TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

DANIEL KOSLOPP

DISTRIBUTED PROCESSING IN FPGA ACCELERATED CLOUD

Master of Science thesis

Examiner: Prof. Timo D. Hämäläinen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 29th August 2018

ABSTRACT

DANIEL KOSLOPP: Distributed Processing in FPGA Accelerated Cloud
Tampere University of Technology
Master of Science thesis, 76 pages
December 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Timo D. Hämäläinen
Keywords: Hardware Accelerator, FPGA, NFV, Cloud, SDN, Machine Learning

Motivated by the need of cost reduction, better energy efficiency and agile update and deployment of new services, telecommunication industry is moving towards virtualization, which lead to Network Function Virtualization (NFV) standard. NFV leverages cloud technologies to deploy network functions that are traditionally implemented using dedicated proprietary hardware. Still, the performance provided by current cloud infrastructure does not fulfill the requirements for demanding NFV's use cases. Thus, hardware acceleration should be deployed.

The hardware programmability of FPGAs allows them to adapt well to many type of workloads, placing them as good candidates to be used as hardware accelerators in virtualized environments. In this thesis, the CRUN framework is proposed to provide FPGA as hardware accelerator resources in cloud, abstracting the integration complexity while enabling sharable and scalable use of such devices.

CRUN architecture allow user's acceleration hardware to be accessed locally and through the datacenter's network. The latter provide flexible connectivity by following the Software-defined Networking (SDN) principles. The architecture enables the same sharable FPGA to be used simultaneously as a co-processor, a network accelerator or as a distributed accelerator in a scalable scenario over several FPGAs.

In its current development state, CRUN was leveraged for inference of a machine learning application composed of a fully connected neural network. The main performance target was to achieve ultra-low latency, less than 40μs, for each inference at software level. Only CRUN fulfilled the requirement among the analyzed alternatives, where the architecture is capable of providing latency in the 30μs range in average. For context, high-end General-Purpose Processor (GPP) and Graphics Processing Unit (GPU) provided latency values of 798μs and 1 897μs respectively for the same application.

PREFACE

At first, I thank God for all the blessings in my life.

I would like to thank Nokia for all the support and working hours I had available to dedicate to this work. Also, I would like to express my gratitude to all my colleagues that are part of it: Anssi Örn, Kalle Holma, Pekka Jokela, Aki Kaihtela and Miika Jokinen.

Special big thanks to Jouni Markunmäki for the immeasurable technical guidance and support, Piia Saastamoinen for the careful review, Hannu Tulla for solving all possible problems with the laboratory and obrigado to Juho Tieaho for SDN development and the daily cooperation in any challenge.

I wish to thank Prof. Timo D. Hämäläinen for the opportunity to conduct my thesis under his supervision and all the assistance.

My most sincere thanks to my father Silvio Koslopp, my mother Maria Elisabeth G. Koslopp and my brother Denilson Koslopp, all of who have always supported me in all aspects and decisions in my life.

At last, my deepest gratitude, admiration and love to the greatest partner of my life, Talita Tobias Carneiro, for being my light and inspiration.

Tampere, 20.11.2018

Daniel Koslopp

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

AHU	Accelerator Hardware Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AVG	Average
BBU	Baseband Unit
BS	Base Station
CAPEX	Capital Expenditure
CLI	Command Line Interface
COTS	Management and Orchestration
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
EM	Element Manager
ETSI	European Telecommunications Standards Institute
FPGA	Field-programmable gate array
GPP	General Purpose Processor
GPU	Graphical Processor Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HWA	Hardware Accelerator
IaaS	Infrastructure-as-a-Service
ID	Identification
IP	Internet Protocol
ISG	Industry Specification Group
IT	Information technology
MANO	Management and Orchestration
MAX	Maximum
MIN	Minimum
MLP	Multilayer Perceptron
NAT	Network Address Translation
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	Network Function Virtualization Orchestrator
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
NOS	Network Operating System

N-PoP	Network Point of Presence
OPEX	Operating Expenses
OS	Operating System
PaaS	Platform-as-a-Service
PCIe	Peripheral Component Interconnect Express
PF	Physical Function
PNF	Physical Network Functions
PR	Partial Reconfiguration
PRR	Partial Reconfigurable Region
RAN	Radio Access Network
RRH	Remote Radio Head
SaaS	Software-as-a-Service
SDN	Software-defined Networking
SFC	Service Function Chain
SR-IOV	Single Root I/O Virtualization
TCO	Total Cost of Ownership
VF	Virtual Function
VHDL	VHSIC Hardware Description Language
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFC	Virtual Network Function Component
VNFM	Virtual Network Function Manager

1. INTRODUCTION

For a long time, the telecommunications industry has relied on physical proprietary devices for providing services. This practice ossifies the infrastructure not allowing them to easily update or innovate services due to the specialized and manual work needed for it. Meanwhile, it also increases complexity to maintain the facilities [87].

The rapid increase of data traffic is well known worldwide [19] along with the diversity and insertion of new services. All this requires scale as well as constant and fast modifications on the underlying infrastructure, which tied to inflexible environments lead to high costs.

For this reason, telecommunication operators joined efforts and proposed virtualization and COTS (Commercial off-the-shelf) hardware as the key solution for enabling a rapidly evolving infrastructure, culminating on the establishment of the NFV (Network Function Virtualization) [22]. The core principle behind this move is the decoupling of the function from the physical equipment that runs it.

Concurrently, virtualization and COTS are also the key idea behind cloud computing, which has been evolving for some time. The benefits provided by this scheme, namely the possibility of offering infrastructure, platform and software as a service directly translates into efficient use, cost savings and flexibility [67].

Cloud providers are constantly improving their facilities for supporting a wide range of use cases. Consequently, more and more the usage of their services is spread over several segments of industry. The flexibility, scalability and usability constitute key elements for developing and deploying such a diversified scenario.

Just as common cloud applications, NFV covers a wide range of services and applications. This means that the requirements also vary considerably. Many NFV's use cases can be directly deployed in current cloud infrastructure. In fact, most of early trials and proofs of concept of NFV applications have used them.

Yet, more demanding NFV services have requirements that are not met by resources provided in common cloud, namely GPP (General Purpose Processors). Cloud RAN

(Cloud Radio Access Network) is an example of such case, but even more straight forward use cases can face issues when executed in GPPs.

Motivated by the goal of supporting more applications and providing increased performance, which directly translate in added value and income, cloud providers have already started to introduce hardware acceleration in their infrastructure. Main common Hardware Accelerator (HWA) deployed is Graphical Processor Units (GPUs), which covers a range of applications but is not flexible and efficient enough for many of NFV's use cases. Similarly motivated by the limitations of current cloud systems, telecommunication operators and academia alike also have been researching and testing HWA solutions.

Recent efforts have been done for the deployment of FPGAs (Field-programmable Gate arrays) as HWA in cloud. This type of HWA adapts better for a wider range of workloads scenarios and use cases than GPUs and GPPs. Meaning that better performance and more energy efficiency may be obtained.

Still, FPGAs bring their own challenges for deployment in a virtualized environment, both for the provider and user. From the user perspective the development languages, such as VHDL (VHSIC Hardware Description Language) and Verilog, as well as tools and flows are significantly different from the ones software engineers are used to. Even when using currently available higher-level description languages the developer should have understanding of hardware design.

From the provider perspective, FPGAs insert heterogeneity in an already complex homogeneous system. This leads to difficulties in how to manage the resources and requires considerable changes in the software that orchestrate the infrastructure. Providers must implement a system that allow the FPGAs to be sharable, scalable and secure, while abstracting the hardware details when exposing the resources for development and deployment to the users.

This work presents a framework developed from the scratch for enabling the usage of FPGA as hardware accelerator in a cloud environment, motivated by NFV but not limited to it. The goal is to enable high performance and provide a scalable and flexible system while abstracting the complexity of managing and using it. There are some efforts available in academia with similar motivations as well as proprietary solutions in industry. Still the architectural details presented here are different, specially the usage of SDN (Software-defined Networking) to provide distribution of workloads over several accelerators.

The architecture developed is named CRUN. It provides abstraction of the connec-

tivity and expose standard interfaces for the user. A scalable system is achieved and allows the distribution of processing over several accelerators. The software management system proposed virtualizes the FPGA as a resource in the cloud. Furthermore, a distributed ultra-low latency machine learning inference report of a trial that leverages CRUN is presented. The trial was developed by a third party.

The work also briefly explains the main associated subjects and their relation, such as NFV, SDN and cloud computing. Cloud RAN is presented and used as an example of the motivations behind this thesis. Moreover, hardware acceleration in cloud is briefly reviewed.

The rest of this thesis is structured as follows. Chapter 2 describes the main virtualization related concepts, such as cloud computing, NFV, SDN. Hardware acceleration and FPGAs in cloud are reviewed in Chapter 3. Chapter 4 discusses and review related efforts in the field. Chapter 5 presents the hardware equipment as well as software tools and libraries leveraged. Chapter 6 details the proposed architecture. Chapter 7 shows and discuss the results obtained from the architecture and its performance comparison provided by the trial. Finally, Chapter 8 presents the final considerations and prospects for future work.

2. VIRTUALIZATION IN MOBILE NETWORKS

In this chapter important concepts like Network Function Virtualization (NFV), Software-defined Networking (SDN) and cloud computing are presented along with how they relate with each other.

Even though there are several efforts for inserting hardware acceleration in common cloud, one can assume that the need is accentuated in the telecommunication industry due to its demanding requirements. Thus, first Cloud RAN is introduced, which is shown here as an example of the motivations for inserting FPGAs as hardware accelerator in cloud.

Cloud RAN demonstrates well the reasons behind virtualization trends in the telecommunication industry, namely the NFV, as well as the challenges it imposes in cloud computing technologies.

2.1 Cloud RAN

The mobile traffic grow is well known, documented and experienced by the industry and users. Recent reports show an increase in global mobile traffic of 18-fold from 2011 (400 petabytes) to 2016 (7.2 exabytes) and forecast an 7-fold grow by 2021 (49 exabytes) [19]. On the other hand, average revenue per user does not compensate for the increase in Total Cost of Ownership (TCO) that traffic grow imposes in mobile operators [76, 13].

A simplified analysis of TCO can be break down to Capital Expenditure (CAPEX) and Operating Expenses (OPEX). CAPEX is related with costs for building the network infrastructure, while OPEX associates with operation and management of the network.

OPEX expense represents about 60% of TCO and is composed mainly by operation and maintenance, site rent and electricity. CAPEX cost examples are site acquisition, civil works, supplementary equipment as air conditioning and the actual hardware and software responsible for the wireless functionality. The latter is

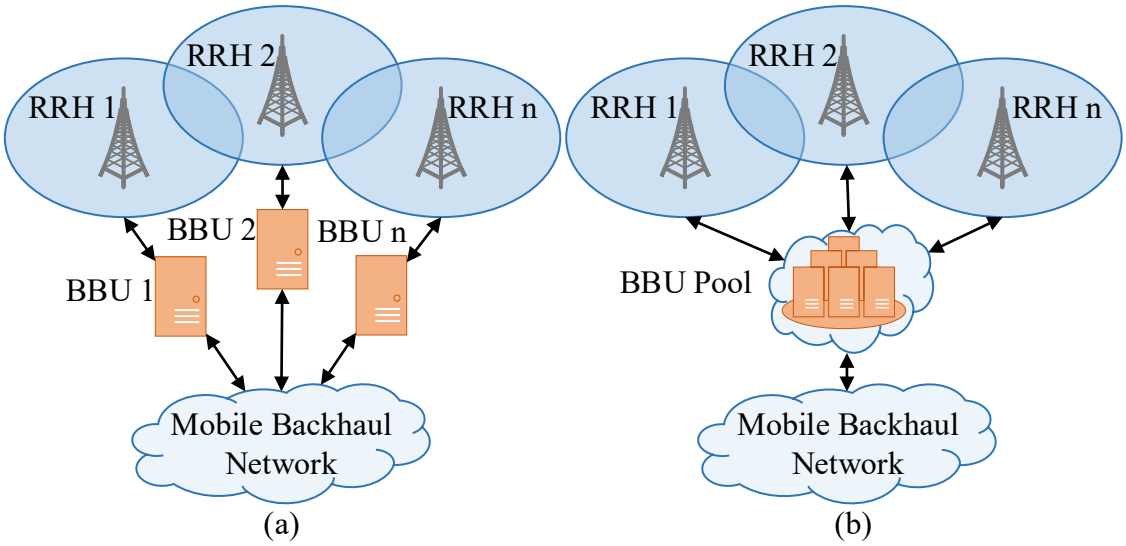

Figure 2.1 *RAN (a) vs Cloud RAN (b). Adapted from [13].*

what actually brings revenues and represents less than 50% of the CAPEX costs [15].

To support the afore mentioned growth, mobile operators have to improve their Radio Access Network (RAN) capacity, which architecture is traditionally designed to scale mostly with inclusion of more Base Stations (BSs). This solution quickly become prohibitively expensive and operators introduced the novel Cloud RAN [15].

Simply summarizing the RAN evolution, in the first wireless mobile architecture generations (1G and 2G) each network cell was a single Base Station consisting of an antenna located few meters away from a radio module. In the third generation (3G), the RAN is divided into Remote Radio Head (RRH), responsible for the analog to digital and vice-versa conversion, filter implementation and power amplification, and a Baseband Unit (BBU) that is mainly responsible for the signal processing tasks. In this configuration BBU could be located in more convenient and cost-efficient locations than beside RRH [13]. Finally, in the fourth generation (4G) and on the road for the fifth (5G), Cloud RAN is the evolution that leverages both wireless and IT (information technology) technologies by virtualising BBUs and sharing its storage and compute resources [62]. A high-level overview of the difference between traditional RAN architecture and Cloud RAN is shown in Figure 2.1.

The main benefits of Cloud RAN can be categorized as follow [76, 38]:

- Reduced Cost: Concentrating computation and sharing resources in a single

datacenter reduces OPEX by simplifying management, maintenance and operation. Also, the more efficient utilization of the equipment achieved through virtualization reduces CAPEX cost.

- Energy Efficiency: The number of individual BBUs are decreased and enables finer control for setting some BBU to low power and even turning it off. Also, there is no need to dimension several BBUs for the peak traffic of its location, since the dynamic loads of various locations may even out each other, i.e. some business area has high demand of traffic during day time, while house areas are mostly idle and vice-versa during night.
- Spectrum Utilization Efficiency: Centralization facilitates low latency sharing of information among BBUs, like base stations and user equipment link, traffic data and control services, which enables multiplexing more streams on the same channel with less mutual interference and consequently increasing capacity.
- Scalability: It becomes easier to add more resources or upgrade them to increase compute and storage capacity in the BBU. Also, RRH can be scaled to increase coverage and capacity faster and at lower cost since installation mainly requires the antenna and feeder systems.

Even tough Cloud RAN is a prime technology for enabling 5G mobile network [46], it does impose some challenges, such as [13]:

- High bandwidth, strict latency and jitter: the fronthaul transport network (between BBU and RRH) requirements may be 50 times larger than the backhaul (among BBUs and Mobile Backhaul Network)
- BBU Cooperation, Interconnection and Clustering: Sharing user data, scheduling and channel handling for interference control require BBU coordination, which in turn requires reliability and security mechanisms.

The challenges may be better understood from Table 2.1, which compares Cloud RAN requirements with common applications in cloud computing. One realizes that current cloud computing technology does not offer a ready made solution for telecommunication operators. Cloud RAN is an example of virtualization trends that motivated the foundation of the Network Function Virtualization standard.

In fact, Cloud RAN is one of the use cases covered by NFV [25, 87]. Hence, it is a narrower example of NFV's requirements since it varies for other use cases and can be even more demanding.

Table 2.1 *Cloud computing and Cloud RAN requirements. Adapted from [13].*

	Cloud Computing	**Cloud RAN**
Data rate	Mbps range	Gbps range
Data profile	Bursts and low activity	Constant stream
Latency	Tens of ms	Hundreds of μs
Jitter	Tens of ms	ns range
Information Life time	Long (content data)	Extremely short
Recovery time	s range	ms range
Number of clients	Thousands to millions	Tens to hundreds

2.2 Network Function Virtualisation

The challenges of virtualization are not limited to mobile operators only but the whole telecommunication industry. To address it, seven world's leading telecommunication operators and the European Telecommunications Standards Institute (ETSI) founded in 2012 the Industry Specification Group (ISG) for Network Functions Virtualization (NFV) [22].

Broadening the scope from Cloud RAN and mobile operators to the telecommunication industry in general, networks traditionally contain several dedicated proprietary hardware to execute network functions, also referred as middle-boxes or Physical Network Functions (PNFs). Example of such middle-boxes are Network Address Translation (NAT), Firewall and Deep Packet Inspection (DPI).

The constant increase in diversity of services and demanding requirements is proportional to the number of PNFs in the network infrastructure. At the same rate, the complexity of deploying them rises due to the specialized and manual work needed for it. Also, incompatibility among middle-boxes is frequent as well as diagnosis of failures or misconfiguration is difficult. Furthermore, the fact that they are fixed in some physical and logical location and the inability to easily move or share them make the network inflexible and ossified. This issues directly translate in slow and costly process for a network provider to install, maintain or update any service [87].

NFV aims to change this scenario by standardizing how to leverage virtualization and change the way telecommunication operators infrastructure their network. Instead of deploying middle-boxes, they are implemented in Commercial-Off-The-Shelf (COTS) equipment in the form of Virtual Network Functions (VNFs) as shown in Figure 2.2 [30]. This effectively decouples hardware from software and brings flexibility for faster update and deployment of new services in the same hardware and enable dynamic scaling [51]. NFV target benefits such as improvement in energy efficiency, decreasing the equipment cost, faster update and deployment of new ser-

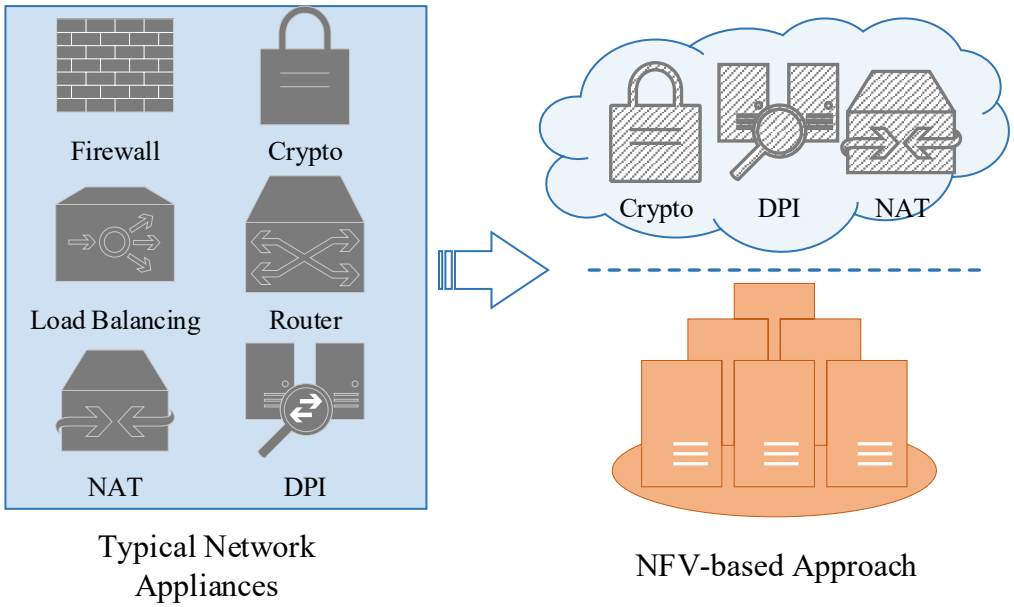

Figure 2.2 *Traditional network functions in middle boxes are deployed as VNFs in COTS hardware. Adapted from [30].*

vices, and provide a scalable and elastic ecosystem [87, 51].

Simply speaking, NFV is the cloudification of network functions. Throughout this work, traditional cloud computing is called common cloud, while NFV cloud refers to a cloud infrastructure used for deploying VNFs.

ETSI divides NFV architecture in three main layers: VNF Layer, Network Function Virtualization Infrastructure (NFVI) and Management and Orchestration (MANO) [23]. Figure 2.3 depicts this architecture.

2.2.1 VNF Layer

VNF is the virtual version of a PNF using virtual resources like Virtual Machines (VMs) in the NFV Infrastructure, providing the same functionalities of their physical counter-part. VNFs may be composed of one or several VNF components (VNFC). For example, one VNF can span several Virtual Machines, where each is one component, across multiple physical servers. The life cycle control of those components, such as instantiation and configuration, is the responsibility of Element Manager (EM). In the same context, one or more VNFs can be grouped to form a service [51].

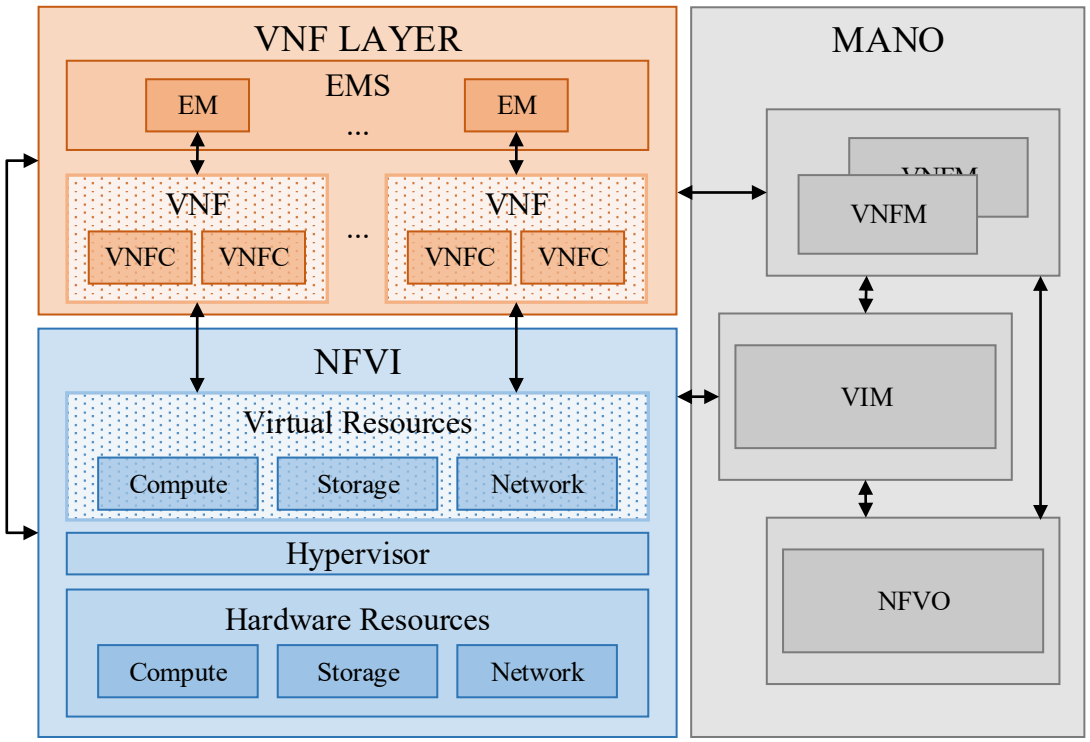

Figure 2.3 *NFV Architecture. Adapted from [87].*

2.2.2 NFVI

NFVI combines both hardware and software where the VNFs are deployed. In this layer hardware is decoupled from software [51].

The physical resources provide compute, storage and network functionalities from COTS equipment. Example of compute resources are x86 servers or hardware accelerators that can be applied for performance improvement. For storage, Direct Attached or Network Attached Storage servers can be used and for network standard switches are applied [87].

Those resources are then abstracted by a virtualization layer. Usually, virtual compute resources are exposed to the VNF layer as Virtual Machines using an hypervisor such Linux KVM [42], but container technology can also be used.

Virtual networking resources interconnect the virtual compute and storage nodes following the physical networking principles, but it must be aware that nodes may be located in the same host or not [31].

Following the same principle, virtual storage resources expose scalable and flexible pools of storage and also bring features such as backup and snapshots [87].

Software acceleration may be implemented in the virtual layer similarly to hardware accelerators in the physical layer, some common examples are Data Plane Development Kit (DPDK) and Single Root I/O Virtualization (SR-IOV) [40].

NFVI is not a complete solution for NFV and different service providers can and are building their own NFVI depending on their requirements [87].

2.2.3 MANO

MANO is responsible of managing and orchestrate hardware and software resources and their life cyle in the NFVI layer. It also manages VNF instances, their placement and their life cycle. Furthermore, it includes database to store information of the VNFs and NFVI [51].

Due to such complex and wide scope MANO is further divided into three sub elements: NFV Orchestrator (NFVO), VNF Manager (VNFM) and Virtual Infrastructure Manager (VIM) as shown in Figure 2.3 [87].

NFVO chains and orchestrate multiple VNFs to provide services, it includes the responsibility of finding the optimal path and placement of the VNFs accordingly to the requirements. VNFM manages multiple instances of any type of VNF, including life cycle from instantiation to termination. Finally, VIM control and manages NFVI compute, storage and network resources [87].

ETSI presents a reference architecture for the MANO and it is known that NFVO, NFVM and VIM borders are blurry. Thus, many implementations of MANO may not be directly mapped to them [87]. Furthermore, the support for heterogeneity, i.e. hardware accelerators, is an open question still [51].

To summarize the NFV architecture, a visualization of the main terminology and how they are related can be seen in Figure 2.4. Virtualized network functions are referred in the standard as (VNF) and the Element Manager (EM) is responsible for controlling its life cycle. A NFV Infrastructure (NFVI) provides the necessary resources where VNFs are deployed. This deployment can be constituted of one or multiple VNFs connected to form a Service Function Chain (SFC). Due to strict requirements and high coupling of VNFs in a SFC, the location of those in the NFVI is important and are referred as Network Point of Presence (N-PoP). The control of resources and connection among VNFs in the NFVI is performed by the Management and Orchestration (MANO) element, which is further divided by the standard in Virtualized Infrastructure Manager (VIM), VNF Manager (VNFM) and NFV Orchestrator (NFVO) [87].

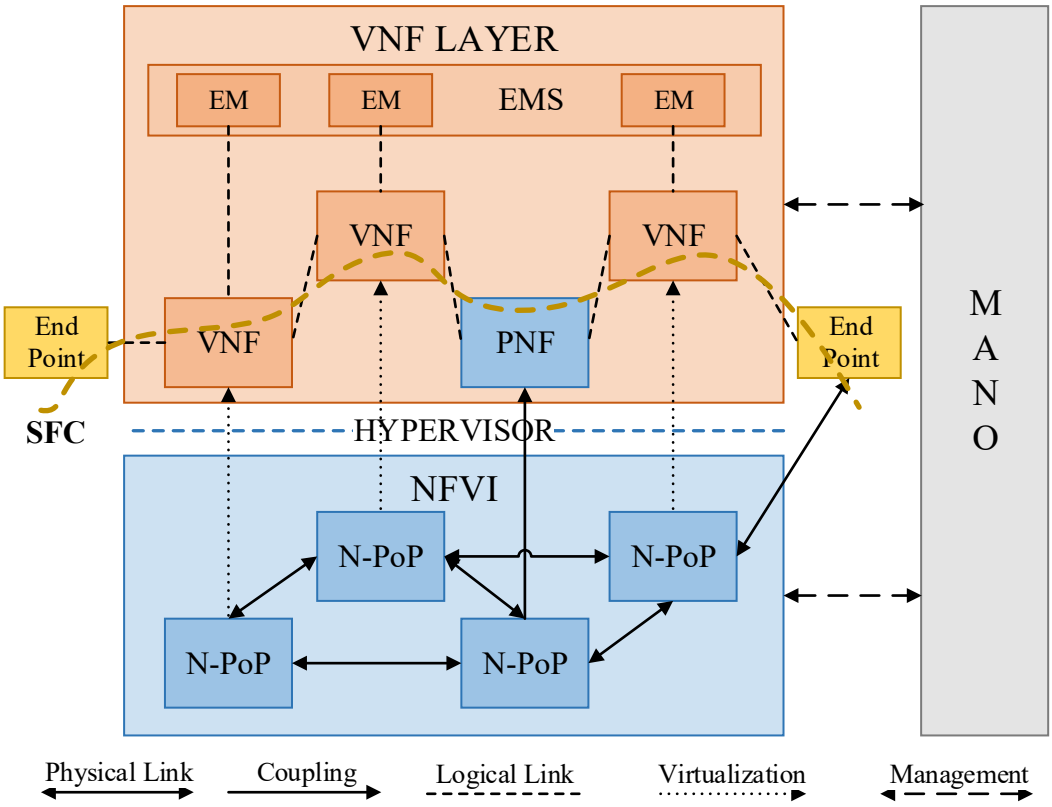

Figure 2.4 *NFV's main terminology and their relations. Adapted from [87].*

Note that a VNF can coexist with a PNF when forming a SFC, it is expected and especially important in the early implementation phase of NFV.

2.3 Cloud Computing

Cloud computing is a key technology in NFV. As many concepts on computer science, the idea behind cloud computing is not as new as it may seem. As early as 1961, Professor John McCarthy suggested the concept of utility computing, in which he envisioned computing as a public utility, just as the electricity and telephone system [1]. This early concept proposed that not only computing power, but specific applications would be sold in a utility-type business model in the future [66]. This idea was revitalized in the past two decades and resurfaced as cloud computing [1, 66].

There are many definitions for cloud computing, industry and academia alike have composed several meanings. According to the National Institute of Standards and Technology (NIST) cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g.,

networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [48, p. 2].

From the NIST definition one can point out five essential characteristics of cloud computing [67]:

- Scalability: Resources must scale up and down fast and as needed;
- Measurable services: Services must be controlled and monitored by cloud provider for billing, access control, resource optimization and other purposes;
- Automation: Tenants can use services on-demand without human interaction;
- Ubiquitous: Cloud is available over the network and can be easily accessed;
- Shared: Physical and virtual resources are assigned and reassigned on consumer demand who usually has no control of its exact location.

Computing is treated as utility in cloud. Thus, the user, also called tenant, pay for its usage as one pay for water and electricity, lowering costs since resources are essentially rented on demand [67]. This means a business paradigm shift in which third parties are contracted for delivering commodities of computing power, data storage and services to enterprises and customers [1].

Virtualization and cloud orchestration are key technologies enabling this paradigm shift and may be considered as one of the foundations of cloud computing [77, 6]. Furthermore, there are two important concepts within this context, the deployment mode and the service model of cloud computing. The next subsections briefly explain these terms.

2.3.1 Virtualisation and Orchestration

Virtualization is a technology used for running multiple independent virtual operating systems on a single computer [1], as such, the underlying physical resources are abstracted away by logical ones. The objectives of this abstraction are agility, flexibility and energy-efficient resource utilization [67], bringing further benefits such as hardware independence, availability, isolation and security [77]. There are two main techniques to achieve virtualization: hypervisor-based and container-based. Figure 2.5 compares them.

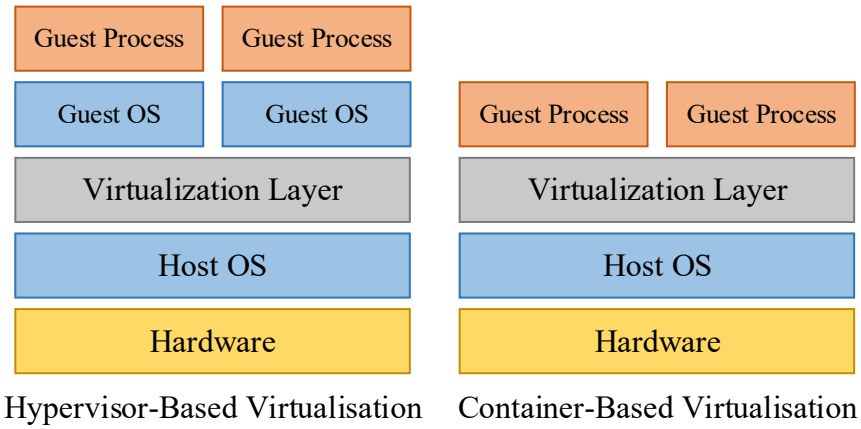

Figure 2.5 *Virtual Machines vs Containers. Adapted from [77].*

Most common form of virtualization is hypervisor-based, which inserts a software layer to provide abstraction of multiple virtual resources on top of a physical one (host). These virtual resources are called Virtual Machines (VMs). The hypervisor runs on host Operating System (OS) and provides an isolated execution environment to each VM allowing them to have their own OS, usually called guest OS. In practice it means that one host OS can execute multiple different guest OSs. Some well known hypervisors are Xen, VMware and KVM [77].

VMs impose overhead and degrade performance. A lightweight alternative is container-based virtualization. Containers are multiple isolated user-space instances that run directly in the physical machine at the OS level [77]. Containers do not provide the same level of isolation as VMs and may introduce security issues but have less footprint [87]. Docker [21] is probably the most well-known container platform.

To realize the full potential of a cloud, resource management is needed. Cloud orchestration controls and arranges the underlying hardware and hypervisors to provide users the required resources as efficient as possible. In practice the orchestrator controls the sharing of resources among several users. This is a complex task due to the need for scalability, heterogeneous resources and several constrain from the limited capacity [6]. OpenStack [58] is a widely used cloud orchestrator.

There are three main requirements for the cloud orchestrator [6]:

- Visibility: The system has to monitor all cloud resources and expose to user their availability, status, placement, cost and any other information required;

- Orchestration: The allocation must guarantee that user is provided with the agreed resource, such as bandwidth and latency, while coordination must ensure correct configuration and execution of the resources;
- Provisioning: Users and provider must coordinate sharing of statistics and resource utilization to optimize the system, using techniques such as auto-scaling and failure recovery.

With proper orchestration and virtualization, a datacenter with its limited number of physical servers can be shared among several users, since one single host can execute many guests simultaneously [67].

2.3.2 Deployment Modes of Cloud Computing

There are four deployment modes as defined by NIST [48], categorized as below. This classification refers to the ownership of the cloud datacenter [61].

- Public Cloud: this category describes the environment own by a third party provider that exposes their services via the Internet [67, 61]. Resources are dynamically provisioned on a self-service basis in which the availability is done in a pay-as-you-go manner to the general public [61]. Famous examples are Microsoft's Azure [50], and Amazon AWS EC2 [4, 1].
- Private Cloud: the management of data and process is handled within the organization. In this sense, there are no restrictions as in public cloud services related to network bandwidth, security exposures and legal requirements. Some examples cited here are Amazon VPC and OpenStack [1].
- Community Cloud: is constituted by a group of organizations sharing the same interests, being specific security requirements an example. The group members share the access to the data and applications [67].
- Hybrid Cloud: is the combination of the Public and the Private Cloud modes, as such, an organization can run some applications on an internal cloud infrastructure while still running others in a Public Cloud. In this sense, the main advantage for a company is to benefit from scalable resources offered by third party service providers while being on control of specific applications or data [61]. Examples are RightScale and QTS [1].

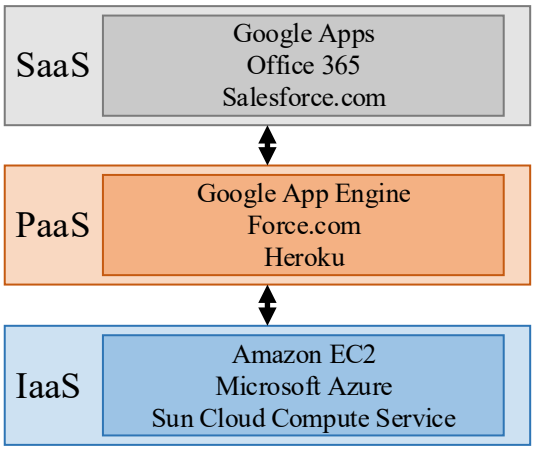

Figure 2.6 *Service Models and the three layers. Adapted from [61].*

The typical choice of telecommunication operators is the deployment of private clouds. Yet, this work is not limited to it since the main difference among the deployment modes is who owns and manages the cloud and not its infrastructure.

2.3.3 Service Models of Cloud Computing

With regard to the type of the service offered, the NIST definition specifies three distinct groups, as showed below [48]. These models are widely known "as a service". Figure 2.6 shows their correlation.

- Software-as-a-Service (SaaS): one or more providers owns the software, its delivery and remote management. They are offered in a pay-per-use manner. It constitutes the most visible service in this context, since the end consumer actually access and uses the software [61]. A single instance of the object code and the correspondent application database must be shared along common resources for supporting multiple customers in a simultaneously manner. Important examples are Salesforce.com and Oracle [1].
- Platform-as-a-Service (PaaS): these offerings are intended to software developers [61]. The key idea here is to provide developers with the systems and environments that they need, from an end-to-end life cycle perspective, comprising developing, testing, deploying, and hosting of applications [1]. In this sense, there is no need to worry about the underlying layer, which is the hardware infrastructure (IaaS), it means an easy to use environment for developing applications and services over the Internet [61]. Key examples are Google App Engine and Microsoft Azure [1].

- Infrastructure-as-a-Service (IaaS): as showed in Figure 2.6 this service model constitutes the lowest abstraction layer. It offers the computing resources directly, as processing power and storage in the format of a service over the Internet [61]. The provided infrastructure can be scaled up or down depending on the needs [67]. Usually IaaS is offered in a virtualized infrastructure, in which they are exposed to the upper layers through standardized interfaces as unified resources, where the user can create its own VMs, for example. As providers one may cite for processing Amazon Web Services with its Elastic Compute Cloud (EC2) and for storage Simple Storage Service (S3) [61].

The scope of this thesis is the IaaS level where the hardware accelerator is exposed to upper levels as a virtualized resource.

2.4 Software-defined Networking

In cloud computing, NFV and networks in general, switches and routers are key elements that enable flow of information around the world in the form of digital packets. Although highly pervasive, they are known to be complex and challenge to manage due to the usage of low-level and often vendor-specific languages. These characteristics lead to low flexibility, halt network evolution and increase costs [41].

Any update, new feature or change in the network functionality is complex since they need to be implemented directly into the network infrastructure [56]. A clear example of the problem is the transition from IPv4 to IPv6, which has taken more than a decade and is still ongoing, even though it is a protocol update only [41]. This environment, also so-called Internet "ossification", is attributed mainly to the tight coupling of data and control planes in the network devices [56].

Software-defined Networking (SDN) principle is exactly the separation between control and data planes [56]. SDN is still recent and growing at very fast pace, consequently its definitions may be fuzzy among literature. The objective of this work is not to debate over different views, as such, to avoid ambiguity the four pillars architecture definition provided by Kreutz at el. (2015) is used to identify the requirements of a SDN enabled device [41]:

- The control and data planes are decoupled;
- Forwarding decisions are flow based, instead of destination based;

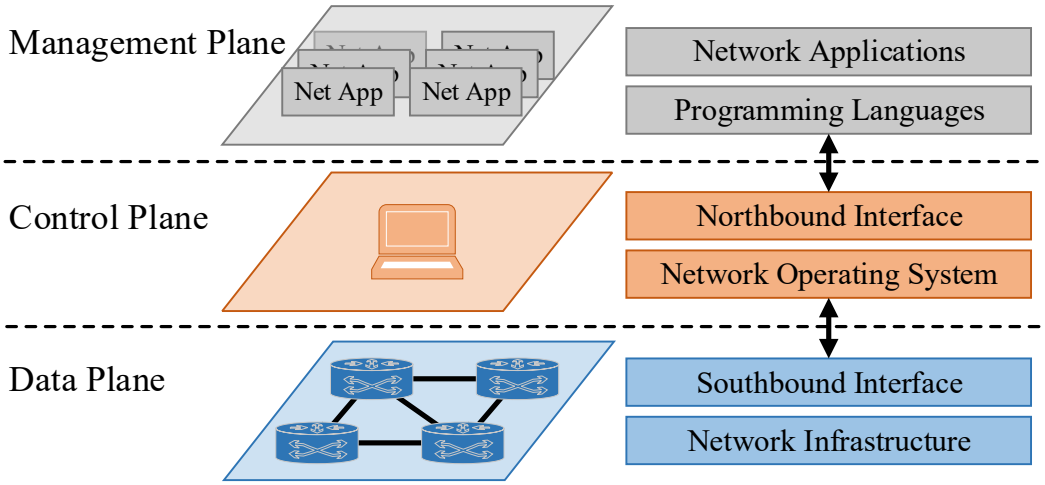

Figure 2.7 *Software-defined Networking planes and layers. Adapted from [41]*

- Control logic is moved to an external entity, called Network Operating System (NOS);
- The network is programmable through software applications running on top of the NOS.

Summarizing SDN architecture, the data plane is responsible for analyzing each packet and efficiently decide what to do with it, for example forward to some port or drop. The control plane is responsible for translating network policies, i.e. forward rules, so they are recognized by the data plane, which in turn will enforce the policy by processing the packet accordingly. On top of the control plane resides the management plane, which contains network applications that define the desired behavior of the network through some programming language that in turn abstracts away the actual implementation of the policies [41]. Figure 2.7 shows plane and layers views of SDN and next subsections go into more details.

2.4.1 Data Plane

Data plane in SDN is simplified and composed basically from forward devices that leave all the intelligence to the control plane. These devices expose some standard interface called southbound [41]. There are multiple standards that can currently be used to fill the southbound interface layer, like OpenFlow, ForCES and POF [20]. Arguably, OpenFlow is the de-facto SDN standard and mostly widespread

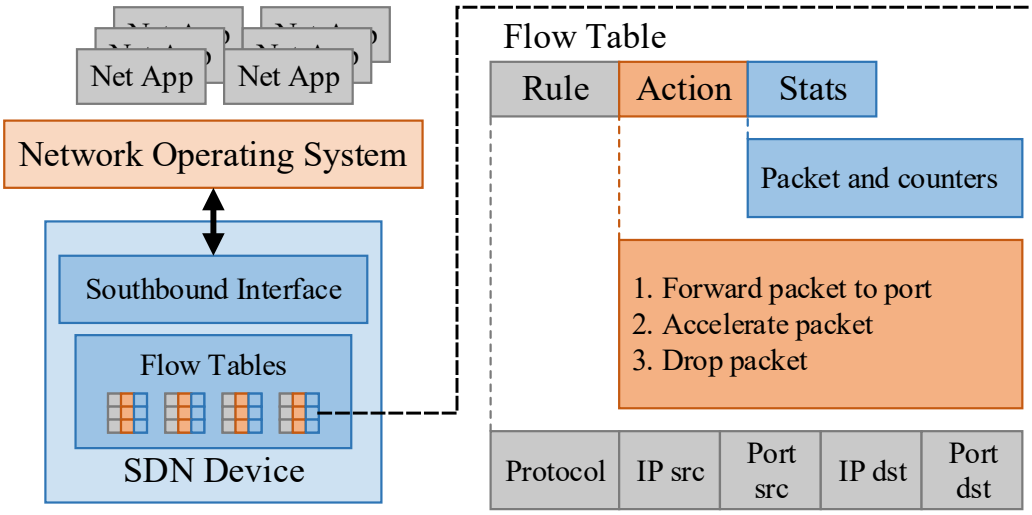

Figure 2.8 *OpenFlow-enabled SDN devices. Adapted from [41].*

[41, 56]. Thus, it is described here how an OpenFlow-enabled device functions to better explain how the data plane actually works and is separated from the control plane.

Figure 2.8 shows the components of an OpenFlow-enabled device. In these equipment, header fields of incoming packets are matched against headers in the flow tables of the device. Depending on whether a match is found a specific action, i.e. forward or accelerate is taken, if a match is not found the device can be configured to drop the packet or forward it to the controller so the tables can be updated accordingly [20]. The tables can be pipelined and also include statistics field that can be fetched by the controller to visualize the network behavior. This functionality enables a device to be controlled to behave as a router, switch or even more complex roles as traffic shaper, load balancer, and other depending on what kind of actions it can execute on the packets [41].

Furthermore, depending on performance requirements no specialized hardware is needed and the forward device can be fully virtualized in COTS hardware [51].

2.4.2 Control Plane

The SDN controller is frequently regarded as the operating system of the network, thus the name NOS, and in practice it abstracts away the application layer from the low level details of the hardware [20, 41, 56].

As traditional operating systems, NOS should provide essential services and common APIs (Application Programming Interfaces) to developers. Among essential services one can mention device management and discovery, shortest path, topology information, statistics, notification and security mechanisms. There are several controllers and platforms due to the number of competitors fighting to be at the forefront of SDN. Hence, there are no clear standard and they vary greatly in architecture and features [41]. The main aspects that differentiate them are:

- Centralized vs Distributed: Centralized controllers can provide enough performance for a dense data center but may suffer scalability issues and is a single point of failure, while distributed can scale better and be more resilient but are naturally more complex. [41]
- Packet vs Flow: Packet is the basic network unit but a per packet control may imply overhead, on the other hand, applications usually send many packets that can be grouped as a flow. [56]
- Reactive vs Proactive: In reactive control every time an unknown packet/flow arrives, it is forwarded to the controller to decide the action and update flow tables, this increases the delay of the first packet, which may or not be a problem. On the other hand, in proactive control new flows are kept in the data plane and the controller do not need to be consulted, usually.

OpenFlow and other southbound languages standardize the hardware interface, but it not necessarily makes the process of configuring them easy. Hence, they are usually compared to low-level language of x86 platforms such as assembly [20, 41]. More complex operations and orchestration of the network are realized in the management plane through applications [20]. Applications and NOS are connected by the northbound interface. Aligned with the controller diversity, no clear standard can be determined currently [20]. Furthermore, an east-west bound interface may be present, especially in distributed NOS architectures, since NOS interact with each other through it. However, these interfaces are usually private and incompatible among different controllers [20].

2.4.3 Management plane

Network applications reside in the management plane. Once more when comparing with x86 platforms, network applications are developed using high-level programming languages and also run on top of NOS. The main purpose of such high-level

languages are to abstract further the task of programming forwarding devices, assist software reusability and speed up development. Several high-level programming languages have been proposed, a comprehensive list and their approach can be found in [41].

As an example, a common network application is load balancing. One can imagine multiple workers executing the same heavy process in packets of a group of flows, the task of this application is to keep the load of the workers balanced so no flow is excessively delayed or dropped due to the limits of a single worker. Furthermore, it can be tuned to reduce power usage in periods of reduced load by directing all flows to a limited number of workers, allowing others to move to a low power state. To achieve this, the application must instruct the controller to install and update the forwarding rules and policies of the devices. [41]

In [41] it is provided detailed network application examples and references to them. Among a wide variety of use cases, SDN applications can usually be categorized as follow [41]:

- Traffic Engineering: Load balancing, energy aware routing, scheduling and Quality of service (QoS);
- Mobility and Wireless: RAN virtualization (Cloud RAN), interference management and programmable virtualised WLANs.
- Measurement and Monitoring: Active and passive measurements and monitoring of QoS parameters;
- Security and Dependability: Attack detection and mitigation, security flow rules privatization and fine-grained access control;
- Datacenter Networking: network utilization optimization, live network migration and workload prediction.

2.5 Cloud Computing, NFV and SDN

Cloud computing, NFV and SDN are enablers for a revolution in how networks are implemented and monetized, being NFV the one that unify them and which brings higher value for telecommunication operators [30]. In [51], they are classified as an abstraction of different resources, being compute for cloud computing, network for SDN and functions for NFV, as such, they are very related. NFV, for example,

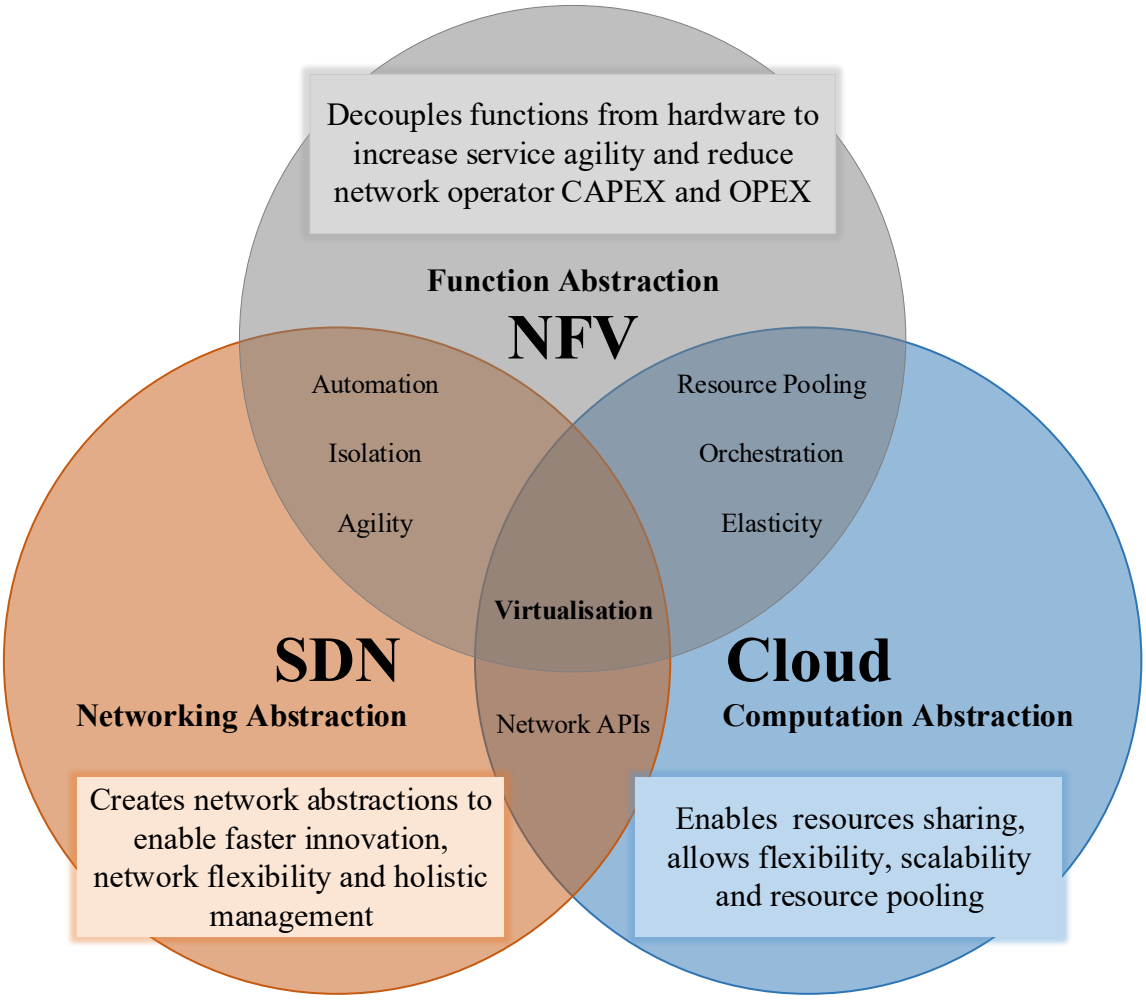

Figure 2.9 *NFV, Cloud Computing and SDN. Adapted from [51].*

leverage cloud computing technologies to deploy VNFs while SDN may use the same technologies to implement one or multiple SDN controllers on demand [51]. Meanwhile, cloud computing may benefit from SDN applications and VNFs to automate and optimize datacenter's network [41]. Figure 2.9 summarize their relationship.

Additionally, NFV and SDN highly complement each other. For example, a SDN application, such as load balancing and monitoring, may be implemented as a VNF in a service chain. In this way SDN benefits from running in the NFVI while NFV can use SDN's features to automate complex service chains deployment [51]. Still, while very powerful when used together, NFV and SDN can be deployed independently of each other [87].

Moreover, the IaaS service model in cloud computing can be directly mapped to the NFVI layer in NFV's architecture, providing both physical and virtual resources.

Thus, most NFV early trials have been deployed using dedicated VMs in common cloud. Furthermore, VNFs and services can be compared with SaaS [51].

Yet, since NFV applications are mainly originated from telecommunication industry, they impose different requirements than commonly deployed cloud applications, such as high pressure on processing performance, harder network demands and stronger availability and reliability needs, as shown in Cloud RAN in Section 2.1 Consequently, NFV will most probably change considerably when compared with common cloud [51].

Although correlated, NFV and cloud are not the same. NFV focuses on function virtualization and open the scope to provision of services, while cloud focuses on resource virtualization [87]. NFV brings new challenges to the common cloud and intensifies the existing ones, such as:

- VNF performance
- Energy efficiency
- VNF deployment and placement
- VNF life cycle control and migration
- Service chaining
- Performance evaluation
- Policy enforcement
- Security, Reliability and Portability

This work concentrates mainly in the performance issue. More about the other items can be found in [87, 51, 31].

Performance requirements of applications should be guaranteed but it is an challenge even in non-virtualized hardware at high speeds [54]. Moreover, COTS are known to be weaker in terms of performance and reliability when compared to specialized hardware [87]. Packet processing, encryption and decryption are examples where GPPs perform poorly.

To address the question whether virtualized hardware can provide high and predictable performance while assuring portability, ETSI created the “NFV Performance & Portability Best Practises” specification [24]. This specification provides

recommendations of minimum features and requirements the hardware and hypervisor should support and also reports performance test results on NFV use cases. The results show that when using high-end servers and applying the recommendations, performance was consistent, predictable and portable as desired for the cases covered [51], showing that COTS hardware can support many applications requirements.

Yet, even though it is desired to have a virtualized environment composed of COTS hardware only [87], not all VNFs may achieve the performance requirements in this scenario as shown in [28]. Hence, studies show that hardware acceleration techniques will also be important in NFV [51]. Specialized hardware is against NFV's concept, nonetheless, in practice a trade-off among performance, cost and flexibility is needed [87, 51].

3. HARDWARE ACCELERATION IN CLOUD

In this chapter the main hardware accelerators used for cloud and how they are deployed are reviewed, along with the connectivity options and what type of workloads they best fit. The final section then go into details of why FPGAs should be leveraged for cloud acceleration, the requirements for doing so and how they are usually deployed.

3.1 Accelerators

Unlike homogeneous systems composed of only General-Purpose Processors (GPP), heterogeneous systems introduce specialized hardware devices, also called Hardware Accelerators (HWAs), that are better suited for certain type of work. The motivation of such systems are performance and energy-efficiency requirements that may not be achieved with GPPs only systems in high demanding applications [29].

Some well-known HWAs currently used are Application-Specific Integrated Circuit (ASIC), Graphic Processing Units (GPUs) and Field-programmable Gate Array (FPGA) [57, 11].

GPUs are deployed mostly as co-processor and are widely used with General Purpose Processor (GPPs) to improve performance. Currently, most big players in cloud computing, such as Azure [5] and AWS [4], already offer scalable GPU-enabled instances in its infrastructure, providing on demand acceleration for application such as machine learning training and inference, streaming, gaming and video encoding.

ASICs are integrated circuits that offer unique features for specific applications, such as PNFs in NFV that are usually composed of such devices. On one hand, ASIC typically provides the highest performance and energy efficiency with smallest chip size. On the other hand, designing such devices that realize these advantages require considerably more development time. Also, each new feature or design error found after taping out of the chip requires a new set of masks for silicon fabrication as well as more time for a new tape out process. The ever increase complexity of designs aggravates it, since huge efforts and time in verification are needed to avoid errors.

This yields a very high cost which is only mitigated in applications that need high volumes of chips [88]. Furthermore, since ASICs have specific purpose, they provide very limited programmability [54].

FPGAs are COTS silicon devices that provide programmable digital circuits. Simply speaking, FPGAs contain several basic elements such as configurable logic blocks, registers and memory that are connected via programmable interconnects. This allows designers to develop custom hardware that takes most advantage of parallelism and data path of an application, in other words, the hardware can be configured at run time to best fit specific applications [43]. As with ASICs, FPGAs offer potential flexibility for many workloads [64], but with no fabrication process, faster development time and the ability to update or fix the design at any time by simply reprogramming the FPGA [26]. On the other hand, FPGAs do not match the performance of ASICs because of their internal structural overhead [39].

3.1.1 Workload Characteristics

As mentioned before, GPPs are capable of providing sufficient performance for many applications but may not be enough in all use cases. These more demanding functions can be broadly classified in two types, Compute-intensive and Network-intensive [7]. They are mainly characterized by the amount of computation needed, latency and how dynamic is the data.

Compute-Intensive functions requires heavy computations and GPP resources in relatively static data. Example of such functions are big data, security, machine learning training and inference, media and games [7]. This type of workload can be further divided into responsive or not. Non-responsive Compute-Intensive workloads process huge chunks of data while latency can be on minutes to day's range. Examples of such applications are machine learning training and scientific calculations. Responsive ones on the other hand require relatively short latency values on moderate amount of data. Example of such application is non-real-time machine learning inference.

Both responsive and non-responsive Compute-Intensive applications can be accelerated using look-a-side model. In this type of acceleration data can be transferred from GPP's memory to accelerator using batches, a group of inputs, where the whole batch is processed and results are send back to GPP's memory. The size of the batches can then be adjusted to match the latency requirements.

Network-Intensive functions on the other hand process highly dynamic amount of

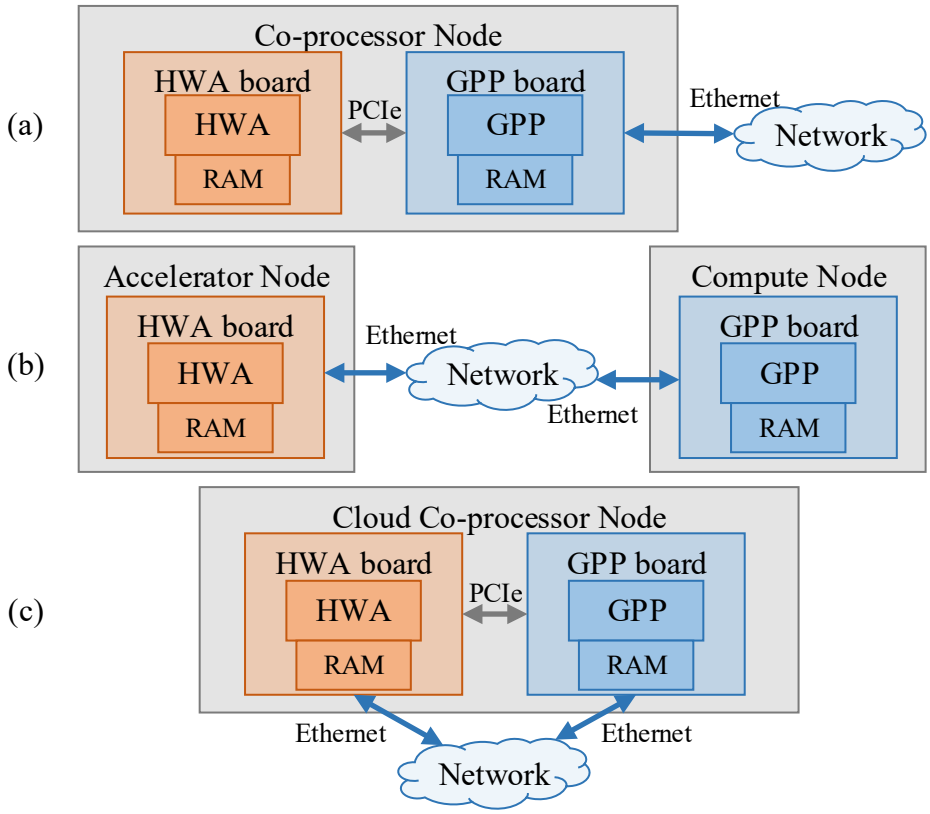

Figure 3.1 *Hardware accelerator attachment options: (a) Tightly coupled; (b) Network attached; (c) Tightly coupled and network attached. Adapted from [39].*

data with very short latency [7]. This type of function tends to exhaust memory bandwidth of GPPs architectures [55]. This workload is usually processed in a stream manner and examples of applications are NAT, load balancing, streaming video processing and machine learning inference with tight latency requirements.

Network-Intensive workloads are good candidates for in-line acceleration model. This type of acceleration processes the packets while they traverse through the accelerator.

3.1.2 Connectivity Options

One can point out three main options in how to connect HWA with GPPs as shown in Figure 3.1: Tightly coupled (a), Network connected (b) and combination of both (c) [39]. Figure 3.1 shows these options using PCIe (Peripheral Component Interconnect Express) and Ethernet as examples of interfaces.

The most prevalent type of accelerators systems are the ones composed of GPPs and co-processors [39], where the GPP offload compute intensive tasks to the co-processor that is tightly coupled to it [29]. Tightly coupled GPPs and HWA are usually connected to each other using some coherent memory mechanism or direct memory access (DMA) [10, 39]. The accelerator chip can be located in a daughter attached card, in the same board as the GPP or even in the same die [33, 73]. For GPUs and FPGAs the most common option is adding a daughtercard using PCIe [39, 73].

Tightly coupling accelerators with GPPs in the same board or die provide better latency values and can potentially easier DMA and coherence [64]. However, this approach suffer from scalability, resilience, size and power issues [73, 64]. They are expected to be used for very specific applications [73]. Using daughtercards and PCIe connectivity partially solves scalability since more chips and/or cards can be added, but if an application needs more devices than the number available it cannot be implemented. Also, if less are needed the system is over-provisioned [64]. In neither case tightly coupled accelerators scale across servers [10].

Network-only connect accelerators are directly hooked to the datacenter's network. Co-processing in this approach may not be efficient due to the higher response time and the need of constant communication with the host [39]. Thus, the accelerator has to work as standalone and capable to communicate with other resources over the network [73]. This approach increases scalability and flexibility compared with tightly attached option, since in this configuration accelerators can be accessed remotely, deployed independently of the number of hosts and allow user defined topologies [74].

Both previous options are good for some workloads but are not generic enough. A third and more flexible alternative is to provide both attachments, tight coupling and network connectivity [39]. This configuration covers more application scenarios, such as, local acceleration over the tight connection, network acceleration over the network connection and global acceleration using pool of remote HWA available from the network [11].

In more details, local HWA occurs when the accelerator works as a co-processor for the GPP and is ideal for Compute-Intensive tasks, as long as the local accelerator has enough capacity to handle it. Network acceleration are good for Network-Intensive workloads like processing incoming or leaving packets of the host. In the case where one accelerator is not enough for large-scale applications, network connectivity provide a pool of accelerators that can be used remotely to distribute the tasks [11].

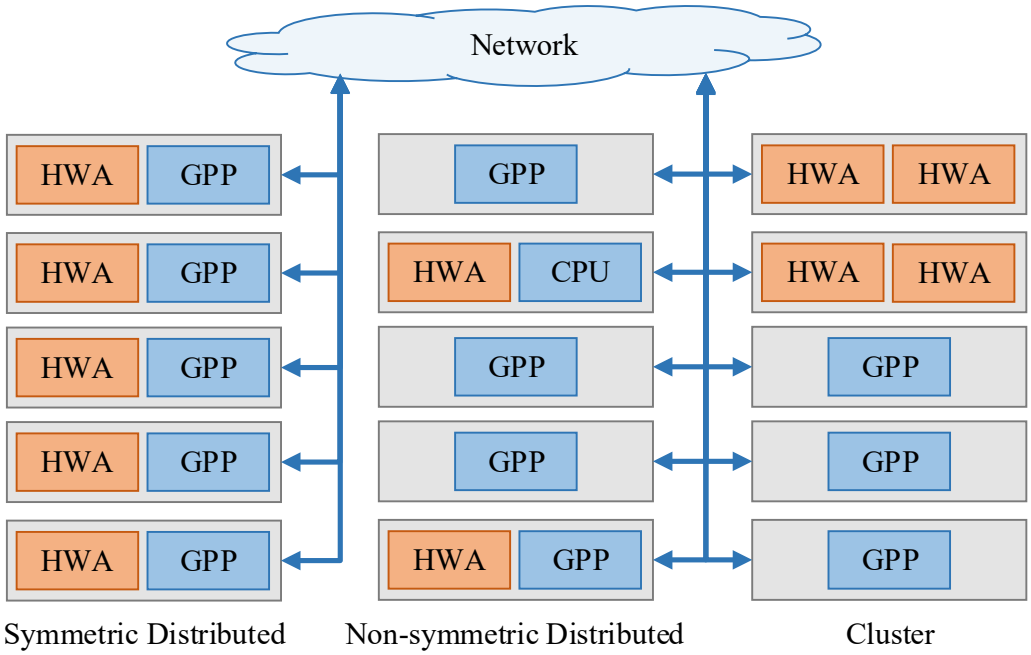

***Figure** 3.2 HWA deployment topologies alternatives.*

3.1.3 Deployment Topologies

In [11], it is mentioned that there are basically two ways to introduce HWA in a datacenter: cluster and symmetrically distributed. This work goes further and also presents a third option, a non-symmetric distributed. Figure 3.2 shows these three deployment topologies.

Cluster of HWA break datacenter homogeneity and limits scalability [11]. Nevertheless, it minimizes disruption to the infrastructure and optimize hardware cost.

Non-symmetric distributed topology maintains optimized costs by providing flexible accelerator and server ratio while allowing the provider to introduce HWA continuously, in smaller steps as needed, to an already existing infrastructure. Also, it applies better than the cluster option for some workloads, i.e. local acceleration. However, the homogeneity is also broken, and management become more complex. For example, mapping some application requires details whether HWA is available in a specific node or not.

Symmetric distributed topology on one hand provide efficient scalability, easies the management, maintain the highly desirable homogeneity [11] and is generic enough for a wide range of workloads. On the other hand, in most cases it requires the

highest hardware investment and may result in underutilization.

Furthermore, the topology choice may be influenced by the physical restrictions in the infrastructure, such as, power limits for the accelerators, physical space, resilience and temperatures [11]. For example, a provider who wishes to insert HWA without buying new servers may have problems to go with the distributed options due to restrictions in its current servers.

3.2 FPGAs in Cloud

The diversity of cloud workloads and its fast change rate is a challenge for HWA. It is highly desirable that any hardware inserted to the infrastructure can adapt to this during its lifetime, in other words, HWA needs programmability. This make FPGAs and GPUs preferable over ASICs whenever possible [11].

GPUs and FPGAs are both already deployed in cloud environments at reasonable scale [11]. GPUs architectures are efficient when processing images and video data, but since they are designed for its specific domain, they may not be so efficient or even decrease performance when processing different types of workload, such as signal processing and ciphering [43]. In fact, GPUs are not suited for tasks that do not contain a fair amount of well-structured data-level parallelism [29]. Furthermore, GPUs power and size requirements are bigger than FPGAs [11], which may make GPUs significantly less energy-efficient [37].

Providing FPGAs as resources in cloud infrastructure fills the gap among efficiency provided by ASICs and flexibility of GPPs [43]. As a matter of fact, AWS already provide FPGA as resources [4] while Azure currently offer it in preview mode for external users [50] and have worldwide deployment trials for its own purposes [64, 11]. An overview of developed HWA with FPGAs used for common cloud applications along with their main metrics is presented [35].

Additionally, resilience and reliability at hyper-scale is required when deploying FPGAs in cloud. Currently only Microsoft have such a high-volume system in production. They report only 0.03% of board failure in one month, all of them during the beginning of deployment, which is an acceptable level specially because the scale of datacenters provides sufficient redundancy [11]. The only restriction is that the management system should be able to detect and isolate problematic nodes.

The diverse range of workload types in NFV use cases [25] turn FPGA into even more promising candidate to be used as HWA in NFV systems. Yet, FPGAs do

come with their own differences and challenges when developing applications to run on it. One can classify these challenges in programming languages and design flow.

3.2.1 Programming Languages

Traditionally, applications for FPGAs are created using low level Hardware Description Language (HDL), such as VHDL or Verilog, this impose a challenge since it is a barrier for most software developers [69].

FPGA and system vendors have been putting high effort in the last years to reduce such a barrier by using well known high-level languages, such as C++, OpenCL and C, to abstract away hardware details. This abstraction is referred as High Level Synthesis (HLS) [35, 69]. Such abstraction usually results in reduced performance when compared with optimized HDL code, but for a wide range of designs, HLS tools can provide average performance around 90% when compared with optimized HDL [29]. Furthermore, usage of HLS facilitates code reuse and portability, even among different accelerators, i.e. the same OpenCL code can be deployed in FPGAs and GPUs [34].

Still, to obtain good results the developer should have understanding of hardware aspects [53], especially for I/O interfaces such as PCIe, Ethernet and off-chip DRAM [69]. This can be mitigated by frameworks that completely abstract the FPGA board and its I/O from the developer [69]. Some example of such frameworks from major vendors are Xilinx© SDAceel™ [81] and Intel® FPGA SDK For OpenCL™ [34].

Frameworks and pure HDL tools allow also another alternative to reduce the burden of developing HDL components. It is the use of Intellectual Property components designed by specialized developers, being third party entities or not. Making Intellectual Property components easily available and facilitating its integration by developers with no prior experience with HDL allow applications to leverage the better performance obtained by optimized HDL code seamlessly [35]. This approach works similarly as with software libraries, and as such, require usage of standard interfaces and is usually provided for tasks that are frequently required.

3.2.2 Design Flow

FPGAs also require a completely different design flow and set of tools than the ones software engineers are used to for compiling the applications. Instead of a set of instructions, the end result of a FPGA "compiler" is a binary file that mainly

describe the internal connectivity of the basic elements inside the device. This binary file, also called bitstream, is then loaded into the FPGA to implement the desired circuit and functionality [26].

The flow to obtain such bitstream is composed of a chain of automated tools that know the details of the available elements and their possible connections in the device target and translate the HDL descriptor accordingly [39].

One can simply describe the flow chain as follows: First, if the design is developed using HLS languages, it is translated to HDL. A synthesis step then take place where the HDL functionality is mapped to the basic elements available in the device. Then in the placement phase, the tool chooses among the elements which one to be used based on its location in the silicon floor. Later, in the routing phase the tool explores the best possible routes to connect the mapped elements. Finally, a time analysis take place, this phase checks each existing path among the elements and verify if they meet time requirements, in other words, it check if the desired clock frequency can be used, ensuring that the hardware functions as expected.

In practice, each step search among a wide range of possibilities and choose optimal configurations with the goal of reaching requested time constrains, while keeping the number of elements (area) and power consumption as low as possible. Due to the wide range of possibilities, designers can constrain the tools to look for solution that optimize time, performance or energy consumption. The whole flow is a heavy and complex process, as such, compiling a design can take from minutes to several hours for each of the steps depending on the complexity [39].

3.3 FPGA Virtualization

Besides challenges from the developer point of view, providing FPGAs as HWA resources in cloud is no simple task for the infrastructure provider either. There are at least four essential requirements that need to be addressed [14]:

1. Sharable: As with all resources in cloud, FPGAs should be sharable among multiple tenants and applications in order to maximize resource utilization.

2. Abstracted: FPGAs must be exposed to tenants as a pool of resources that can easily be requested, allocated and deallocated. Programmability of the FGPAs must be exposed to tenants, similar as with GPPs and GPUs, in other words, FPGAs should not be considered as an ASIC, but a programmable one.

3. Secure: FPGAs should provide proper isolation when multiple tenants and applications are sharing the same resource and one cannot impact the other on purpose or not.

4. Scalable: To not limit applications to resources of a single FPGA at most, tenants should be able to easily scale applications among multiple FPGAs and create their own topology.

Furthermore, to increase productivity, it is highly desirable to expose APIs for controlling the FPGA as well as interfacing application logic and board-level functions while supporting resilience and debugging [64].

In the following it is discussed how those essential requirements are or can be provided.

3.3.1 Sharing

The fact that GPPs applications are a set of instructions allow the same core to execute several applications from multiple users, in practice, the virtualization takes place by sharing the same resource by time slicing and scheduling the instructions from each application accordingly. Even though the synthesis time required by FPGA's design can be mitigated by creating the bitstream beforehand, the process to program an FPGA is still too slow to allow multiple applications and users to share the same area of the device in a time slice manner, limiting its scalability [64].

To allow more applications and users one could simply deploy multiple FPGAs for that, but it easily become costly, consuming more power and wasting resources when not needed. Partial Reconfiguration (PR) is a key enabler for sharing FPGAs and can be used to allow multiple applications to run on the same FPGA by dividing the device's area, instead of time slicing [64].

PR is a technique where only part of the FPGA is reconfigured during run-time instead of the whole device. Beforehand a specific area is reserved to be static, this region is never reconfigured during run-time and usually contain all the management circuitry and communication interfaces, such as PCIe and Ethernet. The remaining area of the device can be divided into multiple Partial Reconfigurable Regions (PRR) that can be reconfigured during run-time without interfering in others PRR [26].

3.3.2 Abstracting

Infrastructure level of abstraction for FPGAs can be achieved by exposing them as a resource in a cloud management system such as OpenStack [58]. Following the cloud nomenclature, in its core this functionality is provided by a hypervisor that allows users to implement and execute their guest Operating System. In the FPGA context, the hypervisor is a set of hardware (shell) and software (manager) components that allows user to program and run their own hardware design [39, 8].

In the hardware side, abstraction is achieved by utilizing a static area that is programmed during start-up and not accessible by the user, the shell. The shell abstracts away all the board connectivity, such as PCIe and Ethernet, while providing common standard interfaces, such as AMBA [3, 2], to the user's hardware, in this work called Accelerator Hardware Unit (AHU). This release the heavy burden of system integration from the user and makes the design reusable among different devices and boards as long as the shell remains compatible, speeding up development and increasing portability [64]. Furthermore, the shell communicates with the manager, providing status and debugging information while receiving global or user-specific configurations commands.

Multiple flavors of HWA can be exposed to users via the manager. It could be for example the whole FPGA or part of it, in both cases PR can be leverage to provide abstraction [39], allowing fine grain selection of resources. Once a resource is selected, the desired acceleration functionality is programed into the FPGA with a compatible bitstream image previously uploaded to the manager, i.e. using *glance* service in OpenStack [8].

3.3.3 Securing

From the security point of view, the infrastructure must guarantee that users cannot propositionally crash other's applications or the system. Also, it must provide complete data isolation [14].

The former requirement can be provided by a robust hypervisor that ensure security access in the manager level [39] and that utilize PR. With Partial Reconfiguration users can only affect their own region in the FPGA and the shell must be designed so that it is able to detect faulty AHU and prevent it to affect others [14].

Data isolation for tightly coupled connections can be solved using DMA. The hypervisor supervises all DMA operations and allow only legal and correct ones, thus

users can only access the memory regions they are allowed to [14]. For network connected devices, common network security approaches can be used while the hypervisor must ensure that the users can only receive and generate packets from and with addresses which they are allowed.

3.3.4 Scaling

FPGA attached through tightly coupled connections have scalability limited to a single server [10]. Hence, to enable truly scalable FPGAs and allow global acceleration [11], at least network connectivity is needed. The hypervisor should allow the user to request as many FPGAs as the design requires, which could be from couple to thousands of devices [10].

Furthermore, it is needed to allow the user to easily create its own topologies for network connection when requesting multiple FPGAs. Integrating SDN capabilities into the manager and shell allow the user to create extremely flexible and scalable architectures that enable powerful solutions.

Besides the four essential requirements above, another is compatibility. It addresses the issue that design flows, tools and libraries for developing FPGAs applications are still very dependent on the vendors and devices. A framework that provide common ecosystem and seamlessly migration among devices versions and vendors available should be exposed. Contrary to [14], here scalability requirement was promoted as essential in place of compatibility. This is done so because this work concentrates on the IaaS layer of the cloud environment. One could argue, but here it is assumed that compatibility could be better addressed in a higher level of abstraction, similar with the PaaS in common cloud, while the four essential requirements must be addressed already in the infrastructure layer.

4. RELATED WORK

In this chapter closely related works are reviewed, this means that the list is reduced to efforts that focus on FPGAs and in at least one of the subjects: HWA virtualization or abstraction; Scaling HWA; HWA for NFV or SDN;

They are divided in four sections: Hardware acceleration only contains efforts that do not provide scalability; Partially scalable hardware acceleration reviews works that have some level of scalability; Fully scalable hardware acceleration provides review of efforts that allow deployment of scalable hardware acceleration in the range of thousands or more; Finally, Hardware acceleration in NFV provides review of related works that focus on it.

4.1 Hardware Acceleration Only

Even though this work focusses on enabling FPGA in a cloud environment for distributed acceleration of NFV systems, FPGAs are currently mainly used for acceleration of specific applications. It is worth to mention some available solutions that focus on this area.

IBM's Coherent Accelerator Processor Interface (CAPI) [75] enables acceleration of Compute-Intensive applications on POWER8 processors by simplifying the complexity of programming the I/O system. The system connects GPP and FPGA through PCIe and provide coherent memory between them, allowing the application to share its memory space with the hardware accelerator and transfer data through simple memory access commands.

Intel's Hardware Accelerator Research Program (HARP) [33] follow similar approach as CAPI, abstracting away the system integration and providing coherent memory. HARP differ from CAPI because it can also connect Xeon® processor with an in-package FPGA. In this configuration, when GPP and FPGA are in the same package, they can be connected through Intel® QuickPath Interconnect (QPI), which improves system bandwidth when compared with PCIe.

In the same line as CAPI and HARP, CCIX Consortium [12] promotes a standard specification to enable coherent interconnect, but instead of FPGA only, the Consortium aims to include more accelerator devices, such as GPUs and ASICs.

All frameworks presented in this section enable easier integration between GPP and FPGA and allows efficient communication. Neither of the solutions above offer scalability, in other words, if one or a limited number of FPGAs is not enough, it cannot be used, or workarounds must be taken, which decreases performance.

4.2 Partially Scalable Hardware Acceleration

In this section it is described works that provide some level of scalability but have some limitation and are not considered fully scalable.

In [26], a framework was developed to integrate FPGAs in cloud using a card connected via PCIe. The system supports multiple accelerators using partial reconfiguration. A shell hardware in the FPGA provides control and standard interfaces to the accelerators and maintain fair bandwidth among them. Also, the software stack was developed to facilitate integration. The software contains FPGA driver for PCIe operation, an API for transfer data between host and FPGA, an hypervisor for resource and security management and a middleware that uses Linux sockets to enable clients to access the cloud services. To ensure that no malicious bitstreams are used, the system authenticate them using bitstream watermarking. The usage of virtual machines in this work is not mentioned and users can only access accelerator through Linux sockets.

Similarly to [26], [14] presents a framework to integrate FPGAs but leverages OpenStack and integrates the system to virtual machines with Linux KVM. It uses a Xilinx's board connected via PCIe and leverages PR for sharing the same device as well. The framework is divided in four layers: hardware; hypervisor; library and application. Hardware layer implement the shell functionality. The hypervisor layer provides the drivers to access the FPGA and the controlling and monitoring system. The library layer exposes an API for the applications and maintain a library of bitstream files. The application layer consists of a modified version of OpenStack that enables users to upload bitstreams and allocate accelerators. DMA is used to transfer data from and back to acceleration, which causes overhead when using VMs since translation between guest and host memory space is needed in this framework. The work compares two techniques to solve this, one is copying the data from guest to host and other is translating addresses. In any case, transferring data among multiple FPGAs add significant overhead.

vFPGAmanager in [63] is a framework that virtualize the communication between FPGA and VMs or containers using DMA with PCIe that leverages SR-IOV for fast data transfer. This framework exposes an API that could be used by some VIM orchestrator. Scalability is limited by the tight couple between FPGA and GPP but the use of SR-IOV should provide better performance than the framework presented by [14].

Microsoft's Catapult project [64] is probably the first medium-scale deployment of FPGAs in a production level cloud. The work describes multiple techniques used to enable such accomplishment and the improvement of 95% in the Bing search algorithm. Contrary with the previously mentioned works, the Catapult's architecture does not contain only PCIe or network connectivity, instead, it leverages PCIe and a secondary dedicated network among up to 48 FPGAs. Also, it uses an in house developed board that is compliant with their requirements. As with others, the FPGA contains a shell that provides easy integration of the user's hardware. Although no details about the proprietary software is provided, they describe additional services that were added to it in order to ensure correct operation, failure detection and recovery as well as debugging.

Amazon offers up to 8 FPGAs as resources in its AWS cloud [4]. The devices are connected through a PCIe fabric and share a memory space. The largest flavor is connected with a bidirectional ring for low-latency and high bandwidth communication. Even though in the cloud, it is still limited to 8 FPGAs.

4.3 Fully Scalable Hardware Acceleration

In [8], the authors show a framework that uses Ethernet as data transfer instead of PCIe and leverage a modified OpenStack to virtualize FPGAs in the cloud system. The FPGAs are instantiated and tear down by the users as it is done for VMs. Each partial reconfigured area of the FPGAs is presented as it would be for the whole FPGA and the shell provides the interface abstraction. A special user has access to scripts that allow them to compile their own accelerator and upload them for use while basic users can only use precompiled hardware. The scripts for compilation allows users to use HLS for fast development. The shell uses MAC (Media Access Control) address to route packets to the correct accelerator and enforce the right MAC addresses in the output, avoiding sniffing and spoofing of data. The management is done through UART and a soft processor inside the FPGA. Even though network connectivity provide scalability, the use of plain Ethernet limits flexibility of the network.

In the same line as [8], [73] also proposes to connect FPGAs thorough the network but argument for more flexibility by adding a Network and Transport Stack that provides hardware implementation of L3-L4 protocols. This wider the scope to applications that use, for example, TCP, UDP or RoCE protocols. Furthermore, this stack applies SDN principles, thus only data plane is implemented in the accelerator while control plane runs on decoupled software. The cloud integration is also achieved by leveraging a modified OpenStack where FPGA is considered a standalone resource and is not attached to some VM. The framework allows the user to create large and distributed applications with its own topology and provide an API to interface it.

Another framework that provides network connectivity is present in [39], with the main difference this also includes PCIe connection. The authors described their own framework that, contrary to others not proprietary ones, is not based on OpenStack. A shell for abstraction is showed, PR is leveraged, and the design flow allows users to use HLS to improve development speed. Furthermore, interesting services models are proposed: Reconfigurable Silicon as a Service (RSaaS) provide full access to an entire FPGA board along with the framework's development flow, drivers and VM. The concept similar to IaaS in cloud. Reconfigurable Accelerators as a Service (RAaaS) provide access to PR regions only and user develop the accelerator and use the provided API to interface with the FPGA. This model could be mapped to PaaS. A third model that could be comparable with SaaS is Background Acceleration as a Service (BAaaS), where the FPGA is not directly available, instead, users benefit from applications that are accelerated by the FPGA in the background.

At last, even though very powerful, the first version of Catapult's architecture still had some limitations, for example, scalability of the system was limited to 48 nodes and a secondary network was expensive and complex. To overcome those limitations Catapult evolved to Configurable Cloud architecture presented in [11], which is currently deployed world-wide in all new Bing and Microsoft Azure cloud servers. In this version, all traffic goes through the FPGA that is connected between the server's NIC and Ethernet switches while still providing PCIe connectivity too. The authors argument that this provide enormous flexibility, especially because it allows local, network and global accelerations, widening the range of applications. The world-wide level of deployment of this system is a strong argument in favor of the potential FPGAs have in cloud. It is further confirmed with some recent examples where the infrastructure is used, such as project Brainwave [17] and Azure Accelerated Networking [27].

4.4 Hardware Acceleration in NFV

Neither of the previous mentioned works take into account NFV. This section reviews some works that cover the deployment of HWA in NFV.

The work presented in [36] proposes the use of FPGA for NFV and SDN. The authors argument that FPGA can provide the flexibility of virtualization and high performance of specialized hardware for NFV systems but no detailed solution is showed.

In line with this idea, in [28] FPGA accelerators are integrated using OpenStack. The work shows how FPGA resources could be provided to VNFs for acceleration in a very similar way as [39] and [26], leveraging PR and using PCIe connectivity. Results shows an order of magnitude improvement for functions such as NAT, DPI and Dedup. Again, scalability is limited due to the tightly coupling between FPGA and GPP and solution for this is not described.

Deployment of HWA in a NFV system requires strong hardware and software support. In the software side, full NFV's MANO implementation is required to achieve such goal and it is out of the scope of this work, as such, attempts to provide complete MANO stack are not reviewed here. Still, for sake of completeness, one can refer to the works in [52, 87] for a comprehensive review on efforts in this area.

Still, it is not possible to enable FPGA as HWA in NFV systems without software support. Even though the MANO architecture may be a bit fuzzy depending on the implementation, in this work the software is limited to the VIM component in MANO's architecture. In other words, the focus is in managing the virtualization of the FPGA similarly as the ones previously reviewed works presented in [14, 39, 8, 73].

Another example of a software architecture of a framework for management and control of a infrastructure that enables creation of heterogeneous NFV service chain is presented in [45]. Their concept is not limited to FPGA only, but allows management of GPUs and IoT sensors, for example. The framework is deployed in the SAVI testbed, a multi-tier and SDN-enabled cloud which contain heterogeneous compute, wireless, and IoT resources. It allows flexible chaining of heterogeneous VNFs leveraging OpenStack for compute virtualization and OpenFLow for network. In this work no details about the hardware is shown.

From all the works reviewed here, it is clear that Microsoft's Configurable Cloud [11] is the most advanced one, nevertheless it is worldwide deployed. Still, being a proprietary technology, details about the whole framework is not disclosed. Be-

sides that, neither of the other efforts provide the same flexibility and scalability. This work describe an early stage of a framework which goal is to be as flexible as Configurable Cloud while aiming acceleration of NFV's services chains.

5. METHODOLOGY

This chapter presents hardware and software tools and libraries leveraged in the development and measurements of the architecture proposed in this work.

5.1 Hardware and Laboratory Setup

A high-level view of the main components and hardware used in this thesis is shown in Figure 5.1.

The laboratory setup contains three servers with dual socket Intel® Xeon® E5-2680 v4 @ 2.40GHz CPUs of 64-bit and x86 architecture, the CPUs have 14 physical cores each and hyperthreading enabled, providing 56 treads in total and 128 GB of DDR4 memory. The Network Interface Controllers (NICs) used for network communication are Intel® 82599ES 10 Gigabit Ethernet Controller. In the network path a switch model QFX5100-48S from Juniper® Networks was applied to connect NICs and FPGAs.

Three Xilinx® Kintex® UltraScale™ FPGA KCU1500 Acceleration Development Kit boards were used. Each board contains a XCKU115 FPGA device, 16GB of DDR4 memory, two x8 interfaces bifurcated to x16 edge connector PCIe Gen3 and two Ethernet QSFP cages. Details about the board can be found in [85].

FPGA development, simulation, verification and debugging were done with Vivado Design Suite [84]. Multiple components, known also as Intellectual Property components, provided in Vivado were leveraged for the design, such as 10G/25G Ethernet Subsystem, DMA/Bridge Subsystem for PCIe and Integrated Logic Analyzer (ILA) for debugging. Own logic with VHDL was used to develop the CRUN's shell top level, ROUT RX component and a glue logic for interfacing with de metadata interface of P4 RX and TX components.

The integration of SDN into FPGA allow users to easily create flexible and scalable distributed architectures. This thesis leverages Xilinx® SDNet™ packet processor [82]. This tool generates components that can be integrated in the FPGA for a wide

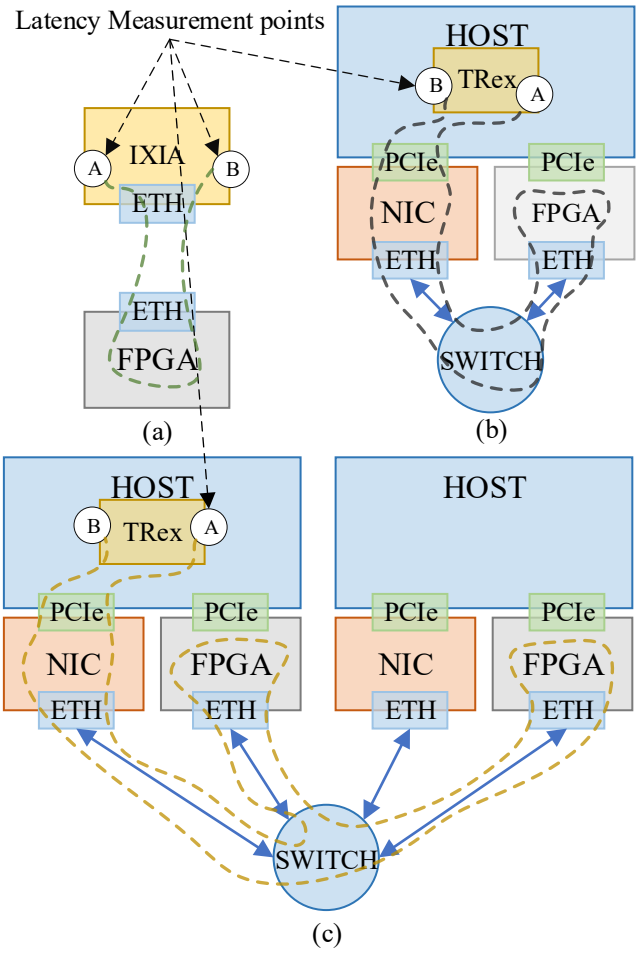

Figure 5.1 *Main components, hardware and test cases: (a) shows the setup for hardware performance measurements; (b) presents the setup for software measurement of a single accelerator; (c) provides the setup to obtain distributed accelerator metrics.*

range of packet processing functions. Designs can be described using P4 language [72], which is a standard to describe SDN's programmable data planes [82].

5.2 Software and Libraries

The host runs CentOS Linux release 7.5.1804 Operating System [70] and virtualization of x86 hardware is achieved with KVM hypervisor [42]. To control virtualization and interact with KVM hypervisor libvirt is leveraged, more specifically its libvirt-python library wrapper [44]. Since libvirt is heavily dependent on XML files for describing VMs and configuration, lxml library [47] is also employed.

For DMA and control access to the FPGA through PCIe a driver provided by Xilinx® is used [86]. For high performance when accessing NIC's PCIe interface from a VM, SR-IOV [49] is employed.

5.3 Test Cases

To proof the architecture functionality and obtain performance values, the test cases in Figure 5.1 were realized together with a third party trial.

For network measurement of the hardware only, IXIA board NOVUS-R100GE8Q28 traffic generator was employed. IXIA was directly connected to the FPGA Ethernet port, as shown in Figure 5.1 (a), for precise measurement of shell throughput and latency without switches or software bottlenecks in the path, providing precise hardware measurements at nanoseconds scale.

To validate the hardware and obtain performance measurement from the application point of view, TRex version 2.45 [18] is used. TRex is an open source traffic generation tool that runs on Linux and employ DPDK [71] to obtain high performance. The throughput and latency measurements obtained from TRex include the Linux software stack and provide a more realistic view of the performance achieved by an application running on the host, in other words, it includes the software limitations and not the hardware only as with IXIA, for the same reason it is not as precise as IXIA values.

Figure 5.1 (b) shows the test case used to obtain performance values with TRex for the case where only one accelerator is used, while Figure 5.1 (c) provide the performance values for a distributed acceleration case. The metrics obtained from (b) and (c) include also the switch, which affects mostly the latency of the system.

6. CRUN ARCHITECTURE

This chapter presents the proposed architecture, which is divided in hardware and software sections. The architecture is named CRUN, as a pun from its main motivation, Cloud RAN, where CRUN means "a cloud that does not walk, it RUNs".

It is important to note that the architecture shown here is a proposal providing the desired functionalities and not the final solution. Currently it is being developed and not yet completed. The details of what has been actually implemented is presented in Chapter 7.

6.1 CRUN FPGA's Hardware

The hardware is described in a top-down fashion, meaning that first server and datacenter are described along with the view of the infrastructure, then the FPGA's shell is presented followed by the Accelerator Hardware Unit (AHU) details.

6.1.1 Server and Datacenter

The high-level view of the server with acceleration can be seen in Figure 6.1. This figure shows only the relevant components for this thesis and does not try to represent all the existing ones.

As shown in Figure 6.1, FPGA has its own on-board memory and both NIC and FPGA are connected through PCIe. Also, FPGA provides two PCIe drivers, one for accessing its control logic and another for data transfer through DMA. The host contains an hypervisor that provides VMs and expose the PCIe's Physical Functions (PFs) of both FPGA and NIC as Virtual Functions (VFs) to the user's VMs using SR-IOV. The BRO-CLIENT is part of the management software application and is responsible of managing the host OS, the hypervisor, NIC and FPGA.

The main physical difference required for the server is the addition of a FPGA daughtercard connected through PCIe. This is an approach similar to the one used in [11], but here the NIC is connected directly to the network instead of the FPGA.

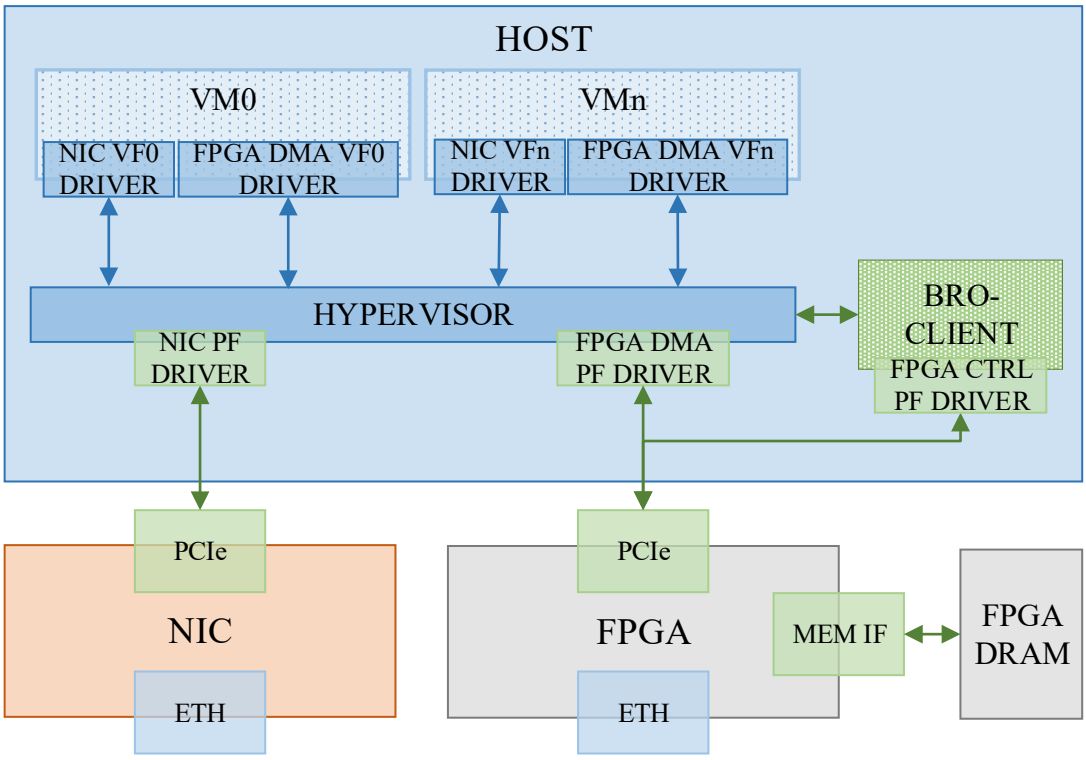

Figure 6.1 *Server architecture.*

The Figure 6.2 shows a high-level view of a datacenter setup composed of three servers as an example. The acceleration framework in this thesis leverages distributed connectivity instead of clustering since it adapts to more workloads and acceleration scenarios then the cluster option.

Here, it is assumed that the datacenter is a symmetrically distributed system in terms of deployment topology, i.e. each server in the datacenter has its own accelerator board as discussed in Subsection 3.1.3. The option for non-symmetric distribution depends on support for it in the management software, since it inserts more constrains when mapping the VMs and accelerators.

Referring again to Figure 6.2, it also contains the data path for the three acceleration scenarios that the system aims to support, as is the case presented in [11]. In local acceleration, path 1 from and back to VM in HOST 1 (green dashed line), the data is transfered through PCIe via DMA straight to accelerator and back. In network acceleration, path 2 from VM in HOST 2 to INTERNET (black dashed line), the packet is in-line processed after leaving the VM and before going out of the datacenter. In distributed acceleration, path 3 from and back to VM in HOST 3 (yellow dashed line), data is transfered to a chain of two FPGAs through the NIC.

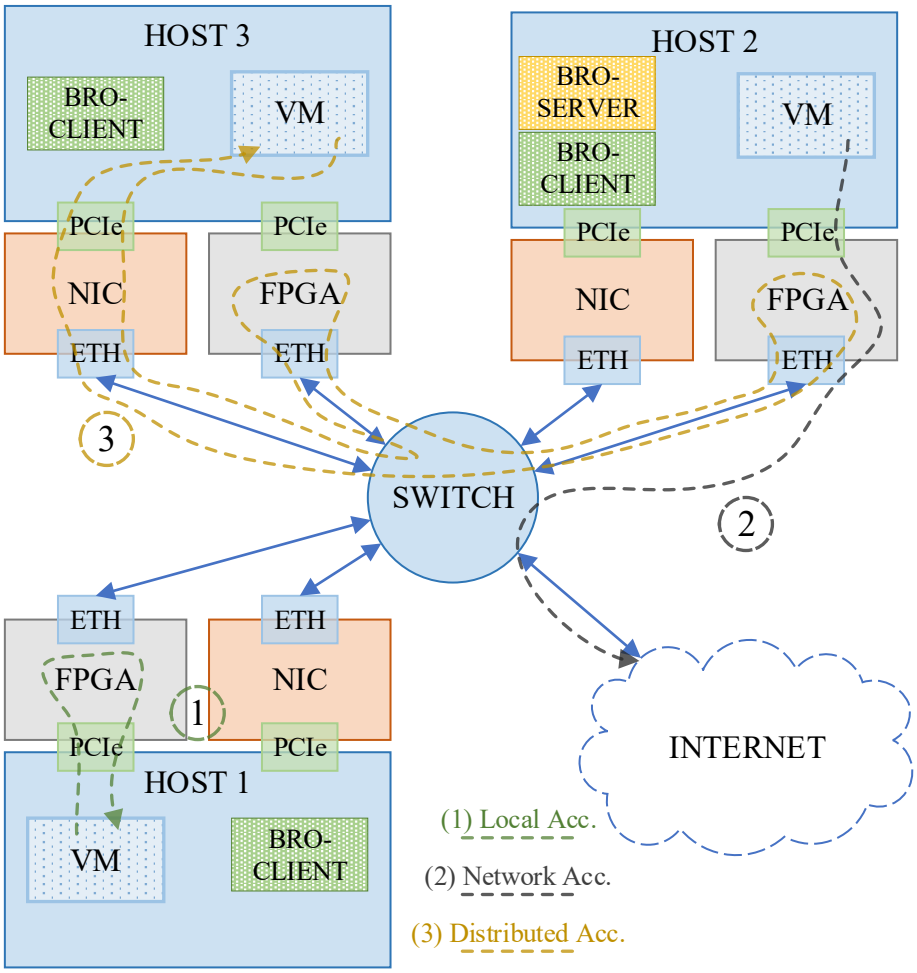

Figure 6.2 *Datacenter architecture with hardware acceleration model.*

Note that the flexibility of the system allows several combinations of paths. For example, depending on the requirements, local acceleration could use the NIC path to access a remote and free FPGA, leaving the local one to others VMs with more critical tasks. In network acceleration more than one FPGA could be on the processing path, becoming a mix of distributed and network acceleration. In distributed acceleration scenario, the first FPGA could receive packet from DMA instead of NIC.

6.1.2 CRUN Shell

The Figure 6.3 shows the CRUN shell architecture and its main components in a high-level view. In the center of referred figure one can see multiple AHUs, each one

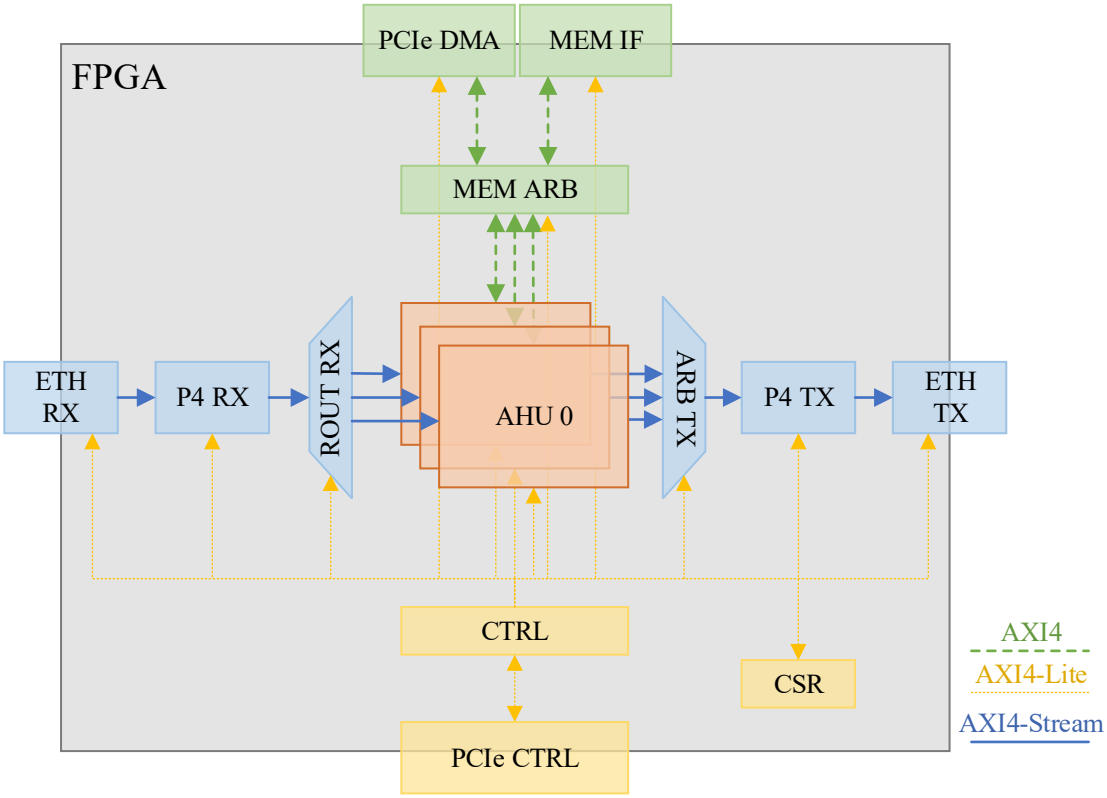

Figure 6.3 *FPGA architecture.*

being a Partially Reconfigurable Region (PRR) that can be programmed independently of the others. This is widely used practice for sharing the FPGA, as can be seen in most of the works presented in Chapter 4. All remaining components except the AHU are static and belong to the shell logic.

Depending on the FPGA device and needs of the users, multiple shell versions can be deployed, each with different number of AHUs and sizes. It allows a fine tune of available resources. When changing the shell version during run-time, the management system should first take care that no AHU is currently being used in the target FPGA, since updating the shell requires full reprogramming of the device.

Each color (or line) in the Figure 6.3 represents one clock domain. Clock domain crossings from AHUs to the interfaces are provided by the shell. Furthermore, AHUs are served with predefined clock frequency inputs and the user can choose which best suits their hardware.

Stream Path

The blue components (ETH RX, P4 RX, ROUT RX, ARB TX, P4 TX and ETH TX) compose the Ethernet stream path. All of them function on line rate so the system supports full throughput and never stalls. This path provides functionality similar to the ones presented in [8, 73] and enable full scalability of the system.

ETH RX main responsibility is checking and translation of incoming physical Ethernet packets from the I/O transceivers into the equivalent AXI4-Stream ones. An Ethernet frame packet is composed of multiple fields and the standard can be checked in [32]. The ETH RX component removes the preamble, Start of Frame Delimiter and Frame Check Sequence fields in the Ethernet frame while checking for possible errors. Those fields do not carry any user information. The component then provides the relevant fields (Destination and Source MAC addresses, Ethertype and Payload) as output in the form of AXI4-Stream packet.

P4 RX is a SDN component created using SDNet^{TM} and was developed together with a third party. A received packet is first parsed by P4 RX in order to identify the flow it belongs to based on 5-tuple definition, which is composed by five fields of an IP (Internet Protocol) packet: protocol, source and destination IP addresses, and source and destination ports. One AHU can receive packets from more than one flow as well as send packets with different flows. This allows multiple sources and destinations to use the same AHU. Thus, each packet is tagged with the corresponding flow ID (Identification) it belongs to, so the AHU can identify the flow and also use it when sending packets out. The flow ID and tables are updated during run-time. Flow ID is defined by user per AHU, it allows the user to identify clearly where the packet comes from and/or is destined to.

Once the packet is parsed, P4 RX does checksum verification and matches against a lookup table. If a match is not found the default action is taken, which is to drop the packet. Otherwise, the component strip out all headers so only payload is forwarded. P4 RX then indicates to ROUT RX the flow ID and which AHU the packet is destined to.

ROUT RX receives the packet payload from P4 RX and routes it to the corresponding AHU along with the flow ID. The component contains small buffers, so it can store a few packets and check whether or not the corresponding AHU is ready to receive it. The stream path must not stall, since it would mean that other AHUs may be affected. Thus, if the AHU is not ready for any reason the packet is dropped and a corresponding user-readable counter is increased. It is user's responsibility

to make sure the AHU supports the desired throughput and the counter provides status of how many packets were dropped, if any.

The ARB TX component is responsible for receiving the packets generated by the AHUs and arbitrating among them which one should be served. Different priority schemes can be used to provide more throughput to some of the AHUs, allowing finer grain selection for the users.

P4 TX is also created with a third party with SDNet^{TM}. This component receives packets with payload only from the AHU with the information of which flow it belongs to and uses it to build the corresponding IP headers. The constant header fields belonging to each flow, such as addresses and ports, are updated at run-time through the control port, while dynamic ones, such as checksums and lengths are determined on the fly.

Similarly to ETH RX, ETH TX converts AXI4-Stream Ethernet packets to the physical correspondent ones and sends back to datacenter's network, where they will be routed to the desired destination.

DMA Path

The DMA path is composed by the green components (MEM ARB, PCIe DMA and MEM IF) in Figure 6.3. AHUs can access the FPGA's on-board memory through MEM IF interface or do Direct Memory Access with host via PCIe DMA interface. This path aims to provide local acceleration to the server in a similar manner as most of the works presented in Chapter 4. Combining the DMA path to the stream path allows in-line accelerator as well.

PCIe DMA is the interface component that translates AXI4 protocol inside the shell to the physical I/O signals in the PCIe interface in order to transfer the required data. MEM IF perform similar tasks, but it interfaces the on-board memory transceivers, providing large amounts of DDR memory to the AHUs.

MEM ARB is the main component in the DMA path and has similar concept as the system presented in [14]. It assures secure data transfers since it knows which memory region each AHU can access either on DMA or on-board memory. Thus, only allowed access transactions are executed. The management system is responsible of controlling the regions each AHU can access by defining its context.

Contexts contain memory regions and sizes information allocated to the AHU and

they are dynamically updated through the control channel. Each AHU contains also its own bank of control registers where users are able to issue read and write requests.

Through the bank of control registers, accessible via the control channel, acceleration tasks may be enqueued to the AHU. In fact, control registers are totally user defined, it allows user to describe their own type of acceleration commands, management and monitoring. For example, a region of the control register could be used as a queue of jobs.

MEM ARB contains a DMA engine that takes care of DMA write and read requests, keeps track of the responses and reorders whenever needed. For high performance when communicating with VM, SR-IOV is leveraged as in [63]. MEM ARB is also able to prioritize AHUs by providing different bandwidths according to the management software control. An internal scheduler manages each AHU context for transferring data with host.

Control System

The control components are represented in yellow (PCIe CTRL, CTRL and CSR) in Figure 6.3. The control system is accessed through PCIe CTRL interface and a specific PCIe driver that do simple memory map access using AXI-Lite to the shell and AHUs components, where each component has its own address space. Users can only access the address space of its own AHUs while management can access all shell components and the mandatory region of the AHUs.

The CTRL block is responsible of accessing the desired component base on the address. It is through this component that P4 RX/TX tables are updated, number of dropped packets per AHU in ROUT RX is requested, ETH TX/RX status is checked, etc.

The control system has also a FPGA global Control and Status Register (CSR) component that enable general control of the shell and status. CSR contains for example resource ID, shell version, total number of AHUs and how many in use, etc.

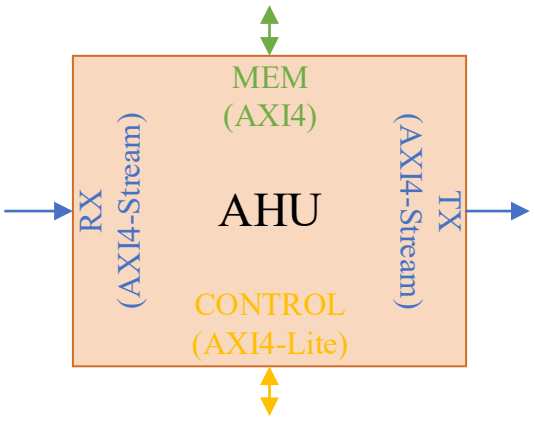

Figure 6.4 *AHU's interfaces.*

6.1.3 Accelerator Hardware Unit

The Accelerator Hardware Unit (AHU) is the hardware component that actually process the data. The user that wishes to accelerate some application has to develop the AHU that executes the desired functionality using HLS or HDL, for example. The provided interfaces of the AHU are shown in Figure 6.4.

The interfaces used are AMBA AXI4 compliant. The protocol is an industry-standard widely known and deployed, with a robust third-party tools environment which many component vendors support. AMBA AXI4 also provide high flexibility, for example, AXI4 is ideal for high-performance memory-mapped operations, AXI4-Stream for high-speed streaming data and AXI4-Lite for lightweight memory-mapped communication interface for simple control and status operations [78]. The complete specification can be found in [3, 2].

Referring to Figure 6.4, the AXI4-Stream RX interface is where incoming packets of the physical Ethernet which are destined to the AHU are received, while the output packets are sent to the corresponding TX.

The AXI4 MEM interface is used to do memory access either direct with host through DMA or the on-board FPGA memory, the exact location is based on address space.

The RX/TX interfaces are ideal for network and distributed acceleration, while MEM interface can be used for local acceleration. It is also possible to use a combination of them. In in-line acceleration, for example, incoming packets from RX can be accelerated and delivered to host via MEM. Similarly, data from host can be

accelerated before going out to the TX interface.

The AXI4-Lite CONTROL interface access register banks in the AHUs for control and monitoring. CONTROL and its respective register bank is not meant for transferring data but for controlling and monitoring the AHU. For example, user can use this channel to indicates the AHU when to start some process and can poll some register to check when it is done. CONTROL can and should also be used to monitor application specific status, such as errors and internal counters.

6.2 BRO Management Software

The difficulties of using the system and managing it must be abstracted away from the user and allow the provider to easily control it, which requires a strong software support. The aim of this section is to explain the functionality of the proposed infrastructure management software, called BRO. The name comes from the fact that providing the MANO's functionality is the ultimate goal of the software and that the word *mano* is a slang for brother in Brazilian's Portuguese as *bro* is in English.

Even though full MANO functionality would be the ultimate target, it is out of the scope of the proposed solution, which covers only the infrastructure level of the system. Thus, BRO could be better mapped to VIM component in MANO's architecture shown in Section 2.2.

BRO is developed mainly using Python language [65] and is divided in two main components, BRO-SERVER and BRO-CLIENT. Their deployment in a datacenter level can be seen in Figure 6.2. The following sections explain them.

6.2.1 BRO-SERVER

BRO-SERVER is the centralized main application that oversees the whole datacenter's infrastructure. Figure 6.5 provides BRO-SERVER's architecture and main components.

Users and administrators can interact with BRO-SERVER through a Command Line Interface (CLI), that can be later expanded to be remotely accessible and provide a graphical user interface. Administrators can have deep access to infrastructure data and may, for example, add, modify or remove resources, change user's access rights and monitor the whole infrastructure. Users on the other hand can only affect its

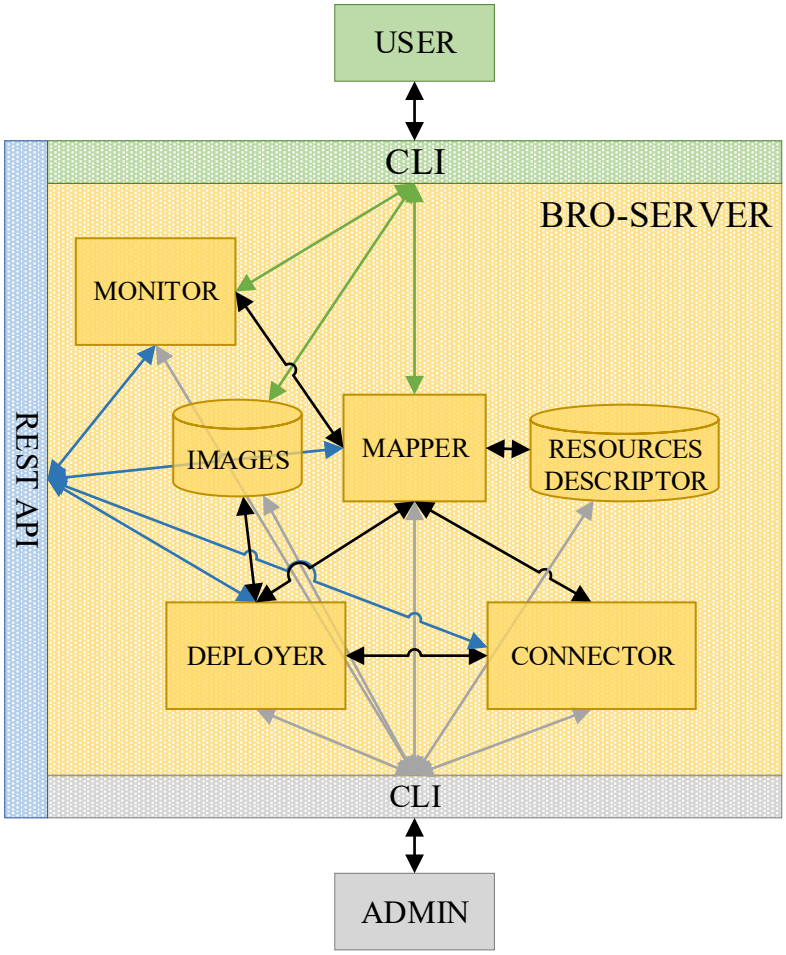

Figure 6.5 *BRO-SERVER architecture*

own services and have reduced view on the infrastructure. Thus, user have limited access to BRO functionality. Example of allowed common user's commands are upload of images, request and release of services and owned resource monitoring.

From Figure 6.5 one can see that the BRO-SERVER architecture contains two databases and is divided in four main components.

The RESOURCE DESCRIPTOR database maintains information about resources available and their details, such as servers and HWAs. IMAGES database is where shell images and user's ones like VMs and AHU's bitstreams are stored. The databases are developed using SQLAlchemy [68].

Each of the main components in the BRO-SERVER, being them the MAPPER, DEPLOYER, CONNECTOR and MONITOR, is a single process running on the

host. They communicate with each other and their clients through REST API developed using and following the microservice architecture principles.

The MAPPER is the component that directly interacts with users and administrators as well as coordinates the other BRO's components. MAPPER maps requests based on the information contained in the RESOURCE DESCRIPTOR database, return information to users about services status and issue jobs to other components in order to satisfy users and administrators requests accordingly to the infrastructure constraints.

DEPLOYER receives resource requests from MAPPER, such as boot some VM in a specific server or program some bitstream in a specific FPGA. DEPLOYER then is responsible for keeping track and pass the required information to the respective DEPLOYER-CLIENT running in the servers targeted, in order to deploy the VMs and/or AHUs.

Similarly, CONNECTOR receives requests from the MAPPER with information in how the resources required should be connected. CONNECTOR then interacts with the targeted CONNECTOR-CLIENTs in order to achieve the configuration requested.

When either DEPLOYER or CONNECTOR have completed or failed their task, MAPPER is notified. Once both are done and a service is started successfully, MAPPER then can provide the required information, so the user may start using it.

MONITOR functions more independently of other components. Its main responsibility is to inquiry from MAPPER and MONITOR-CLIENTs status and debugging information about the services to provide it to the users and administrators.

6.2.2 BRO-CLIENT

In each server of the datacenter the BRO-CLIENT application must be running in order to manage the local resources. BRO-CLIENT is responsible for the management and configuration of the host's OS, hypervisor, NIC and the FPGA. Figure 6.6 shows BRO-CLIENT's main components.

BRO-CLIENT is composed of client components that receives jobs from their respective main server versions. Responsibilities that the BRO-CLIENT covers are divided in the following by component.

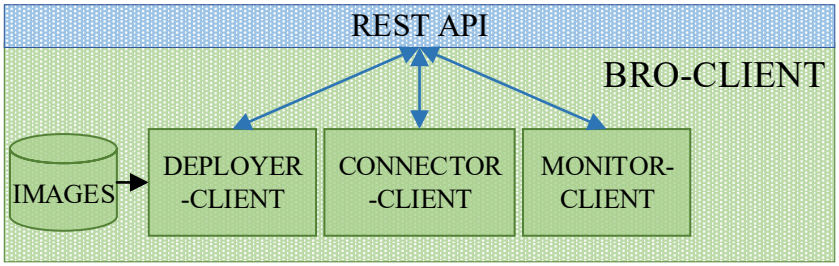

Figure 6.6 *BRO-CLIENT architecture*

The DEPLOYER-CLIENT is mainly responsible to deploy VMs and AHUs in the local server. Some examples of its tasks are: Interact with hypervisor in order to boot up and down VMs; Control VM's resources such as number of cores and amount of memory; Configure OS's Ethernet and PCIe interfaces and drivers; Maps FPGA's SR-IOV physical and virtual functions; Program AHUs in the FPGA.

CONNECTOR-CLIENT manages all network aspect of VMs and AHUs in the local server. Example of tasks it is responsible for are: Keep track and manage VM's IP addresses; Control hypervisor's network; Maps NIC's SR-IOV physical and virtual functions; Configure network lookup tables in CRUN shell.

MONITOR-CLIENT takes care of handling monitoring and status requests either from VMs or FPGAs.

The IMAGES database is a reduced version of the main IMAGES database in the BRO-SERVER. This local one contains only the images and their information that are or were recently used in the local host.

6.2.3 BRO Usage

The BRO functionality is better explained by showing what each component does in a typical service request. Figure 6.7 shows what steps BRO goes through and each components responsibilities when a service is requested by the user.

Assuming that all required images were already uploaded and the resources available were discovered and mapped before-hand, a user requests through the CLI a service providing its description. The service descriptor contains mainly the resources required and its details as well as a graph of how they must be logically connected. In a more advanced stage, the BRO could support also constraints such as maximum latency and/or minimum throughput between resources.

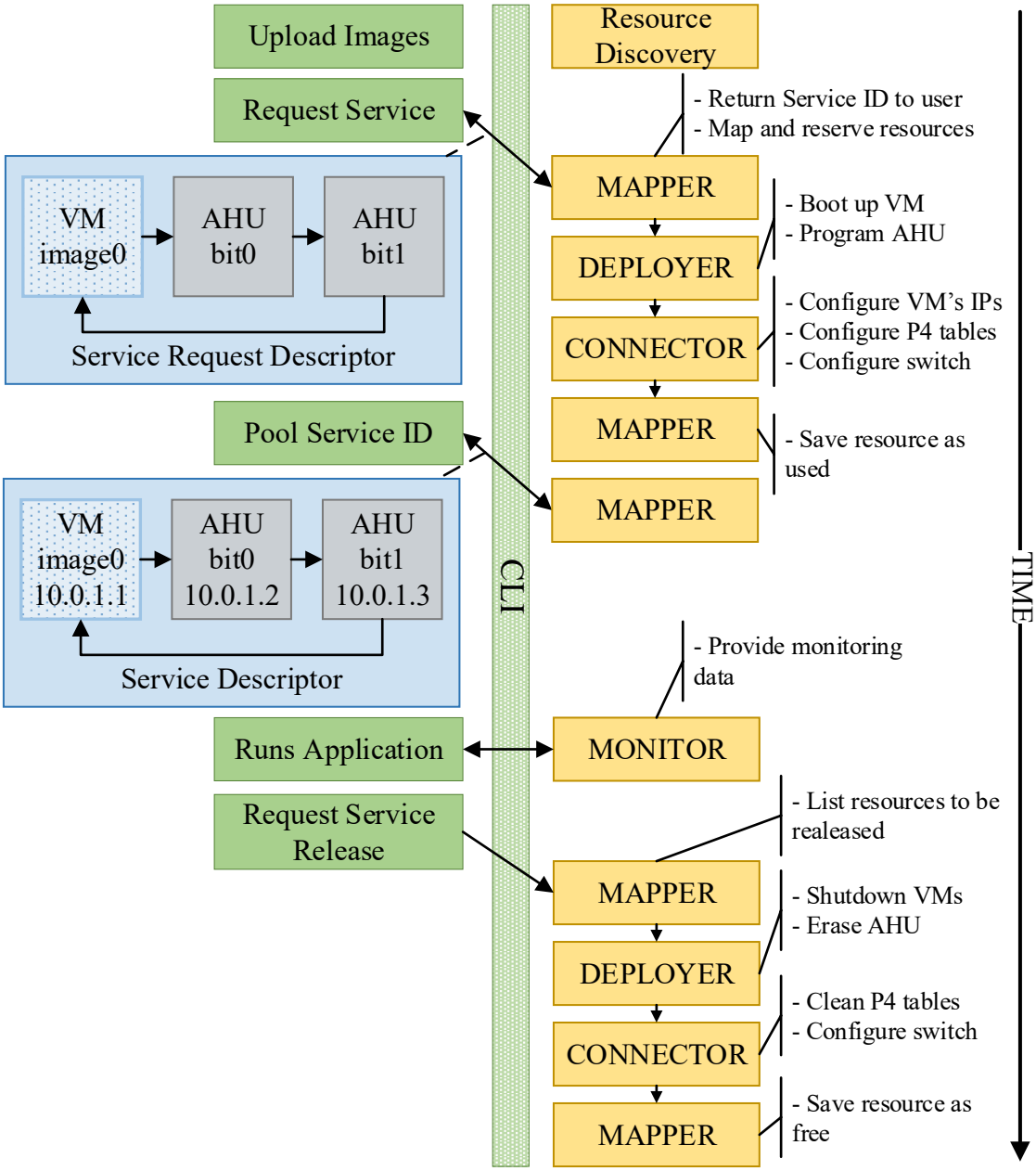

Figure 6.7 *BRO typical usage flow.*

Upon service request, MAPPER first creates a service ID and return it, so the user can check later its status. In the background then MAPPER checks the resources available, maps the request and reserve them. Subsequently, MAPPER issues tasks to DEPLOYER and CONNECTOR to provide the resources and connectivity, respectively.

DEPLOYER boot up the VMs and program the AHUs in the FPGA as requested, by interacting with the DEPLOYER-CLIENTs. Similarly, CONNECTOR interact with

the CONNECTOR-CLIENTs in order to set switches, VM's network and FPGA's SDN components. Assuming everything was successfully completed, once all tasks are done the MAPPER is notified, which in turn set the resources as used in the database.

With the service ID returned when issuing the request, user can poll MAPPER to check its status. Upon successful creation of the service, MAPPER returns a service descriptor with information on how the user can reach the resources. It means that user receives VM's IPs and AHU's flows, so they can be accessed and used. Once the service is up, user can accelerate their application and monitor or debug it through the MONITOR.

When the user wishes to terminate the service, a release request is issued with the service ID. MAPPER then communicate it to DEPLOYER and CONNECTOR which resources and connections should be released. In turn, they issue commands to the targeted DEPLOYER-CLIENTs and CONNECTOR-CLIENTs in order to shut down VMs and remove AHUs as well as reseting the corresponding network configurations. Again, MAPPER is notified when tasks are done, and resources are marked as free in the database and are ready to be used in new service requests.

7. EVALUATION

In this chapter are described the current development state of the CRUN architecture and the first trial realized with it. Furthermore, preliminary results in terms of area, throughput and latency are shown. Finally, the whole system and results are discussed.

7.1 Development State

As mentioned in Chapter 6, CRUN has not yet been completely developed. On the CRUN shell side, the blue (stream path) and yellow (control) components in Figure 6.3 are functional and in advanced state. Currently, one AHU is supported with no partial reconfiguration. Also, multiple flows per AHU has not been tested yet. Furthermore, the green components (DMA path) development has been not started.

In the BRO software, currently the MAPPER, CLI, REST APIs, RESOURCES DESCRIPTOR and IMAGES database as well as jobs queue with Redis Queue are functioning. DEPLOYER and DEPLOYER-CLIENT main functionalities, such as hypervisor management through libvirt and FPGA remote programming have been tested but not integrated. CONNECTOR and CONNECTOR-CLIENT are in similar state, where the tested functionalities such as IP management in hypervisor and host OS as well as SDN control were tested. MONITOR and MONITOR-CLIENT development has not been started.

Thus, BRO is not functional yet and it is not possible to automatically start a service. Using the current state of CRUN shell it is possible to accelerate distributed or network applications by manually starting the system and configuring it.

7.2 Hardware Metrics

The shell should consume as little of the FPGA's resources as possible, so users can fit bigger applications in the same device. The Table 7.1 shows the preliminary values for the current development state of the shell.

Table 7.1 *Shell's resource utilization.*

Resource	Utilization	Available	Utilization %
LUT	102 234	663 360	15.41
LUTRAM	16 242	293 760	5.53
FF	179 912	1 326 720	13.56
BRAM	373	2 160	17.26

From Table 7.1 one can see that the shell consumes about 15% of the resources available, which is a reasonable amount. The most area hungry components in the shell are the P4 RX and P4 TX, which is expected due to their complexity. There are not much that can be done in this regard since these components are automatically generated. Fortunately, the use of P4 by the SDNet™ tool is recent, and it is expected that it will provide considerable improvements and optimizations in the near future.

The shell should not impact performance as well, thus it must not limit throughput and inserts as little latency as possible. All the components in the stream path work in line rate. It means that the shell does not limit the throughput and the total 10Gbps of the available Ethernet port link are provided to the AHUs. In terms of latency, Table 7.2 shows the average (AVG), maximum (MAX) and minimum (MIN) latency values per packet size when the total 10Gbps of throughput is fed to the FPGA.

Table 7.2 *Shell latencies per packet size at 10Gbps.*

Packet Size [bytes]	AVG latency [ns]	MAX latency [ns]	MIN latency [ns]
78	4 122	4 132	4 105
1 088	4 255	4 270	4 240
1 856	4 624	4 637	4 607

The latency values are between 4.1μs and 4.7μs depending on the packet size. They were obtained with the setup shown in Figure 5.1 (a), with IXIA directly connected to the FPGA ports. This range is acceptable but for ultra-low latency applications it may become significant, especially in distributed acceleration where the data travels among multiple shells. P4 RX and P4 TX are the components that limit those values, in fact they are responsible for about 90% of the total latency. Again, this is expected due to the look up operations and complexity of their tasks. In this regard, it is possible to increase the frequency that P4 components use for its internal operations, thus speeding up its processing time. This test has not been done yet.

7.3 Trial

As mentioned previously, the main motivation of this thesis is to enable hardware acceleration for NFV use cases due to its demanding requirements, more specifically for Cloud RAN. Among several applications that are currently executed in Cloud RAN, it is expected a significant increase of machine learning applications on the next mobile network architecture generation (5G).

The CRUN architecture was leveraged by a third party to implement an acceleration trial for inference of a neural network application. In this work, only an overview of the trial and results obtained are shown. A review about machine learning in mobile networks and detailed description and analysis of the results shown here are provided in [9].

The goal of the trial was to run inference of the application with ultra-low latency, between 20μs and 40μs per inference at software level. The application consists of a fully connected neural network, also known as Multilayer Perceptron (MLP), that was trained to detect anomalies in mobile networks.

Figure 7.1 shows a high-level view of the AHU's architecture developed for inferring the MLP, which consists of four fully connected layers. Each layer possess a NEURAL ENGINE that computes multiplications and additions (MAC) among weights and inputs, add biases and applies the activation function for one neuron at a time. The bigger the layer the longer it takes to finish its computation, thus layers can be unrolled to achieve lower latency. In Figure 7.1 LAYER 2 was unrolled once for this purpose. The remaining blocks are responsible for controlling the inference process and buffer the data.

The trial compared the inference performance obtained from CRUN and multiple others implementation options, such as GPP, GPU, GEMX and SDAccel. It also leveraged CRUN for distributing the acceleration among two FPGAs. Table 7.3 shows the performance values of each implementation and its details.

The simplest and straight forward implementations were realized with GPP and GPU using Keras framework [16]. The GPP used was Intel® Xeon® Gold 6130 CPU @ 2.10GHz and GPU used was NVIDIA® Tesla® V100 Data Center.

GEMX from Xilinx [79] is a General Matrix Operation library used for acceleration of BLAS-like matrix operations in FPGAs.

The same AHU developed for CRUN was also deployed in the Xilinx SDAccel™ Environment [80]. SDAccel™ is a framework that offers the possibility of developing

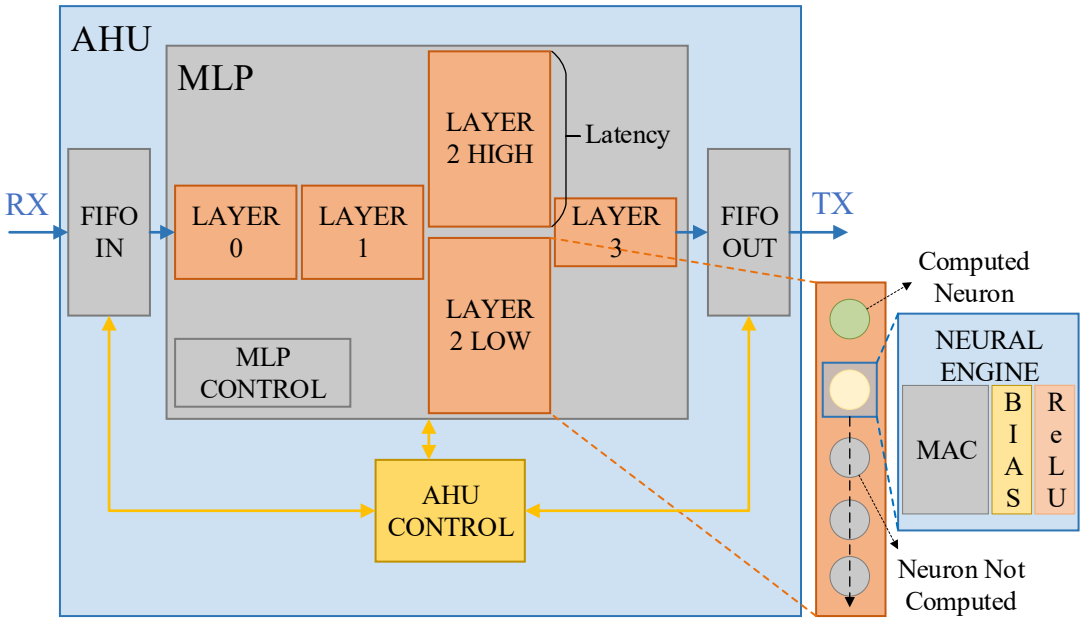

Figure 7.1 *MLP's AHU developed by a third party and used in the CRUN trial.*

Table 7.3 *Results for different implementations of the MLP neural network.*

Solution	**NN Model**	**Latency/ Batch μs**	**Inferences per second**	**Batch Size**	**Freq.** MHz	**FPGA Board**
GPP-1	Keras	798	1 253	1	NA	NA
GPP-16	Keras	3 694.6	4 330	16	NA	NA
GPU-1	Keras	1 897.4	527	1	NA	NA
GPU-16	Keras	1 973.4	8 107	16	NA	NA
GEMX-32	Python API	1 500	21 333	32	60	VCU1525
SDAccel-16	Baseline	602.5	26 556	16	100	VCU1525
SDAccel-1	Baseline	272.5	3 662	1	100	VCU1525
CRUN-B	Baseline	30.96	49 499	1	156.25	KCU1500
CRUN-U	Unrolled	24.40	72 568	1	156.25	KCU1500
CRUN Dist.	Unrolled	32.55	150 488	1	156.25	KCU1500

and delivering accelerated data center applications on FPGAs.

The GPP used for GEMX and SDAccel™ implementations was the same for GPP trial and both were implemented in VCU1525 Reconfigurable Acceleration Platform board [83]. VCU1525 is similar to the KCU1500 used in CRUN but should provide better performance.

The CRUN implementation was realized with the laboratory setup shown in Figure 6.2 and hardware described in Chapter 5. Three different versions are presented: CRUN baseline (CRUN-B), which uses the same MLP core as SDAccel™; CRUN

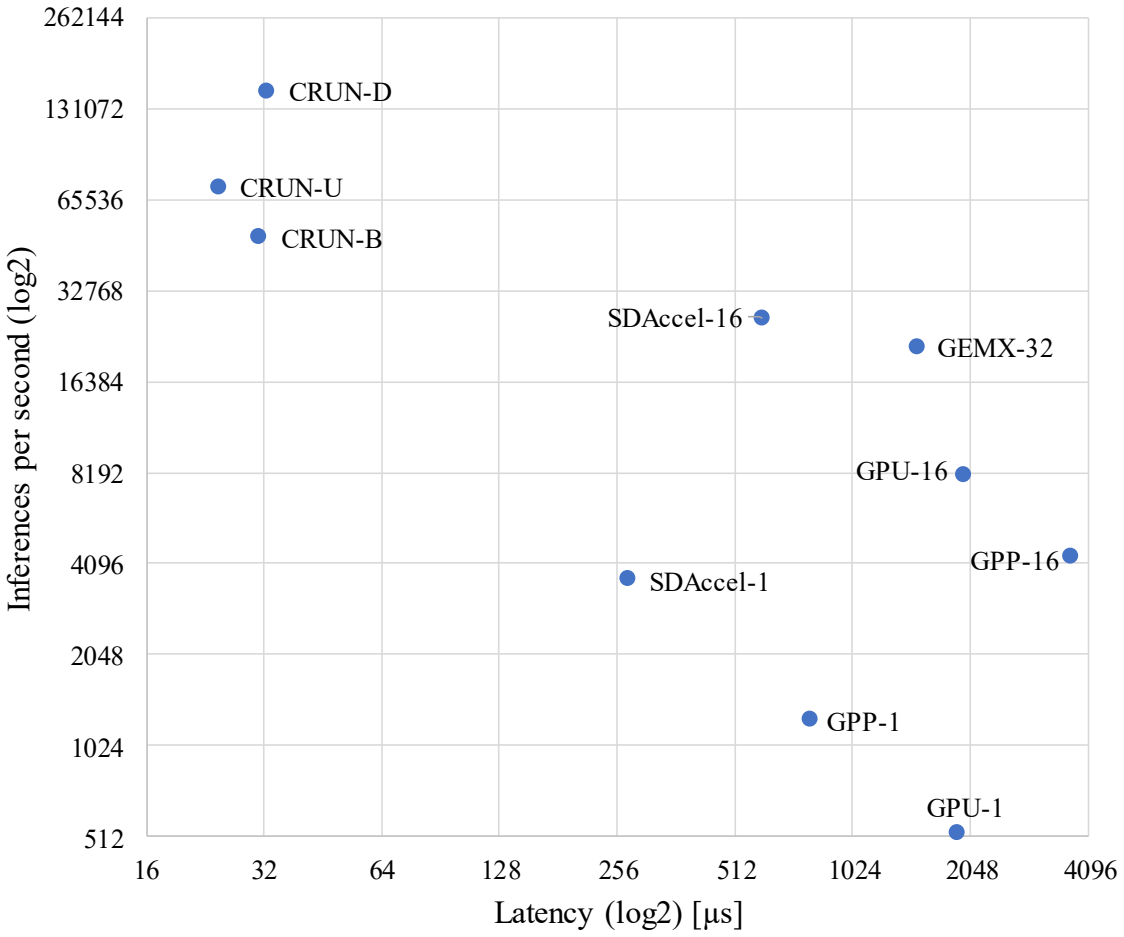

Figure 7.2 *Trial's inferences per second vs. latency results graph.*

Unrolled (CRUN-U) where the LAYER 2 of the MLP was unrolled once; CRUN Distributed (CRUN-D) in which the MLP was distributed into two FPGAs, allowing further optimization.

The latency measurements were obtained from TRex application as shown in Figure 5.1 (b) for CRUN-B and CRUN-U, while CRUN-D points of measurement are shown in (c).

Figure 7.2 shows the inference per second vs latency of each implementation to better present how they compare with each other. From Table 7.3 and Figure 7.2 one can see that only CRUN was able to provide the required latency values.

Here, inferences per second is the actual metric of interest instead of throughput. They are different because GPP and GPU uses 32-bits floating point precision, while FPGA's implementations uses 16-bit fixed point quantization. This quantization

step is common when porting designs to FPGA platforms and for the application in question it did not affect the precision. Still, throughput and inferences per second are used seamlessly in the text since they are proportional.

For all implementations except the CRUN, the batch size is used as a trade-off between latency and throughput. With bigger batch sizes, less is paid in expensive memory transfers between host and accelerator.

When analyzing GPP, even though it performs better than GPU with batch size of 1, if more throughput is desired the price paid in latency is huge and the gain is not as good as the gain with GPU.

GPU on one hand can dramatically increase the inferences per second with bigger batch sizes without much effect on the latency. In fact, batch of 16 utilized only around 1% of the available GPU resources in the trial, meaning that much more could be obtained than the one shown in the graph. This shows well why GPUs are suitable for tasks with large amount of parallelism and also indicates that it is not a good fit for all scenarios, especially when low latency is the goal.

The GEMX implementation shows results on pair with GPU in terms of throughput and slightly better latency. Still, GPU's inference per second overcome GEMX with bigger batch sizes. Since GEMX implementation uses a relatively easy Python API for designing the FPGA's hardware, it is considered here as an interesting alternative.

SDAccel is the best among the other solutions in term of latency, providing also the higher inference per second with same batch sizes. It is an example of the performance that specific designed hardware can achieve in FPGAs. Again, GPU may beat SDAccel throughput with larger batch sizes, but it is not a match in terms of latency.

GEMX and SDAccel implementations can provide good latency values but are still limited, especially due to the costly memory access operations that their architecture uses. For example, in the SDAccel-1, from the total latency of 272.5μs, approximately 245μs are consumed by memory accesses, which represents almost 90% of the total.

CRUN was the only implementation capable of achieving the latency requirements while also providing the best inference per second. The results obtained here place CRUN in a completely different level when compared with the other implementations.

The possibility of distributing the application provided more area and allowed the MLP to be further optimized, which in turn improved the total inferences per second even further. For that, a small price in latency was paid since data travels through two shells and do more hops in the switch.

7.4 Analysis

CRUN enables all three acceleration scenarios, local, network and distributed. In the datacenter, the FPGA can be easily managed through the PCIe and users can have access to local or multiple remote FPGAs seamlessly. Since its deployment aims to be distributed through the datacenter instead of in clusters, no extra resources in the FPGA is consumed to turn it into a standalone device.

Even in its preliminary state, the CRUN proves its potential. The machine learning trial presents considerable gain in performance and is the only suitable solution for the requirements among other solutions studied.

It is important to note that such latency values in CRUN trial was only possible due to the use of TRex that leverages DPDK to bypass the Linux kernel stack. Also, all trials were realized in bare-metal, which means that some performance degradation is expected when running the application in a VM. This performance degradation can be mitigated with the usage of SR-IOV, which is supported in most of modern NICs.

The CRUN latency values presented in Table 7.3 are average ones. In fact, it was observable that the maximum latency value could reach around 290μs. This must be taken into account when using the system. Still, it is expected jitter also in the others implementations, but such values were not obtained.

The use of DPDK and SR-IOV should also be leveraged when building the DMA path for local acceleration. With this, it is expected even better results in terms of latency for both bare-metal and virtualized systems, since there is no need for the data to traffic through the switches in the network. Still, the local acceleration has somewhat limited scalability.

7.4.1 Hardware

CRUN also is the most complex implementation among all other implementations used for comparison. The simple fact that the application uses HDL already add a

barrier for most developers. On the other hand, the usage of standard interfaces such as AXI4-Stream and the abstraction of the I/O signals facilitate the development. As an example, only a few modifications in the control were needed to adapt the hardware design between SDAccel™ and CRUN. Still, work is needed in order to study options in how to provide a platform where the AHU can be easily integrated and developed with support of HLS.

When analyzing the network connectivity, it is clear the importance of the P4 components built from SDNet™ packet processor. On one hand, they can be easily developed, modified and integrated with the set of tools provided by Xilinx®. On the other hand, they are also the components that have major effect on area and latency of the shell. Currently, it seems that the only other option that could provide similar functionality is the project P4FPGA [60], but it has not been investigated.

Another option would be to develop similar functionality using HDL or HLS. Even though the performance and area of the final design could be improved by optimization, it is a very challenging, time consuming and error prone task that would most probably not result in such complete tool. Thus, SDNet™ seems to be the only design option that is easy, powerful and flexible enough for the functionalities required here. Yet, this is a vendor specific and proprietary tool, limited to Xilinx's FPGAs only.

7.4.2 Software

In any cloud environment, software support is needed to allow management of the whole datacenter's infrastructure in an automated fashion, specially with HWA support. From the related work presented in Chapter4, one could point it out that most of the efforts leverages a modified version of OpenStack [58]. In fact, OpenStack is a strong candidate to fill this position in NFV clouds [87], at least for open source solutions.

HWA support from OpenStack is being developed by the Cyborg project [59], which aims to provide a management framework for various types of accelerator resources, such as FPGA, GPU and ASIC. Official OpenStack releases are still not mature in this area, also Cyborg requires that vendors deliver their own Cyborg's driver so the HWA can be deployed. Specifically for FPGAs, unfortunately there is no support yet from vendors available. Furthermore, in the first phase it is probably expected that only PCIe connectivity will be provided.

Thus, the in-house development of a software management called BRO was proposed.

The motivation is to avoid the steep learning curve barrier to modify OpenStack and to obtain a simple and quick VIM like functionality for proof of concept purposes only.

The required functionality could be mapped to the VIM component in MANO's architecture, but one can point out that MAPPER is a rather complex component and goes above the VIM responsibilities of the MANO architecture. Again, MANO's architecture division can be fuzzy, but MAPPER would be better compared with NFVO, providing service level management, while DEPLOYER and CONNECTOR would provide the automation of the infrastructure.

8. CONCLUSIONS

The need for hardware acceleration in cloud infrastructure is accentuated due to the virtualization trend of applications with demanding performance requirements, such as the ones covered by NFV.

This need is not limited to NFV's use cases only. Thus, hardware accelerators in cloud, such as GPUs, are already commonly available for end user. Still, GPUs do not fit well for many workloads and requirement scenarios while FPGAs can adapt efficiently to a broader scope. Hence, FPGAs are good candidates for hardware accelerator resource as GPUs currently are. In fact, FPGAs are already being deployed in commercial cloud, although only in proprietary solutions and at limited scale for end users.

The virtualization of FPGAs requires significant updates in infrastructure's management software as well as support from the hardware accelerator itself. In this thesis, the CRUN framework for enabling FPGAs as accelerator resources in cloud environment was proposed. The architecture contains hardware and software components that provide sharable and scalable FPGA acceleration, while abstracting away complex tasks from users, such as setting interfaces and network configuration.

Partial Reconfiguration is proposed to enable a sharable FPGA resource, in which CRUN shell leverages static components that provides the necessary infrastructure. This allows users to independently deploy their accelerators in the same FPGA.

Scalability is obtained by providing network connectivity powered by SDN components that enables a flexible and automated network. With this approach users can scale up their applications from a single region of a shared FPGA to several FPGAs that are automatically logically connected through the network.

The proposed BRO software offers the necessary functionality for management of the cloud infrastructure and support the hardware and virtualized resources. Also, BRO exposes to users a simple interface for requesting, configuring and logically connecting VMs and FPGA's hardware accelerators (AHU).

Some features and improvements are still missing from the CRUN framework and are subject for future development. For example, even though CRUN employ basic mechanism to avoid users from accessing each other's data and affecting each other's performance, security is a requirement that have not been deeply analyzed and one cannot assure it is guaranteed. Also, a higher level of abstraction for development and integration of the accelerator's hardware, such as HLS support and APIs for controlling AHUs and transferring data needs further investigation and development. Furthermore, the BRO software management aims to support proof of concept level development and the system should be integrated to a tool that provide all the necessary features for managing the cloud infrastructure, such as OpenStack.

The CRUN framework has not yet been fully implemented. The current CRUN shell uses about 15% of the main resources available in the FPGA and provide the full 10Gpbs of throughput available in the Ethernet link to the user's accelerators, while adding around 4μs of latency for packet processing.

Even though not completely implemented, in its current stage CRUN can provide distributed and network acceleration. Indeed, a machine learning application trial was carried out independently as a proof of concept of the framework. The trial achieved excellent results and was the only solution to fulfill the application's ultra-low latency requirements. In fact, it overcame all of the other implementations in terms of inference per second as well, although with more optimizations GPU may probably provide better values, but not at the same latency range.

Summarizing the trial's results, CRUN was able to provide 24.4μs of latency in average at best, while the second best was obtained from SDAccel implementation and achieved 272.5μs. Common implementations, such as GPP and GPU provided 798μs and 1 897μs respectively at best.

The results show the FPGA and CRUN architecture potential for providing high performance in cloud environments. Furthermore, CRUN enables distributed hardware acceleration in the cloud.

BIBLIOGRAPHY

[1] N. Antonopoulos and L. Gillam, *Cloud Computing: Principles, Systems and Applications.* Springer Publishing Company, Incorporated, 2012.

[2] ARM, *AMBA® 4 AXI4-Stream Protocol.*

[3] ARM, *AMBA® AXI™ and ACE™ Protocol Specification.*

[4] AWS. Amazon EC2 instance types: Accelerated computing. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[5] M. Azure. GPU optimized virtual machine sizes. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu

[6] K. Bousselmi, Z. Brahmi, and M. M. Gammoudi, "Cloud services orchestration: A comparative study of existing approaches," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, May 2014, pp. 410–416.

[7] Z. Bronstein, E. Roch, J. Xia, and A. Molkho, "Uniform handling and abstraction of nfv hardware accelerators," *IEEE Network*, vol. 29, no. 3, pp. 22–29, May 2015.

[8] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 109–116.

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

[10] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, S. Heil, J. Kim, D. Lo, M. Papamichael, T. Massengill, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," *IEEE Micro*, pp. 1–1, 2018.

[11] 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, and D. Burger, "Configurable clouds," *IEEE Micro*, vol. 37, no. 3, pp. 52–61, 2017.

[12] CCIX. About CCIX consortium. [Online]. Available: https://www.ccixconsortium.com/about/

[13] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann, "Cloud RAN for mobile networks—a technology overview," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 405–426, 2015. [Online]. Available: https://doi.org/10.1109/comst.2014.2355255

[14] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2597917.2597929

[15] China Mobile, "C-RAN: The road towards green ran," China Mobile Institute, Withe Paper, 2011.

[16] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[17] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, M. Ghandi, D. Lo, S. Reinhardt, S. Alkalay, H. Angepat, D. Chiou, A. Forin, D. Burger, L. Woods, G. Weisz, M. Haselman, and D. Zhang, "Serving dnns in real time at datacenter scale with project brainwave." IEEE, March 2018. [Online]. Available: https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/

[18] Cisco. Trex. [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_book.pdf

[19] Cisco, "Global mobile data traffic forecast update," Cisco Visual Networking Index, White Paper, 2017. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html

[20] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, "Advancing software-defined networks: A survey," *IEEE Access*, vol. 5, pp. 25 487–25 526, 2017.

[21] Docker Inc. Docker. [Online]. Available: https://www.docker.com/

[22] ETSI. Network functions virtualisation. [Online]. Available: https://www.etsi.org/technologies-clusters/technologies/nfv

[23] ETSI. Network functions virtualisation (nfv); architectural framework. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf

[24] ETSI. Network functions virtualisation (nfv); nfv performance & portability best practises. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-PER/001_099/001/01.01.02_60/gs_NFV-PER001v010102p.pdf

[25] ETSI. Network functions virtualisation (nfv); use cases. [Online]. Available: https://www.etsi.org/deliver/etsi_gr/NFV/001_099/001/01.02.01_60/gr_NFV001v010201p.pdf

[26] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, pp. 430–435.

[27] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[28] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "Openanfv: Accelerating network function virtualization with a consolidated framework in openstack," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 353–354, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2740070.2631426

[29] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 46–56.

[30] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, Feb 2015.

[31] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "NFV: state of the art, challenges, and implementation in next generation mobile networks (vepc)," *IEEE Network*, vol. 28, no. 6, pp. 18–26, Nov 2014.

[32] IEEE, "Ieee standard for ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, March 2016.

[33] Intel. HARP: Hardware accelerator research program. [Online]. Available: https://software.intel.com/en-us/hardware-accelerator-research-program

[34] Intel. Intel®fpga sdk for opencl™: Product brief. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/ps-opencl.pdf

[35] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–10.

[36] C. Kachris, G. C. Sirakoulis, and D. Soudris, "Network function virtualization based on fpgas: A framework for all-programmable network devices," *CoRR*, vol. abs/1406.0309, 2014.

[37] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE Computer Society Annual Symposium on VLSI*, July 2010, pp. 288–293.

[38] E. J. Kitindi, S. Fu, Y. Jia, A. Kabir, and Y. Wang, "Wireless network virtualization with SDN and c-RAN for 5g networks: Requirements, opportunities, and challenges," *IEEE Access*, vol. 5, pp. 19 099–19 115, 2017. [Online]. Available: https://doi.org/10.1109/access.2017.2744672

[39] O. Knodel, P. Lehmann, and R. G. Spallek, "RC3E: Reconfigurable accelerators in data centres and their provision by adapted service models," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 19–26.

[40] M. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, Nov 2015, pp. 74–78.

[41] D. Kreutz, F. M. V. Ramos, P. E. VerÃssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[42] KVM. Kernel virtual machine. [Online]. Available: https://www.linux-kvm.org/page/Main_Page

[43] J. Lallet, A. Enrici, and A. Saffar, "Fpga-based system for the acceleration of cloud microservices," in *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, June 2018, pp. 1–5.

[44] libvirt. libvirt: Virtualization api. [Online]. Available: https://libvirt.org/

[45] T. Lin, N. Tarafdar, B. Park, P. Chow, and A. Leon-Garcia, "Enabling network function virtualization over heterogeneous resources," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Sept 2017, pp. 58–63.

[46] Y. Luo, S. Huang, J. Chou, and B. Chen, "A computation workload characteristic study of c-ran," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 1599–1603.

[47] lxml. lxml - XML and HTML with python. [Online]. Available: https://lxml.de/

[48] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," Gaithersburg, MD, United States, Tech. Rep., 2011.

[49] Microsoft. Single root i/o virtualization (SR-IOV). [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/single-root-i-o-virtualization--sr-iov-

[50] Microsoft Azure. Deploy a model as a web service on an FPGA with azure machine learning. [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-deploy-fpga-web-service

[51] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, Firstquarter 2016.

[52] R. Mijumbi, J. Serrat, J. Gorricho, S. Latre, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, January 2016.

[53] N., P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software-defined networking," *Proceedings of the IEEE*, vol. 103, no. 7, pp. 1102–1124, July 2015.

[54] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in nfv environments," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, March 2015, pp. 1–5.

[55] L. Nobach, B. Rudolph, and D. Hausheer, "Benefits of conditional fpga provisioning for virtualized network functions," in *2017 International Conference on Networked Systems (NetSys)*, March 2017, pp. 1–6.

[56] B. A. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, Third 2014.

[57] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 77–84.

[58] OpenStack. OpenStack. [Online]. Available: https://www.openstack.org/

[59] OpenStack. OpenStack cyborg. [Online]. Available: https://wiki.openstack.org/wiki/Cyborg

[60] P4FPGA. P4FPGA. [Online]. Available: https://github.com/p4fpga/p4fpga

[61] S. Patidar, D. Rane, and P. Jain, "A survey paper on cloud computing," in *2012 Second International Conference on Advanced Computing Communication Technologies*, Jan 2012, pp. 394–398.

[62] M. Peng, Y. Sun, X. Li, Z. Mao, and C. Wang, "Recent advances in cloud radio access networks: System architectures, key techniques, and open issues," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2282–2308, 2016. [Online]. Available: https://doi.org/10.1109/comst.2016.2548658

[63] S. Pinneterre, S. Chiotakis, M. Paolino, and D. Raho, "vFPGAmanager: A virtualization framework for orchestrated fpga accelerator sharing in 5g cloud environments," *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pp. 1–5, 2018.

[64] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, June 2014, pp. 13–24. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/

[65] Python. Python. [Online]. Available: https://www.python.org/

[66] J. Rittinghouse and J. Ransome, *Cloud Computing: Implementation, Management, and Security*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[67] S. B. Shaw and A. K. Singh, "A survey on cloud computing," in *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*, March 2014, pp. 1–6.

[68] SQLAlchemy. SQLAlchemy. [Online]. Available: https://www.sqlalchemy.org/

[69] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for fpgas in the cloud," *IEEE Design Test*, vol. 35, no. 1, pp. 23–29, Feb 2018.

[70] The CentOS Project. The CentOS project. [Online]. Available: https://www.centos.org/

[71] The Linux Foundation. Data plane development kit. [Online]. Available: https://www.dpdk.org/

[72] The P4 Language Consortium. $P4_{16}$ language specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[73] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling fpgas in hyperscale data centers," in *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)*, Aug 2015, pp. 1078–1086.

[74] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Disaggregated fpgas: Network performance comparison against bare-metal servers, virtual machines and linux containers," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016, pp. 9–17.

[75] B. Wile, *Coherent Accelerator Processor Interface (CAPI) for Systems*, IBM Systems and Technology Group.

[76] J. Wu, Z. Zhang, Y. Hong, and Y. Wen, "Cloud radio access network (c-ran): a primer," *IEEE Network*, vol. 29, no. 1, pp. 35–41, Jan 2015.

[77] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240, 2013.

[78] Xilinx, *AXI® Reference Guide.* [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf

[79] Xilinx. General matrix operation library. [Online]. Available: https://github.com/Xilinx/gemx

[80] Xilinx, *SDAccel Environment User Guide.* [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1023-sdaccel-user-guide.pdf

[81] Xilinx. SDAccel™ development environment. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[82] Xilinx, *SDNet Packet Processor.* [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf

[83] Xilinx, *VCU1525 Reconfigurable Acceleration Platform.* [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf

[84] Xilinx. Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html#documentation

[85] Xilinx. Xilinx kintex ultrascale FPGA KCU1500 acceleration development kit. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/dk-u1-kcu1500-g.html

[86] Xilinx. Xilinx PCI Express DMA drivers and software guide. [Online]. Available: https://www.xilinx.com/support/answers/65444.html

[87] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212 – 262, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128618300306

[88] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel, "A hybrid asic and fpga architecture," in *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, Nov 2002, pp. 187–194.