Tampere University

Juho Tieaho

DATA PLANE PROGRAMMABILITY
in network-attached FPGA accelerators

Faculty of Information
Technology and Communication
Sciences
Master of Science thesis
June 2020

ABSTRACT

Juho Tieaho: Data plane programmability in network-attached FPGA accelerators
Master of Science thesis
Tampere University
Master’s Degree Programme in Electrical Engineering
June 2020

To increase capital efficiency and flexibility in networking, virtualization methods, following the concept of Network Function Virtualization (NFV), can be used. In NFV, network functions conventionally implemented on proprietary hardware appliances are migrated to Commercial Off-The-Shelf (COTS) hardware as software-implemented virtualized functions. This may come at the cost of performance, and some performance-critical functions may require the usage of specialized hardware as hardware accelerators.

This work is focused around the reconfigurable Field-Programmable Gate Array (FPGA) accelerators, and more specifically, FPGA accelerators that are network-attached, as in accessible directly via network. In this thesis, a data plane programmability language, P4 (Programming Protocol-independent Packet Processors), was trialled as a method for implementing packet processors in the FPGA ingress and egress paths as networking logic surrounding the core accelerator functionality. This was done to map its usability as an alternative to a Register-Transfer Level (RTL) Hardware Description Language (HDL).

For the study, three design variants were implemented, all providing the same networking functionality of Virtual Tunnel Endpoint (VTEP) termination and a five-tuple based firewall. The design variants were a software interfaced P4 design, a reference hard-coded VHDL (Very High Speed Integrated Circuit Hardware Description Language) design, and finally, a hard-coded P4 design for more comparable hardware resource utilization metrics. As the P4 language is a platform-independent high-level description language, a third-party back end compiler was used in the hardware design.

The P4-based implementations were compared against the VHDL-based implementation in terms of FPGA resource utilization, performance, as in latency and throughput, and design automation, as in lines of source code. From the variants, the VHDL design proved to be superior by the lowest resource utilization. Additionally, the VHDL design achieved the lowest latency from the variants, being able to process 1kB frames in 0,5μs, whereas the P4 software interfaced and hard-coded design variants achieved latencies of 1,1μs and 1,3μs, respectively. However, the P4 proved to provide a more automated implementation design flow, indicated by the lines of code: the VHDL description consisted of 8,1x more lines than the P4 software interfaced variant.

Keywords: Data plane programmability, FPGA, hardware accelerator, P4, NFV

TIIVISTELMÄ

Juho Tieaho: Reititystason ohjelmoitavuus verkkoon kytketyissä FPGA-kiihdyttimissä
Diplomityö
Tampereen yliopisto
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Kesäkuu 2020

Verkkofunktioiden virtualisointikonseptin (engl. Network Function Virtualization, NFV) mukaisia menetelmiä voidaan käyttää tiedonsiirtoverkkojen kustannustehokkuuden ja joustavuuden lisäämiseksi. Tämä konsepti tarkoittaa verkkotoimintojen toteutusta virtuaalisina ohjelmistofunktioina. Tällöin perinteisten, erikoiskäyttöisten ja patentoitujen verkkolaitteiden käyttöä voidaan korvata yleiskäyttöisellä ja yleisesti saatavilla olevalla laitteistolla. Ohjelmistototeutuksien käyttö voi kuitenkin näkyä suorituskyvyn heikkenemisenä, jolloin vaativimpien toimintojen suorittamisessa voidaan käyttää apuna erillisiä laitteistokiihdyttimiä.

Tämä työ keskittyy verkkoon kytkettyihin uudelleenohjelmoitaviin FPGA-kiihdyttimiin (engl. Field-Programmable Gate Array). Työssä koekäytettiin reititystason (engl. data plane) ohjelmointiin tarkoitettua P4-kieltä (engl. Programming Protocol-independent Packet Processors) FPGA-kiihdyttimen verkkotoiminnallisuuden toteutuksessa. Työn tavoitteena oli kartoittaa P4-kielen käytettävyyttä tässä käyttökohteessa vaihtoehtona perinteiselle rekisterisiirtotason (engl. Register-Transfer Level, RTL) laitteistokuvauskielelle (engl. Hardware Description Language, HDL).

Tutkimuksessa tuotettiin kolme reititystason toiminnallisuudeltaan vastaavaa toteutusta, jotka toteuttivat virtuaalitunnelin päätepisteen (engl. Virtual Tunnel Endpoint, VTEP) terminnoin, sekä protokollikenttien avulla muodostettuun monikkoon pohjautuvan palomuurin. Toteutuksina olivat ohjelmistorajapinnallinen P4-totetutus, vertailukohtana toimiva kovakoodattu VHDL-toteutus (engl. Very High Speed Integrated Circuit Hardware Description Language), sekä tarkemman resurssien käyttöasteen vertailun mahdollistava kovakoodattu P4-toteutus. P4-kielen kuvaukset käännettiin käyttäen kolmannen osapuolen kääntäjää.

P4-toteutuksia vertailtiin VHDL-toteutukseen käyttäen vertailukohtina FPGA:n resurssien käyttöastetta, suorituskykyä, sekä suunnitteluvuon automaatiota lähdekoodiriveissä mitattuna. VHDL-toteutuksen resurssien käyttöaste osoittautui matalimmaksi. VHDL-toteutus kykeni myös matalimpaan käsittelyviiveeseen, joka oli noin 0,5 µs käsiteltävien pakettien ollessa 1 kilotavun kokoisia. Täyden ohjelmistorajapinnallisen P4-toteutuksen viive oli 1,4 µs, ja kovakoodatun P4-toteutuksen viive 1,1 µs. Automaatioltaan, tässä työssä lähdekoodirivien lukumäärässä mitattuna, P4-toteutus oli ylivertaisin: VHDL-toteutuksessa käytettyjen koodirivien määrä oli 8,1-kertainen P4-toteutukseen nähden.

Avainsanat: Reititystason ohjelmoitavuus, FPGA, laitteistokiihdytin, P4, NFV

PREFACE

Big thanks to Prof. Timo D. Hämäläinen and Arto Oinonen from Tampere University and Jouni Markunmäki from Nokia for supervision and help throughout the writing process of this thesis.

I also want to express my gratitude for Nokia and my excellent colleagues in the company. Special thanks to Hannu Tulla for assistance in all technical matters, as well as Talita Tobias Carneiro and Daniel Koslopp for collaboration throughout the CRUN project.

Tampere, 12.06.2020.

Juho Tieaho

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

4G	Fourth Generation
5G	Fifth Generation
API	Application-Programming Interface
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Product
BCAM	Binary Content-Addressable Memory
BRAM	Block Random-Access Memory
BSV	Bluespec SystemVerilog
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DMA	Direct Memory Access
DUT	Device Under Test
EM	Element Manager
EMS	Element Management System
ETSI	European Telecommunications Standards Institute
FF	Flip-flop
FIFO	First In, First Out
ForCES	Forwarding and Control Element Separation
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware-Description Language
ID	Identifier
ILA	Integrated Logic Analyzer
iNIC	Intelligent Network Interface Controller
IP	Intellectual Property
IPv4	Internet Protocol version 4
JTAG	Joint Test Action Group
LOC	Lines of Code
LUT	Lookup Table
LUTRAM	Lookup Table Random-Access Memory
MANO	Management and Orchestration
NFP	Network Flow Processor
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NIC	Network Interface Controller
NOS	Network Operating System
NPU	Network Processing Unit
OS	Operating System
OVS	Open vSwitch
P4	Programming Protocol-independent Packet Processors
pcap	Packet capture
PCIe	Peripheral Component Interconnect Express
PHY	Physical layer
POC	Proof of Concept
PP Eg. P4 HC	Hard-coded P4 implementation of the egress packet processor
PP Eg. P4 SW IF	Software interfaced P4 implementation of the egress packet processor
PP Eg. VHDL	VHDL implementation of the egress packet processor
PP Ing. P4 HC	Hard-coded P4 implementation of the ingress packet processor

PP Ing. SW IF	Software interfaced P4 implementation of the ingress packet processor
PP Ing. VHDL	VHDL implementation of the ingress packet processor
RAM	Random-Access Memory
RAN	Radio Access Network
RTL	Register-Transfer Level
RX	Receive
SDN	Software-Defined Networking
Shell P4 HC	Shell design with hard-coded P4 implementations of the packet processors
Shell P4 SW IF	Shell design with software interfaced P4 implementations of the packet processors
Shell VHDL	Shell design with VHDL implementations of the packet processors
SR-IOV	Single Root Input/Output Virtualization
TX	Transmit
UDP	User Datagram Protocol
UVM	Universal Verification Methodology
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNF	Virtual Network Function
VNFM	Virtual Network Function Manager
VNI	Virtual Network Identifier
vRAN	virtualized Radio Access Network
VTEP	Virtual Tunnel Endpoint
VxLAN	Virtual Extensible Local Area Network

1. INTRODUCTION

Mobile traffic is increasing, both in volume and variety, due to growing amounts of smart phones and other connected devices [1], creating an increasing demand for lower latency and greater capacity in mobile networks [2]. The telecommunications industry is pressured for higher data rates by the subscribers [3], and for example with the fifth generation (5G) mobile networks, the data rate demand can be up to 100x compared to the fourth generation (4G), while the end-to-end latency must be reduced to a fifth [4]. The higher capacity requirements force the communications service providers to invest in the network, while concurrently finding ways to preserve profitability [1].

For a telecommunications service provider, the Radio Access Network (RAN) causes a major part of capital and operating expenses, respectively up to 80% and 60%. This makes RAN a compelling choice for expense reduction [3]. One method to achieve this is to utilize the concept of Network Function Virtualization (NFV).

In the NFV concept, network functions are separated from proprietary hardware devices and implemented as software virtual functions, running on Commercial Off-The-Shelf (COTS) hardware, for example x86 architecture high volume servers [5]. In the domain of a virtualized RAN (vRAN), this means the virtualization of baseband unit functions to run on a shared physical infrastructure, separating the baseband units from their dedicated remote radio units [3].

With the usage of COTS hardware and virtualized, software-implemented functions, the NFV concept brings benefits such as capital efficiency and flexibility to the network [6]. However, some virtual network functions may require performance not attainable with standard servers and require the usage of specialized hardware as hardware accelerators [7].

One candidate for a hardware accelerator is the Field-Programmable Gate Array (FPGA), which is a reconfigurable platform, and due to its hardware structure can offer improved performance compared to General-Purpose Processors (GPP). In addition to serving as local accelerators for their host server, they can be connected directly to a data centre network. As network-attached, the FPGA is additionally enabled for network acceleration as well as global acceleration [8]. This work is focused on the networking logic on these FPGAs, bringing in the concept of data plane programmability.

Software-Defined Networking (SDN) is a networking paradigm which aims to decouple the control plane from the data plane and to centralize it. The data plane then forwards traffic based on the control plane instructions via a well-defined Application-Programming Interface (API). [9]

On the data plane level, its programmability can be enabled with different programming models and abstractions that are exposed to the control plane. One of the abstractions is the match-action abstraction, where the controller configures the packet processing of the device by managing entries in flow tables. [10] The target is programmed to match values from protocol headers against the values in flow tables and based on the match result and the configuration of the table a certain action is executed. A programming language using this abstraction is the P4 (Programming Protocol-independent Packet Processors).

P4 is a platform-agnostic, domain specific language for the programming of protocol-independent packet processors. In a P4 program, the programmer defines the supported set of protocols, a protocol parser and control programs. The control programs contain the match-action tables and action definitions, which determine how the packets are processed. [11]

The goal of this thesis is to map the feasibility and use cases of using a data plane programmability method, the P4 language, in the implementation of networking logic in network-attached FPGA accelerators. The thesis was done as a part of a larger in-house framework, the CRUN, which presented a cloudified datacentre architecture, where FPGAs could be flexibly provided as hardware accelerator resources, accessible both locally and via network.

The feasibility study was done by implementing packet processors, i.e. the networking logic surrounding the accelerator on the FPGA, with both the P4 language and VHDL (Very High Speed Integrated Circuit Hardware Description Language). The resulting hardware designs were compared against each other by performance metrics, latency and throughput, utilization, as well as by the degree of automation in the design flows.

This thesis is structured as follows. Chapter 2 presents the main networking paradigms behind this work, the NFV and SDN, focusing on an architectural description. Chapter 3 presents hardware acceleration with the NFV framework, as well as an introduction to FPGA accelerators in this context. Chapter 4 opens the domain of data plane programmability, alongside a description of the P4 language as a part of this work. Chapter 5 describes the methodology, tools, and platforms used in this work. Chapter 6 presents the project framework, implemented top level FPGA design, and the implemented networking functionality. Chapter 7 gives a more detailed description of the VHDL and P4 packet processing designs, and of how they were implemented. Chapter 8 presents the utilization, performance and degree of automation results for all design variants, and chapter 9 concludes the thesis.

2. NETWORKING PARADIGMS

In conventional networking, the implementation of network functions often includes the usage of proprietary hardware appliances. Addition of new features is costly and complex, as it requires the purchase of new devices [12]. A large variety of equipment, such as switches and middleboxes, further increases the complexity and slows innovation, as these devices often come with closed and proprietary control software and configuration interfaces varying across vendors [9]. This chapter presents the networking concepts of SDN and NFV, which aim to address these issues.

2.1 NFV

Network Function Virtualization is a networking paradigm which aims to bring capital efficiency and flexibility to networking by replacing proprietary hardware devices with COTS hardware and providing the network functions implemented by them as virtualized functions (Virtualized Network Function, VNF) [6]. This is purposed for a more cost-effective, shareable and homogenous hardware architecture. Additionally, with VNFs flexibly assignable to hardware, functionality is decoupled from location, scalability is increased, and the software-based deployment model enables faster innovation for new services. In conclusion, the European Telecommunications Standards Institute (ETSI) group specification in [6] summarizes the service provisioning differences in NFV compared to non-virtualized traditional networks as follows:

- Decoupled software and hardware, enabling independent evolution for both, and leading to
- flexible network function deployment and dynamically scalable operation, adjusting the performance capacity as required by traffic in the network.

To better understand the NFV framework, it can be divided into 3 main working domains, as in [6]:

- VNFs, software-implemented network functions running on top of the
- NFV Infrastructure (NFVI), which in turn includes all the physical resources and their virtualization methods, and finally the
- NFV Management and Orchestration (MANO), including the orchestration and management of hardware and software resources supporting the virtualization and the VNFs.

How these domains are connected is depicted in the NFV architectural framework in figure 1. The NFVI can be seen as the data plane of the network [13]. It consists of hardware resources, which are virtualized as virtual resources by the virtualization layer. The computing hardware is realized by general-purpose COTS compute nodes. Storage hardware consists of data storage devices divisible into for example shared network attached storage and server-specific storage. Finally, the networking hardware is a combination of switches, routers, and wired or wireless links. [6]

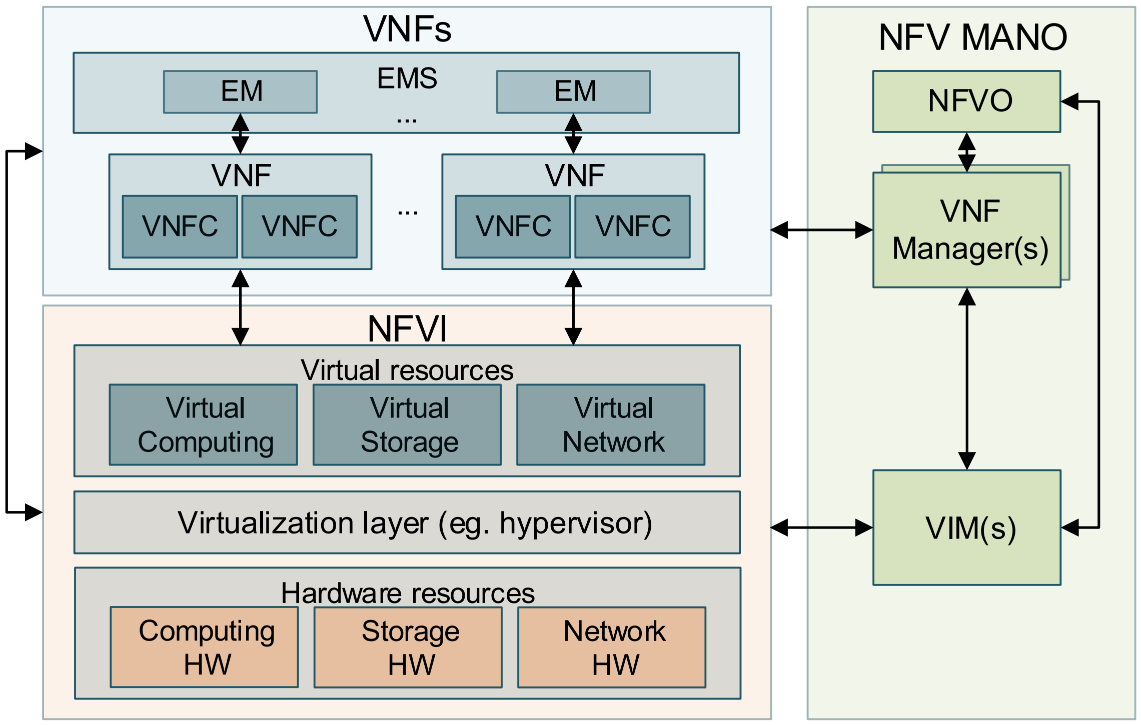

Figure 1. *The architectural framework of NFV. Adopted from [6]*

The virtualization layer decouples the hardware resources from the VNF software. The physical resources are abstracted and partitioned as virtual resources for the VNFs to use. A typical solution to provide the virtual resources is the usage of hypervisors, which in turn provide Virtual Machines (VMs). A VNF can then be implemented on one or several VMs. [6] Virtual resources are interconnected with typically software-based virtual networking, implemented with for example virtual switches [13]. Techniques such as virtual networks and network overlays, e.g. Virtual Local Area Network (VLAN) or Virtual Extensible Local Area Network (VxLAN) can be used to create virtualized paths to interconnect VMs and VNFs [6].

The VNF domain consists of VNFs and the Element Management System (EMS). The VNFs are software-implementations of network functions, providing the same functionality and external operational interfaces as physical implementations on dedicated hardware. The implementation of a VNF can be distributed to components (VNF Component, VNFC) on different VMs. The VNFs are managed by Element Managers (EM), which together form the EMS. [13]

In the NFV MANO, the responsibility of the NFV Orchestrator (NFVO) is the management and orchestration of the NFVI and its resources, and realizations of the networking services in it. The VNF lifecycle from instantiation to termination is managed by VNF managers (VNFM). A VNF is tied to a single VNFM, whereas a VNFM may manage several VNFs. Finally, the Virtual Infrastructure Managers (VIM) are responsible for resource management and monitoring of the NFVI, including tasks such as VM allocations to hypervisors, resource adjustments to VMs, and fault information collection from the NFVI. [6]

2.2 SDN

Whereas NFV focused on the separation of software and hardware, Software-Defined Networking focuses on the decoupling of the data plane from the control plane [14]. Conventionally, a network consists of separate network devices, which in turn are entities of tightly coupled hardware and software, performing both data and control plane functions (figure 2 a)). In SDN, the planes are separated by centralizing the control of the network onto SDN controllers, which configure the data plane according to rules set by network applications in the application plane (Figure 2 b)). [15]

The SDN survey in [14] describes a software-defined network by an architecture based on 4 principles:

1. Decoupled control and data planes, resulting in network devices becoming simple forwarding elements.
2. Forwarding rules are based on flows instead of destinations, i.e. sets of packet field values matched for a set of actions.
3. Control is centralized and moved to the SDN controller, or the network operating system (NOS), which is running on server hardware.
4. The programming of the network is done by applications on top of the NOS.

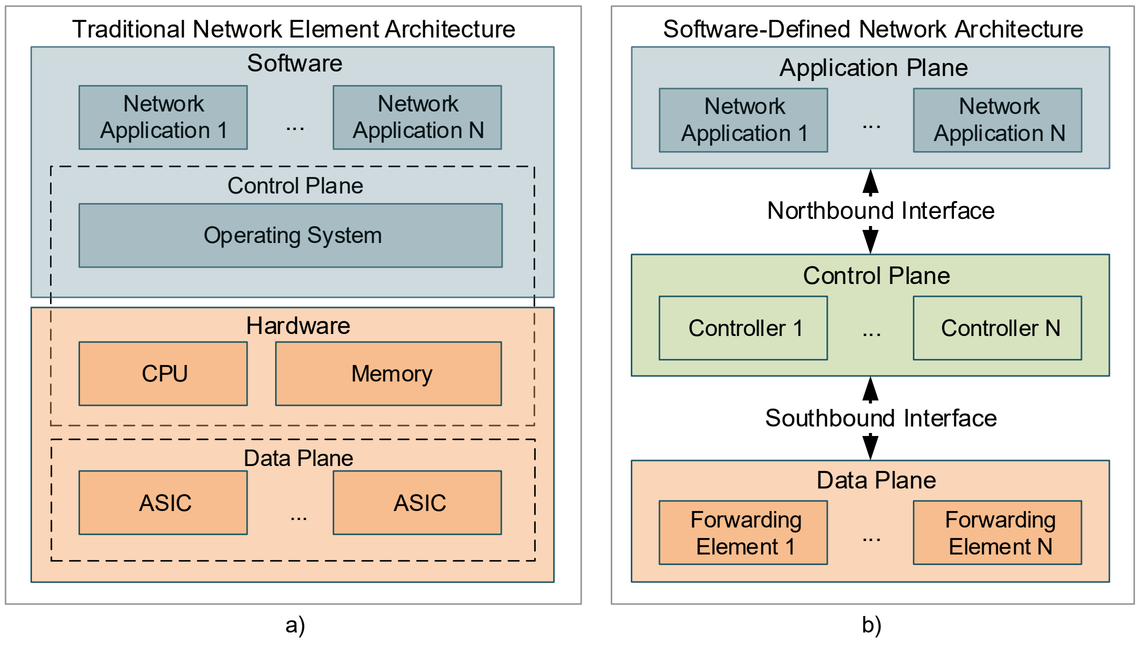

Figure 2. *The architecture of a) a traditional network element and b) SDN. Adopted from [15].*

The centralized control is purposed to maintain a global view of the network, and to provide a more abstracted model of the underlying hardware, enabling the use of high-level programming languages and software components. Through this centralization, state and information of the network is available to all applications, and the applications are less tied to location. The higher-level abstractions are also more shareable and reusable between applications. [14]

The decoupled planes in the SDN architecture, and the interfaces between them, can be seen in figure 2 b). In the bottom of the figure reside the forwarding elements, also referable as forwarding devices as in [14]. These devices, e.g. routers and switches [15], perform actions such as forwarding and dropping of packets, or header modifications [14]. These actions are taken based on flow rules, which are received from the control plane through the southbound interface [14, 15]. Interconnected by wireless or wired connections, together these devices form the data plane [14].

The control plane is the centralized intelligence of the network [14], consisting of a controller or controllers [15]. The controllers generate configuration rules derived from the application plane, and pass these rules to the data plane devices via the southbound interface [14, 15]. The southbound interface, implemented with for example OpenFlow [16], defines the communication protocol between the forwarding elements and the control plane [14].

On the top of the figure is the application plane, also referable as the management plane [14]. This plane consists of applications for example routing [15], quality of service mechanisms [15], firewalling [14] and load balancing [14]. These applications use an API, the northbound interface, and its functions to generate and deliver rules for network traffic treatment to the control plane [14].

3. HARDWARE ACCELERATION IN THE NFV CONTEXT

In NFV, networks gain flexibility, scalability and capital efficiency by replacing the proprietary hardware middleboxes with COTS hardware. However, this comes with a trade-off: using software virtualizations to run network functions in place of specially tailored ASICs (Application-Specific Integrated Circuit) can have a negative impact in throughput and latency [17]. To improve performance, whether the goal is in relation to e.g. cost, power, area, or to reach the sheer maximum, acceleration techniques can be introduced into the NFVI. The usage of specialized hardware to gain this performance improvement, is called hardware acceleration. [7]

Hardware accelerators can be such as custom ASICs, FPGAs, NPUs (Network Processing Unit) or GPUs (Graphics Processing Unit). A further classification for VNF hardware accelerators can be done by their type, and according to [18] these types can generally be divided in the following categories:

- in-line accelerators, which process packets in-line with software [7], as they traverse to or from the network, i.e. on the fly [18].
- look aside accelerators, which typically operate on data and commands submitted by software. Based on the command, the accelerator processes the data and sends a response. [7]

Look aside accelerators are typically associated with compute-intensive algorithmic acceleration, such as crypto or compression [7]. Compute-intensive functions characterize with the complexity and dynamism being in the calculations, while the processed data is more static, in relation to network-intensive functions. Network-intensive functions, e.g. network address translation and load balancing, have high throughput constraints and the data is dynamic, while the processing code itself can be relatively small. [19] Due to the data being mediated by a Central Processing Unit (CPU), look-aside acceleration can introduce higher latencies and more limited throughput by the CPU I/O in comparison to in-line acceleration [18].

Additional taxonomy for the hardware accelerators can be brought by their housing. The ETSI group specification for NFV acceleration [7] lists accelerator housings as

- integrated CPU, as in the accelerator (e.g. ASIC, GPU, FPGA) is implemented as a hardware function in the CPU socket

- iNICs (intelligent Network Interface Controller) or smartNICs, which are programmable and can be based around CPU or NPU cores (multicore system-on-chip-based, or an FPGA (FPGA-based). Additionally, the programmable cores on a SmartNIC can be accompanied by custom hardware blocks as acceleration engines. [4]
- bus attached, or
- network-attached, where the accessing is done over the network.

All of the above housings support both inline and look-aside acceleration, excluding the network-attached housing model, which is tied to only in-line acceleration in the group specification.

For the location of the accelerators, as in deployment models, generally two options for data centres can be identified [20] as in having the accelerators deployed in clusters, in centralized pools, or each server is coupled with acceleration hardware. Pooling of the accelerator hardware is a way of retaining uniformity in the core server infrastructure. On the other hand, from the perspective of the complete data centre, the homogeneity of the infrastructure is reduced. Whether to deploy the accelerators with each server, or in a subset of servers with the downside of more complex management and configuration is essentially a matter of cost-effectiveness. [20]

3.1 FPGA accelerators

An FPGA is a reprogrammable silicon device used to implement hardware circuitry. It consists of a certain amount of basic circuit elements, which are used and interconnected according to an architecture definition. Generally, the programmer writes this definition using an HDL, such as VHDL or Verilog. With automated tools, the hardware design is translated into a binary file, a bitstream, which, once loaded onto the FPGA, implements the circuit. As computational data paths are customisable and parallelization can be exploited, an FPGA can offer considerable performance gain in comparison to software implementations running on GPPs. [21]

The reconfigurability of an FPGA is a crucial feature in an accelerator, as cloudified environments come with a large variety of workloads changing in a fast pace. [20] With FPGAs, as opposed to ASICs, there is no manufacturing process as the functionality can be changed with a binary file, enabling more rapid design changes. [21] In comparison to another commonly used [20, 21] accelerator, the GPU, FPGAs are less demanding on size and power [20]. Besides, even though well-suited for their original purpose of video and image processing offloading, the GPU-provided performance gain in domains such as signal processing and ciphering is neglectable [22].

A common way of bringing FPGA accelerators to data centres is by tightly-coupling them with a host CPU in a co-processor manner [23] (figure 3 a)), most commonly on by using a daughter-card with a point-to-point connector such as PCIe (Peripheral Component Interconnect Express) [24]. A tighter coupling could be achieved by integrating the CPU and the FPGA on the same board for latency and memory access benefits [25], but the approach breaks server compute module homogeneity, power and size limits for a server board could be exceeded, and a fault on the CPU would lead to the waste of the FPGA resource [24]. The tightly-coupled option in general can be effective on local compute acceleration [8], but without network connectivity, the accelerators are more prone to under- or overutilization by their host CPUs [23].

By making the FPGA network-attached as in figure 3 b), it can be used as a standalone appliance essentially a peer processor in the network with CPUs [24]. Additionally, for more efficient hardware and software co-processing, the FPGA can be made both tightly coupled and network attached [23], as in figure 3 c). For example in [8], FPGA accelerators are network-attached and PCIe-connected to a server, enabling local compute acceleration via PCIe, as well as network acceleration and global acceleration. In network-acceleration, the FPGA can function as an in-line, bump-in-a-wire accelerator for tasks such as network encryption and deep-packet-inspection. In global acceleration, the FPGAs unused by their hosts can function as remote accelerators for large-scale applications, e.g. machine learning. [8]

Network-attachment also brings varying amounts of required logic on the FPGA, as some functionalities, such as protocol parsing [26], are essential on all network-attached devices. In for example [24], where the FPGA is a stand-alone network appliance, networking layer is done completely on the FPGA itself, removing the need for an external Network Interface Controller (NIC) and enabling the implementation of a protocols as demanded by the network environment. NIC functionalities, however, utilize resources which could be otherwise used for accelerator functionalities [8].

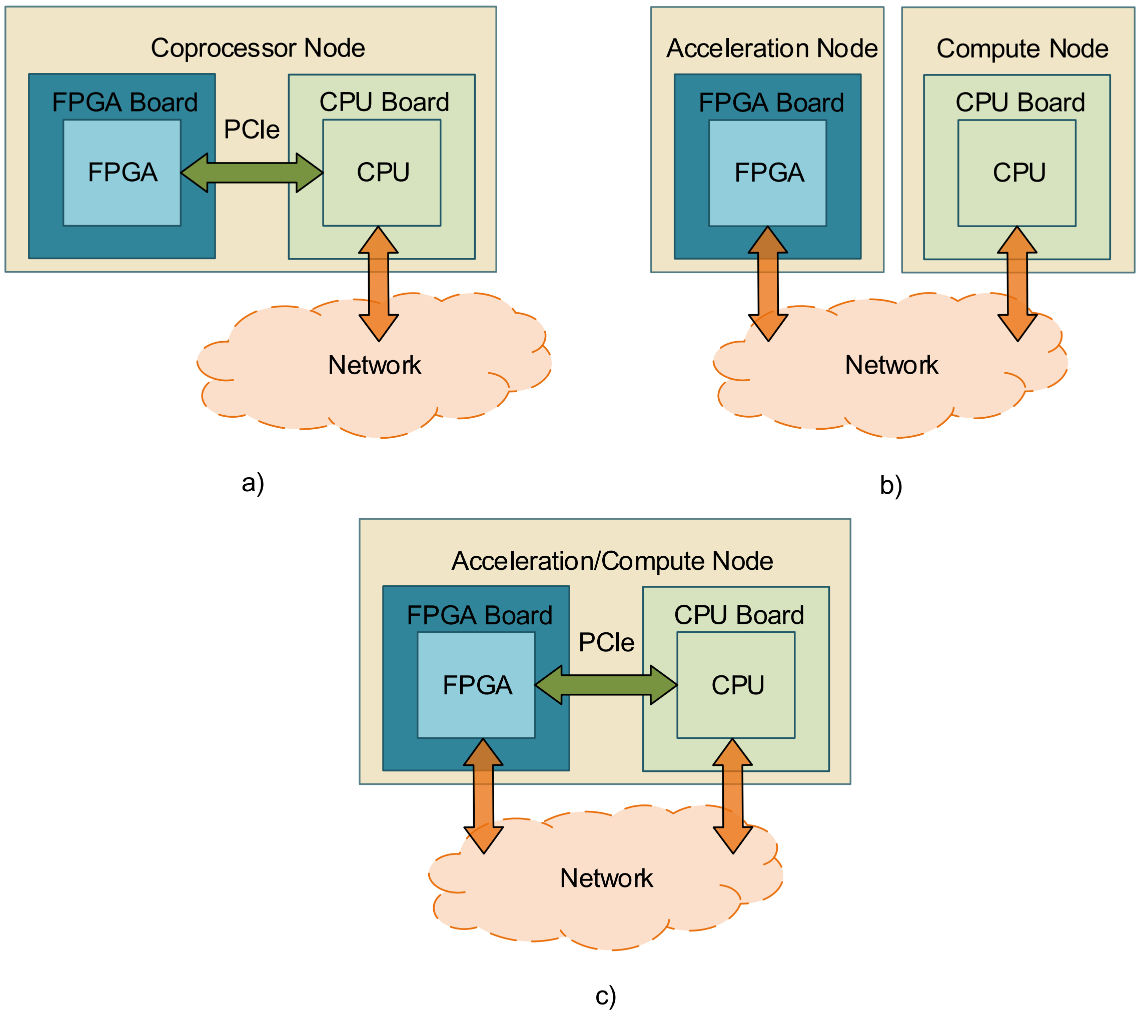

Figure 3. *FPGA accelerator attachment options: a) a tightly coupled coprocessor model, b) a network-attached, network appliance model, c) a tightly coupled and network-attached model. Adapted from [23].*

4. DATA PLANE PROGRAMMABILITY

A network device, be it a router, switch or a NIC, may have a varying amount of network functions to fulfil. These functions can range from switching and routing to for example firewalling, network telemetry, quality of service management and load balancing. [10, 15] Traditionally, the devices often implement these functions utilizing vendor-specific protocols, algorithms and interfaces, [15] in a way leaving the devices black boxes for the network operators [27]. Because of this, the operators dependent on the device vendor when it comes to device configuration, maintenance and re-deployment [15, 27]. Additionally, as compatibility between devices in larger networks needs to be ensured, the requirement of interface and protocol standardization can be seen as an obstacle for innovation, driving development time up alongside the cost [15].

At the same time as the operators are depending on the vendors, the network device vendors are facing the burden of implementing and supporting new functionalities on top of pre-existing ones, as per requests from the operators. New protocols and functions, such as new encapsulation methods in datacentre networks [28], are following the requirements set by evolving trends in for example 5G, machine learning and cloud computing [10]. With a rigid data plane, as in one implemented with dedicated hardware, adding these new features requires continuous development and manufacturing of new and increasingly complex devices. Additionally, an overly extensive supported feature set on a device might lead into unnecessary hardware resource utilization or performance degradation, should a specific deployment of the device not require it in its entirety [10].

Bringing programmability to the network devices to address these issues is not exactly a novel idea, as research on active networks was started in the 1990’s. Active network research was based on the idea of bringing the analogy of a programmable computer to networking, with usage of smart packets to program the network device for wanted functionality. [29] More recently, in the 2000s, the architecture standardization for SDN began with the Forwarding and Control Element Separation (ForCES) specification [30] by Internet Engineering Task Force and with OpenFlow [15, 16] . Both ForCES and OpenFlow stated, that the programmability in the network devices requires the decoupling of the control and forwarding planes, with an open, standardized API in between. For this interface in SDN, OpenFlow has gained the most traction [27].

With SDN, the OpenFlow provides a standard API to give the network control plane a possibility for configuring the data plane [16]. However, OpenFlow and the switches utilizing it are limited by a predefined set of supported protocols, and the programmability of the forwarding device is limited to flow-rule setting with a predefined set of actions [28].

4.1 Programmability to the forwarding plane: How?

One fundamental factor to the programmability and flexibility of the data plane is the choice of hardware platform in the network devices. SDN is targeted to utilize general-purpose hardware in the network for programmability [31], but at the same time, from the network parts, the data plane is the most heavily constrained by performance requirements [10].

The highest programmability comes with software switches implemented on CPUs, or GPPs. High abstraction levels in programming languages and design tools provide flexibility and speed in the implementation, at the cost of limited performance due to the general-purposed nature of the hardware architecture. [31]

Network processing units or Network Flow Processors (NFPs), similarly as GPPs, are hardware platforms for software switches. Unlike GPPs, these platforms are specifically designed with network processing in mind, e.g. by using dedicated accelerator units [32] and an architecture enabling processing parallelization. Utilization of this architecture, on the other hand, requires more specific programming than with GPPs, at the cost of flexibility. [31]

Moving to the domain of hardware switches, programmable logic devices, such as FPGAs, offer the highest flexibility [31]. These reconfigurable hardware devices enable parallel and pipelined processing with wanted functionality. As a downside, the available logic is limited per-chip, and compared to ASICs, these chips consume more power, come with a higher per-chip-cost, and are more limited in performance due to the general-purpose architecture.

Application-Specific Standardized Products (ASSP) are designed to implement functions commonly used or targeted for high-volume products. ASSP use cases can be for example physical and data link layer products [31] and switching fabric implementations [33] in network devices. Performance for the targeted function come with the downside of functionality configuration limitations. [31]

Performance-wise, the Application-Specific Integrated Circuits (ASICs) reside at the top of the curve. These devices are custom made for a certain purpose, designed for the applications where either the required features are outside the scope of standard products, or the performance requirements are too strict for programmable platforms [31]. Consequence from the custom application-specification, the ASICs have the poorest reconfigurability from the platform choices.

In reality, forwarding element devices are not strictly limited to a one certain platform, and the optimal trade-off between performance, cost and programmability can be also reached with hybrid platforms [10, 31]. A hardware switch appliance can for example use a CPU for functions less demanding on performance, and vice versa, a software switch could use external hardware components for efficiency. [10]

As seen in survey [10], one classification criteria for data plane programmability methods is the abstraction model of the data plane exposed to the control plane. These abstractions provide language constructs for an architectural model and means to configure the programmable target with. The survey identifies three common abstractions for the data plane:

- The data flow graph abstraction,
- the match-action pipeline abstraction, and finally,
- the hybrid-switch abstraction.

Data flow graph abstractions are based on division of processing logic into smaller entities, nodes, which are connected by edges. This abstraction model lets the programmer choose and connect the processing functions modularly, in the wanted order. An example of a data flow graph software switch architecture is Click [34]. Click implements the processing nodes of a software switch, called elements, as C++ objects, which are interconnected with pointers, called connectors.

The match-action pipeline abstraction, as used in for example OpenFlow [16], P4 [28], Protocol-Oblivious Forwarding [27] and Domino [35], describes the packet processing logic with lookup tables containing flow rules. The lookup keys, based on the processed packets' protocol headers, are matched for values in these flow tables. These values are stored together with corresponding actions, which determine the following processing steps, such as additional lookups on another flow tables or dropping of the packet. Thus, the configuration of the packet processing functionality is done by managing the entries in the lookup tables.

The last of the abstractions presented in the survey is the hybrid-switch abstraction, used by architectures combining features from both data flow graph and match-action abstractions [10]. One presented example application falling into this category is the disaggregated Reconfigurable Match-Action Table [36].

4.2 P4 brought into focus

P4, a high-level programming language, receiving its name from its intended use of Programming Protocol-independent Packet Processors, is defined to describe the data plane packet processing logic of a forwarding element [11]. As stated in the original P4 paper [28], it is designed around three main goals:

- Reconfigurability of the packet parsing and processing logic, post-deployment, in the field.
- Protocol independence through a protocol header stack defined by the control plane, alongside the parser extracting these headers, and the set of match-action tables to process them.
- Target independence by a high-level functionality description, leaving the generation of a target-dependant program for compilers.

P4 is a domain-specific language, and provides the match-action pipeline abstraction of the data plane of a forwarding element with programs containing the following main elements [11, 28]:

- Header definitions,
- parsers,
- match-action tables,
- actions,
- control programs.

All of the protocol headers accessed in the program are defined with their set, order, and bit widths of fields. How these headers can be sequenced, and how the sequences are identified, is defined in the parser. The parser also defines which of these headers are extracted from the packets. [11, 28]

Program 1 presents a simple example snippet of a parser state machine definition, based on the P4 programs used in this work and written according to the P4 specification [11]. Keywords reserved by P4 are bolded. The parser declaration with its interface starts on line 1, where the input is a P4 core library extern object packet_in [11], and the output is a user-defined struct of headers. Lines 3 to 10 describe the initial state, where, in this example, the Ethernet header is extracted from the packet and based on the value of the type-field of the header the next state is chosen. On line 8, the default transition is defined to be accept, which results into ending the parsing in the initial state for unmatched packets. Functionality of other states on lines 11 to 17 are abstracted away from the snippet. The match-action unit abstraction is provided by the tables and the actions tied to these tables. The tables are defined with lookup keys, which can be header fields or other values calculated in the P4 program, and actions which are executed based on the matches in the table. The actions are functions, which may have optional input parameters from the table. Tables and actions are contained inside control programs, which determine the order of execution of the match-action units. Additionally, the re-assembling of the packet, deparsing, can be defined in a control program. [11]

```
    parser MyParser(packet_in pkt, out hdrs_s hdr) {
 2
      state start {
 4      pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.type) {
 6        0x0800  : parse_ipv4;
          0x8100  : parse_vlan;
 8        default : accept;
        }
10    }
      state parse_ipv4 {
12      … // state transition rules
      }
14    state parse_vlan {
        … // state transition rules
16    }
      …
18  }
```

Program 1. *Example program of parser declaration in P4.*

Program 2 presents an example snippet from a control program including a match-action unit, again based on the P4 programs used in this work, written according to the specification [11]. Line numbers 9 to 16 define the lookup table. Lookup key is set as a source-field from an Ethernet field, which in turn belongs to a user defined header struct (hdr). Matching method is chosen as exact match, but P4 core library additionally supports longest prefix and ternary matching with “don’t care” bits [11]. Actions tied to each table value are declared on lines 11 to 14. Each value in the table will trigger either of the two actions, “Forward_pkt” or “Drop_pkt”, with the latter one being also declared as a default action on line 16. The default action is triggered if the lookup results in no match. Maximum amount of entries in the table is defined on line 15 with the “size” parameter being set to 1024.

```
   action Forward_pkt (bit<4> route_id) {
 2   md.route_ID = route_id;
   }
 4
   action Drop_pkt () {
 6   md.route_ID = DROP_ID; //constant
   }
 8
   table Eth_match {
10   key     = { hdr.ethernet.src : exact; }
     actions = {
12         Forward_pkt;
           Drop_pkt;
14   }
     size = 1024;
16   default_action = Drop_pkt;
   }
```

Program 2. *Example of a match-action unit declaration in P4.*

Actions definitions are on lines 1 to 7. The action “Forward_pkt” has been defined with an input parameter, which is received from the table, set by control plane. This received parameter is set as the value of a user-defined metadata (md) field “route_ID”. Action “Drop_pkt” similarly sets the value of the metadata field, in this case to a value “DROP_ID” depicting a user-defined constant.

4.2.1 P4 targets and compilers

As a domain-specific and target-independent language, P4 is designed to be targetable for both software switches and hardware platforms such as NICs, FPGAs and ASICs. [11] To produce an actual target-specific data plane configuration and a control plane API, an implementation framework, architecture definition and a target-specific P4 compiler is required from the target manufacturer [11]. An open-source reference compiler is available in [37], designed as modular to provide a standard front end compiler to be combined with a platform-specific back end compiler.

An example of a software switch target for P4 is PISCES [38]. It is based on the OpenFlow-enabled Open vSwitch (OVS) [39]. OVS has gained wide use in data centres, running inside a hypervisor and switching traffic among virtual and physical interfaces. PISCES prototype brings protocol-independency to the OVS by three main modifications, possibly required by a P4 program, and a P4-to-OVS compiler. The modifications are the addition of arbitrary encapsulation and decapsulation with new header adding and removal primitives, conditional action executions, and checksum optimizations. The PISCES compiler compiles a P4 program into OVS C code, with the parse, match and action codes replaced according to the P4. This modified OVS can then be compiled with a C compiler into switch binary. [38]

P4-programmable smartNICs can be found from Netronome. Their Agilio class SmartNICs are based around Netronome NPUs. [40] The compilation process uses the open-source P4 front end compiler together with a Netronome back end compiler for a target-specific C implementation of the data path. Finally, firmware for the SmartNIC is generated from these C files and downloaded to the device. [32]

On the FPGA side, several P4 compilers and projects exist [41-44]. SDNet is a design environment from Xilinx, which, supported with a P4 back end compiler, compiles P4 descriptions into packet processor IPs for Xilinx FPGAs. In addition to the IP, the tool generates a testbench for simulations.[41]

P4FPGA is an open source compiler and runtime, presented in [42]. Similarly as SDNet, it is targeted for generating HDL code for FPGAs from P4. P4FPGA uses the P4 front end compiler for an intermediate representation, which is then compiled with the P4FPGA compiler into a Bluespec SystemVerilog (BSV) representation. The P4FPGA BSV-based runtime includes support for external IPs and management units for transceivers and host communication, enabling FPGA-targets from multiple vendors. Additionally, the P4FPGA generates a C++ based API for table management and debugging. [42]

Examples of ASIC targets for P4 programming are the switching ASIC Barefoot Tofino [45] and RMT [46]. Leveraging the common abstractions in the chip, a P4 compiler targeting the latter is presented in [47].

5. METHODOLOGY

This work evaluates the feasibility of data plane programmability in network-attached FPGA accelerators. More precisely, the evaluation is fixed on evaluating the usability of P4 language in the design of protocol processing blocks on an accelerator FPGA, in comparison to implementing such functionality directly in RTL. Surrounding the actual accelerator Intellectual Property (IP) on the FPGA, these blocks are responsible for networking related tasks, such as protocol parsing and header modifications.

For the evaluation, an accelerator shell design was implemented on the FPGA. The top level in the design hierarchy consists of an accelerator wrapper for the acceleration functions, and the shell design providing connectivity, packet processing, and routing functionalities. Three variants of the shell were implemented, each having a different implementation of the packet processors, yet providing the same data plane functionality. These different implementations were

1) a P4 implementation with a software interface for dynamic table updates,
2) a hard coded VHDL implementation without a software interface, and finally, for more accurate comparison with the VHDL variant,
3) a hard coded P4 implementation without a software interface.

The shell designs with P4 implementations were compared against the shell design with the VHDL implementation in terms of utilization of FPGA resources, performance, and degree of automation in the design flow.

The FPGA has a fixed amount of programmable logic and on-chip memory, and therefore the utilization of resources by the shell dictates the resources available for the main acceleration functionality. To maximize the logic available for the accelerator, the shell design should be aimed to provide the wanted functionality with the lowest utilization percent possible. The utilization is measured as the use of Lookup Tables (LUTs), Flip-Flops (FFs), random-access memories (RAMs) implemented with LUTs (LUTRAMs), and on-chip block RAM memory tiles (BRAMs).

The performance is measured with two parameters, latency and throughput. Latency is measured as time between the moment the chosen test data is sent out from a traffic generator and the moment it has been processed by the Device Under Test (DUT) and is received back at the traffic generator. More specifically, latency is measured as cut-through latency, meaning the measurement begins when the first bit of data is sent out and ends as the first bit is received back. While latency of a network-attached device is generally desired to be as minimal as possible, throughput, on the other hand, is most feasible as maximal. Throughput is the amount of data the DUT can process in a time unit, measured by bits per second in this work.

While being a less quantitative parameter, the degree of automation in the design flow still provides a valuable insight to the feasibility of an implementation method. As the design flows differ between the three implementations, the degree of automation can be measured by comparing which of the intermediate steps between a design specification and an implemented design in each variant is automated, and by how much, in terms of lines of code required.

5.1 Hardware design methodology

The VHDL-implemented packet processors developed in this work were manually written, and verified with an in-house developed Universal Verification Methodology (UVM) [48] testbench. For the P4 implementations, the packet processor design and verification steps were done with the Xilinx SDNet [41], which is a design environment providing a back end compiler for P4 designs targeting Xilinx FPGAs. Additionally, the tool provides a flow for RTL simulations with user-provided stimuli, and therefore no user-made testbenches were required. For more reference on the tool, newer SDNet documentation is available by contacting Xilinx.

Packet data leveraged in the verification of all packet processor variants was generated and captured with a traffic generator, TRex version 2.45, from Cisco [49].

The integration of the packet processor components to the top-level FPGA design, and the synthesis, implementation, and generation of bitstreams, was done by using the Vivado Design Suite [50].

5.2 Test and measurement setup

The system used in the FPGA design comparisons and measurements (figure 9) consists of three main hardware components: the FPGA, the host CPU, and the traffic generator.

The FPGA used in this work was a Xilinx VU9P Virtex Ultrascale+, attached to a Xilinx VCU1525 PCIe development board. More detailed description of the board can be found in [51]. The FPGA has three interfaces in use: an Ethernet interface for packet data, a

JTAG (Joint Test Action Group) interface used for programming and resetting the FPGA, and a PCIe interface for runtime dynamic configurations from the host CPU.

The test data traffic was generated by an IXIA NOVUS-r100GE8Q28 load module. A single port from the traffic generator was connected to one of the VCU1525 board transceivers with the configured link speed of 10 Gbps and the IXIA software used for test configurations was IxNetwork.

The host CPU has a 64-bit x86 architecture and is running a release 7.4.1708 CentOS Linux operating system. With the software interfaced P4 design variant, a C language program code was compiled into an executable user application, which uses a Xilinx PCIe driver for PCIe access to the FPGA. These PCIe accesses are table initializations and updates.

A Lab Edition of Vivado Design Suite was used on the host CPU for programming the FPGA with JTAG connection. The JTAG was converted from an USB connection by the USB to JTAG converter of the VCU1525 board. Vivado was additionally used for controlling the system reset on the design with a Virtual I/O [52], and monitoring ILA (Integrated Logic Analyzer) debug cores [53] in the testing phase. The ILA cores were removed before the utilization measurements.

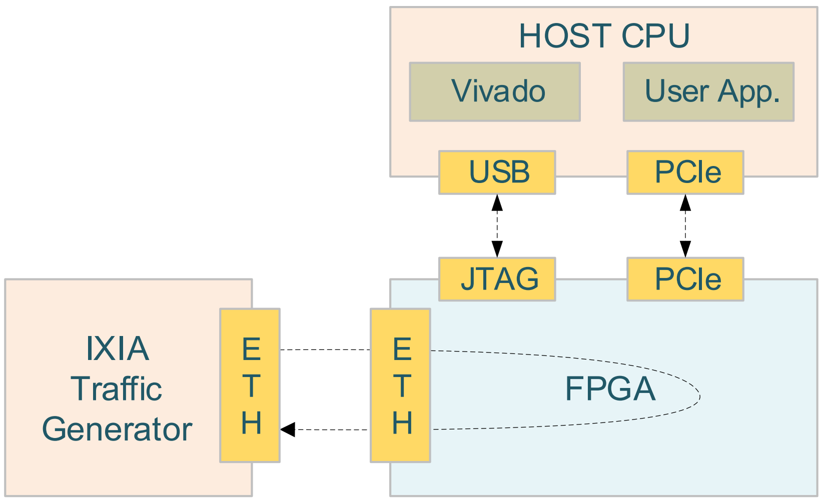

Figure 4. *Testing and measurement system description.*

Two different base traffic items, originally generated with TRex,. were used in the testing:

1) a packet with a maximum supported header stack of outer Ethernet, outer VLAN, outer IPv4, outer UDP, VXLAN, inner Ethernet, inner VLAN, inner IPv4 and inner UDP, and
2) a packet with the minimum supported header stack of inner Ethernet, inner IPv4, and inner UDP.

These two different packet types were chosen to expose possible performance differences between design variants caused by the varied set of headers parsed and encapsulation and de-encapsulation processes.

The performance measurement method used was an implementation of the RFC 2544 Throughput/Latency test [54] by IXIA, with frame sizes of 74, 512, 1024 and 1522 bytes with the minimal header stack, and 136, 512, 1024 and 1522 bytes with the maximal header stack. IXIA requires a certain amount of payload for measurement-related tagging of the packet, hence the difference in the smallest possible frame sizes with different header stacks. The latency was measured as average, minimum and maximum cut-through latency, and maximal throughput was tested by incrementing the transmission rate from 10% of the maximum line rate of 10Gbps, until a frame loss threshold of 0 was crossed. Test duration for each transmission rate was 20 seconds.

6. SYSTEM DESCRIPTION

This work was done as a part of a larger in-house developed CRUN project. The project was based around a concept of a cloud architecture with virtual overlay networks on top of a physical network, consisting of server nodes interconnected by switches. These server nodes were attached with virtualizable hardware accelerator FPGAs. This chapter describes the CRUN architecture, the top level of the FPGA design architecture used as the evaluation framework, and the specific use-case the packet processors in the design are designed for.

6.1 CRUN framework

This section briefly presents the CRUN in a top-down manner, starting from the network architecture and then the server architecture, presenting the framework for the FPGA design, which is the main subject of this thesis. The layered networking scheme of the framework is presented in figure 4. The two main layers depicted are

- the physical underlay network, consisting of the hardware components building the cloud infrastructure, and
- the overlay network, a virtual network consisting of virtual machines and accelerators, interconnected in VxLAN segments.

Virtualized network functions are provided by VMs, accelerators and their combinations. These VMs and accelerators are physically located on hardware server nodes, but virtually contained and connected in overlaying virtual networks. For example, a VM on a server node 1 might operate together in the same virtual network with an FPGA accelerator unit on a server node 2, together providing a VNF. Their connection is established over a virtual tunnel, with the virtual tunnel endpoints residing on the server node 1 hypervisor and the FPGA networking logic on server node 2. In the underlay network, the server nodes are connected to switches via NIC and FPGA Ethernet transceivers, forming a physical cloud infrastructure connection.

Figure 5 presents the CRUN server architecture, depicting its most relevant components. The main hardware components are the host GPP, the PCIe-attached NIC, and the optional PCIe attached FPGA accelerator.

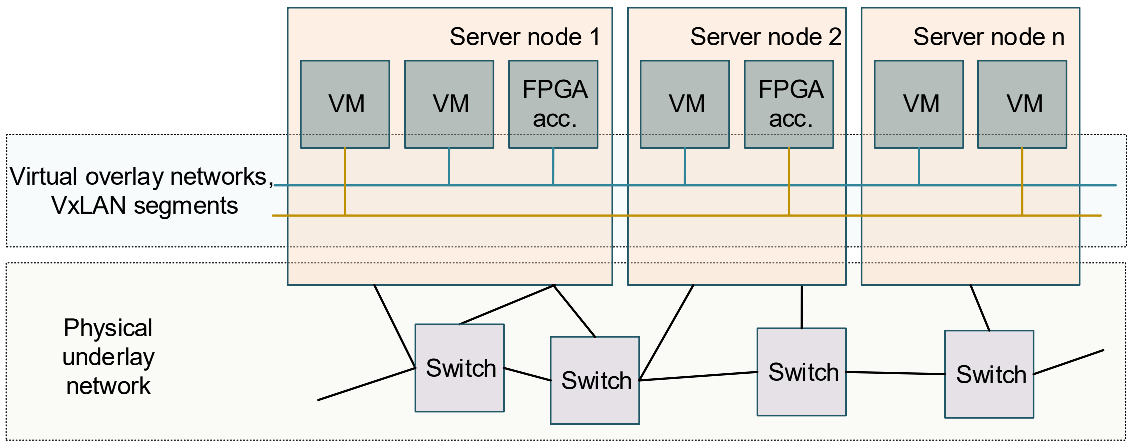

Figure 5. *Network architecture in CRUN.*

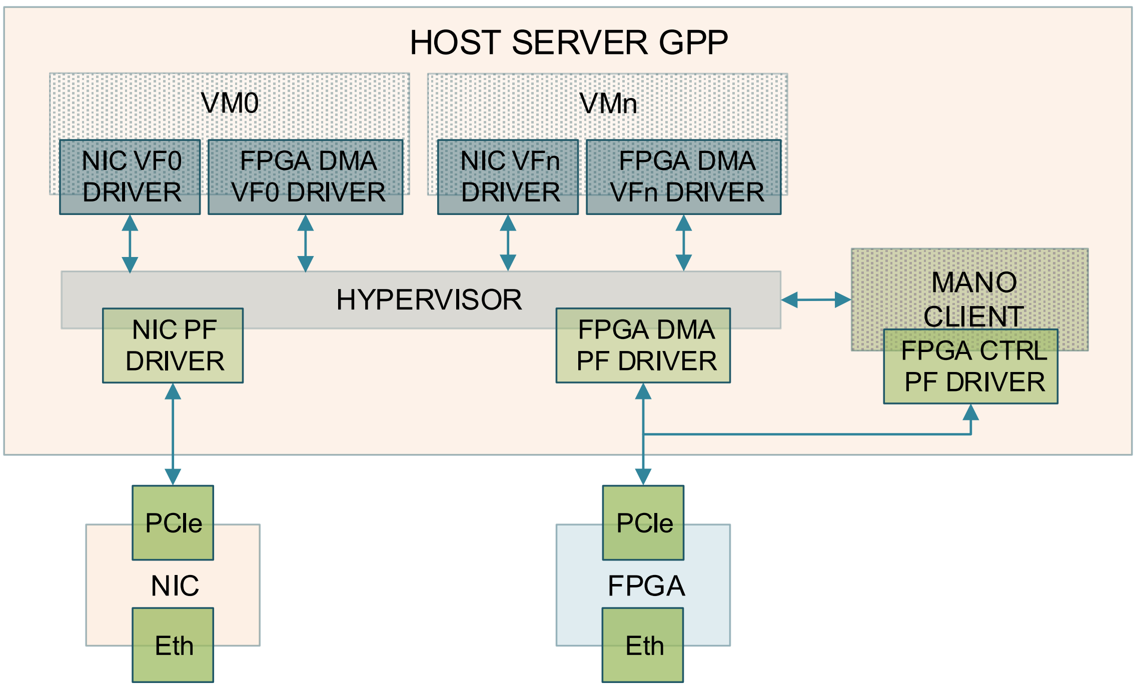

Figure 6. *Server architecture in CRUN.*

The host GPP is running a hypervisor and management and orchestration client application on top of a host Operating System (OS). The hypervisor creates and manages VMs and utilizes SR-IOV (Single Root Input/Output Virtualization) to provide the VMs virtual functions to access the PCIe physical functions. The hypervisor is in turn managed by the MANO client application.

The MANO client is managing the host OS, the hypervisor and the FPGA, and is a part of a larger centralized management application. A more detailed description of the management and orchestration software used in CRUN is given in [55].

The FPGA, attached to the host GPP via PCIe, is an optional hardware accelerator component. Its main data plane interface is the Ethernet network interface, but CRUN defines an additional possibility of Direct Memory Access (DMA) to and from VMs on the host server via PCIe. The main purpose of the PCIe connection, and the only purpose, in the scope of this work, is to provide a control plane interface for flow control of the networking logic on the FPGA. The FPGA itself is functioning as a network-attached hardware accelerator.

The CRUN project also set performance requirements for the FPGA accelerators:

- The FPGA networking logic cannot limit the throughput of the accelerator and it must operate at line rate.
- The FPGA networking logic latency must be in the scale of microseconds, with the maximum of 8μs.

The maximum latency for the networking logic originates from a neural network inference acceleration trial done with CRUN, presented in [56]. A goal in the trial was to achieve an ultra-low latency of 20μs to 40μs in the software level. In the trial, this set the requirement for the latency on the FPGA hardware between 10μs and 30μs. The largest latency on the FPGA came with the baseline implementation of the neural network, itself causing a latency of roughly 22μs. This left the networking logic with 8μs to meet the maximum limit of 30μs hardware latency.

6.2 Top level FPGA design

The top level of the FPGA design used in this work is depicted in the figure 6. The implementation, excluding the accelerator, is designed to operate at line rate and with minimal latency. Throughout this thesis, the design excluding the accelerator is generally referred to as the shell design. In figure 6, the blocks independent of the shell design variant are coloured as orange. The software interface (SW IF) section, consisting of the PCIe DMA IP and the AXI4-Lite interconnection (AXI4-Lite XBAR) and its related AXI4-Lite interfaces are present only in the full P4 implementation with the software interface, whereas the packet processor blocks (PKT PROC. INGRESS and PKT PROC. EGRESS) coloured with blue are present in each variant, but with different implementations.

The data plane interface in and out of the FPGA is implemented with a Xilinx Ethernet Subsystem (Eth-SS) IP [57] for receive (RX) and transmit (TX) operations. On the RX side, the IP converts the incoming physical layer (PHY) traffic into AXI4-Stream (AXI4-S) protocol. The IP is configured for the speed of 10 Gigabits per second, with the clock frequency of 161MHz, and a data bus width of 64 bits. This bus width is used in the AXI4-

S interfaces throughout the design, and the majority of the design operates in the same clock domain, with only the SW IF segment functioning within a clock domain dictated by the PCIe DMA IP.

The ingress packet processor parses the incoming packets' headers and does possible lookup operations and header modifications before forwarding the data to the packet router (PKT RTR) as AXI4-Stream traffic, alongside a metadata bus carrying possible packet-specific information for following blocks, as well as a route identifier (ID).

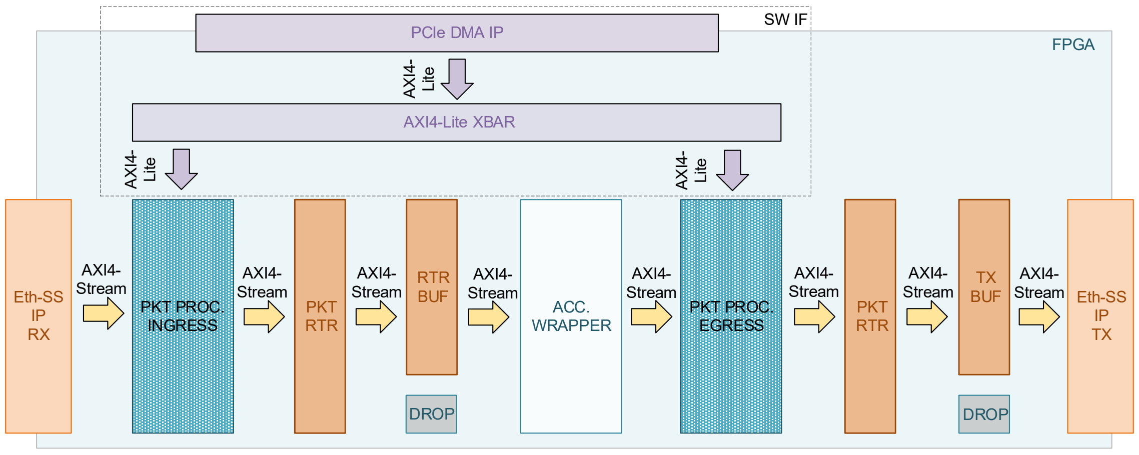

Figure 7. *Top level accelerator shell design implementation on the FPGA.*

The packet router connects the AXI4-Stream interface to either the router buffer (RTR BUF) or drops the packet, based on the received route ID. The RTR BUF component is a BRAM FIFO (First In, First Out) buffer generated with the Vivado FIFO Generator IP [58], and is purposed for migitating backpressure from the accelerator, ensuring that only complete packets are forwarded.

The accelerator wrapper, in this work, does not contain any acceleration logic, and its entity consists of direct connections from its inputs to outputs, as the evaluation is focused on the shell design.

The egress packet processor does use-case dependent protocol parsing and header modifications, and forwards the packet to the output packet router. This packet router and the buffer component connected to it are instances of the same IPs as the packet router and the FIFO buffer described earlier.

Finally, following the buffer, the outoing packet is converted from AXI4-Stream into PHY by ETH-SS IP TX, and sent out from the FPGA.

For the control plane, the FPGA is connected to software via PCIe, with the Xilinx DMA (XDMA) IP [59] converting the PCIe procotol into AXI4-Lite protocol. The AXI4-Lite interfaces are operating in a clock domain of 125MHz. The XDMA AXI4-Lite interface is connected to the ingress and egress packet processors with an AXI4-interconnect IP [60], which maps certain address segments into separate AXI4-Lite interfaces. This design segment is only present in the P4 implementation with the software interface, and is used there for dynamic table updates for the lookup engines in the packet processors.

6.3 Use case description

The chosen application for the packet processors is VTEP termination, meaning VxLAN decapsulation in the ingress, and VxLAN encapsulation in egress. In addition to this, the packet processors are inspecting and modifying the protocol headers underlaying the VxLAN encapsulation. This depicts the accelerator to function in VM, or VNF context, while the broader system is operating in cloud infrastructure context. More about VxLAN can be found in [61].

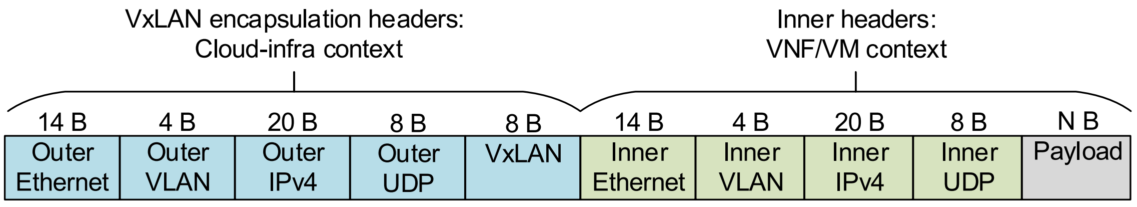

Figure 8. *Protocol stack supported by the packet processors. Fixed header lengths in bytes (B).*

The set of networking protocol headers supported and processed by the packet processors are visible in the figure 7. For the data (payload in figure 7) to reach the accelerator in the design, it must be included in a User Datagram Protocol (UDP) packet. This UDP packet must in turn be encapsulated in an Internet Protocol version 4 (IPv4) packet and contained inside an Ethernet frame with an optional VLAN tag. In figure 7, the headers of these protocols are shown as the inner headers. In case of VxLAN encapsulation being present in the input packet, these inner headers are further encapsulated by a VxLAN header, an outer UDP header, an outer IPv4 header and an outer Ethernet header with an optional VLAN tag. In the following sections, the operations of the packet processors are divided into two contexts: The Cloud Infrastructure (Cloud-infra) context, which includes the operations related to the VxLAN encapsulation headers, and the VNF (or VM) context, which includes the operations related to the inner headers.

The processing flow of the packet processors is described in the figure 8, where the figure 8 a) describes the ingress operations, and 8 b) the operations in the egress packet processor.

The ingress side operation begins with protocol parsing, starting from the outer headers. Once the parsing reaches the UDP header, the destination port field is checked. If the field holds the value 4789, the parsing continues to VxLAN and inner headers. If the VxLAN and inner headers are parsed successfully, the processing starts in the cloud infrastructure context. If the outer UDP destination port value differs from 4789, the parsing ends and the processing starts directly in the VNF context.

In cloud infrastructure context, the VxLAN header field Virtual Network Identifier (VNI) is matched for VNI values in the on-chip memory. If a match is found, the processing moves to VNF context, and if not, the packet is dropped.

In VNF context, a five-tuple lookup operation determines whether the packet is forwarded or dropped. This five-tuple value consists of the values of the following header fields in the innermost headers: IPv4 protocol, IPv4 source and destination addresses, and UDP source and destination ports. If the lookup operation returns a match, and the five-tuple value is found in the on-chip memory, the packet is forwarded. If the packet reached the VNF context through the cloud infrastructure context, the VxLAN encapsulation headers are removed, and a metadata signal is sent out with the packet, indicating that the original packet is to be VxLAN encapsulated in the egress.

The egress processing, similarly to ingress processing, begins with protocol parsing. If the parsing of the inner headers is successful, the processing checks the metadata for information if the original packet in ingress had VxLAN encapsulation. If the inner headers are not parsed successfully, the packet is dropped.

If the metadata indicates that VxLAN encapsulation was indeed present in the original packet received in ingress, a lookup operation is initiated with the source address of the Ethernet header. These source addresses are saved as keys in the on-chip memory as key-value pairs where the return value is a set of outer headers (Ethernet, VLAN, IPv4, UDP and VxLAN). If the lookup results in a match, these returned headers are used to re-encapsulate the packet. If the on-chip memory is missing a configuration for a particular Ethernet source address, and a match is not found, the packet is dropped.

Finally, if the packet has not been set to be dropped, the innermost headers are modified with a swap operation. This operation swaps the Ethernet source and destination addresses, IPv4 source and destination addresses, and the UDP source and destination ports.

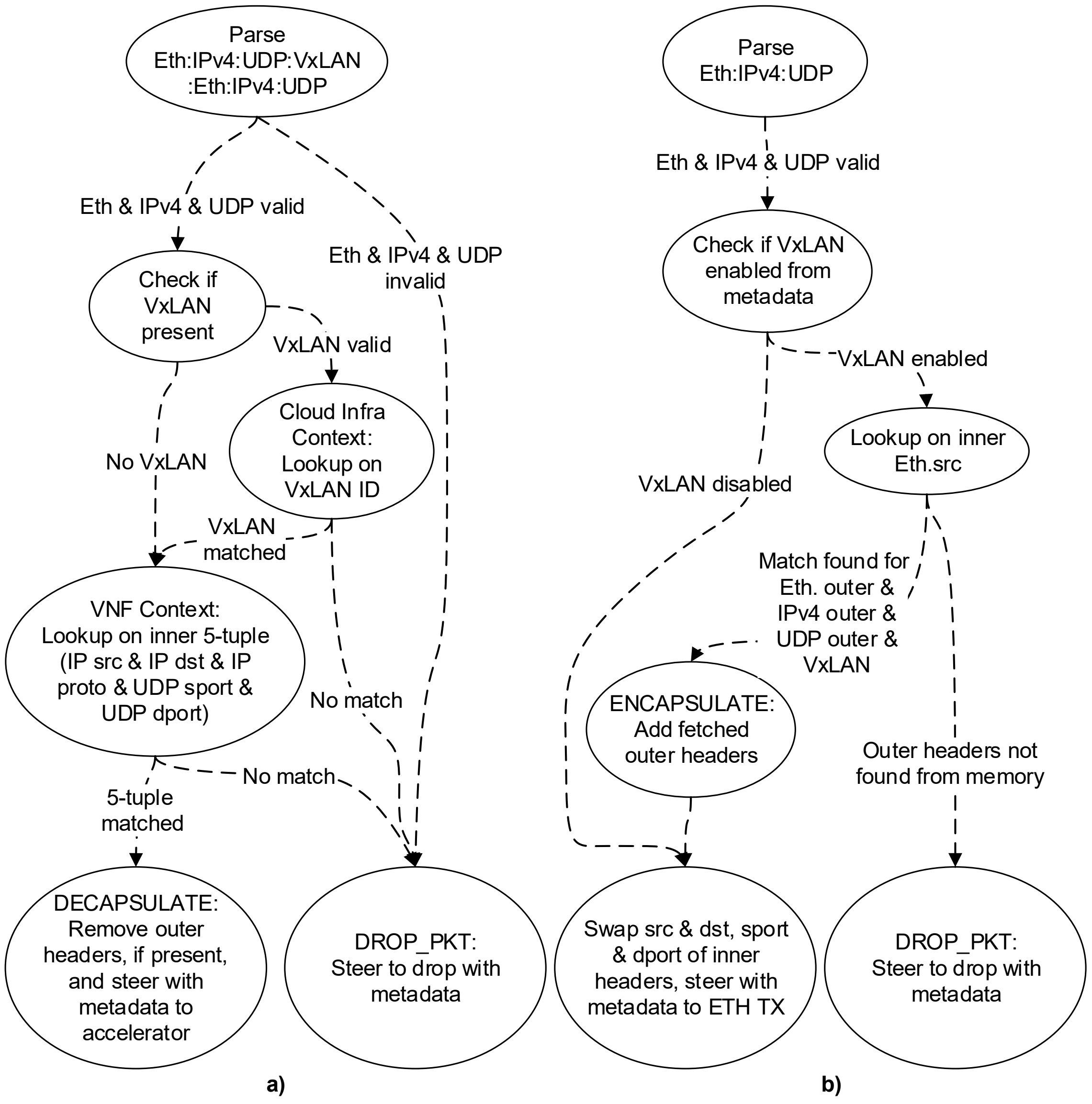
Parse Eth:IPv4:UDP:VxLAN :Eth:IPv4:UDP
Eth & IPv4 & UDP valid
Eth & IPv4 & UDP invalid
Check if VxLAN present
VxLAN valid
Cloud Infra Context: Lookup on VxLAN ID
No VxLAN
VxLAN matched
VNF Context: Lookup on inner 5-tuple (IP src & IP dst & IP proto & UDP sport & UDP dport)
No match
No match
5-tuple matched
DECAPSULATE: Remove outer headers, if present, and steer with metadata to accelerator
DROP_PKT: Steer to drop with metadata
a)
Parse Eth:IPv4:UDP
Eth & IPv4 & UDP valid
Check if VxLAN enabled from metadata
VxLAN enabled
Lookup on inner Eth.src
VxLAN disabled
Match found for Eth. outer & IPv4 outer & UDP outer & VxLAN
ENCAPSULATE: Add fetched outer headers
Outer headers not found from memory
Swap src & dst, sport & dport of inner headers, steer with metadata to ETH TX
DROP_PKT: Steer to drop with metadata
b)

Figure 9. *Packet processing flow diagram for a) ingress and b) egress packet processors.*

7. IMPLEMENTATION

This chapter describes the hardware design steps from specification to testing which were gone through in this work and presents the design architectures which were created to implement the functionality based on the use case description in chapter 8.2. Figure 10 pictures the design steps on a generic level to apply to all three design variants.

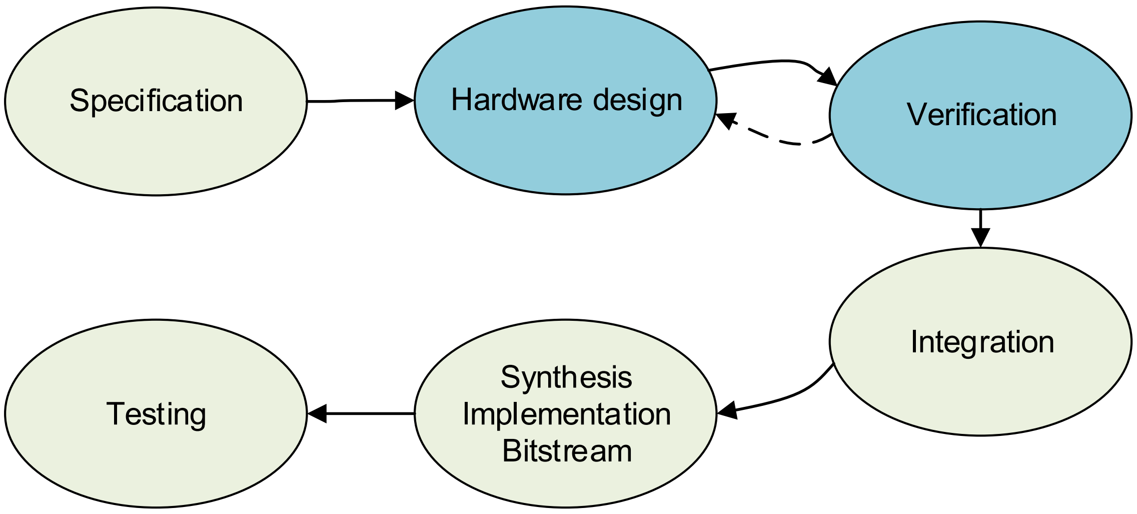

Figure 10. *Generic flow graph of the design process for all packet processor variants.*

The specification step roughly translates into the process of specifying the use case, describing the main functionality, parse trees and IP interfaces. The following steps, production of RTL and verification, are executed with different methods, depending on whether the design is implemented with P4 or VHDL. These steps are described in the subchapters 9.1, 9.2 and 9.3 for VHDL, P4 with the software interface, and hard-coded P4 implementations respectively. A common sub-step for all design variants in verification was packet stimulus generation. All three implementations were using packet capture (pcap) files as test stimulus.

The integration step execution is similar between all design variants. The packet processors were integrated to the surrounding shell architecture by connecting interfaces in a VHDL top level entity.

As the design includes multiple Xilinx IPs, and the target platform was a Xilinx FPGA, the synthesis, implementation and bitstream generation were done using Vivado Design suite [50] from Xilinx. In the Vivado tool flow, the implementation step essentially means mapping the synthesized design into actual hardware resources available on the FPGA, and then placing and routing the design. Vivado was also used to insert ILA debug cores into the design. These debug cores could then be connected to probe certain signals, and thereby further verify correct functionality on the top level of the design in the testing phase.

In the testing phase, the target FPGA was programmed with the generated bitstream. With the software interfaced P4 design variant, a C code was prepared and compiled into an executable for hardware register access for table population. The design was then tested with the IXIA traffic generator by transmitting varying types of packets to the FPGA. The rebound packets were then inspected to verify the wanted functionality. Error cases were investigated with the hardware debuggers and resolved on the RTL. The hardware debuggers were removed after the design was deemed functional for more exact utilization reports.

7.1 VHDL implementation

In the VHDL implementation, the production of the RTL description was manual. This made the hardware design process lengthier, but at the same time more accurate in comparison to a higher-level P4 language implementation. As the RTL description was written, it was frequently simulated with a UVM testbench. The test stimulus was generated from pcap files, and the DUT output was checked against another pcap file presenting the wanted output. Three different premade pcap files were used:

- one representing the input data for ingress packet processing received from ETH RX,
- one representing the data at the output of the ingress packet processor, and
- one representing the output data at the output of the egress packet processor.

This enabled automated verification of the DUT output against the reference pcap files in three different stages of the design:

1) complete functionality of the ingress packet processor implemented,
2) complete functionality of the egress packet processor implemented,
3) complete functionality of both packet processors implemented.

The intermediate stages of the design were simulated and verified only visually from simulation waveforms.

Figure 11 describes the implemented ingress packet processing architecture. The design functions in line rate and handles back-pressure by back-propagating it. The parser is designed to function in a pass-through manner, i.e. it is inspecting and extracting header fields while passing through the input packet stream without registering the data stream. It handles the defined parse-tree with Finite State Machine (FSM) structure, with deviations from the expected protocols causing a parsing error signal passed in the metadata along the packet stream. A successful parse results in a tuple valid pulse, which is passed along the extracted header fields and a VxLAN-valid signal to the lookup emulator.

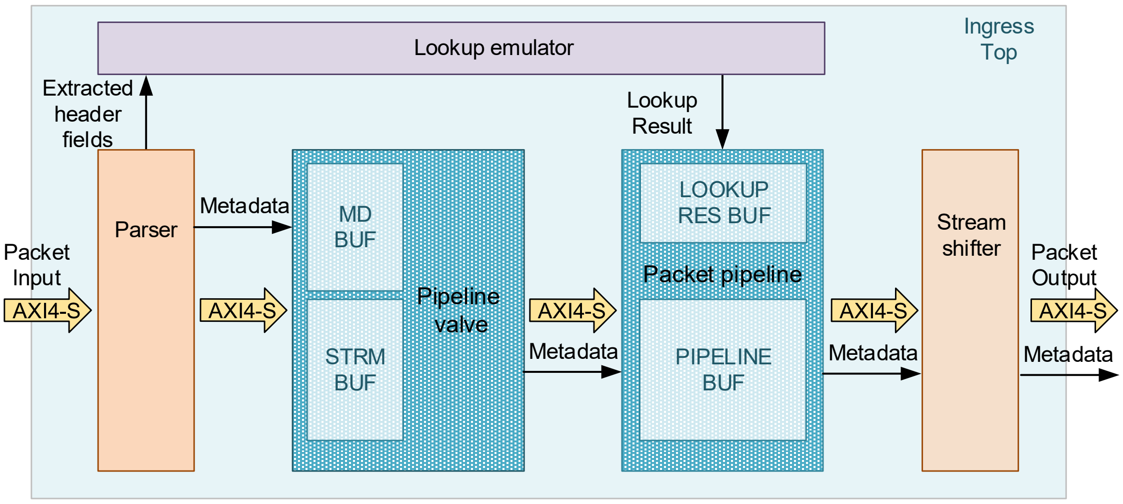

Figure 11. *Design architecture of the VHDL-implemented ingress packet processor.*

The pipeline valve functions as an intermediate buffer to prevent the packet stream entering further until either a successful parse or a parse error. The valve opens only after receiving a tuple valid or a parse error signal, staying open until the last beat of the packet, during which the metadata buffer is read for the parsing status of the next packet.

The packet pipeline functions as a delay to buffer the stream until a lookup result is read. It has a similar valve structure to prevent the packet stream, as in the pipeline valve. In the case of a parse error, the packet is streamed through directly, with metadata indicating the latter block to stream through and route the packet for dropping. If the parse error signal is zero, the valve opens only after receiving a lookup result. The lookup result is passed on as metadata, alongside the stream to the two final pipeline stages.

The stream shifter is polling for valid beats from the two final pipeline stages on the packet pipeline. In the case of the metadata indicating a drop, resulting from a parse error or a no-hit lookup, the packet is streamed through with metadata guiding it to drop. If, instead, the lookup resulted in a hit, the packet is streamed through with the valid as zero, until the count of bytes indicated by the metadata has passed. After the discarded beats, the valid signal is set to follow the input, and the stream is aligned correctly by left shift. The shift amount is read from the input metadata. Alongside the output stream, the stream shifter outputs a metadata consisting of route ID, VxLAN enabled, and outer

VLAN enabled. This metadata enables the egress packet processor to encapsulate the packet correctly.

Figure 12 describes the egress packet processing architecture, which is similarly operating in line rate and backpropagating backpressure. The egress side parser has the same operation principle as the ingress side, although the parse tree consists of only inner headers. It passes a tuple valid pulse to the lookup emulator along with the ethernet source address if the inner header stack is parsed successfully. Otherwise, a parse error signal is passed to the pipeline valve. Alongside the packet AXI4-stream, a metadata bus carries the information of whether VxLAN encapsulation is enabled for the packet, and if the outer headers carried a VLAN tag in the original packet received in ingress.

The lookup emulator has Ethernet source addresses and corresponding outer headers saved as constants. Once a valid, extracted Ethernet source address is received from the parser, it conducts a lookup operation. If a match occurs, the outer headers are streamed to the lookup result buffer in the Stream assembler. If not, the negative result is passed in a single transfer.

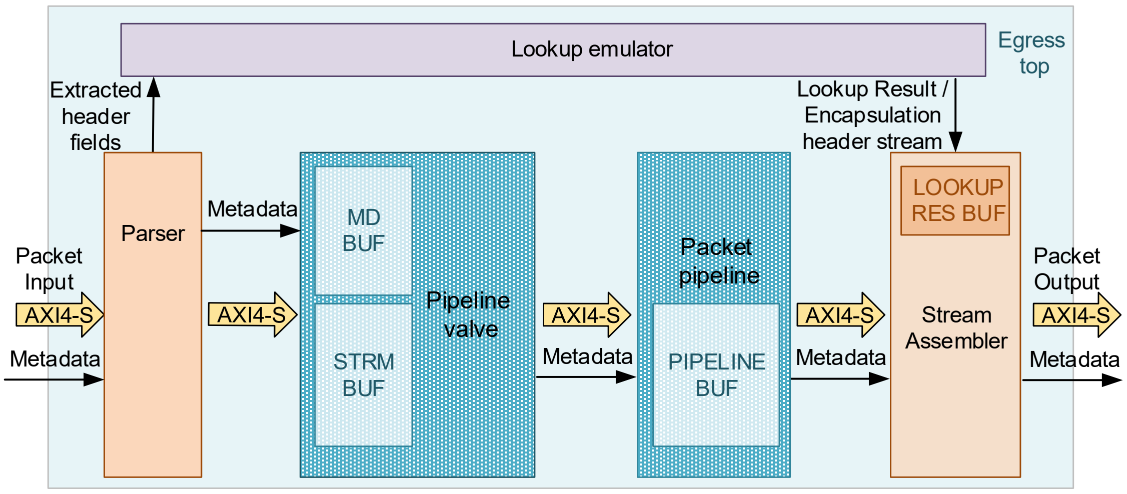

Figure 12. *Design architecture of the VHDL-implemented egress packet processor.*

The pipeline valve and packet pipeline components in egress are similar with those in ingress, with the slight differences in metadata handling. In packet pipeline, the lookup results are polled from the lookup result buffer, if the packet is enabled for VxLAN encapsulation. If the packet is not due to encapsulation or a parse error has occurred, it is directly passed through to stream assembler. The stream assembler polls for valid metadata from the packet pipeline. The metadata indicates if the incoming packet is

1) directed to drop because of a parse error or a failed lookup,
2) streamed through without encapsulation,

3) VxLAN encapsulated and forwarded with the outer headers from the lookup result buffer.

In the case of 1), the packet is streamed through without modifications with the metadata field route ID set to 0, resulting it to be routed into drop. In case of 2), the packet is streamed through while the Ethernet and IP addresses and the UDP port fields are swapped. In the case 3), the outer headers are first streamed from lookup result buffer and the rest of the packet stream starting from the inner headers is concatenated into this encapsulation header stream. The same swap operations as in case 2) are also executed.

In general, the VHDL implementation was not done with resource optimization in mind, and the goal was to attain the wanted functionality with as low as possible latency, regardless of the levels of logic between registers, as long as the timing constraints were met.

7.2 Software interfaced P4 implementation

In the P4 Software Interfaced design variant, the RTL is generated as IPs from P4 language description with a third-party tool. The written P4 code for ingress and egress matches closely to the use case description described previously in chapter 8.2. Both P4 descriptions consist of 3 logical entities: the parser, the match-action pipeline, and the deparser, which is the final control block emitting the packet with wanted protocol headers. Both P4 descriptions have the same headers struct defined, consisting of the outer and inner headers.

The ingress parser definition has a parse tree which starts at the outer Ethernet. If a VXLAN header is encountered, an indicator from this is passed in a metadata field to the match-action pipeline. The lookup-table and action declarations of the ingress match-action pipeline are presented in program 3. As an initial process, the match-action pipeline control section code checks the validity of the VxLAN header. If it is valid, a VNI matcher table is applied. As seen in program 3 lines 26 to 31, the matching method of the table is exact match, and the key is the VNI-field of VxLAN. Size is defined as two. The table is defined to have to two actions: a default action “dropPacket” (lines 10-12), which sets the metadata route ID field as 0, and the action “move_to_VNF_cntxt” (lines 1-3), which sets an internal variable of “vnf_cntxt_enable” as ‘1’. Both actions are defined without input parameters.

If the VxLAN header was not valid, or the “vnf_cntxt_en” is ‘1’ after VNI matcher table was applied, the table “five_tuple_matcher” (lines 14-24) is applied. The table uses five-tuple as the key, with exact match as the matching method. Size of the table is two. Actions tied to the table are “TerminateVT” (lines 5-8), which sets the metadata route ID-field as 1 and the variable “terminate_enabled” to ‘1’, and the previously explained action “dropPacket”.

Finally, should the “terminate_enabled” be ‘1’ and VxLAN header valid, the outer headers are set as invalid and thereby removed from the packet in the deparser. However, if the VxLAN is invalid, but terminate_enabled was ‘1’, the packet is not modified but still forwarded. In any other case the route ID-field value ‘0’ results in the packet to be dropped by the external router IP (PKT RTR in figure 6).

In egress P4 description, the parser starts directly from inner Ethernet and parses through the inner header stack. The match-action section processing begins with an if-clause conditioning the validity of the complete inner header stack. Should they prove valid, a nested if-clause checks the value of the metadata field “VXLAN_ena”. If it is “1”, the tables “addEth”, “addIPUDP”, and “addVxlan” are applied. These tables and related action declarations are presented in program 2.

All three tables (lines 17-36) have the same lookup key, which is the Ethernet source address, and each table has the exact match lookup method. All the tables have one common action, the dropPacket (lines 13-15), which has no input parameter and writes the metadata route ID-field as zero, resulting in the packet to be directed for drop. Additionally, each table has a unique action: action “PushEth” (lines 1-3) for table “addEth”, “PushIPUDP” (lines 5-7) for “addIPUDP”, and “PushVxlan” (lines 9-11) for “addVXLAN”. Action PushEth has an input parameter of a 144-bit vector consisting of Ethernet and optional VLAN headers, PushIPUDP a 224-bit vector consisting of IPv4 and UDP headers, and PushVxlan a 64-bit vector consisting of the VXLAN header. If these actions are hit and the input parameters were found in memory, the outer headers are set as valid and their values set from the input parameters of the actions. These action functionalities are omitted for brevity.

Finally, if no “dropPacket” actions were hit, or VxLAN encapsulation was not set in the metadata, the route ID-value is set as “1”, resulting in the packet being forwarded for Ethernet TX. Due to tooling issues at design time, the software interfaced P4 implementation does not include the inner header swap operations.

```
action move_to_VNF_cntxt() {
  vnf_cntxt_en = 1;
}

action TerminateVT() {
  metadata.ROUTE_ID = 0x1;
  terminate_enabled = 1;
}

action dropPacket() {
  metadata.ROUTE_ID = 0x0;
}

table five_tuple_matcher {
  key = { dst   : exact;
          src   : exact;
          proto : exact;
          dport : exact;
          sport : exact;
   }
   actions        = { TerminateVT; dropPacket; }
   size           = 2;
   default_action = dropPacket;
}

table vni_matcher {
  key = { hdr.vxlan.vni : exact;
  }
   actions        = { move_to_VNF_cntxt; dropPacket; }
   size           = 2;
   default_action = dropPacket;
}
```

Program 3. *P4 declarations for ingress match-action tables and actions for the software interfaced P4 implementation.*

Figure 13 describes the SDNet-generated packet processor IP with its main interfaces. The parser, match-action pipeline, deparser, and the interior interfaces represent the P4 description, and are not actual internal hardware components in the SDNet IP. The packet processing IPs in this work are interfaced in data plane by a 64-bit AXI4-Stream packet data interfaces and supporting metadata interface, providing once-per-packet 6-bit sideband data accompanied with a valid signal. The control plane interface is an AXI4-Lite interface, used for managing the contents of the lookup tables in the memory search IP cores of the design.

The user application for table management is using and compiled together with an API, and the compiled executable is using a PCIe driver for access from the host CPU to the FPGA. It is essentially responsible for populating the tables with key-value pairs.

```
action PushEth(bit<144> hdrs_1) {
  … //Ethernet+VLAN set valid and added
}

action PushIPUDP(bit<224> hdrs_2) {
  … //IP+UDP set valid and added
}

action PushVxlan(bit<64> hdrs_3) {
  … //VXLAN set valid and added
}

action dropPacket() {
  md.ROUTE_ID = 0x0;
}

table addEth   {
  key      = { eth_tmp : exact; }
  actions  = { PushEth; dropPacket; }
  size            = 4;
  default_action  = dropPacket;
}

table addIPUDP  {
  key      = { eth_tmp : exact; }
  actions  = { PushIPUDP; dropPacket; }
  size           = 4;
  default_action = dropPacket;
}

table addVxlan    {
  key      = { eth_tmp : exact; }
  actions  = { PushVxlan; dropPacket; }
  size           = 4;
  default_action = dropPacket;
}
```

Program 4. *P4 declarations for egress match-action tables and actions for the software interfaced P4 implementation.*

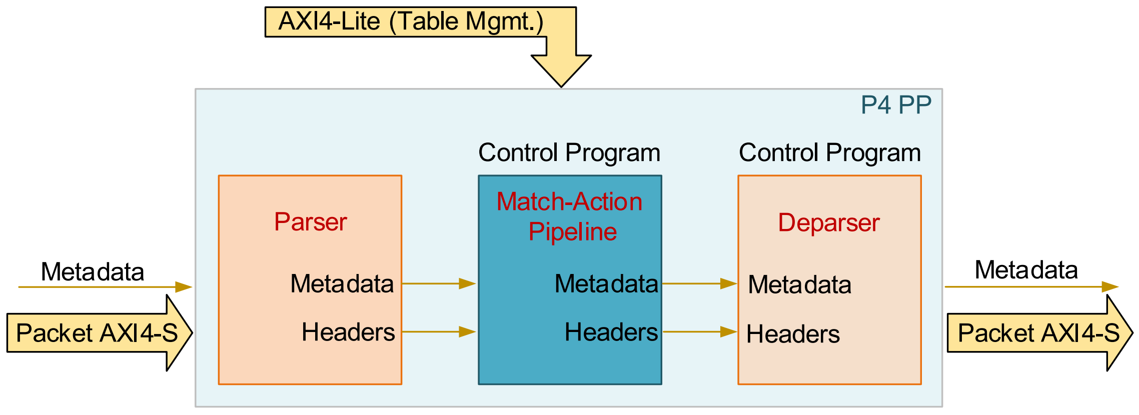

Figure 13. *P4-defined packet processor IP.*

7.3 Hard-coded P4 implementation

The hard-coded P4 implementation of the packet processors has the same implementation flow as the software interfaced P4 implementation. There are, however, three main differences:

1) The P4 code does not include any table structures, as the match-action structures are made with conditional statements. Therefore,
2) there are no software interfaces in the packet processors and the top level of the shell design is coherent with the VHDL implementation.
3) Without a software interface, no control plane application for table management is used.

The conditional statements replacing the match-actions are presented in programs 5 and 6, for ingress and egress, respectively. In program 5, the matching of the VNI field is done by comparison to constants “const_vni_1” and “const_vni_2” (lines 1-6), which hold pre-defined VNI values. Match or no match results in same functionality as in software interfaced ingress program. The five-tuple value matching is done (line 8-15) by comparing the input five-tuple value to pre-defined constants “const_fivetpl_1” and “const_fivetpl_2”.

Hard-coded P4 egress match-action declaration is shown in program 6. The Ethernet source field of the input packet is compared against 4 pre-defined constants, resulting in either header additions with a match, or packet being guided to drop with route ID-field as “0”.

With these code structures the design is to match more closely to the VHDL implementation, as the software interface and possible related hardware is made unnecessary.

Data plane functionality is the same as in the software interfaced variant, as is the simulation process.

```
if ((hdr.vxlan.vni == const_vni_1) || (hdr.vxlan.vni == const_vni_2)) {
  vnf_cntxt_en = 1;
}
else {
  vnf_cntxt_en = 0;
}
…
if ((fivetpl == const_fivetpl_1) || (fivetpl = const_fivetpl_2)) {
  metadata.ROUTE_ID = 0x1;
  terminate_enabled = 1;
}
else {
  metadata.ROUTE_ID = 0x0;
  terminate_enabled = 0;
}
```

Program 5. *The Conditional statements of the hard-coded P4 ingress implementation, which replace the software interfaced match-action table structures.*

```
//if VXLAN encapsulation is enabled
if (hdr.eth.src = const_eth_src_1) {
  … //Outer headers set valid and added from constants #1
}
else if (hdr.eth.src = const_eth_src_2) {
  … //Outer headers set valid and added from constants #2
}
else if (hdr.eth.src = const_eth_src_3) {
  … //Outer headers set valid and added from constants #3
}
else if (hdr.eth.src = const_eth_src_4) {
  … //Outer headers set valid and added from constants #4
}
else {
  md.ROUTE_ID = 0x0;
}
```

Program 6. *Conditional statements of the hard-coded P4 egress implementation, replacing the software interfaced match-action table structures.*

8. RESULTS AND ANALYSIS

This chapter presents the FPGA resource utilizations, latency, and throughputs of the three design variants.

8.1 Utilization

The numerical amounts of FPGA resources available in total on the VU9P FPGA are presented in table 1. The LUTRAMs are a subset of the LUTs.

Table 1. *Amount of resources available on the VU9P FPGA.*

RESOURCE	AVAILABLE TOTAL (CT.)
LUT	1182240
LUTRAM	591840
FF	2364480
BRAM	2160

Table 2 describes the count of resources used in the VHDL (PP Ing. VHDL), hard-coded P4 (PP Ing. P4 HC), and software interfaced P4 (PP Ing. P4 SW IF) implementations of the ingress packet processor. For comparison clarity, these numerical values are also visualized as percentages of total available resources from table 1 in the figure 14. While inspecting the utilization percentage figures, one should bear in mind the actual percentages are platform-dependant, and therefore the key point of the figures is the relations of these percentages between implementations.

As seen from the figure 14, the resource utilization in the software interfaced P4 implementation is significantly higher the amount used in two other design variants. The generated RTL is third party IP, and one can only make hypothetical assumptions on the cause for this difference. Logical entities in the software interfaced variant, which are missing in the other design variants, are the software interfaced lookup tables. They may have surrounding logic in forms of e.g. protocol converters, buffers and arbiters.

Table 2. *Amount of resources utilized by each ingress packet processor variant.*

DESIGN VARIANT	RESOURCE UTILIZATION (CT.)			
	LUT	LUTRAM	FF	BRAM
PP ING. VHDL	1110	96	930	0
PP ING. P4 HC	1239	393	2697	0
PP ING. P4 SW IF	9656	2284	18403	12

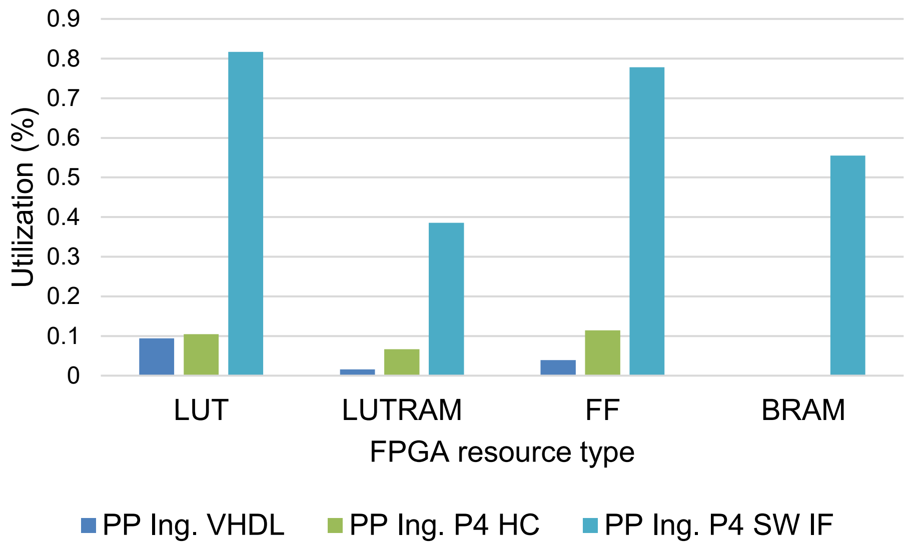

Figure 14. *FPGA resource utilization percentages in ingress packet processors.*

Comparing the VHDL and hard-coded P4 implementations of the ingress packet processors, one can see that the VHDL implementation has the lower utilization percentage in all the selected FPGA resource types, excluding the BRAMs, which are at even 0 in both variants. This result could be expected, as the VHDL implementation is written directly to fulfil its purpose, without any overhead in performance or functionality. Particularly visible is the difference in FF usage: the hard-coded P4 implementation is using 190% more flip-flops than the VHDL implementation. This is in-line with the design principle of the VHDL design, as it was created to have as-low-as-possible latency, with minimal register stages. On the other hand, the VHDL variant is actually using more combinatory logic than the hard-coded P4 variant: as seen in table 2, the hard-coded P4 ingress processor is utilizing 1239 LUTs, of which 393 are used as LUTRAMs, leaving 846 LUTs used as combinatory logic. For the VHDL variant, these numbers are 1110 LUTs, of which 96 are used as LUTRAMs, leaving 1014 LUTs used as combinatory logic.

Finally, the hard-coded P4 ingress variant is using almost fourfold the LUTRAMs comparing to the VHDL variant. In the VHDL variant, the memories are used in pipeline FIFOs to stall the stream while the lookup is finished and their capacity is minimized. The use-case of the P4 design variant for the LUTRAMs is unknown.

Table 3 presents the numerical counts of resources utilized by the VHDL (PP Eg. VHDL), hard-coded P4 (PP Eg. P4 HC), and software interfaced P4 (PP Eg. P4 SW IF) implementations of the egress packet processor. Utilizations as percentages of total available resources are depicted in figure 15. The software interfaced P4 variant of the egress processor is by far the most resource expensive of the variants, which is again expected as the design includes software interfaced lookup tables. The VHDL variant of the egress packet processor has the lowest utilization ratio across all resource types.

Table 3. *Amount of resources utilized by each egress packet processor variant.*

DESIGN VARIANT	RESOURCE UTILIZATION (CT.)			
	LUT	LUTRAM	FF	BRAM
PP EG. VHDL	2415	104	2054	0
PP EG. P4 HC	3730	1875	7818	0
PP EG. P4 SW IF	21148	5999	43236	44

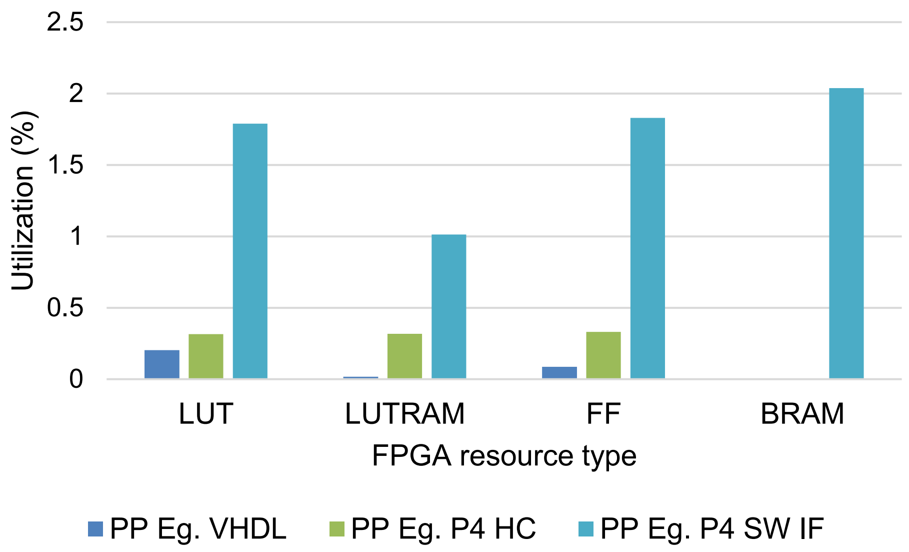

Figure 15. *FPGA resource utilization percentages in egress packet processors.*

On the egress side, the most drastic percentual resource utilization difference occurs with the LUTRAMs: the hard-coded P4 egress processor is using 1875 LUTRAMs, which is roughly 18x the 104 LUTRAMs used by the VHDL variant. These counts leave 1855 and 2311 LUTs used as combinatory logic for hard-coded P4 and VHDL variants, respectively. For FFs, the difference between hard-coded P4 and VHDL egress processors is 5764 units, making the FF utilization of the hard-coded P4 design 3.8x the utilization of the VHDL variant.

Table 4 presents the percentual increase in resource utilization from ingress to egress in the VHDL, the hard-coded P4 (P4 HC) and the software interfaced P4 (P4 SW IF) packet processor implementations. One should remember that the logical functionality in ingress is essentially comparing header values to lookup keys, and possibly remove headers and route the packet with metadata. In egress however, the lookup leads into header addition operation and modification, apart from the software interfaced P4 variant where the header swap operations do not occur. From table 4 one can notice, that the egress side functionality is more demanding on the hardware resources across all three design variants.

Table 4. *Percentual increases in resource utilization from ingress to egress.*

DESIGN VARIANT	**INCREASE OF UTILIZATION FROM INGRESS TO EGRESS (%)**			
	LUT	LUTRAM	FF	BRAM
VHDL	117,6	8,3	120,9	0
P4 HC	201,0	377,1	189,9	0
P4 SW IF	119,0	162,7	134,9	266,7

For P4 SW IF, the usage of on-chip BRAM experienced the most drastic increase. An increase is expected, as the stored lookup values are increased from just five-tuple values and VNIs into Ethernet source addresses and full VxLAN encapsulation headers, therefore requiring more memory space. In the VHDL variant, the increase is not directly visible in memory resources, i.e. LUTRAMs or BRAMs, but the P4 HC variant egress packet processor shows a rather significant growth of 377% in LUTRAM usage comparing to the ingress packet processor. The storage of the VHDL variant is implemented with constants mapping to FFs and LUTs, but most of the increase in these resources happens in the egress assembler. The assembler holds a shift register buffer for the outer header stack stream to be concatenated into the packet stream, plus logic to swap the inner header addresses and ports.

Altogether, the VHDL variant can be deemed as the most resource optimized with the least overhead in both the ingress header removal, and the egress header addition and modification. Naturally, the comparison of utilizations between the P4 SW IF and other variants might not be the most meaningful, as the P4 SW IF variant holds logic not present in others. In terms of utilization scaling with different functionalities, on the other hand, the VHDL variant is the best across all variants.

8.2 Performance

Tables 5 and 6 present the maximal throughputs of shell designs with VHDL (Shell VHDL), hard-coded P4 (Shell P4 HC), and software interfaced P4 (Shell P4 SW IF) implementations of packet processors. Table 5 measurements are done with packets having only inner headers, whereas table 6 measurements are done with VxLAN encapsulated packets, including also the outer headers. Looking at the throughputs, one can notice that the values are the same per frame size across all design variants. It can be assumed that all the packet processors are capable the same throughput. The delta between the theoretical line rate of 10Gbps and the measured throughputs seems independent of the shell design variants and is most likely caused by factors external to the packet processors, such as the traffic generator or Ethernet SS IP limitations. Affecting factor to the throughput seems to be the frame size, and in a manner that would show that more frequent packet boundaries, i.e. smaller frames, cause more delta from the maximum throughput.

Table 5. *Measured throughputs from the shell variants with basic Ethernet frames.*

FRAME SIZE	**THROUGHPUT (Mbps)**		
(BYTES)	Shell VHDL	Shell P4 HC	Shell P4 SW IF
74	7871,543	7871,543	7871,543
512	9623,097	9623,097	9623,097
1024	9807,447	9807,447	9807,447
1522	9869,311	9869,311	9869,311

Table 6. *Measured throughputs with VxLAN encapsulated Ethernet frames.*

FRAME SIZE	**THROUGHPUT (Mbps)**		
(BYTES)	Shell VHDL	Shell P4 HC	Shell P4 SW IF
136	8717,076	8717,076	8717,076
512	9623,097	9623,097	9623,097
1024	9807,448	9807,447	9807,447
1522	9869,311	9869,311	9869,311

Latency measurements for each shell variant similarly presented with only inner headers in table 7 and figure 16, and with VxLAN encapsulated frames in table 8 and figure 17. These results show that the VHDL variant of the shell design has by far the lowest latency independent of the case of only inner headers or VxLAN encapsulation present in the packet. With only inner headers, the Shell P4 HC has a roughly a twofold latency compared to the VHDL variant. For the Shell P4 SW IF, the latency is roughly 2,5-fold the latency of the VHDL variant. It is also notable, that with the smallest tested frames (74 Bytes), the P4 variants are able to process the packets with around 40 ns (roughly 6,5 clock cycles with the frequency of 161 MHz) smaller latency than larger packets, whereas the latency of the VHDL variant is somewhat constant across all frame sizes.

The VxLAN encapsulated frames induce larger latencies for all design variants, but the lowest latencies are still caused by the VHDL variant. The header removal, addition and modification operations cause an average increase of 90 ns (14,5 clock cycles) in latency in the VHDL variant comparing to the latencies measured with the frames without VxLAN encapsulation. For the P4 variants, the increase in latency with VxLAN capsulation operations is around 40 ns (roughly 6,5 clock cycles). This would indicate that the P4 variants hold already more overhead in processing time even for packets with only inner headers, and the VxLAN encapsulation header processing can be more parallelized with other operations than with the VHDL variant. The latency of the VHDL variant is dominantly caused by header removals and additions.

The latency difference between the two P4 designs is over 250 ns, adding up to roughly 42 clock cycles. The root cause for this is unknown, given the encrypted nature of the designs, but assumptions can be made. As the P4 SW IF variant holds the software interfaced lookup table logic, it could be causing additional latency in the packet processing match-action pipeline with for example protocol converters (such as AXI-protocol to a native memory interface protocol), extended buffering, arbiters, and naturally the delay caused by the lookup operation itself. In the case of the VHDL design, the lookup latency is one clock cycle, as there are no memory structures and the lookup is essentially a logical bitwise AND-operation between the extracted header fields and the constant values. However, the lookup emulator is specifically designed to be an external component, where a lookup engine could be inserted, without affecting the parsing, pipeline and deparsing stages of the design.

Table 7. *Measured cut-through latencies with Ethernet frames.*

FRAME SIZE	**LATENCY (ns)**		
(BYTES)	Shell VHDL	Shell P4 HC	Shell P4 SW IF
74	544	1048	1301
512	542	1091	1344
1024	542	1091	1344
1522	544	1092	1345

Table 8. *Measured cut-through latencies with VXLAN encapsulated Ethernet frames.*

FRAME SIZE	**LATENCY (ns)**		
(BYTES)	Shell VHDL	Shell P4 HC	Shell P4 SW IF
136	632	1137	1390
512	632	1135	1389
1024	632	1135	1388
1522	633	1137	1390

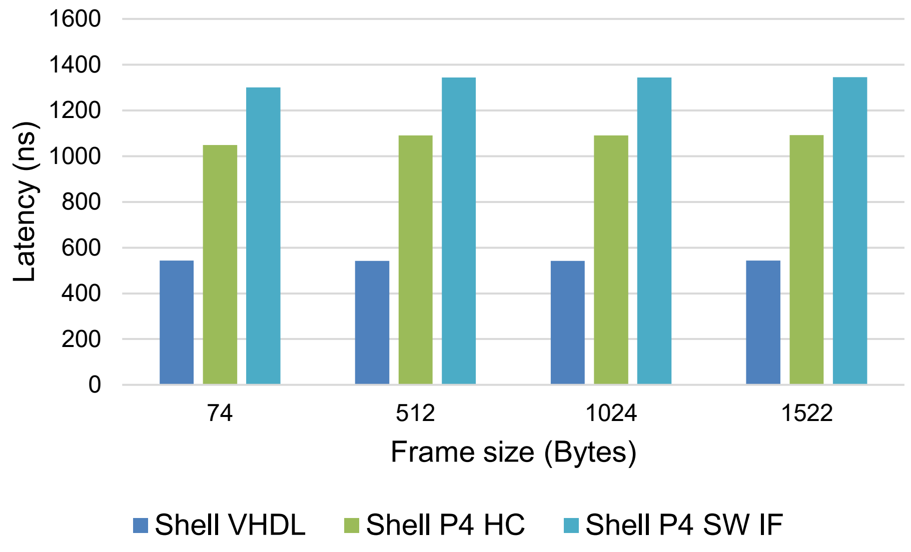

Figure 16. *Latencies with only inner headers for each design variant per frame size.*

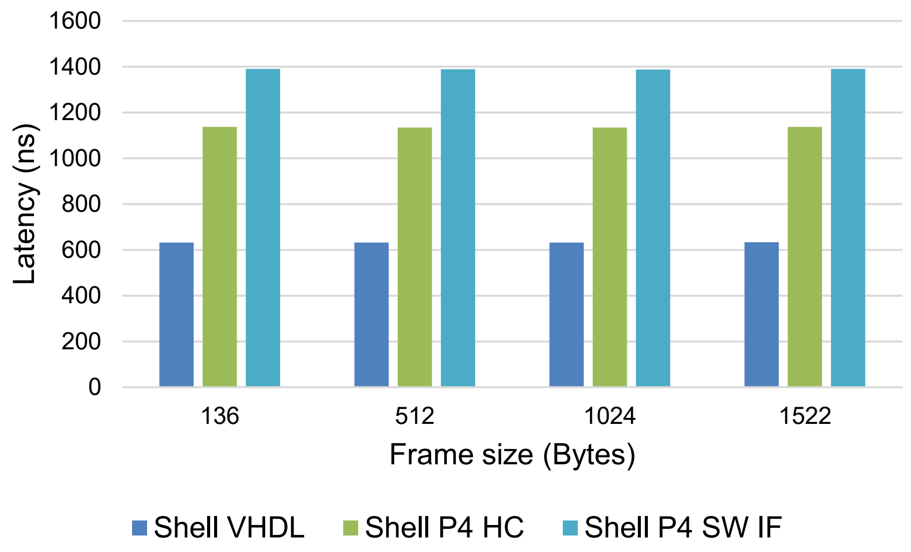

Figure 17. *Latencies with VxLAN encapsulation headers on top of inner headers for each design variant per frame size.*

The minimum size of an Ethernet frame is 64 bytes (512 bits) [62], which with a 64-bit data bus-width and back-to-back frames means a new packet every 8 clock cycles. This clock cycle count gives a theoretical maximum latency for a packet-specific lookup to not affect the throughput of the design. In other words, the throughput of the lookup engine itself has to be 1 lookup per 8 clock cycles. For example, the Xilinx BCAM engine (Binary Content-Addressable Memory) for SDNet version 2018.1 [63] is reported to have a lookup latency of 3 clock cycles. The rest of the design has a buffering capability for a lookup latency of 12 clock cycles, which would still leave clock cycles for pipelined arbitration and protocol conversions. With a 12-clock-cycle lookup latency, the overall latency of the VHDL design variant would be increased by roughly 70 ns (with the clock frequency of 161MHz).

Adding an actual memory search IP to the lookup-emulator remains as possible future work. This would naturally introduce added logic to the design, for memory access arbitration between the lookups and table updates, and the resource utilization of the design would grow along with some added latency. Also, one should remember that these performance metrics do not consider the runtime table updates, which will cause either added latency with buffering, or dropped packets in the case of table updates exceeding the clock cycle window for the lookup operations. Nevertheless, in this use case, the VHDL variant is superior in terms of latency minimization and should prove to be superior even with an added lookup engine, providing lower latency than the P4 designs.

8.3 Degree of automation

Figure 10 in chapter 9 depicted the steps of the workflow on a general level, common to all design variants. In figure 18, the steps of actual hardware design (as in hardware description code writing) and verification are pictured with slightly more detail, but still on a very general level, showing what sub-steps they include in VHDL design flow (fig. 18 a)) and in P4 design flow (fig. 18 b)). Feedback paths from verification steps back to hardware design are abstracted from the picture but naturally exist, as the design is fixed based on simulation results.

As figure 18 shows, the main difference in the HW design processes with VHDL and P4 design flow is production of RTL description of the design, which is written manually (IP-reuse is not taken into account) in VHDL flow, but generated automatically in the P4 design flow, based on the P4 description. To give an insight on the degree of automation in the P4 flow, table 9 presents the lines of code (LOC) required to develop the combined packet processors, the ingress packet processor, and the egress packet processor per design flow. Even though in general the LOC-based analysis of code might not be particularly meaningful due to personal differences and taste in coding styles and code readability levels, it is used in this work to give the reader a sense of the level in abstraction and simplicity in P4 compared to VHDL. Empty and commented lines are excluded from the LOC counts.

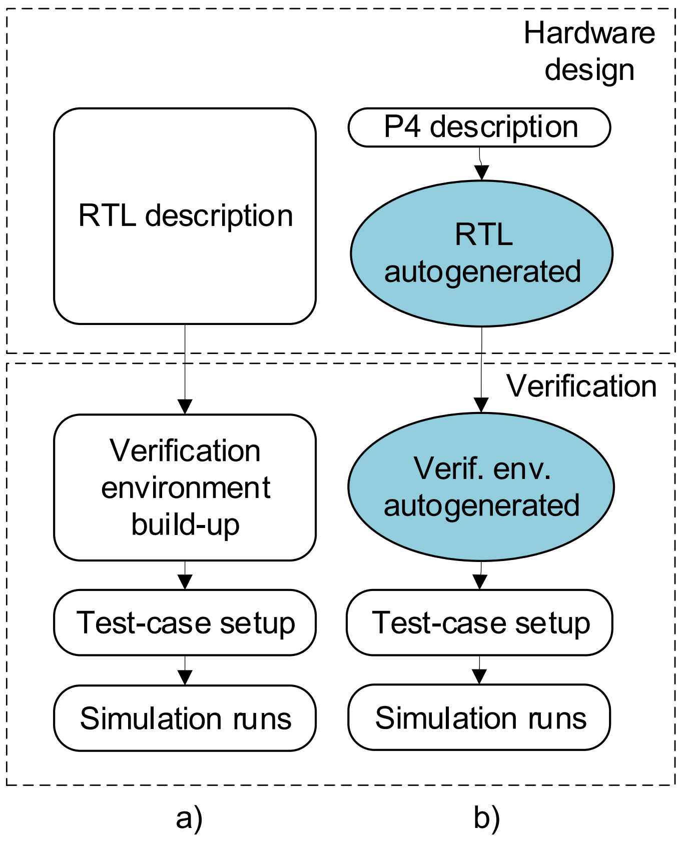

Figure 18. *Design and verification workflow graphs for a) VHDL and b) P4 design in this work.*

As seen in the table 9, the combined LOC count of the VHDL ingress and egress packet processors is roughly 6,3-fold the LOC count of the P4 HC variant, and 8,1-fold the respective count in the P4 SW IF variant, with a difference of over 3 thousand lines to both. As P4 is a higher-level language, this is expected: the P4 code is more of a functional description with no clock cycle-accurate operations as in RTL, such as VHDL. For example, the highest LOC count for a single IP in the VHDL variant is 446 in the ingress parser. This IP implements the parse tree with a sequential process FSM, which alone has 40 states and 356 LOC. In comparison, the ingress parse tree definition in P4 takes 10 states and 71 LOC.

On the egress side, the dominant IP of the VHDL variant, in terms of LOC, is the stream assembler. Supporting different header structures and swapping headers on the fly requires again a large FSM structure. This increase in code length in egress side altogether is however mitigated by a shorter parse tree (130 LOC), as there are no VXLAN encapsulation headers to parse. In P4 HC variant the LOC count increases due to the if—clause structure presented in chapter 8.3, and thus the source code in both ingress and egress is lengthier than in the P4 SW IF design units. In general, the P4 SW IF variant seems to handle scaling better across ingress and egress, and has consistently the smallest count of LOC. At the same time, it is the design variant with the most logic, as presented in chapter 9.1, and the most complex design with the added SW IF section.

Table 9. *Counts of lines of code per design unit for each design variant.*

DESIGN UNIT	LINES OF CODE		
	VHDL	P4 HC	P4 SW IF
PP TOTAL	3624	576	446
PP INGRESS	1836	247	239
PP EGRESS	1786	329	207

The verification done with this work was not thorough, and as seen in the figure 15, fall more under the category of simulation. Tasks included

1. building the framework of a verification environment, a testbench, such as an UVM class hierarchy structure,
2. defining a set of test cases, the stimuli, and implement the feeding of this stimuli to the DUT.
3. Simulation of the DUT with the testbench and verifying the wanted functionality corresponding to the input stimulus

From these three steps, only the first one is somewhat automated in the P4 flow with Xilinx SDNet. The IP developer is left with the task of choosing the stimuli and running the simulations, as well as verifying the output. In this work, the UVM testbench structure was legacy design and reused. How much of work this kind of a testbench architecture requires in terms of LOC or work hours, is dependant for example of the IP, its interfaces, requirements on testing, and developer experience. Additionally, in the case of developing IPs which have the same interface, such as packet processors with different protocol-specific operations, the testbench architecture would not possibly need much rework in principle between implementations, and reuse to some degree should be possible.

Whereas exactly quantitative analysis on the degree of automation gained from the P4 design flow cannot be performed, a few conclusions can be made. Even with the simplified VHDL implementation without memory search IPs, the P4 design with a software interface provided a significant advantage in terms of code lines to be written. This is a natural consequence of the high-level nature of the P4 language, coming with the disadvantage of losing control in clock cycle accurate functionality description. With the supporting simulation test environment provided by SDNet, the P4 language could very well lead into a smaller time to testing and prove itself useful in for example Proof of Concept (POC) projects, such as [56].

9. CONCLUSIONS

In this thesis, a data plane programming language, P4, was trialled as an implementation method for networking logic on a network-attached FPGA. The goal was to find out if the high-level nature of the language with a current state compiler could prove an efficient way to implement packet processing logic without an overly extensive supported feature set and sufficiently in-par with a traditional VHDL implementation in terms of performance and resource utilization.

The functionality to be implemented on the FPGA was VTEP termination combined with a five-tuple based firewall. Three design variants were implemented: a full P4 implementation with a software interface for table management, a reference VHDL design providing the same data plane functionality but without a software interface, and finally, a hard-coded P4 design without a software interface, purposed for a more exact resource utilization comparison with the VHDL variant.

The results of this work suggest that the lowest utilization and lowest latencies can be reached with VHDL. However, one must consider that overall the trialled designs are relatively simple. Even the highest-utilizing design variant, the software interfaced P4 design, uses only around 2,6% of the LUTs available on the FPGA for the packet processing logic. Additionally, the highest latency measured with the same design was around 1,4µs, which is well under the 8µs limit for the accelerator shell design latency maximum set by the CRUN framework. Combined with the fact that the VHDL design took 8.1x the lines of source code in comparison to the software interfaced P4 implementation, these measurements suggest that there could be use for P4, in for example POC projects, where the performance and resource limitations are more relaxed, in comparison to commercial products.

In conclusion, the study was successful, and proved a valuable look into the current state of the P4 language with network-attached FPGA accelerator platforms, paving way for continued and more thorough research, which remains future work. For a deeper analysis, a full VHDL reference design must be implemented, including a lookup engine, e.g. a content-addressable memory component, and a software interface. Additionally, the utilization and performance scaling with varying packet processing functionality must be tested, as well as reachable performance maximums with higher data rates and clock frequencies.

REFERENCES

[1] vRAN: The Next Step in Network Transformation, Wind River Systems, White paper. Available (accessed on 8.5.2020): https://events.windriver.com/wrcd01/wrcm/2017/10/vRAN-The-Next-Step-in-Network-Transformation-White-Paper.pdf.

[2] Cloud RAN Architecture for 5G, Telefonica, White paper. Available (accessed on 8.5.2020): http://www.hit.bme.hu/~jakab/edu/litr/5G/WhitePaper_C-RAN_for_5G_-Telefonica_Ericsson.PDF.

[3] R. Mijumbi, J. Serrat, J. Gorricho, J. Rubio-Loyola, S. Davy, Server placement and assignment in virtualized radio access networks, 2015 11th International Conference on Network and Service Management (CNSM), 2015, pp. 398-401.

[4] D. Rajan, Achieving High Performance with Virtualized Data Plane Workloads for 5G Networks, 2019 Sixth International Conference on Software Defined Systems (SDS), 2019, pp. 236-241.

[5] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. López, F.J. Ramón Salguero, F. Ruhl & P. Sen, Network Functions Virtualisation - An Introduction, Benefits, Enablers, Challenges & Call for Action, White paper. Available (accessed on 8.5.2020): https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[6] Network Functions Virtualisation (NFV); Architectural Framework, ETSI, White paper. Available (accessed on 11.4.2020): https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf.

[7] Network Functions Virtualisation (NFV); Acceleration Technologies; Report on Acceleration Technologies & Use Cases, ETSI, White paper. Available (accessed on 30.4.2020): https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/001/01.01.01_60/gs_nfv-ifa001v010101p.pdf.

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, D. Burger, A cloud-scale acceleration architecture, 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, pp. 1-13.

[9] N. Feamster, J. Rexford, E. Zegura, The Road to SDN: An Intellectual History of Programmable Networks, SIGCOMM Comput.Commun.Rev., Vol. 44, Iss. 2, 2014, pp. 87–98. Available (accessed on 7.5.2020): https://doi.org/10.1145/2602204.2602219.

[10] R. Bifulco, G. Rétvári, A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems, 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), 2018, pp. 1-7.

[11] P416 Language Specification, version 1.2.0, The P4 Language Consortium, Available (accessed on 26.2.2020): https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf.

[12] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, R. Boutaba, Network Function Virtualization: State-of-the-Art and Research Challenges, IEEE Communications Surveys & Tutorials, Vol. 18, Iss. 1, 2016, pp. 236-262.

[13] B. Yi, X. Wang, K. Li, S. Das, M. Huang, A Comprehensive Survey of Network Function Virtualization, Computer Networks, Vol. 133, 2018,

[14] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-Defined Networking: A Comprehensive Survey, Proceedings of the IEEE, Vol. 103, Iss. 1, 2015, pp. 14-76.

[15] E. Kaljic, A. Maric, P. Njemcevic, M. Hadzialic, A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking, IEEE Access, Vol. 7, 2019, pp. 47804-47840.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling Innovation in Campus Networks, SIGCOMM Comput.Commun.Rev., Vol. 38, Iss. 2, 2008, pp. 69–74. Available (accessed on 20.2.2020): https://doi.org/10.1145/1355734.1355746.

[17] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, G. Bianchi, Survey of Performance Acceleration Techniques for Network Function Virtualization, Proceedings of the IEEE, Vol. 107, Iss. 4, 2019, pp. 746-764.

[18] H. Eran, D. Levi, L. Liss, M. Silberstein, NFV acceleration: the role of the NIC, SFMA'18, 2018,

[19] Z. Bronstein, E. Roch, J. Xia, A. Molkho, Uniform handling and abstraction of NFV hardware accelerators, IEEE Network, Vol. 29, Iss. 3, 2015, pp. 22-29.

[20] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, D. Burger, Configurable Clouds, IEEE Micro, Vol. 37, Iss. 3, 2017, pp. 52-61.

[21] S. A. Fahmy, K. Vipin, S. Shreejith, Virtualized FPGA Accelerators for Efficient Cloud Computing, 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), 2015, pp. 430-435.

[22] J. Lallet, A. Enrici, A. Saffar, FPGA-Based System for the Acceleration of Cloud Microservices, 2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), 2018, pp. 1-5.

[23] O. Knodel, P. Lehmann, R. G. Spallek, RC3E: Reconfigurable Accelerators in Data Centres and Their Provision by Adapted Service Models, 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), 2016, pp. 19-26.

[24] J. Weerasinghe, F. Abel, C. Hagleitner, A. Herkersdorf, Enabling FPGAs in Hyperscale Data Centers, 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and

2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015, pp. 1078-1086.

[25] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, D. Burger, A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services, IEEE Micro, Vol. 35, Iss. 3, 2015, pp. 10-22.

[26] G. Gibb, G. Varghese, M. Horowitz, N. McKeown, Design principles for packet parsers, Architectures for Networking and Communications Systems, 2013, pp. 13-24.

[27] H. Song, Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane, Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, ACM, New York, NY, USA, pp. 127-132.

[28] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesigner, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: Programming Protocol-Independent Packet Processors, ACM SIGCOMM Computer Communication Review, Vol. 44, Iss. 3, 2014. Available (accessed on 26.2.2020): http://www.sigcomm.org/sites/default/files/ccr/papers/2014/July/0000000-0000004.pdf.

[29] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, G. J. Minden, A survey of active network research, IEEE Communications Magazine, Vol. 35, Iss. 1, 1997, pp. 80-86.

[30] L. Yang, R. Dantu, T. Anderson, R. Gopal, Forwarding and Control Element Separation (ForCES) Framework, RFC 3746, 2004. Available (accessed on 20.2.2020): https://tools.ietf.org/html/rfc3746.

[31] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, N. Rao, Are we ready for SDN? Implementation challenges for software-defined networks, IEEE Communications Magazine, Vol. 51, Iss. 7, 2013, pp. 36-43.

[32] H. Harkous, M. Jarschel, M. He, R. Priest, W. Kellerer, Towards Understanding the Performance of P4 Programmable Hardware, 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019, pp. 1-6.

[33] Network Services Switching Platform, NXP Semiconductors, website. Available (accessed on 28.2.2020): https://www.nxp.com/docs/en/fact-sheet/NETSSFS.pdf.

[34] R. Morris, E. Kohler, J. Jannotti, M.F. Kaashoek, The Click Modular Router, SIGOPS Oper.Syst.Rev., Vol. 33, Iss. 5, 1999, pp. 217–231. Available (accessed on 26.2.2020): https://doi.org/10.1145/319344.319166.

[35] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, S. Licking, Packet Transactions: High-Level Programming for Line-Rate Switches, Florianopolis, Brazil, Association for Computing Machinery, New York, NY, USA, pp. 15–28.

[36] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S. Chuang, I. Keslassy, DRMT: Disaggregated Programmable

Switching, Los Angeles, CA, USA, Association for Computing Machinery, New York, NY, USA, pp. 1–14.

[37] P4_16 prototype compiler, website. Available (accessed on 26.2.2020): https://github.com/p4lang/p4c.

[38] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, J. Rexford, PISCES: A Programmable, Protocol-Independent Software Switch, Florianopolis, Brazil, Association for Computing Machinery, New York, NY, USA, pp. 525–538.

[39] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado, The Design and Implementation of Open vSwitch, may, USENIX Association, Oakland, CA, pp. 117-130.

[40] Netronome smartNIC, Netronome, website. Available (accessed on 3.3.2020): https://www.netronome.com/products/smartnic/overview/.

[41] SDNet, Xilinx, website. Available (accessed on 9.2.2020): https://www.xilinx.com/support/documentation-navigation/development-tools/software-development/sdnet.html.

[42] H. Wang, Soul\'e Robert, H.T. Dang, K.S. Lee, V. Shrivastav, N. Foster, H. Weatherspoon, P4FPGA: A Rapid Prototyping Framework for P4, Santa Clara, CA, USA, ACM, New York, NY, USA, pp. 122-135.

[43] P. Benácek, V. Pu, H. Kubátová, P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers, 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 148-155.

[44] J. Santiago da Silva, F.c. Boyer, J.M.P. Langlois, P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs, Monterey, CALIFORNIA, USA, ACM, New York, NY, USA, pp. 147-152.

[45] Barefoot Tofino, Barefoot Networks, website. Available (accessed on 28.2.2020): https://www.barefootnetworks.com/products/brief-tofino/.

[46] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, M. Horowitz, Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN, Hong Kong, China, Association for Computing Machinery, New York, NY, USA, pp. 99–110.

[47] L. Jose, L. Yan, G. Varghese, N. McKeown, Compiling Packet Programs to Reconfigurable Switches, Oakland, CA, USENIX Association, USA, pp. 103–115.

[48] Universal Verification Methodology (UVM) 1.2 User's Guide, Accellera Systems Initiative. Available (accessed on 9.2.2020): https://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[49] TRex, Cisco, website. Available (accessed on 9.2.2020): https://trex-tgn.cisco.com/trex/doc/index.html.

[50] Vivado Design Suite, Xilinx, website: Available (accessed on 9.2.2020): https://www.xilinx.com/products/design-tools/vivado.html#documentation.

[51] VCU1525 Reconfigurable Acceleration Platform User Guide, Xilinx. Available (accessed on 10.1.2020): https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf.

[52] Virtual Input/Output v3.0 LogiCORE IP Product Guide, Xilinx. Available (accessed on 10.1.2020): https://www.xilinx.com/support/documentation/ip_documentation/vio/v3_0/pg159-vio.pdf.

[53] Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide, Xilinx. Available (accessed on 10.1.2020): https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf.

[54] S. Bradner, J. McQuaid, Benchmarking Methodology for Network Interconnect Devices, RFC 2544, 1999. Available (accessed on 9.2.2020): https://tools.ietf.org/html/rfc2544.

[55] D. Koslopp, CRUN: Distributed Processing in FPGA Accelerated Cloud, Master of Science thesis, Tampere University of Technology, 2018.

[56] T. T. Carneiro, Distribution of ultra-low latency machine learning algorithm, Master of Science thesis, Tampere University of Technology, 2018.

[57] 10G/25G High Speed Ethernet Subsystem v3.0, Xilinx. Available (accessed on 3.12.2019): https://www.xilinx.com/support/documentation/ip_documentation/xxv_ethernet/v3_0/pg210-25g-ethernet.pdf.

[58] FIFO Generator v13.2 LogiCORE IP Product Guide, Xilinx. Available (accessed on 3.12.2019): https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_2/pg057-fifo-generator.pdf.

[59] DMA/Bridge Subsystem for PCI Express v4.1 Product Guide, Xilinx. Available (accessed on 3.12.2019): https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf.

[60] AXI Interconnect v2.1LogiCORE IP Product Guide, Xilinx. Available (accessed on 3.12.2019): https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf.

[61] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, C. Wright, Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, RFC 7348, 2014. Available (accessed on 8.2.2020): https://tools.ietf.org/html/rfc7348.

[62] IEEE Standard for Ethernet - Amendment 3: Media Access Control Parameters for 50 Gb/s and Physical Layers and Management Parameters for 50 Gb/s, 100 Gb/s, and 200 Gb/s Operation, in: IEEE Std 802.3cd-2018 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018 and IEEE Std 802.3bt-2018), 2019, pp. 1-401.

[63] Exact Match Binary CAM Search IP for SDNet SmartCORE IP Product Guide, Xilinx. Available (accessed on 9.2.2020): https://www.xilinx.com/support/documentation/ip_documentation/cam/pg189-cam.pdf.