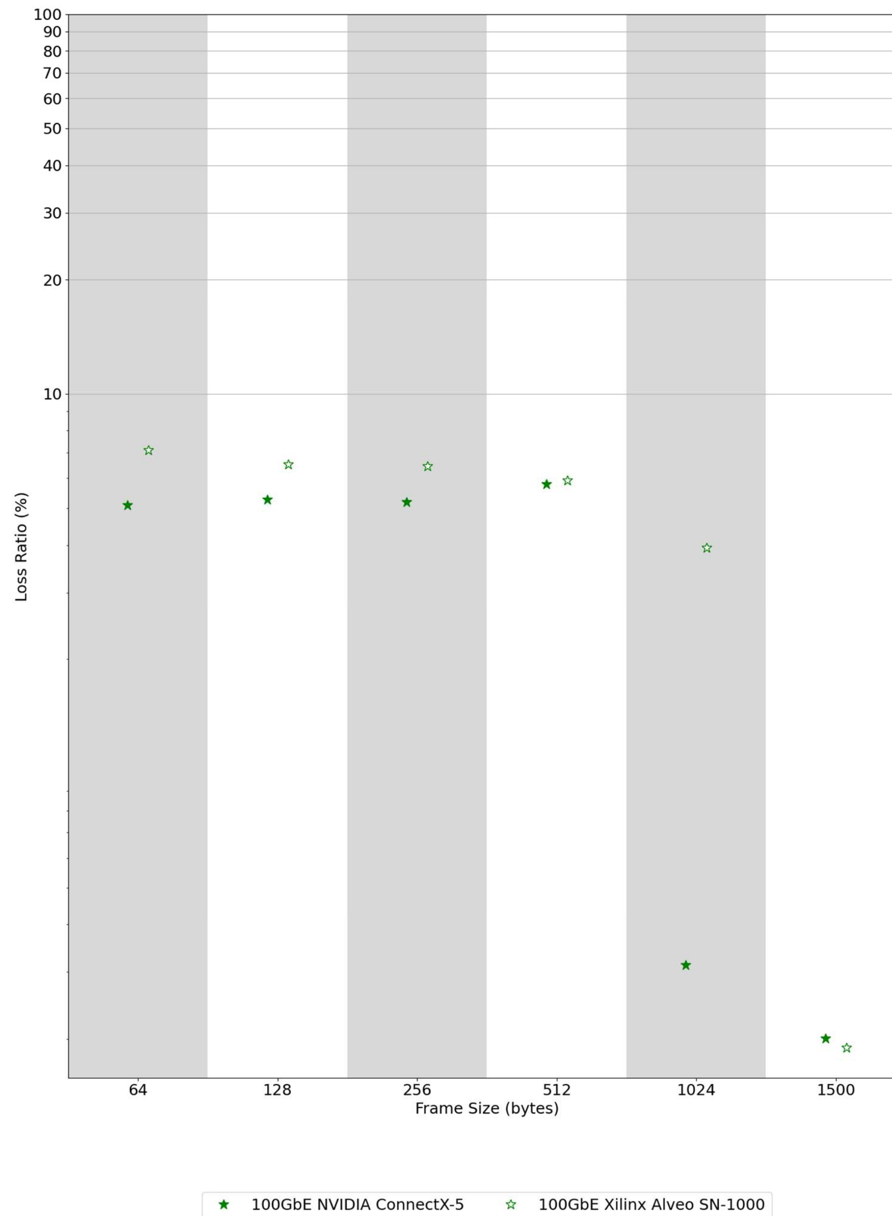

Figure 23: Loss Ratio graph for OVS-DPDK over different frame sizes

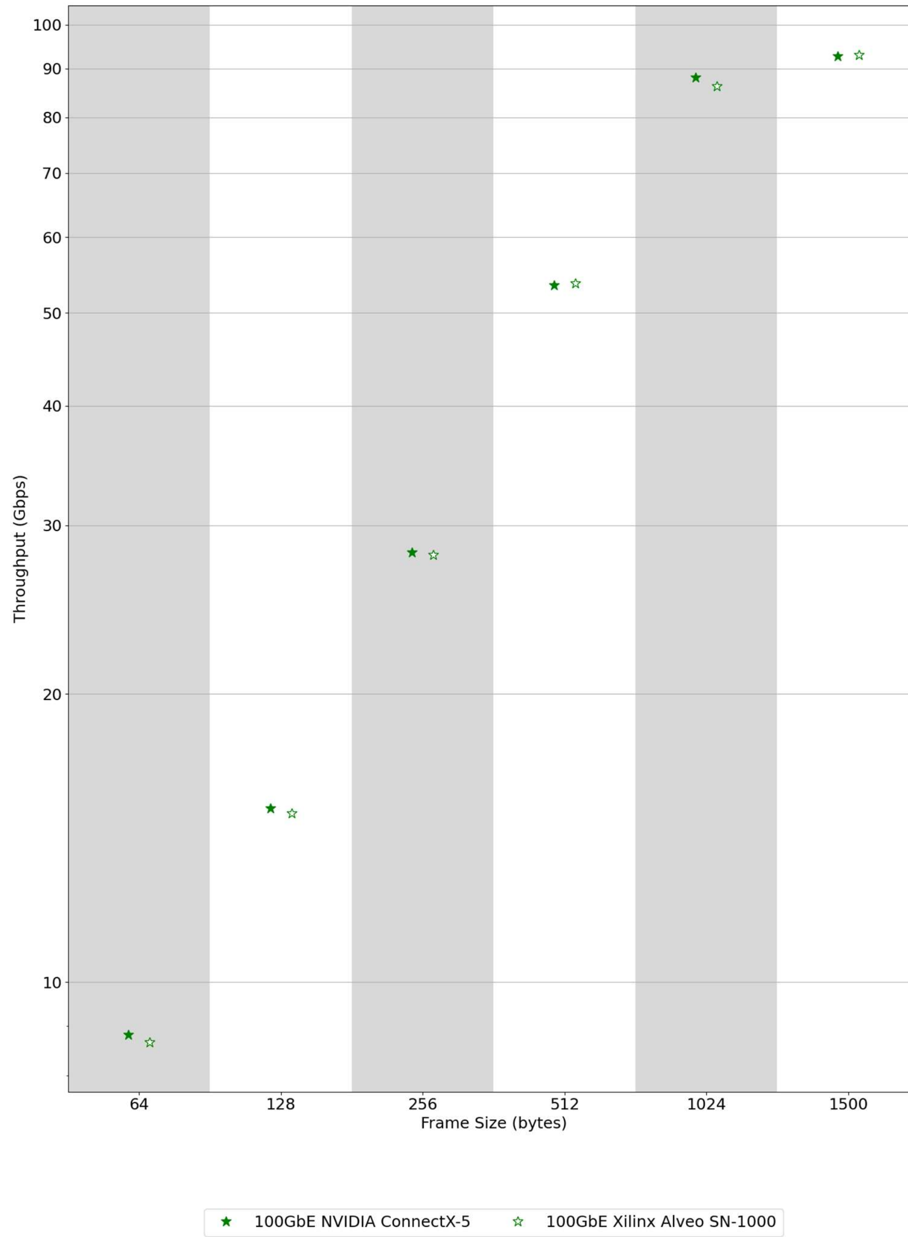

Figure 24: Throughput graph for OVS-DPDK over different frame sizes

Table 7: OVS-DPDK measurements

Frame Size / Metric	64		128		256		512		1024		1500	
	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000
Loss Ratio (%)	5.10	7.10	5.27	6.51	5.18	6.44	5.79	5.92	0.31	3.94	0.20	0.19
Throughput (Gbps)	8.81	8.66	15.19	15.03	28.10	27.93	53.42	53.70	88.02	86.20	92.71	92.99

The above observation about 1024 packet size is also true in terms of latency (Figure 25), where SN1000 seems to have better measurements than CX5, even if less robust. As such, we can mark this particular case as an outlier and not the standard behavior for both cards.

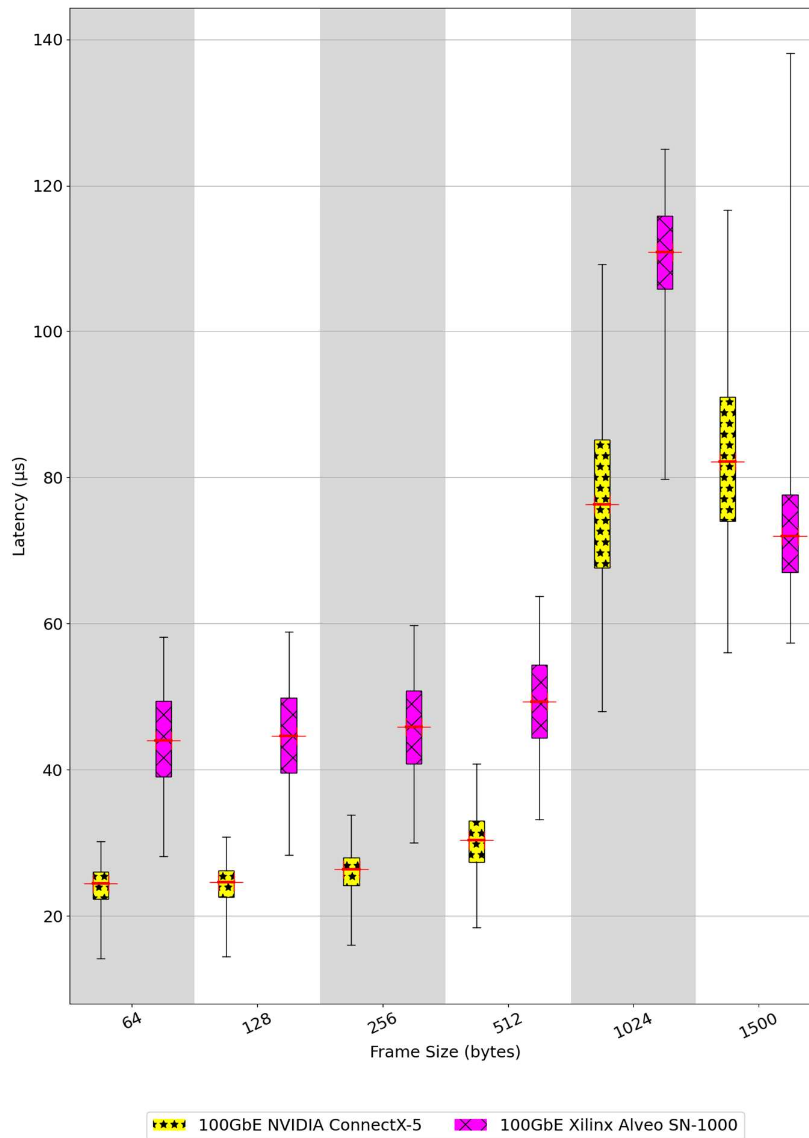

Figure 25: Latency graph for OVS-DPDK over different frame sizes

4.4 P4-DPDK

As mentioned in chapter 3, P4-DPDK is on version 0.1.0 and as such it should not be considered a production grade switch. This can be seen by the results too, as shown in Figure 26 and Figure 27, which show the packet loss and throughput in logarithmic scale, respectively. Again, CX5 and SN1000 results are comparable, which shows that the main problem in P4-DPDK's case is on the software switch itself. Disappointingly the packet loss rate is over 65% and, for most frame sizes, revolves around 75%-80%. This high percentage seems normal if we take into account the low throughput which is under 40 Gbps for all frame sizes and even under 2 Gbps for the 64-byte packet

size! All in all, it seems that there is still a long way to go for P4-DPDK to be a software switch that can be used in any real life use case.

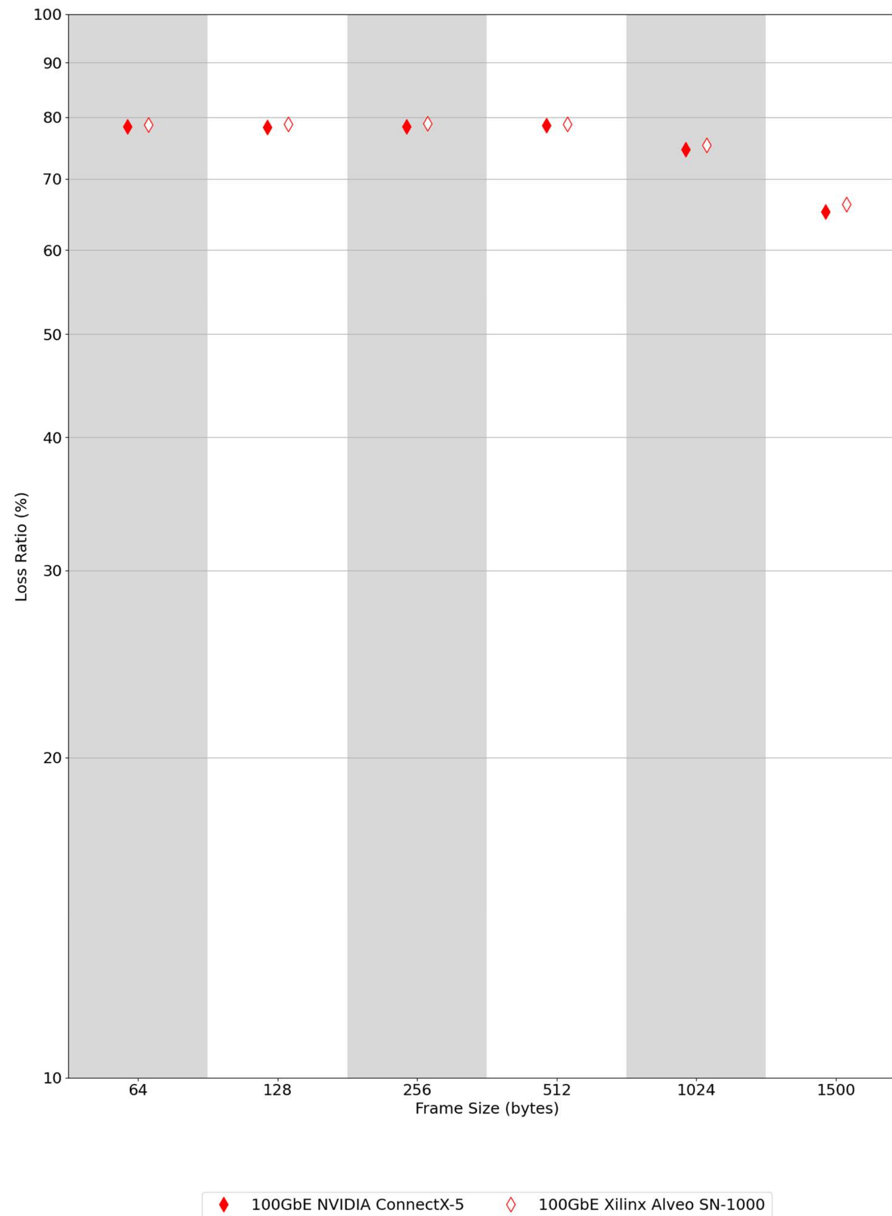

Figure 26: Loss Ratio graph for P4-DPDK over different frame sizes

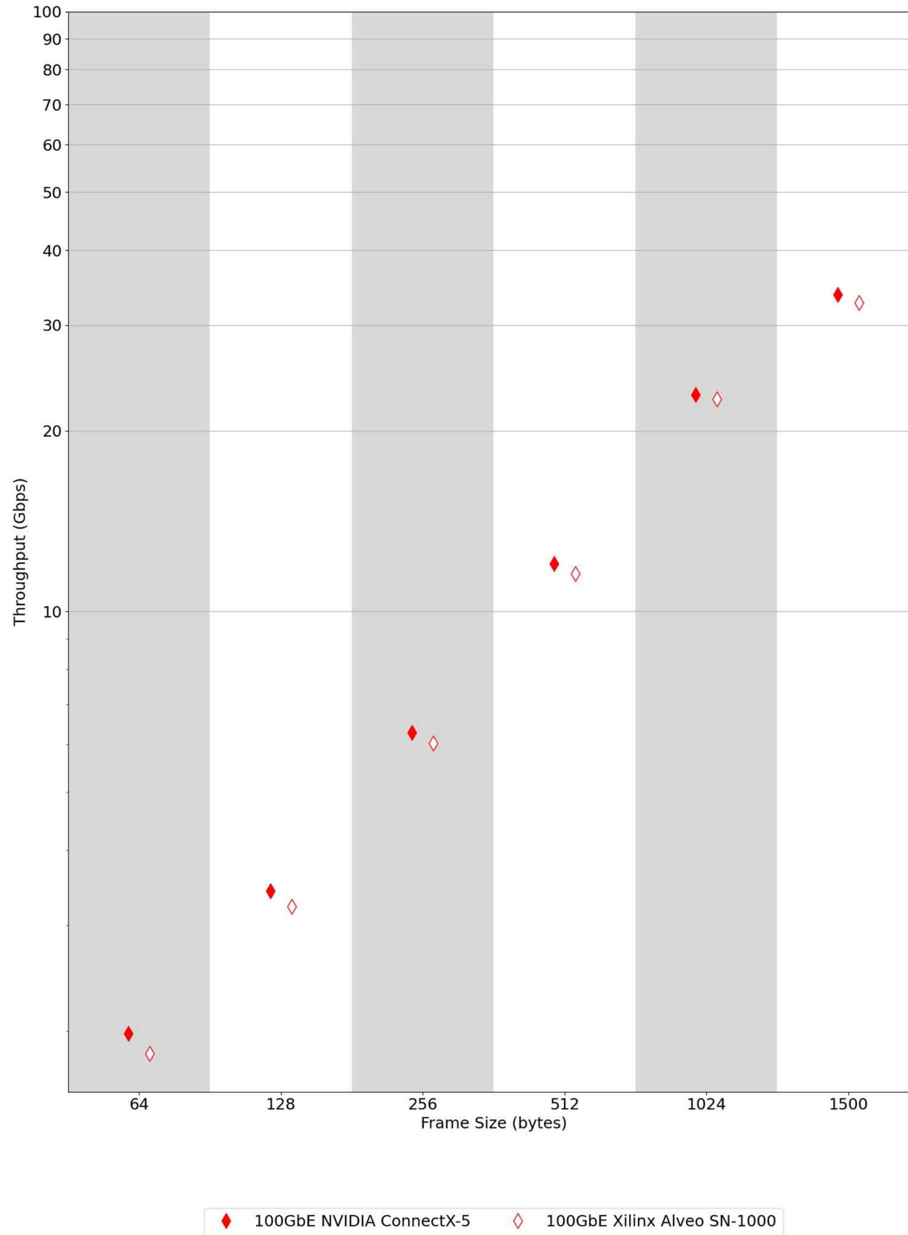

Figure 27: Throughput graph for P4-DPDK over different frame sizes

Table 8: P4-DPDK measurements

Frame Size / Metric	64		128		256		512		1024		1500	
	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000	CX5	SN1000
Loss Ratio (%)	78.35	78.75	78.24	78.77	78.43	78.90	78.55	78.82	74.63	75.30	65.24	66.26
Throughput (Gbps)	1.98	1.83	3.42	3.22	6.28	6.04	12.01	11.56	23.00	22.61	33.71	32.68

On the other hand, the latency measurements are not that bad, with most cases under 100μs, being comparable to OVS-DPDK! Among the two cards, again CX5 seems to perform much better than SN1000.

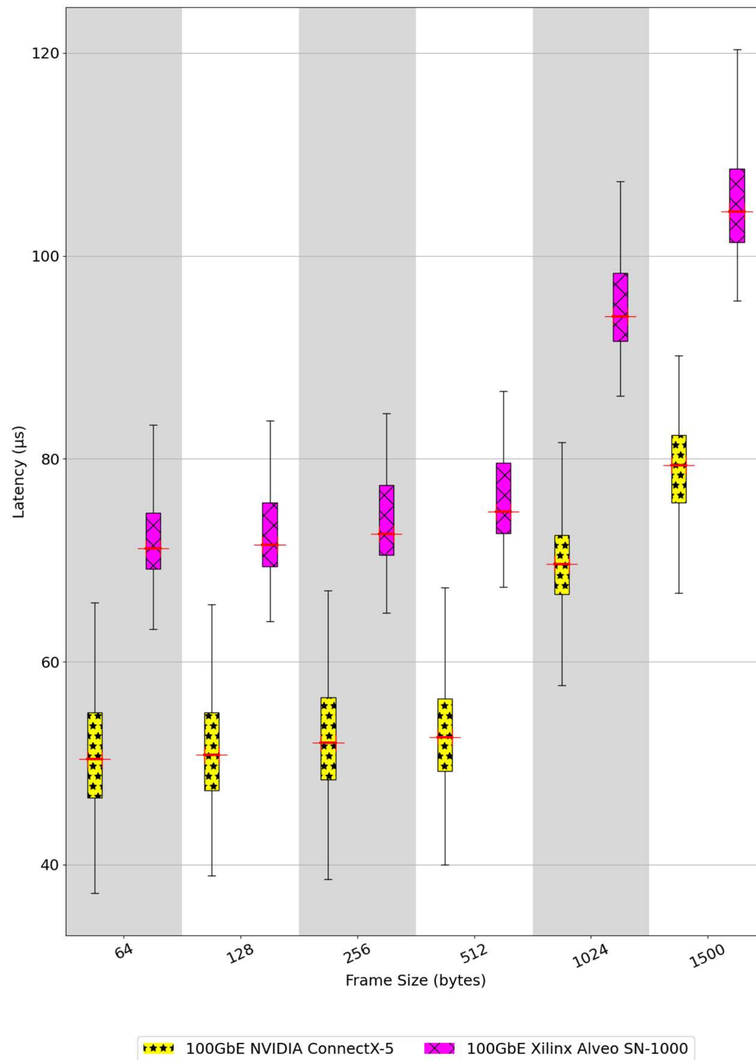

Figure 28: Latency graph for P4-DPDK over different frame sizes

4.5 Xilinx SN1000 FPGA

Unfortunately, we were not able to take measurements of our plugin for the SN1000 FPGA. At high data rates, the card would go out of sync with the driver, enter an infinite reset loop and become inoperative. It is worth noting that this behavior could not be fixed with a soft reboot but needed a cold reboot, making sure that no power was fed to the card, to fix it. As a result, running many experiments with the FPGA design onboarded, was not an easy task.

Seeing that the card would stop functioning with large data rates, we tried to limit the transmission rate. However that proved problematic too, as even at the low rate of 500 Mbps, the card would stop functioning. Further trying to understand SN1000's behavior, we proceeded with

some simpler experiments using ping[35] to control the count of packets sent (-c flag) and the interval between them (-i flag). Ultimately, it seems that the SN1000 breaks in a combination of the number of packets sent and the rate they are sent. Low number of packets (<100) would not break the card at the rates we know that it did with more packets (~500Mbps). On the other hand, increasing the number of packets to 200 would break the card at even lower rates than what we had successfully tried previously (~0.1 Mbps).

These indicate that there is probably a memory management problem (e.g., buffer overflow). However, it is not clear where the problem lies, i.e., in the software (driver) or the hardware (the way P4 is converted into a hardware design). Moreover, Xilinx is aware of that problem but considers SN1000 an end-of-life card, which means that we cannot rely on a fix from their side.

4.6 Remarks on the Comparisons

Figure 29 and Figure 30 show the packet loss and throughput respectively for all the switches implemented on both cards. There are clearly two main findings. First of all, about the switches, as expected bmv2 has the worst results, as it employs the kernel driver and it is used mainly for development and validation purposes. On the other hand, OVS-vanilla, even though it uses the kernel driver, shows promising performance, which justifies the place it has in the industry as the go-to software switch solution. Of course, good performance goes even further when coupled with DPDK, where the results are very good and nearing the maximum values. Unfortunately, P4-DPDK does not perform as good, even though it uses DPDK. Many optimizations still need to be performed before it can be considered as a real-world solution. This is sensible considering that P4-DPDK is still in alpha stages. One interesting observation is about the difference in packet loss and throughput regarding P4-DPDK. In the case of throughput, while not comparable to OVS-DPDK, it seems to be of an order of magnitude better than OVS-vanilla (for SN1000). The same cannot be said for packet loss, which is in the same class as OVS-vanilla. This shows that there is space for improvements in that area, which would surely bring small improvements in throughput too. It must be noted, however, that in terms of latency, P4-DPDK performs comparably with OVS-DPDK.

Secondly, regarding the two smartNICs, NVIDIA ConnectX5 outperforms Xilinx Alveo SN1000 in nearly all cases. What was most surprising is that OVS-vanilla, using the kernel driver, produced comparable results to OVS-DPDK! This means that NVIDIA has dedicated effort to optimize the driver of the card in order to cater to the needs of modern data centers. On the other hand, SN1000 not only did it produce worse results in most of the experiments, but also failed on its selling point, which was easy programmability of its FPGA by using P4. Of course, this could stem from the immaturity of the used toolchain, whose problems could be fixed in the future.

While the ability to produce FPGA bitstreams from P4 is of great value and we believe that this domain will flourish in the future, programming the SN1000 proved to be a troublesome procedure. Writing the P4 program in the required way, to correctly achieve the functionality needed, proved to be more difficult than just writing P4. This was because we had to change the source code to accommodate for the internals of the card, by correctly checking and changing specific flags that alter the path of a packet inside the card. The documentation about these flags was not always clear

[35] https://manned.org/ping.8 (accessed 28/06/2023)

and, in some cases, contained errors. Additionally, while the way the final code is converted into an FPGA design seemed to be straightforward, the synthesis took a really long time and most importantly, in the end, the produced design was not able to achieve to even a hundredth fraction of the nominal speed of the card or at least give us some results. Instead, when pushing with high data rates, a card-driver synchronization problem was occurring.

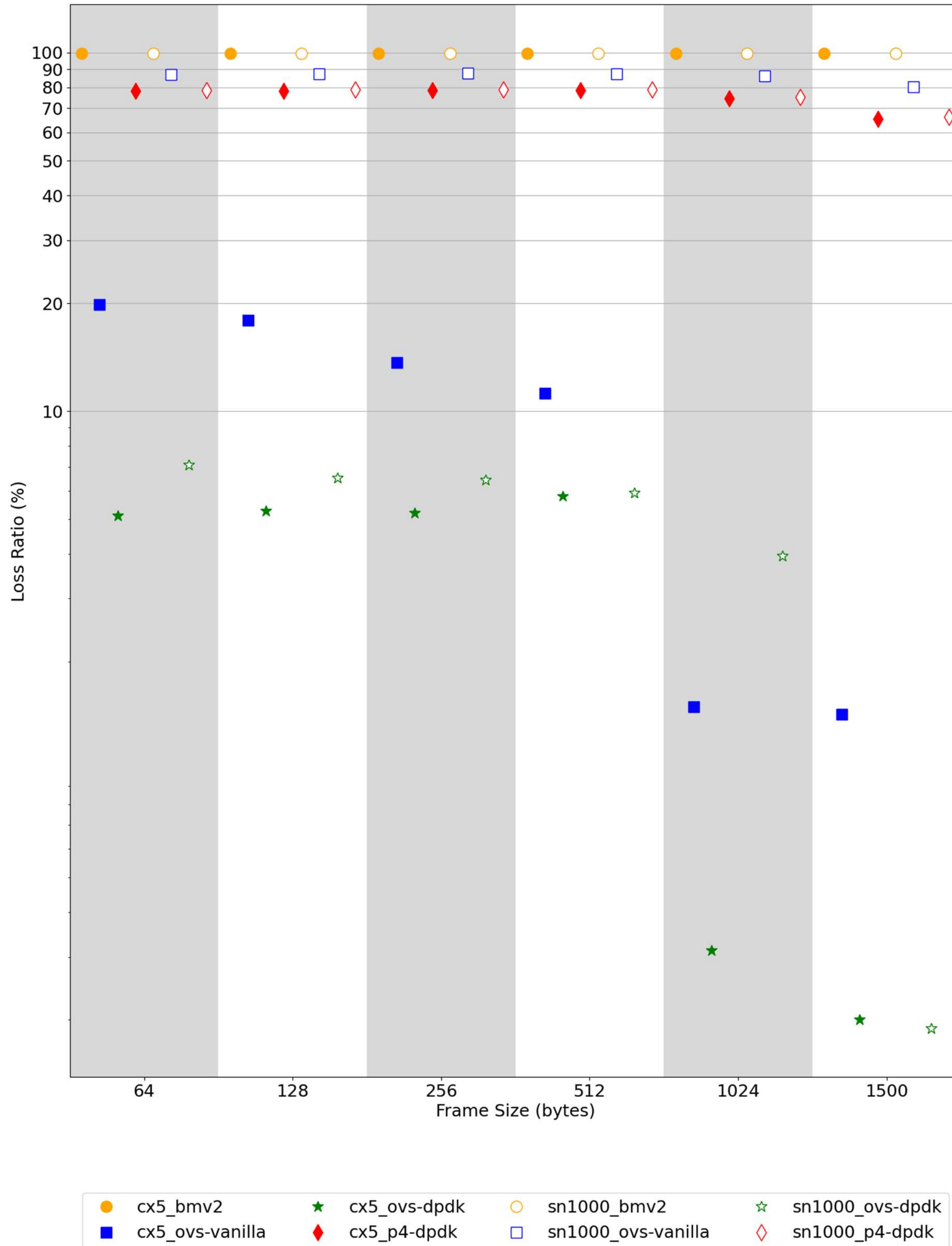

Figure 29: Comparative results for packet loss

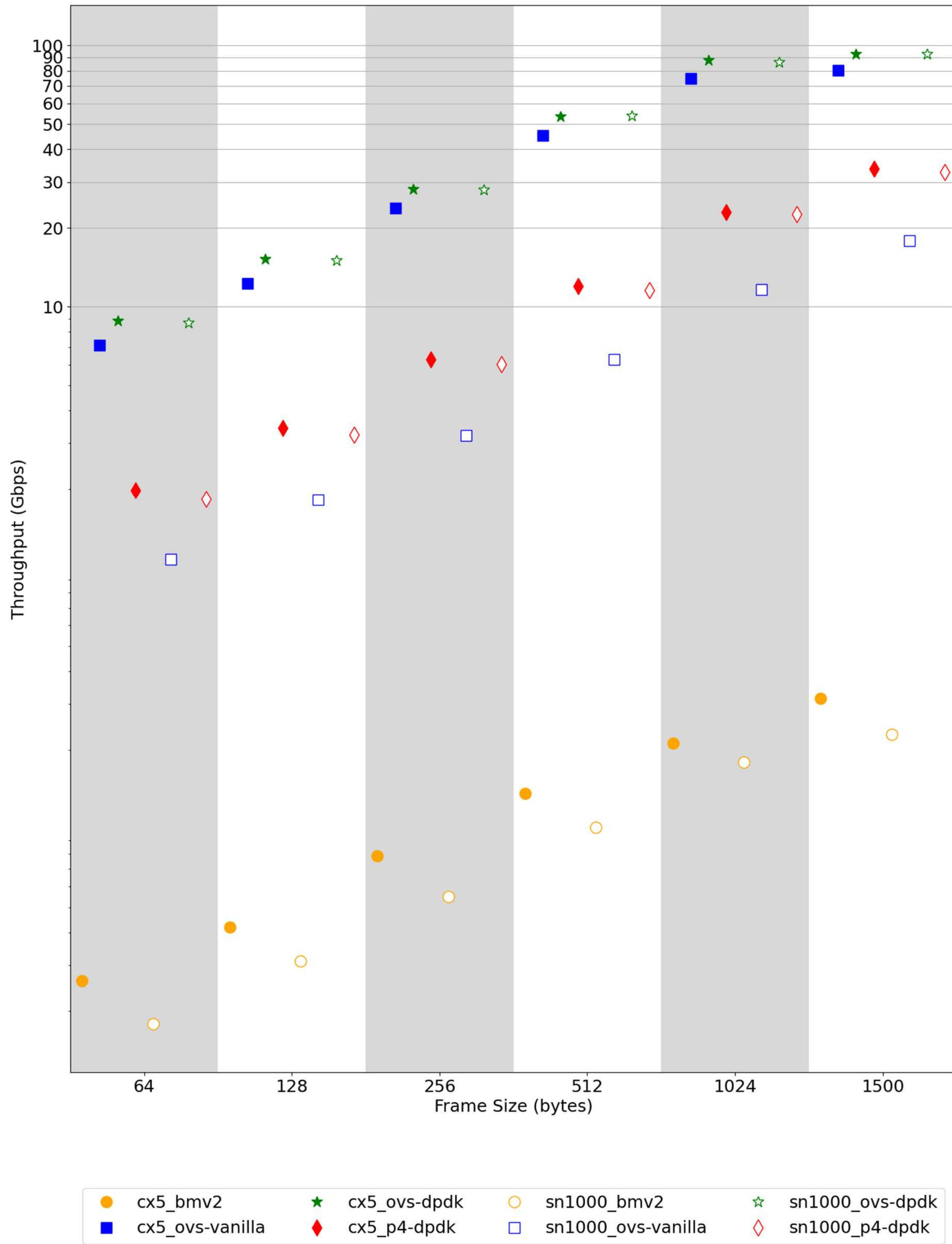

Figure 30: Comparative results for throughput

5

Conclusions

5.1 Closing Remarks

Closing this assignment, we will attempt to recapitulate and summarize its results. The aim is to draw specific conclusions regarding the objectives set in the Introduction and to outline future research proposals that have emerged during working on this thesis.

In the Introduction, we have described specific objectives for the specific master thesis. By concluding it, we see that computer networks have gone a long way since first conceived as an extension of the telephone networks. As in almost all fields of science, a lot of concepts seem to arise again and again, waiting for the right time, in terms of maturity and technological feasibility, to have an impact. SDN brought with its introduction, a paradigm shift in how we think, design and manage networks. With it came new programming languages, like P4, new management software, like controllers, and new hardware, like smartNICs.

In Chapter 2 we provided a thorough survey of all these changes, beginning with the need for SDN and its first steps to it standardization and the most important available controllers. Following this, we introduced P4, a DSL to program the data plane that we consider to be of utmost importance, due to the way it provides abstractions over the networks. We also described the new specialized hardware that has appeared in the market, smartNICs, along with their uses and their programming capabilities. As our initial object was to compare different smartNICs using different software switches, we surveyed the state of the art among different papers that have engaged with that subject.

In Chapter 3 we described our experiments, beginning with the testbed we set, including the description of the servers in which the experiments ran and detailing the procedures we have used. An important tool in that regard was NPF, which allowed us to run experiments and collect measurements in a structured and easily replicated way. Moving on from the common setup of the experiments, we described each experiment separately, along with its different parts and specific configurations. A little more detailed was the description of the Xilinx Alveo SN1000 smartNIC, which was used in our experiments, as it includes an FPGA device. Thus, we delved a bit deeper into its programmability.

In Chapter 4, we presented the results that we collected from the aforementioned experiments. For each switch, we ran several iterations, we calculated mean values, and comparisons were made for the two smartNICs under test. That allowed us to observe what is the relative performance between the two switches, but also how the two cards compare with themselves. We saw that NVIDIA ConnectX 5 outperformed Xilinx Alveo SN1000 in almost all the cases, especially when OVS-vanilla was employed, where the difference was equal to, nearly, an order of magnitude. Regarding the switches, most of the results were expectant, apart from P4-DPDK which is, however, in its early stages of development.

5.2 Future Work

While our original objectives were accomplished, we believe that further work can be done towards the direction followed in this master theses. Our plans include running similar experiments with other kinds of software switches. T4P4S is certainly in our scope, as it is an alternative to P4-DPDK. As we have mentioned, it is not a true software switch, as it does not allow for hot-swapping of its configuration. Nevertheless it is worth comparing how it stands against P4-DPDK.

Additionally, other smartNICs should be investigated. For example, NVIDIA ConnectX 6 is an interesting candidate and upgrade of the ConnectX 5, which is programmable using NVIDIA's DOCA that includes P4 programmability. Moreover, we will continue to monitor Xilinx's efforts towards using P4 as an HLS language. Unfortunately, Alveo SN1000 is in its end of life, but Vitis Networking P4 seems to be alive and under development, which means that they will continue to compete in the smartNIC market.

Another interesting path would be to compare software switches with hardware P4 programmable switches, like Intel's Tofino. It is most probable that the hardware switches would outperform the software switches, but it is interesting to see how much and in what ways. Additionally, it is interesting to check how much of the computation needed in a network can be moved from switches to smartNICs.

Finally, P4 itself is a really promising language. For us, P4 is a disruptive innovation, in the same sense that high level languages were. High level languages provided the necessary abstractions that allowed developers to break free from the burdens of assembly languages and create more complex and more interesting applications. In the same sense, P4 allows network engineers to break free from the ossification of protocols and create custom made solutions, optimized for their specific needs. At the same time, it opens the road for in-network computation, painting a new landscape on what we consider computer networks to be. In that sense, we hold P4 in very high regard and we hope that we will be able to advance the ecosystem supporting it, for example through active development in the TeraflowSDN controller. But also, this will be done, by extending its capabilities as a programming language, extending its application to other domains, e.g. optical networks, and by evolving and being used as an HLS for FPGA programming.

Bibliography

[1] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Inf. Syst. Front.*, vol. 17, no. 2, pp. 243–259, Apr. 2015, doi: 10.1007/s10796-014-9492-7.

[2] M. Fedor, M. L. Schoffstall, J. R. Davin, and J. D. Case, "Simple Network Management Protocol (SNMP)," Internet Engineering Task Force, Request for Comments RFC 1157, May 1990. doi: 10.17487/RFC1157.

[3] M. Casado, N. McKeown, and S. Shenker, "From ethane to SDN and beyond," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 5, pp. 92–95, Nov. 2019, doi: 10.1145/3371934.3371963.

[4] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: an intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014, doi: 10.1145/2602204.2602219.

[5] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, doi: 10.1145/1355734.1355746.

[6] L. L. Peterson, C. Cascone, B. O'Connor, T. Vachuska, and B. Davie, *Software-Defined Networks: A Systems Approach*. Systems Approach LLC, 2021.

[7] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology," Internet Engineering Task Force, Request for Comments RFC 7426, Jan. 2015. doi: 10.17487/RFC7426.

[8] M. Monaco, O. Michel, and E. Keller, "Applying operating system principles to SDN controller design," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, in HotNets-XII. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 1–7. doi: 10.1145/2535771.2535789.

[9] "ONOS | Proceedings of the third workshop on Hot topics in software defined networking." https://dl.acm.org/doi/abs/10.1145/2620728.2620744 (accessed Jul. 30, 2023).

[10] R. Vilalta *et al.*, "TeraFlow: Secured Autonomic Traffic Management for a Tera of SDN flows," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, Jun. 2021, pp. 377–382. doi: 10.1109/EuCNC/6GSummit51104.2021.9482469.

[11] P. Famelis *et al.*, "P5: Event-driven Policy Framework for P4-based Traffic Engineering," in *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2023, pp. 1–3. doi: 10.1109/HPSR57248.2023.10148012.

[12] Open Networking Foundation, "OpenFlow Switch Specification." Mar. 26, 2015. Accessed: Jul. 29, 2023. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[13] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Commun. Surv. Tutor.*, vol. 20, no. 1, pp. 333–354, 2018, doi: 10.1109/COMST.2017.2782482.

[14] P. Bosshart *et al.*, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000, doi: 10.1145/354871.354874.

[16] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends," *IEEE Access*, vol. 9, pp. 87094–87155, 2021, doi: 10.1109/ACCESS.2021.3086704.

[17] R. Doenges *et al.*, "Petr4: formal foundations for p4 data planes," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, p. 41:1-41:32, Jan. 2021, doi: 10.1145/3434322.

[18] The P4 Language Consortium, "The P4 Language Specification," Jan. 2016. Accessed: Jan. 08, 2023. [Online]. Available: https://p4.org/p4-spec/p4-14/v1.1.0/tex/p4.pdf

[19] The P4 Language Consortium, "P4~16~ Language Specification," May 2017. Accessed: Jul. 27, 2023. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[20] The P4.org Architecture Working Group, "P416 Portable Switch Architecture (PSA)," Apr. 2021. Accessed: Jan. 08, 2023. [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[21] The P4 Language Consortium, “P4 Portable NIC Architecture (PNA),” May 2023. Accessed: Jan. 08, 2023. [Online]. Available: https://p4.org/p4-spec/docs/PNA.html

[22] The P4.org API Working Group, “P4Runtime Specification,” Jul. 2021. Accessed: Mar. 08, 2023. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[23] P. Göransson, C. Black, and T. Culver, “Chapter 10 - Network Functions Virtualization,” in *Software Defined Networks (Second Edition)*, P. Göransson, C. Black, and T. Culver, Eds., Boston: Morgan Kaufmann, 2017, pp. 241–252. doi: 10.1016/B978-0-12-804555-8.00010-7.

[24] C. Tipantuña and P. Yanchapaxi, “Network functions virtualization: An overview and open-source projects,” in *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, Oct. 2017, pp. 1–6. doi: 10.1109/ETCM.2017.8247541.

[25] “NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC) | IEEE Journals & Magazine | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/6963800 (accessed Jul. 31, 2023).

[26] Open Networking Foundation, “OpenFlow-enabled SDN and Network Functions Virtualization,” Feb. 2014. Accessed: Jul. 31, 2023. [Online]. Available: https://www.ramonmillan.com/documentos/bibliografia/OpenFlowEnabledSDN&NFV_ONF.pdf

[27] G. P. Katsikas, T. Barbette, D. Kostić, JR. G. Q. Maguire, and R. Steinert, “Metron: High-performance NFV Service Chaining Even in the Presence of Blackboxes,” *ACM Trans. Comput. Syst.*, vol. 38, no. 1–2, p. 3:1-3:45, Jul. 2021, doi: 10.1145/3465628.

[28] European Telecommunications Standards Institute, “Network Functions Virtualisation (NFV); Architectural Framework.” Dec. 2014. Accessed: Jul. 31, 2023. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf

[29] A. Caulfield, P. Costa, and M. Ghobadi, “Beyond SmartNICs: Towards a Fully Programmable Cloud: Invited Paper,” in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2018, pp. 1–6. doi: 10.1109/HPSR.2018.8850757.

[30] E. D’Arnese, D. Conficconi, M. D. Santambrogio, and D. Sciuto, “Reconfigurable Architectures: The Shift from General Systems to Domain Specific Solutions,” in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, M. M. S. Aly and A. Chattopadhyay, Eds., in Computer Architecture and Design Methodologies. Singapore: Springer Nature, 2023, pp. 435–456. doi: 10.1007/978-981-16-7487-7_14.

[31] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen, and C. Zhang, “P4 to FPGA-A Fast Approach for Generating Efficient Network Processors,” *IEEE Access*, vol. 8, pp. 23440–23456, 2020, doi: 10.1109/ACCESS.2020.2970683.

[32] H. Wang *et al.*, “P4FPGA: A Rapid Prototyping Framework for P4,” in *Proceedings of the Symposium on SDN Research*, in SOSR ’17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 122–135. doi: 10.1145/3050220.3050234.

[33] A. Yazdinejad, A. Bohlooli, and K. Jamshidi, “P4 to SDNet: Automatic Generation of an Efficient Protocol-Independent Packet Parser on Reconfigurable Hardware,” in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct. 2018, pp. 159–164. doi: 10.1109/ICCKE.2018.8566590.

[34] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, “Comparing the performance of state-of-the-art software switches for NFV,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, in CoNEXT ’19. New York, NY, USA: Association for Computing Machinery, Dec. 2019, pp. 68–81. doi: 10.1145/3359989.3365415.

[35] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, and G. Q. Maguire, “What You Need to Know About (Smart) Network Interface Cards,” in *Passive and Active Measurement*, O. Hohlfeld, A. Lutu, and D. Levin, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 319–336. doi: 10.1007/978-3-030-72582-2_19.

[36] H. Ghasemirahni *et al.*, “Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets,” presented at the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2022, pp. 807–827. Accessed: Sep. 03, 2023. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni

[37] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. M. Jr, “Metron: {NFV} Service Chains at the True Speed of the Underlying Hardware,” presented at the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), 2018, pp. 171–186. Accessed: Sep. 03, 2023. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/katsikas

[38] R. Kundel, F. Siegmund, R. Hark, A. Rizk, and B. Koldehofe, “Network Testing Utilizing Programmable Network Hardware,” *IEEE Commun. Mag.*, vol. 60, no. 2, pp. 12–17, Feb. 2022, doi: 10.1109/MCOM.001.2100191.

[39] G. Chen, G. Zeng, and L. Chen, “P4COM: In-Network Computation with Programmable Switches.” arXiv, Jul. 28, 2021. doi: 10.48550/arXiv.2107.13694.

[40] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa, and M. Ghobadi, “Challenging the Stateless Quo of Programmable Switches,” in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, in HotNets ’20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 153–159. doi: 10.1145/3422604.3425928.

[41] “The design and implementation of open vSwitch | Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation.” https://dl.acm.org/doi/10.5555/2789770.2789779 (accessed Aug. 27, 2023).

[42] T. Osiński, H. Tarasiuk, P. Chaignon, and M. Kossakowski, “A Runtime-Enabled P4 Extension to the Open vSwitch Packet Processing Pipeline,” *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 3, pp. 2832–2845, Sep. 2021, doi: 10.1109/TNSM.2021.3055900.

[43] M. Shahbaz *et al.*, “PISCES: A Programmable, Protocol-Independent Software Switch,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, in SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 525–538. doi: 10.1145/2934872.2934886.

[44] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff, “revisiting the open vSwitch dataplane ten years later,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, in SIGCOMM ’21. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 245–257. doi: 10.1145/3452296.3472914.

Annex A: P4 program used for Software Switches

```
/* -*- P4_16 -*- */

#include <core.p4>
#include <psa.p4>

/*************************************************************************
 **********************  H E A D E R S  *********************************
 *************************************************************************/

/*  Define all the headers the program will recognize             */
/*  The actual sets of headers processed by each gress can differ */

/* Standard ethernet header */
header ethernet_h {
    bit<48>   dstAddr;
    bit<48>   srcAddr;
    bit<16>   etherType;
}

/*************************************************************************
 *************  I N G R E S S   P R O C E S S I N G   ******************
 *************************************************************************/

    /***********************  H E A D E R S  ***********************/

struct my_ingress_headers_t {
    ethernet_h   ethernet;
}

    /******  G L O B A L   I N G R E S S   M E T A D A T A  *********/

struct my_ingress_metadata_t {
}

struct empty_metadata_t {
}

    /***********************  P A R S E R  *************************/
parser Ingress_Parser( packet_in pkt,
                       out my_ingress_headers_t hdr,
                       inout my_ingress_metadata_t meta,
                       in psa_ingress_parser_input_metadata_t ig_intr_md,
                       in empty_metadata_t resub_meta,
                       in empty_metadata_t recirc_meta)
{
     state start {
        transition parse_ethernet;
    }

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

```
    /**************** M A T C H - A C T I O N  ********************/

control ingress( inout my_ingress_headers_t hdr,
                 inout my_ingress_metadata_t meta,
                 in psa_ingress_input_metadata_t ig_intr_md,
                 inout psa_ingress_output_metadata_t ostd
)
{
    action my_drop() {
        ostd.drop = true;
    }

    action set_mac(bit<48> dst_addr, bit<48> src_addr) {
        hdr.ethernet.dstAddr = src_addr;
        hdr.ethernet.srcAddr = dst_addr;
    }

    table mirror{
        key = {
            hdr.ethernet.dstAddr: exact;
        }
        actions = {
            set_mac;
            my_drop;
        }
        default_action = my_drop;
    }

    apply {
        mirror.apply();
    }
}

    /********************  D E P A R S E R  **********************/

control Ingress_Deparser(packet_out pkt,
        out empty_metadata_t clone_i2e_meta,
        out empty_metadata_t resubmit_meta,
        out empty_metadata_t normal_meta,
    inout my_ingress_headers_t hdr,
    in    my_ingress_metadata_t meta,
    in psa_ingress_output_metadata_t istd)
{
    apply {
        pkt.emit(hdr);
    }
}


/*************************************************************************
 ***************  E G R E S S   P R O C E S S I N G   *****************
 *************************************************************************/

    /**********************  H E A D E R S  **********************/

struct my_egress_headers_t {
}

    /********  G L O B A L   E G R E S S   M E T A D A T A  *********/
```

```
struct my_egress_metadata_t {
}

    /**********************  P A R S E R  ************************/

parser Egress_Parser(
        packet_in pkt,
    out my_egress_headers_t hdr,
    inout my_ingress_metadata_t meta,
    in psa_egress_parser_input_metadata_t istd,
        in empty_metadata_t normal_meta,
        in empty_metadata_t clone_i2e_meta,
        in empty_metadata_t clone_e2e_meta)
{
    state start {
        transition accept;
    }
}

    /**************** M A T C H - A C T I O N  *******************/

control egress(
    inout my_egress_headers_t hdr,
    inout my_ingress_metadata_t meta,
    in psa_egress_input_metadata_t istd,
        inout psa_egress_output_metadata_t ostd)
{
    apply {
    }
}

    /********************  D E P A R S E R  **********************/

control Egress_Deparser(packet_out pkt,
        out empty_metadata_t clone_e2e_meta,
        out empty_metadata_t recirculate_meta,
    inout my_egress_headers_t hdr,
    in my_ingress_metadata_t meta,
    in psa_egress_output_metadata_t istd,
        in psa_egress_deparser_input_metadata_t edstd)
{
    apply {
        pkt.emit(hdr);
    }
}

#if __p4c__
bit<32> test_version = __p4c_version__;
#endif

/*********** F I N A L   P A C K A G E ****************************/

IngressPipeline(Ingress_Parser(), ingress(), Ingress_Deparser()) ip;

EgressPipeline(Egress_Parser(), egress(), Egress_Deparser()) ep;

PSA_Switch(ip, PacketReplicationEngine(), ep, BufferingQueueingEngine())
main;
```

Annex B: P4 program used for Xilinx SN1000's FPGA

```
/*****************************************************************/
/*************** D E F S / C O N S T A N T S **********************/
/*****************************************************************/

// P4 core definitions
#include <core.p4>
// Xilinx Pipeline target architecture definitions
#include <xsa.p4>

// Address field types
typedef bit<48> MacAddr;

/**************************************************************************/
/*********************** H E A D E R S  ***********************************/
/**************************************************************************/

/*****************************/
/****** Standard Headers *******/
/*****************************/

// The following are based on Xilinx's documentation and examples

// Network packet types
#define NP_TYPE_H2C_DMA  0
#define NP_TYPE_PRE_MAE  1
#define NP_TYPE_POST_MAE 2
#define NP_TYPE_C2H_DMA  3

header ef100_cap_netpkt_post_mae_t {
    bit<224> data;
}
```

```
#define CH_BYTE_ADDR_MASK 0xfff8

#define CH_BYTE_ADDR(addr_) ((addr_) & CH_BYTE_ADDR_MASK)

#define CH_BYTE_END(addr_) (223 - CH_BYTE_ADDR(addr_))

#define CH_BYTE_START(addr_) (223 - CH_BYTE_ADDR(addr_) - 7)

#define CH_VAL_MASK(len_) ((1 << (len_)) - 1)

#define CH_VAL_OFFSET(addr_) ((addr_) - CH_BYTE_ADDR(addr_))

#define CH_BYTE(ch_, addr_)                         \
  ch_[CH_BYTE_END(addr_):CH_BYTE_START(addr_)]

#define CH_READ_40(ch_, addr_, val_)                \
{                                                   \
    val_[7:0] = CH_BYTE(ch_, addr_);                \
    if (CH_BYTE_START(addr_ + 8) >= 0)              \
        val_[15:8] = CH_BYTE(ch_, addr_ + 8);       \
    if (CH_BYTE_START(addr_ + 16) >= 0)             \
        val_[23:16] = CH_BYTE(ch_, addr_ + 16);     \
    if (CH_BYTE_START(addr_ + 24) >= 0)             \
        val_[31:24] = CH_BYTE(ch_, addr_ + 24);     \
    if (CH_BYTE_START(addr_ + 32) >= 0)             \
        val_[39:32] = CH_BYTE(ch_, addr_ + 32);     \
}

// This macro is needed to read fields from capsule header with 1-32 bit sizes.
// The returned value is BE.
#define CH_READ_FIELD_32(ch_, addr_, len_, ret_)    \
{                                                   \
    bit<40> tmp_val_ = 0;                           \
    CH_READ_40(ch_, addr_, tmp_val_);               \
    tmp_val_ = (tmp_val_ >> CH_VAL_OFFSET(addr_)) & CH_VAL_MASK(len_);   \
    ret_[len_ - 1:0] = tmp_val_[len_ - 1:0];        \
```

```
}

// This macro is needed to write fields in capsule header with 1-32 bit sizes.
// The input value should be BE.
#define CH_WRITE_FIELD_32(ch_, addr_, len_, val_)           \
{                                                           \
    bit<40> tmp_val_ = 0;                                   \
    CH_READ_40(ch_, addr_, tmp_val_);                       \
    tmp_val_ = tmp_val_ & ~((bit<40>)(CH_VAL_MASK(len_) <<
                  CH_VAL_OFFSET(addr_)));                                  \
    tmp_val_ = tmp_val_ | (((bit<40>)(val_)) << CH_VAL_OFFSET(addr_));     \
    CH_BYTE(ch_, addr_) = tmp_val_[7:0];                    \
    if (CH_BYTE_START(addr_ + 8) >= 0)                      \
        CH_BYTE(ch_, addr_ + 8) = tmp_val_[15:8];           \
    if (CH_BYTE_START(addr_ + 16) >= 0)                     \
        CH_BYTE(ch_, addr_ + 16) = tmp_val_[23:16];         \
    if (CH_BYTE_START(addr_ + 24) >= 0)                     \
        CH_BYTE(ch_, addr_ + 24) = tmp_val_[31:24];         \
    if (CH_BYTE_START(addr_ + 32) >= 0)                     \
        CH_BYTE(ch_, addr_ + 32) = tmp_val_[39:32];         \
}

#define CH_ROUTE_RDP_B_PL_OFFSET 11
#define CH_ROUTE_RDP_B_PL_WIDTH 1
#define CH_ROUTE_RDP_C_PL_OFFSET 12
#define CH_ROUTE_RDP_C_PL_WIDTH 1
#define CH_ROUTE_RDP_OUT_NET_OFFSET 14
#define CH_ROUTE_RDP_OUT_NET_WIDTH 1

#define CH_LENGTH_OFFSET 24
#define CH_LENGTH_WIDTH 14

#define CH_NP_TYPE_OFFSET 38
#define CH_NP_TYPE_WIDTH 6
#define CH_SRC_MPORT_OFFSET 108
#define CH_SRC_MPORT_WIDTH 16
```

```
#define CH_DST_MPORT_OFFSET 124
#define CH_DST_MPORT_WIDTH 16

#define CH_USER_MARK_OFFSET 140
#define CH_USER_MARK_WIDTH 32

#define CH_PRE_MAE_RSVD_OFFSET 173
#define CH_PRE_MAE_RSVD_WIDTH 40
#define CH_PRE_MAE_IMPL_DPU_PCP_OFFSET 213
#define CH_PRE_MAE_IMPL_DPU_PCP_WIDTH 3
#define CH_PRE_MAE_IMPL_L2_DADDR_TYPE_OFFSET 216
#define CH_PRE_MAE_IMPL_L2_DADDR_TYPE_WIDTH 2
#define CH_PRE_MAE_IMPL_PRIO_CHAN_FIFO_OFFSET 218
#define CH_PRE_MAE_IMPL_PRIO_CHAN_FIFO_WIDTH 6

#define CH_POST_MAE_USER_FLAG_OFFSET 173
#define CH_POST_MAE_USER_FLAG_WIDTH 1
#define CH_POST_MAE_VSWITCH_STATUS_OFFSET 174
#define CH_POST_MAE_VSWITCH_STATUS_WIDTH 1
#define CH_POST_MAE_LACP_PLUGIN_OFFSET 175
#define CH_POST_MAE_LACP_PLUGIN_WIDTH 1
#define CH_POST_MAE_LACP_INC_L4_OFFSET 176
#define CH_POST_MAE_LACP_INC_L4_WIDTH 1
#define CH_POST_MAE_RSVD_OFFSET 177
#define CH_POST_MAE_RSVD_WIDTH 44
#define CH_POST_MAE_IMPL_L2_DADDR_TYPE_OFFSET 221
#define CH_POST_MAE_IMPL_L2_DADDR_TYPE_WIDTH 2
#define CH_POST_MAE_IMPL_FIRST_REPLAY_OFFSET 223
#define CH_POST_MAE_IMPL_FIRST_REPLAY_WIDTH 1

// Ethernet Header definition
header eth_mac_t {
    MacAddr dmac; // Destination MAC address
    MacAddr smac; // Source MAC address
    bit<16> type; // Tag Protocol Identifier
}
```

```
// ************************************************************************ //
// ******************* S T R U C T U R E S  ******************************* //
// ************************************************************************ //

// header structure
struct headers {
    ef100_cap_netpkt_post_mae_t post_mae_hdr;
    eth_mac_t                   eth;
}

// user metadata structure
struct metadata {
    bit<16> hdr_from_metadata;
    bit<1>  udp_found_out;
}

// ************************************************************************ //
// ************************ P A R S E R  ********************************** //
// ************************************************************************ //

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t smeta) {

    state start {
        transition parse_pseudo_hdr;
    }

    state parse_pseudo_hdr {
        packet.extract(hdr.post_mae_hdr);
        transition parse_eth;
    }

    state parse_eth {
```

```
        packet.extract(hdr.eth);
            transition accept;
    }

}

// ********************************************************************** //
// **********************  P R O C E S S I N G   *********************** //
// ********************************************************************** //

control MyProcessing(inout headers hdr,
                     inout metadata meta,
                     inout standard_metadata_t smeta)
{

    // ***************************//
    // **** Actions *************//
    // ***************************//

    // Drop Packet
    action drop_pkt() {
        smeta.drop = 1;
    }

    apply {
        if (hdr.post_mae_hdr.isValid()) {
            bit<32> val = 0;

                  // ***************************//
                  // **** Capsule Headers ******//
                  // ***************************//

                  // Transform capsule header between MAE2P and P2MAE plugin
                                                                     sockets.

                  // First we have to route the packet to the correct path.
```

```
      // There are three flags for this
      // Route the capsule towards MAE
val = 0;
CH_WRITE_FIELD_32(hdr.post_mae_hdr.data, CH_ROUTE_RDP_B_PL_OFFSET,
                                      CH_ROUTE_RDP_B_PL_WIDTH, val);

// Route the capsule away from MAE2P
val = 0;
CH_WRITE_FIELD_32(hdr.post_mae_hdr.data, CH_ROUTE_RDP_C_PL_OFFSET,
                                      CH_ROUTE_RDP_C_PL_WIDTH, val);

// Route the capsule towards NET_TX
val = 1;
CH_WRITE_FIELD_32(hdr.post_mae_hdr.data,
     CH_ROUTE_RDP_OUT_NET_OFFSET, CH_ROUTE_RDP_OUT_NET_WIDTH, val);

// Set NP_TYPE to PRE_MAE
val = NP_TYPE_PRE_MAE;
CH_WRITE_FIELD_32(hdr.post_mae_hdr.data, CH_NP_TYPE_OFFSET,
                                             CH_NP_TYPE_WIDTH, val);

// Copy SRC_MPORT value to DST_MPORT (SRC_MPORT is the net port
                                          that the packet came from)
val = 0;
CH_READ_FIELD_32(hdr.post_mae_hdr.data, CH_SRC_MPORT_OFFSET,
                                           CH_SRC_MPORT_WIDTH, val);
CH_WRITE_FIELD_32(hdr.post_mae_hdr.data, CH_DST_MPORT_OFFSET,
                                           CH_DST_MPORT_WIDTH, val);

// PRE_MAE 137:135 (w=3) set to 0
// 0 is the default PCP value as defined by IEEE 802.1Q-2005
val = 0;
CH_WRITE_FIELD_32(hdr.post_mae_hdr.data,
CH_PRE_MAE_IMPL_DPU_PCP_OFFSET, CH_PRE_MAE_IMPL_DPU_PCP_WIDTH, val)

// PRE_MAE 139:138 set to POST_MAE 144:143
```

```
            val = 0;
            CH_READ_FIELD_32(hdr.post_mae_hdr.data,
                                             CH_POST_MAE_IMPL_L2_DADDR_TYPE_OFFSET,
                                         CH_PRE_MAE_IMPL_L2_DADDR_TYPE_WIDTH, val);
            CH_WRITE_FIELD_32(hdr.post_mae_hdr.data,
                                              CH_PRE_MAE_IMPL_L2_DADDR_TYPE_OFFSET,
                                        CH_POST_MAE_IMPL_L2_DADDR_TYPE_WIDTH, val);

            // PRE_MAE 145:140 set to 0
            // Not sure what this should be. Setting it to 0, according to
                                                                 Xilinx's examples.
            val = 0;
            CH_WRITE_FIELD_32(hdr.post_mae_hdr.data,
                                             CH_PRE_MAE_IMPL_PRIO_CHAN_FIFO_OFFSET,
                                        CH_PRE_MAE_IMPL_PRIO_CHAN_FIFO_WIDTH, val);

            // ***************************//
            // *** Ethernet Headers ******//
            // ***************************//

            // Swap source and destination address
            tmp = hdr.eth.smac;
            hdr.eth.smac = hdr.eth.dmac;
            hdr.eth.dmac = tmp;
        } else {
                drop_pkt();
            }
    }
}

// ********************************************************************** //
// ********************  D E P A R S E R  ******************************** //
// ********************************************************************* //
control MyDeparser(packet_out packet,
                   in headers hdr,
                   inout metadata meta,
```

```
                   inout standard_metadata_t smeta) {
    apply {
        packet.emit(hdr.post_mae_hdr);
        packet.emit(hdr.eth);
    }
}

// ********************************************************************** //
// *****************************  M A I N  ****************************** //
// ********************************************************************** //

XilinxPipeline(
    MyParser(),
    MyProcessing(),
    MyDeparser()
) main;
```