Abstract

Telecommunication providers need an infrastructure which is used to host network services like internet, telephone, mobile, etc. This infrastructure is traditionally built using dedicated hardware appliances with fixed functionality. Because of this fixed functionality it is necessary to acquire new appliances when technology evolves. Therefore a lot of telecommunication providers started looking at alternatives and payed a lot of attention to cloud platforms. With the help of hardware virtualization techniques, cloud platforms are able to use a large pool of general purpose hardware resources. These platforms are mostly used for data driven applications like web hosting, multimedia streaming, etc. Telecommunication providers aim to move their network infrastructure into cloud platforms where network functions run in software. This evolution is already in place under the term Network Function Virtualization (NFV). At this moment Openstack is a cloud platform which has already proven a reliable solution for data driven applications. Because of the open source license and large community it has been identified as a well suited candidate for hosting network services. NFV applications have different performance requirements which means that Openstack needs to improve certain parts of the platform. For example, the data path which interconnects network functions. This thesis will evaluate the improvements which have already been adopted by the Openstack community. And also an improvement of the data path which at the moment is in an experimental phase.

Contents

Glossary

ACL Access Control List.

API Application Programming Interface.

BIOS Basic Input/Output System.

BNG Border Network Gateway.

CDN Content Delivery Network.

COTS Commercial off-the-shelve.

CPE Customer Premises Equipment.

CPU Central Processing Unit.

DHCP Dynamic Host Configuration Protocol.

DMA Direct Memory Access.

DNAT Destination Network Address Translation.

DPDK Data Plane Development Kit.

EPC Evolved Packet Core.

FCAPS Fault, Configuration, Accounting, Performance, Security.

GPU Graphical Processing Unit.

GRE Generic Routing Encapsulation.

IMS IP Multimedia Subsystem.

IOMMU Input/Output Memory Management Unit.

IP Internet Protocol.

KVM Kernel-based Virtual Machine.

LLC Last Level Cache.

MANO Management and Orchestration.

MPLS Multiprotocol Label Switching.

NFV Network Functions Virtualization.

NFVi Network Functions Virtualization infrastructure.

NIC Network Interface Card.

NUMA Non-uniform memory access.

OPNFV Open Platform for NFV.

OS Operating System.

OVS Open vSwitch.

PCI Peripheral Component Interconnect.

PF Physical Function.

PoP Point of Presence.

QEMU Quick Emulator.

QOS Quality of Service.

QPI QuickPath Interconnect.

RAM Random Access Memory.

SNAT Source Network Address Translation.

SR-IOV Single Root Input/Output Virtualization.

SSH Secure Shell.

SUT System Under Test.

TLB Translation Lookaside Buffer.

UDP User Datagram Protocol.

VF Virtual Function.

VLAN Virtual Local Area Network.

VM Virtual Machine.

VNF Virtual Network Function.

VXLAN Virtual Extensible Local Area Network.

Chapter 1

Introduction

1.1 NFV background

As technology evolves telecommunication operators constantly need to change the infrastructure to provide services. For example, ever since smartphones became popular, the demand for mobile internet increased significantly. This resulted in the demand for new equipment which supports this functionality. At the moment this thesis is being written, a lot of telecommunication operators are still rolling out 4G services while 5G is already emerging. Each time a new type of service becomes available they need to acquire new hardware.

Looking at data center technology it became evident that there is an alternative way of using hardware. Virtualization technology has matured a lot in the last decade because of the efforts made by hardware producers like Intel and AMD. This has received a lot of attention from telecommunication operators which started to adopt this technology in the recent years. For example AT&T, one of the major telecommunications vendors in the United States, has announced that it plans to virtualize 75% of its network[1] using Software Defined Networking (SDN) and Network Functions Virtualization (NFV) by 2020.

This upcoming demand has also been identified by the European Telecommunications Standards Institute (ETSI) in 2012. ETSI is a non-profit organization which provides standardization of telecommunication technology including fixed, mobile, radio, converged, broadcast and internet technologies. It is a collaboration between 800 members including including hardware manufacturers, network operators, service providers, etc. It provides a whole section for NFV[2] where there are specifications available describing the elements that are part of the virtualized infrastructure.

The advantages of using a virtualized infrastructure include the following:

Faster deployment of network services Traditional hardware is designed to support specific functionality. The hardware manufactures cannot predict the future, therefore telecommunication operators need to acquire new devices when technology evolves. Using virtualized technology it is possible to deploy network services as software applications which allow the use of already acquired hardware for new technology.

Dynamic scaling It is difficult to estimate the amount of required hardware for network services. Virtualization allows a network service to scale at peak moments.

Placement of network functions Using virtualization technology the physical placement of a network function becomes controllable. This has a lot of advantages, for example it allows an operator to move all the network functions to the same part of the data-center during the night. The unused machines can be shut down which reduces power consumption.

Expanding the infrastructure Expanding traditional infrastructure is a very cumbersome process. It requires a lot of configuration and technical knowledge. Cloud platforms are able to use new hardware in an orchestrated way. In practice, it can be as easy as installing the hardware and adding them to an infrastructure manager which deploys the platform automatically.

[1] `http://about.att.com/innovation/sdn`
[2] `http://www.etsi.org/technologies-clusters/technologies/nfv`

Using a virtualized platform allows flexible placement of network functions. This requires a platform which manages the hardware resources. For traditional data centers there are already cloud platforms available. One of the most popular platforms is Openstack. It has therefore been the platform of choice for the Open Platform for NFV project, also called OPNFV.

OPNFV is a collaborative project which aims to identify shortcomings of existing cloud technology in the context of network services. It can be seen as a major collaboration between most of the big telecommunication companies and hardware providers to create a cloud platform for NFV. In practice it provides extra capabilities to Openstack. It consists of a number of projects which provide a specific capability. One of these projects, called *Open vSwitch For NFV*, is used in this thesis.

This thesis uses OPNFV Brahmaputra which doesn't represent a full orchestrated cloud environment but focuses on the part that manages the hardware resources, also called the Virtual Infrastructure Manager (VIM). The goal is to create an open source cloud platform which can be integrated by telecommunication providers into their orchestrator.

In February 2016 ETSI announced a new project which aims to provide a open source orchestration platform[1] called Open Source MANO (OSM). One of the solutions provided by OSM is called OpenMANO which can be used for certain orchestration functionality. At Mobile World Congress 2016 OSM has successfully hosted a demonstration of a virtual network service using OpenMANO and Openstack as the virtual platform[12].

1.2 NFV data center

Data centers have become a very common concept in various environments. At the moment this thesis is being written these environments have proven to be very useful for application level services. Examples of these are web hosting, eCommerce, big data, DBaaS, etc. These have a common property that they store content for users, in other words the data-center owns the data. This property requires certain characteristics of the data center, for example they often have some kind of replication mechanism.

Telecommunication providers are concerned with providing communication channels between users. For example providing internet connectivity, providing mobile networks, etc. These communication channels often consists of a chain of services which together form an end-to-end service. In this kind of environment the data center does not own data, but instead it is responsible for delivering it from point A to B. This also means that they are mostly focused on L2, L3 and L4 of the networking stack instead of the application layer. The requirements become different in the sense that it is more important to provide fast and reliable communication channels than storing content. This introduces the concept of NFV data centers which are specific for delivering network services.

To make optimal use of hardware in data centers, they use virtualization technology. The concept of using virtualization in combination with many hardware resources is also called cloud computing. In practice this can been seen as a large pool of servers which acts as hosts where each host can run one or multiple virtual machines. Each virtual machine requires a data path to the outside network, in other words between the network interface and the virtual machine.

The data path between the network interface and the virtual machine used in existing cloud platforms is not well suited for delivering network services. Therefore this data path needs to be optimized. This problem can be solved in 2 ways, either by providing *direct hardware access* from the virtual machine or by using a virtual switch which is optimized. This thesis will compare 2 popular approaches which are available today, it can therefore be classified as a *comparative study*: *SR-IOV* makes use of direct hardware access and *Open vSwitch with DPDK* is a popular virtual switch which is enhanced to accelerate the data path.

These 2 techniques will be compared using Openstack as the cloud platform. Openstack is an open-source platform which gets a lot of attention from NFV communities. It is the platform used by the Open Platform for NFV community which provides a platform which is optimized for NFV use cases.

At the moment of writing this thesis Openstack is already used in telecommunication environments. One example is the integration with OpenMANO. In this environment the data path is optimized by allowing direct hardware access form the VM to the NIC, also called PCI passthrough[23]. PCI passthrough of the NIC has its own limitations which have led to the development of SR-IOV which can be combined with PCI-passthrough. SR-IOV is a hardware feature on the NIC which allows the NIC to be used as a L2 switch. This introduces multiple network interfaces in the OS which are called virtual functions. Each of these interfaces has its own MAC address.

Although SR-IOV overcomes the performance issues with NFV data path, there are still some shortcomings. This has led to the demand for alternatives. One alternative is the use of an optimized

implementation of a virtual switch. Open vSwitch is a popular virtual switch which can be combined with a library called DPDK to achieve better performance. Since Open vSwitch is an open source project, it is a promising alternative to SR-IOV when using Openstack.

1.3 Research questions

This thesis has been combined with an internship with Intel as a part of a market development group. The goal of this group is to characterize Intel hardware which provides performance data. The main question posed was the following: *How can a virtualized environment (based on Openstack) be used by telecommunication operators to deploy their network services?* Openstack has already adopted features to improve support for NFV use cases, these are often referred to as Enhanced Platform Awareness (EPA)[10]. These features include PCI passthrough, SR-IOV and exposing hypervisor configuration options. Since there is a great demand for virtual switches OPNFV has initiated a project called *Open vSwitch for NFV*. This project uses Open vSwitch with DPDK as a data plane accelerator. In contrary to PCI passthrough and SR-IOV, Open vSwitch with DPDK is in an experimental phase and not yet adopted by the Openstack community. This leads to the following questions which are clarified in this thesis:

1. Why is it necessary to optimize the data path between the NIC and the VM?
2. Why is the current technology, SR-IOV with PCI passthrough, insufficient? Is an alternative really necessary?
3. Is *Open vSwitch with DPDK* a valid alternative? If so, what are the shortcomings?

1.4 Thesis structure

The structure of the thesis looks as follows. Chapter 2 describes the NFV framework defined by ETSI. This framework identifies the different components of an architecture for telecommunication providers focused on virtualization. In chapter 3 the shortcomings of the native networking stack of a general purpose OS are discussed and alternatives are presented.

Chapter 4 and 5 discuss Openstack which is a cloud resource management platform. This platform can be used to manage COTS hardware. The platform needs to be configured with a data path between the NIC and the VM which is used for data plane packet processing. The different alternatives are described in chapter 6.

Finally the different alternatives are evaluated and compared in chapter 7 and 8. In chapter 9 the results are evaluated and discussed.

Chapter 2

NFV architectural framework

2.1 Introduction

Telecommunication operators need network services for various reasons. They are the core for providing internet, cellphone are any kind of connectivity. They provide various functions such as security, user subscribing, etc. These services require a high networking throughput and a low latency. To meet these high demands, the network services are mostly provided as dedicated hardware appliances.

Although these appliances satisfy the demand for high performance, there are a number of downsides. The configuration is often vendor specific and requires trained personnel. If a network operator chooses appliances of a certain vendor, it is later hard to move to a different vendor. Most of the time these appliances perform a single or a limited number of related functions which are programmed in the circuit. Therefore the operator needs to buy different appliances for different functions, and they can't always be upgraded.

Because of their physical appearance, it is not trivial to insert or replace certain services. Besides the need for space and power it could also take a while before the services are in place. If there is a sudden or temporary need for more services, the only choice is to add extra appliances in advance which would lead to unused infrastructure. Furthermore, the life cycle of these appliances keeps decreasing as technology evolves.

These issues where also very common in other computer areas where the solution to this problem is provided by hardware virtualization. Virtualization allows software to be isolated from the hardware, in most cases the whole operating system runs as a virtual instance or virtual machine (VM). The use of VMs has many benefits, it allows multiple operating systems to run on the same hardware device and it allows dynamic access to resources like processing power. Besides the benefits there are also some downsides, running an application or operating system as a virtual machine has a performance penalty. Because of this penalty some processor manufacturers added support for virtualization on their processors, for example Intel VT-x[38] and AMD-V.

To fully benefit from virtualization techniques a lot of companies started building data centers hosting thousands or even millions of computers. These data centers provide compute, networking and storage resources to a large amount of users. These resources can be accessed on demand, this allows the users to launch new VMs without the need for new hardware and it allows software to be migrated to different physical devices. This concept of providing shared resources on demand is called cloud computing.

Cloud computing has been a well established solution for many years in different areas. To solve the problem of manual configuration of network functions we can apply the concept of cloud computing to network functions, this is called Network Functions Virtualization (NFV). NFV aims on deploying Virtual Network Functions (VNF) on standard high volume servers, switches and storage. This collection of software and hardware resources which build the environment for VNFs is called the NFV Infrastructure (NFVi).

The use of virtualization in a network deployments creates new challenges that where not present in traditional non-virtualized environments. The European Telecommunications Standards Institute (ETSI) has published an architectural framework that focuses on the aspects that are unique to virtualization [16] in a telecommunication network. This framework describes different functional blocks and their interaction through reference points. Different vendors can build their own functional blocks which can be combined in a fully interoperable multi-party NFV solution.

Figure 2.1 shows the different elements which form a complete NFV solution. The three main domains

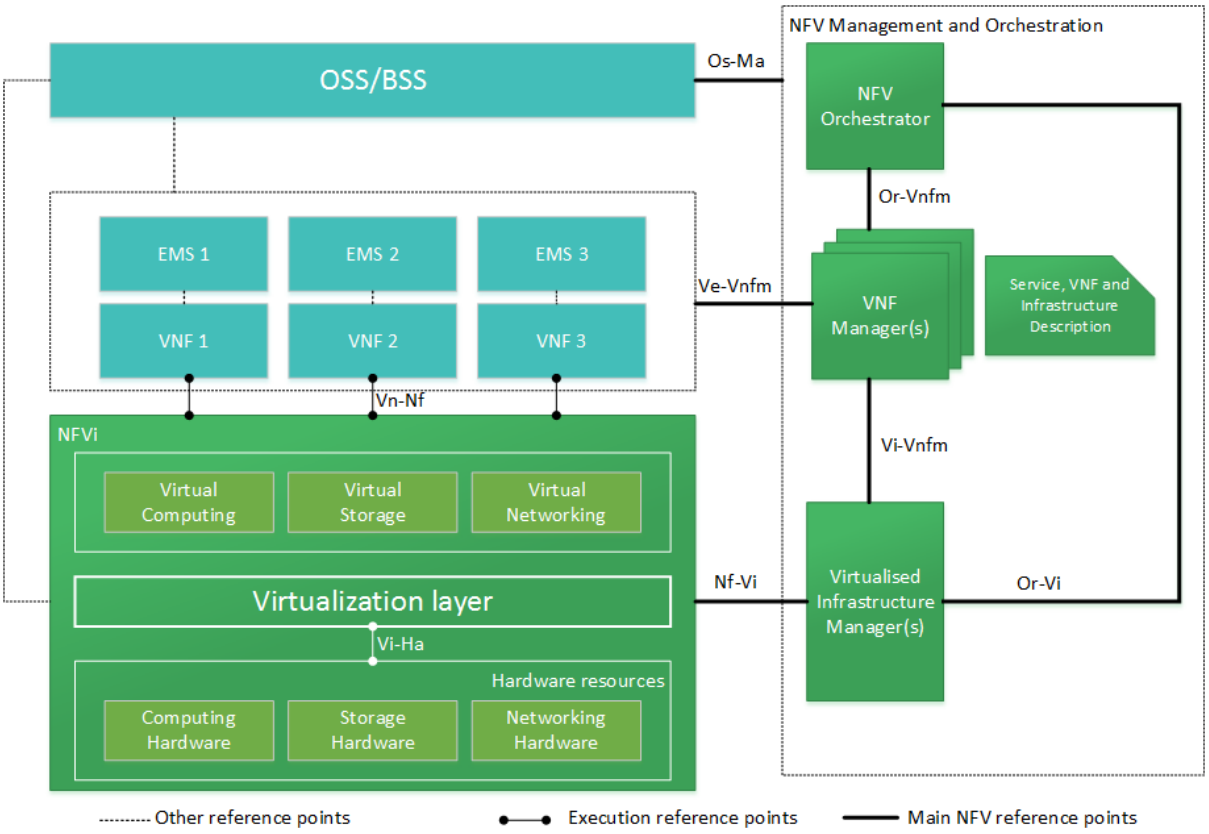

Figure 2.1: NFV reference framework as described by ETSI. The framework defines the different elements which can be implemented independent and accessed through their reference points.

are:

Virtualised Network Function: The goal of NFV is to move existing network functions on dedicated hardware into a virtualized environment in software, these network functions are referred to as VNFs. A VNF can be a single component or a combination of multiple components, for example as a deployment over multiple VMs. Each VNF can be accompanied with a Element Management System (EMS) which can be used for configuration and monitoring.

NFV Infrastructure: The collection of hardware and software resources which build the environment used for the deployment of VNFs is called the NFV infrastructure. This includes the computing, storage and network hardware resources and the abstraction layer which decouples the hardware from software.

NFV Management and Orchestration: the virtualization specific management tasks are handled by NFV Management and Orchestration (NFV MANO). It is responsible for the management of the NFVi, the life cycle management of the VNFs and the orchestration of end-to-end services.

A telecommunications service provider will eventually need to offer end-to-end services to customers and create revenue. The Business Support Systems (BSS) refers to the applications which are responsible for customer facing activities. These include account management, billing, customer support, and service modification. On the other hand service providers need to make sure that the end-to-end services are provisioned and maintained. The systems that are responsible for this are called Operations Support Systems (OSS). Both systems work together to provide telecommunication services to the customers and are already present in traditional networks.

The different functional blocks in the architectural framework interact with each other through reference points. These are called reference points since it exposes an external view of a functional block. The execution reference points deal with the hosting of VNFs and the virtualisation of hardware resources. This allows VNFs to be ported to other NFVi environments and it allows different underlying hardware.

Most of the reference points are between functional blocks in the NFV management and orchestration domain. These reference points are concerned with the management and operation of the network environment. The other (dotted) reference points are already present in current network environments and might need extension for NFV.

2.2 NFV infrastructure

As mentioned before NFVi is defined by the collection of software and hardware resources which build the environment for VNFs. These resources can be placed on different locations where each location is referred to as a NFVi Point of Presence (PoP). The hardware resources in a NFVi include compute, network and storage hardware which are abstracted by the virtualisation layer. The following domains are identified by the ETSI NFV infrastructure overview[18]:

- Compute domain
- Hypervisor domain
- Network domain

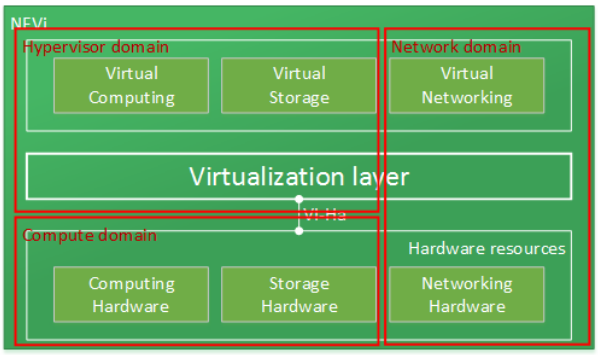

Figure 2.2: Overview of the different NFVi domains identified by the ETSI NFV infrastructure overview. The hypervisor domain is concerned with the virtualization layer on top of the compute and storage hardware. The compute and storage hardware form the compute domain and the network domain is concerned with networking devices and their virtualization capabilities, for example OpenFlow enabled switches.

2.2.1 Compute domain

The main components in this domain are the servers and the storage devices. The servers which are responsible for running the different VMs are also called compute nodes. The connection with the physical part of the network domain is provided by the NIC. More complex NICs may contain offload features or other acceleration features like SR-IOV. Chapter 6 gives a more detailed explanation of SR-IOV.

A lot of network functions need to hold some kind of state to function. A router for example has a routing table which is used for its functionality. The preferred characteristics for storage can be different for each VNF. The primary characteristics of storage devices are:

- Latency
- Capacity
- Volatility or persistence

A server has CPU registers, CPU cache and RAM which provide fast storage but usually don't have a very high capacity and lose all its state when the device reboots. A larger and persistent form of storage is provided by Hard Disk Drives (HDD) which can be part of the server which hosts the VMs. This kind of storage is called Direct Attached Storage (DAS).

A HDD can also be placed in a separate storage device and provide storage to the compute nodes through a network channel. This can be done either by Network Attached Storage (NAS) or a Storage Area Network (SAN). NAS provides a file system which can be attached over the network. It typically provides access to files using network file sharing protocols. A SAN provides storage at block-level which can be used to build a file system.

2.2.2 Hypervisor domain

The software layer which exposes the physical resources on a server as virtual resources is defined as the hypervisor domain. The hypervisor itself allows multiple VMs to run on a single server. Running an operating system as a VM often comes with a performance penalty. Some CPU instructions need to be translated which introduces a context switch. Hardware vendors quickly saw the benefits of running operating systems as VMs and modified their products to improve performance. Intel VT-x and AMD AMD-V are two technologies which allow VMs to run on a server without the need for translated CPU instructions. Another performance penalty is a consequence of sharing resources between multiple VMs. The host operating system can move virtual cores to different physical cores at any time which introduces a context switch.

Another major part of the hypervisor domain is the connection of VMs with each other and outside the host. The most straight-forward way to do this is to assign a device directly to the VM, this is called device pass-through. This limits the number of possible VMs to the number physical devices. To overcome this problem it is possible to use SR-IOV to create VFs which can be passed to the VM. This also limits the number of VMs, but it is possible to create much more VFs than it is to attach physical devices. Another way of interconnecting VMs is by using a virtual switch. This switch will run in software and is able to interconnect VMs with each other and with the NIC. More information is available in chapter 6.

2.2.3 Network domain

The network domain is responsible for the following communication channels:

- between the different components of a VNF.
- between VNFs.
- between the VNFs and their orchestration and management.
- between components of the NFVi and their orchestration and management.

To make optimal use of the network resources it is necessary to create virtual networks. This allows us to use the same network infrastructure for different connections between VNF components. One way of defining virtual networks is by using VLANs where each network is associated with a VLAN identifier. This identifier is limited to 4096 different virtual networks which is not sufficient in a data center context. Other techniques like VXLAN which support up to 16 million virtual networks can be used to overcome this problem.

The Network domain will provide an interface to manage the physical network infrastructure which can be used by the VIM. SDN technologies like OpenFlow can be used for this purpose. For example, the NFVi can provide a centralized OpenFlow controller which is used to control OpenFlow enabled switches. In this case OpenFlow is used to manage and control the network infrastructure. When a new VNF is inserted the OpenFlow controller will update the necessary devices. This approach provides a great benefit to service providers since they only need to provide switches which can be controlled by the OpenFlow protocol. This prevents vendor lock-in since OpenFlow is an open protocol. SDN and OpenFlow will be described in more detail in section 6.4.1.

2.3 NFV Management and Orchestration

A NFV environment exists of many devices with different functionalities. It is important that all these devices work together to create a network service which can be offered to the customer. The service provider is most interested in delivering these services to the customer without manual configuration. This requires automating the different tasks which create and manage the network service. The collection of these automated tasks is called orchestration.

Service providers use orchestration in traditional networks in order to automate tasks across technologies and organizations. This decreases the time between the request for a network service and the actual deployment. Network environments which are build using NFV introduce additional requirements which are unique to the virtualised environment [17].

The NFV Management and Orchestration (NFV MANO) should be able to handle multiple NFVi Point of Presences (PoP). Each NFVi PoP can have its own Virtual infrastructure manager (VIM) or a higher level of abstraction can present multiple NVFi PoPs.

Traditional management functions for network functions like fault management, configuration management, performance management and security management are still present. The decoupling of hardware and software introduces new management functions which are responsible for managing the VNFs lifecycle. These include the following:

- VNF creation
- VNF scaling
- VNF updating
- VNF termination

The main goal of a service provider is to provide network services to customers. A network service may include 1 or more VNFs and Physical NFs. The NFV orchestrator is responsible for managing these different NFs and their interaction. This interaction is described in a forwarding graph.

The network resources in a NFVi PoP are managed by the *Virtual infrastructure manager (VIM).* The VIM abstracts hypervisors and network controllers into a northbound interface. This northbound interface is used by the VNF manager and NFV orchestrator to interact with the available resources. The available resources are collected in an inventory where the VIM keeps track of the allocation by VMs. It also collects performance and fault information of hardware and software resources.

Each VNF is associated with a *VNF manager* which is responsible for its life cycle. A VNF manager can be assigned to a single or multiple VNFs. The deployment and behavior of each VNF is described in an inventory by a VNF descriptor. This VNF descriptor is used by the VNF manager to create and manage the VNF.

The *NFV orchestrator* has two responsibilities, resource orchestration and network service orchestration. The main focus of resource orchestration is to coordinate and manage resources from different VIMs. The orchestrator needs to determine the most optimal placement for each VNF. This placement is based on policies which describe certain rules. For example, an affinity rule can describe that multiple VNFs need to be hosted on the same physical device.

The network service orchestrator deploys and manages network services which are requested by the customer. A network service is described by one or more forwarding graphs which describe the network functions and how they are interconnected. When a customer makes a request for a service, the NFV orchestrator will launch or update the necessary VNFs and VNF managers.

2.4 Conclusion

The NFV architectural framework document describes the components which are needed to integrate a *virtual platform* for network services. They aim to identify components which can be accessed through reference points. This should make it possible to interconnect components from different vendors by their reference points which prevents vendor lock-in.

The NFVi infrastructure part contains the virtual infrastructure used to run VNFs. These include servers and switches as the hardware components. Chapter 3 discusses the issues with general purpose operating systems in the context of NFV. And chapter 6 explains the different design choices for creating data path between the networking hardware and the VNF.

Virtualization of network services introduces new management and orchestration challenges which are identified as NFV MANO. One part of the NFV MANO is a platform which abstracts the NFVi. This part is called the NFV virtual infrastructure manager. Chapter 4 and 5 describe an implementation of such a platform called Openstack and the way it is used for NFV purposes. This platform is also used in chapter 8 to run performance tests.

Chapter 3

Data-plane packet processing

3.1 Introduction

The transition from dedicated appliances into a virtualized environment requires commercial off-the-shelve (COTS) hardware. Instead of a single appliance for each network function, all network functions can be placed on any device. This raises the question, can we use COTS hardware for high performance packet processing?

COTS hardware is managed by an operating system (OS) which manages the different components. An OS allows multiple software applications to run on a single device. This chapter is concerned with the performance issues which are raised when running NFV applications on a general purpose OS.

3.2 Fast packet processing

Most common operating systems have a native networking stack for processing packets, this allows a general purpose computer to be used as a networking device. This network stack receives packets from the Network Interface Card (NIC), delivers it to the right user space applications and the other way around. The process of receiving a packet and delivering it to the right application requires an interrupt and often multiple copies of the packet. The flow of a packet in the Linux networking stack starts when the NIC receives the packet from the network [19], this flow is presented in figure 3.1. The packet is stored in a circular queue on the NIC until it gets copied to main memory, this copy happens via the Direct Memory Access (DMA) without intervention of the CPU. At the arrival of each packet an interrupt is generated to inform the operating system, this interrupt copies the packet from the DMA memory region to a packet buffer in kernel space memory, it decides which user space application it should go to and copies the packet from kernel space to user space.

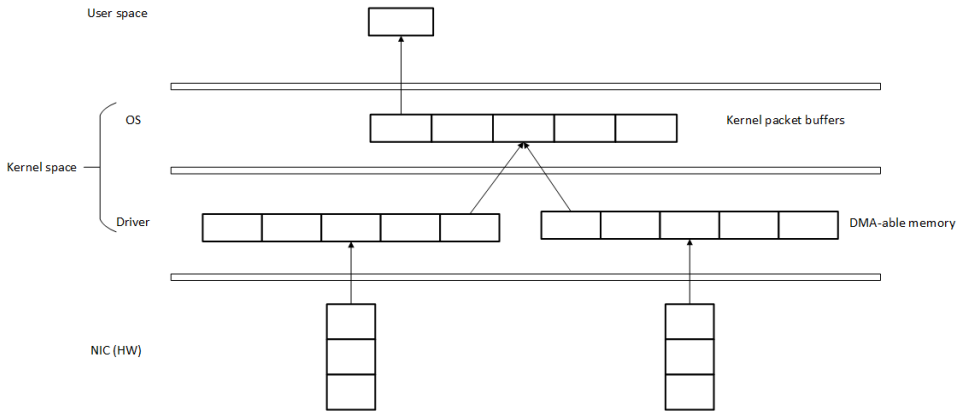

Figure 3.1: standard Linux network stack [19]. The image visualizes the path that a packet travels, and the necessary copies, to get in the application.

Network interfaces keep increasing their performance and network interfaces with 10Gbit/s or even 40Gbit/s rates are becoming more common in various high speed network environments [8, 9] The

standard mechanism of handling packets in an operating system is not sufficient for handling packets at these rates. A lot of CPU cycles are wasted on copying packets between memory regions with different privileges and the interrupt mechanism suffers from the interrupt livelock phenomenon [15]. In the latter case the operating system spends most of its time handling interrupts instead of processing the packet.

A number of techniques have been suggested to deal with these issues [2]. In order to reduce the number of copies made we can use the zero-copy technique [36]. This technique reduces the number of copies by creating regions memory which are visible for both the NIC and the user space application, instead of copying the entire packet a pointer to the packet is passed to the application.

A lot of packet processing applications don't need the entire networking stack, therefore it is often beneficial to bypass the kernel and pass raw packets to the user space application. This allows the programmer to create applications that can be optimized for specific scenarios.

In order to avoid generating interrupts packets for single packets we can transfer multiple packets as a group to the application, this is called batching. To eliminate the overhead of interrupts altogether, the application can use a polling mechanism, in this case the application runs in an infinite loop where it constantly checks if packets are available for processing. This should only be the case if the application needs to handle packets of a 10Gbit/s NIC at line-rate, otherwise it is possible that a lot of CPU are wasted just looking for packets.

In next sections we will look at some frameworks [37] that have adopted these techniques to achieve packet processing at line-rate.

3.2.1 NAPI

As mentioned before the standard Linux networking stack suffers from livelock under heavy load. In this case a polling mechanism could be used to solve this problem but it would waste a lot of CPU cycles under light loads. NAPI provides an alternative interface for networking devices in the Linux kernel using a hybrid approach to process packets [35]. Instead of generating an interrupt for each packet the NIC will generate a interrupt for the first packet in a batch. Next the NIC will periodically poll for remaining packets, if there are no packets left in the queue it will switch back to generating an interrupt. This reduces the overhead created by using the interrupt mechanism while avoiding wasted CPU cycles under light loads.

When the process starts the NIC is in a state where the polling mode is turned off and interrupts are enabled [34]. The first packet that arrives will generate an interrupt, this interrupt will move the NIC to a new state where interrupts are disabled and polling is turned on. The packets that arrive in the polling state are received in a DMA enabled region. At this point it is possible that the CPU is busy handling other processes. When this is the case, the packets will stay in the DMA enabled region until they are processed.

When the CPU is scheduled handle the to the next packet in the ring it will be processed by the network stack the same way as it would have been handled by an interrupt. This is repeated for all the packets in the ring or until a certain amount of packets is handled. If all the packets are handled the NIC will move back to the state where polling is turned of and interrupts are enabled.

When the NIC is under heavy network load it will stay in polling mode and the CPU will be scheduled frequently to handle the buffered packets. If there is not a lot of traffic, the NIC will mostly stay in interrupt enabled mode. The main benefit of NAPI is a significant improvement of CPU utilization.

NAPI is part of the Linux kernel and can be turned on if necessary. It acts as an addition to the Linux network stack and thus allowing Linux to be used in more demanding network environments as a packet processing framework.

3.2.2 PF_Ring

PF_Ring improves packet processing by avoiding packet copies between kernel space and user space. This is achieved by creating a memory mapped packet buffer which is shared by the kernel and user space applications. To achieve this goal, the NIC is bound to a new type of socket which has a circular buffer where the packets are copied on arrival. This buffer is allocated at the creation of the socket, this avoids memory allocation costs at the arrival of packets. When a packet arrives it is copied to the circular using a DMA mechanism, at this point all the work is done by the kernel and the NIC. To copy packets from the NIC to the buffer it is possible to use NAPI as shown in figure 3.2.

When a user space application wants to read these packets it maps to the circular buffer using a mmap() system call which returns a pointer. After the application reads the packet it updates the

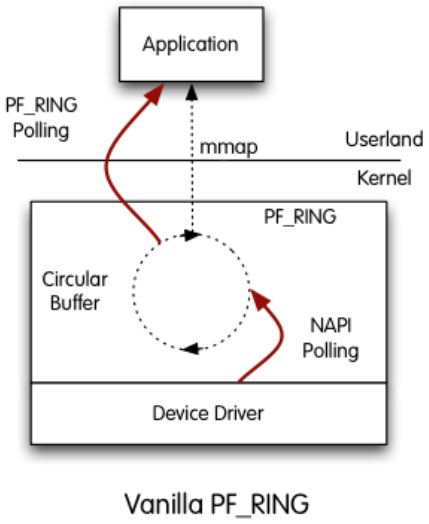

Figure 3.2: PF_Ring socket using NAPI (Source: http://www.ntop.org/products/packet-capture/pf_ring/). In this case the stack takes advantage of both NAP and PF_ring features to boost the performance.

pointer to release the packets location in the buffer.

Performance can be improved even more by using PF_Ring DNA (Direct NIC Access) [32]. In this case the user space application is mapped to memory directly on the NIC. Unlike NAPI approach there is no CPU utilization necessary when packets arrive. This approach has two important disadvantages, it he applications misbehaves it can cause a kernel crash and the NIC can only be used by one application at a time.

Until now we have only discussed the flow of packets on the receiving side. The sending side follows a similar approach where packets are store in the memory mapped area by the user space application. By updating the necessary pointer to the the circular buffer the application informs the NIC that packets can be sent.

3.2.3 Netmap

Most frameworks which implement these performance acceleration techniques are not easily integrated in existing applications. The Netmap framework tries to integrate more easily with existing applications by using existing I/O system calls like select() or poll(). The makes it possible to speedup network applications with a limited amount of new code.

Netmap uses several techniques to achieve better performance [30, 31]. It uses zero-copy techniques by granting the application protected access to the packet buffers in kernel space. These packet buffers are preallocated in order to avoid the cost of allocations and deallocations. It also makes use of certain hardware features such as multiple queues on a NIC.

The meta-data representation of packets is compact and hides device-specific features, therefore Netmap is independent of specific devices and hardware. This representation also supports processing a large number of packets at a single system call to minimize its cost.

Netmap allows to switch the NIC between host mode and netmap mode. In netmap mode the NIC is partially disconnected from the host stack and can be controlled by certain data-structures in shared memory visible by the user space application. The communication between the application and the NIC is validated to prevent a kernel crash even if the application misbehaves.

The switching of the NIC to netmap mode starts by passing a file descriptor to the /dev/netmap device and the interface name to the ioctl() system call. At this moment the NIC is partially disconnected from the host. Next the kernels memory area can be mapped to the application using a mmap() call. At this point the application can start processing packets by making a select() or poll() call.

When the NIC is in netmap mode it is still visible to the host. This allows the application to forward packets to the host network stack.

3.2.4 PacketShader

A very different approach of improving performance is to process packets parallel on multiple cores. Graphical Processing Units (GPUs) often have hundreds or even thousands of cores originally used for graphical applications. Today, GPUs are also used for more general applications which benefit from the highly parallel computing capabilities. PacketShader takes advantage of GPUs to create a user space framework for fast packet processing [20].

Just as other frameworks PacketShader uses a modified technique for handling the reception and transmission of packets via the NIC. To avoid per-packet memory allocations it preallocates a huge packet buffer which holds fixed sized cells which can be reused. The transmission to and from the NIC is done by DMA. The framework uses batch processing in different stages of the transmission. By mapping the huge packet buffer to I/O memory it is possible for the NIC to transfer multiple packets at once.

Packets are also processed in batches at between kernel- and user space, this introduces a copy for better abstraction. To avoid the live-lock from interrupts PacketShader has implemented a mechanism which switches between interrupts and polling. This is similar to NAPI except that NAPI only works in kernel space memory.

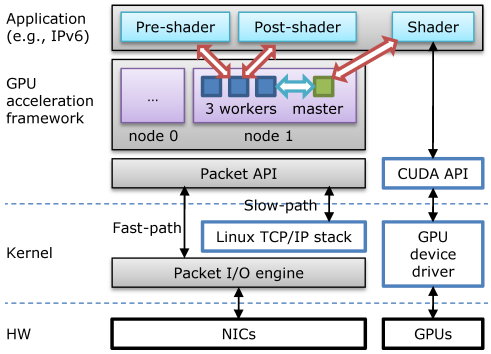

Figure 3.3: PacketShader architecture [20]. The worker and master threads run on the CPU to run administrative tasks, the master thread communicates with the GPU which executes the necessary work.

After the packets are transferred to the application into user space it is possible to use the GPU as a accelerator. Figure 3.3 shows the basic architecture. The framework uses CUDA as the API for programming the GPU. The CPU runs multiple threads to make optimal use of multi-core systems, these threads are divided in one master thread and multiple worker threads. The master thread is the only thread that communicates with the GPU since the performance of CUDA degrades when multiple threads access the same GPU. The worker threads are responsible for packet I/O.

The workflow is divided in three steps, pre-shading, shading and post-shading. The pre- and post-shading run on the worker threads, the shading step is handled by the master thread which communicates with the GPU (Figure 3.4). Each worker thread fetches packets from receive queues. Malformed packets are dropped and the normal packets are prepared for GPU processing. These packets are then transferred to GPU memory by the master thread. After they are done processing by the GPU the master threads transfers the packets back to host memory into the output queue of the worker thread. This worker thread prepares the packets for transmission to the NIC.

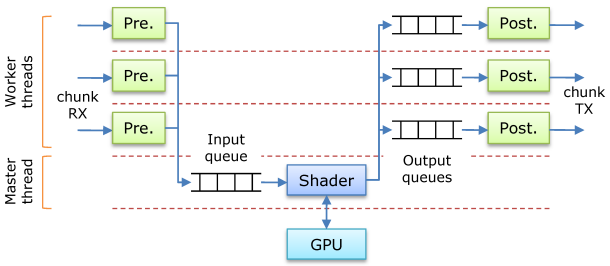

Figure 3.4: PacketShader workflow [20]. There are multiple worker threads to pre- and post-process the packets. The master delegates the actual work on the GPU.

3.2.5 Data Plane Development Kit

DPDK provides a set of libraries used to create data plane applications for high performance packet processing [6]. The set of libraries are available through an Environment Abstraction Layer (EAL), which allows access to hardware related resources.

The framework can use a run-to-completion model or a pipeline model for processing packets. The run-to-completion model uses processing units which execute the same code on each packet. The pipeline model runs a different part of the functionality in each processing unit, each packet is processed by all the processing units which are interconnected using lockless rings. Each model has its own advantages and disadvantages, both models can be used in the same application as an hybrid approach.

Maximum performance is achieved by using optimization techniques which take the hardware into account. Each CPU on a system with multiple CPU sockets has a different memory access time depending on the memory location. The framework is aware of this and allocates memory close to each CPU, this is called NUMA-awareness [22]. An alignment helper ensures that objects are evenly distributed across memory channels to balance memory bandwidth utilization.

To avoid allocation- and deallocation costs for holding packets, memory is allocated by the Memory Pool Manager. This memory is allocated using huge pages to avoid Translation Lookaside Buffer (TLB) misses .

The framework uses a Poll Mode Driver to avoid the overhead of interrupts. Packets are transferred to the configured memory locations using Direct Memory Access, these memory locations are visible by the user space application. After that the user space application receives the packets as a group and they are moved around using pointers to their location.

Communication with the NIC and between different processing units in a pipeline is done by using lockless rings. Traditional multi-core applications use locks to avoid race conditions when accessing shared objects. The use of locks introduces overhead because only one process can access a shared object at a time. To avoid using locks, shared objects can be passed by updating pointers using atomic operations.

DPDK uses an open-source licensing model which allows anyone to use it. In this thesis DPDK will be used to create Virtual Network Functions (VNFs) which will be used to test the performance of the NFV Infrastructure.

3.2.6 Comparison

Table 3.1 shows a comparison between the features supported by the different frameworks.

	NAPI	PF_Ring	Netmap	PacketShader	DPDK
Zero copy		x	x		x
Polling	x	x	x	x	x
Batch processing	x	x	x	x	x
Preallocated memory		x	x	x	x
NUMA awareness				x	x
Huge pages					x
GPU processing				x	

Table 3.1: Comparison between the features supported by different frameworks.

The main goal of these frameworks is to provide an interface to the NIC which allows packet processing at 10Gbit/s or more. Although their main goal is the same, they all take a different approach. NAPI and PF_Ring provide a solution to a certain cause of the performance bottleneck. NAPI is focused on the overhead caused by interrupts generated for each packet and PF_Ring decreases the amount of packet copies made. Both technologies can be combined or used separately.

Netmap tries to provide an alternative stable interface which is easy to adopt in existing application. Data Plane Development Kit on the other hand takes a more aggressive approach where each hardware limitation is optimized as best as possible. This requires a lot more consideration when writing an application and is more difficult to integrate in existing solutions.

To accelerate the processing step frameworks can benefit from multi-core systems. Most frameworks are able to use multiple CPU cores, but PacketShader goes a step further by using the highly parallel computing capabilities of GPUs.

3.3 Conclusion

Traditional network environments are build using dedicated appliances. These appliances lack flexibility and configuration is often vendor-specific. An alternative to these appliances is the use of network functions in software.

We described how general purpose computers can be used to process packets in high-performance network environments. This required acceleration techniques which are applied in a number of frameworks. These frameworks are designed to handle traffic from 10Gbit/s NICs. Today, NICs with multiple 10Gbit/s or 40Gbit/s ports are becoming more common [9, 8]. For example, the Intel XL710-QDA2 has 2 40Gbit/s ports.

In the context of the internship with Intel, *DPDK* is used to run the tests in this thesis. The framework originates as a proprietary product of Intel, but has been made open source. Because it heavily benefits from hardware features to achieve maximal performance it is the most suited framework to use on Intel hardware. In this thesis it is used in 2 ways:

In chapter 6 we explain how DPDK is used on the host OS to improve the performance of Open vSwitch. This is concerned with the data transfer between the NIC and the VM and aims to achieve a higher throughput and a lower latency. Using DPDK is not the default configuration of Open vSwitch, instead it uses the OS networking stack which suffers from the performance issues described in this chapter.

The DPDK frameworks is also used in chapter 7 to implement the applications that represent the VNFs. These applications run in the guest OS.

Chapter 4

Openstack

4.1 Introduction

Data centers often contain hundreds or thousands of servers to form a large pool of computing, networking and storage hardware. By using virtualization techniques we can share these hardware resources among many different users without any interference from each other. To make optimal use of the hardware it is necessary that we see all these individual devices as one big pool of resources. This requires a platform that controls all these devices. If a user wants to launch a new virtual machine, he doesn't want to be concerned about which machine it will be hosted on.

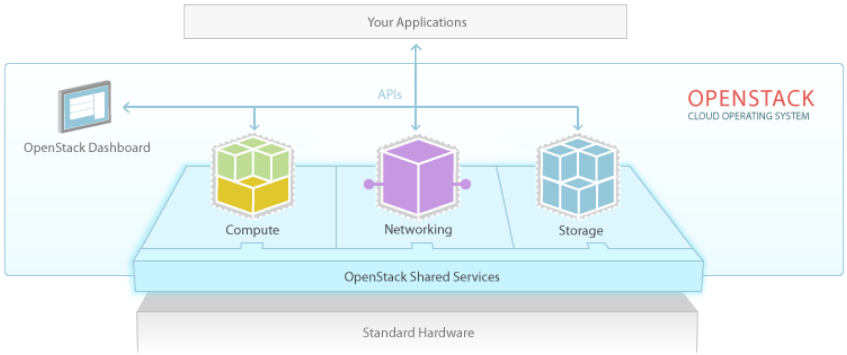

Figure 4.1: High level overview of the functionality provided by Openstack. (Source: https://www.openstack.org/software/)

Openstack provides a platform where multiple devices are managed using a web based dashboard or a REST API. This allows the user to launch a virtual machine, allocate storage and create virtual networks by just making a few clicks. The Openstack platform consists of multiple individual projects, also called services, which provide a specific functionality. The following are the core services:

Nova: is responsible for launching and managing virtual machines.

Glance: stores images containing operating systems which can be used by other services. An operating system is normally stored on a disk volume which is part of a physical storage device. It is also possible to store a disk volume as single file, these files are called (disk) images.

Neutron: provides network components for other services, Nova depends on this service to provide connectivity between virtual machines.

Keystone: provides authentication for other services.

Swift: is used to store objects such as documents, images, and so on.

Cinder: provides block storage which can be attached to virtual machines.

It is not necessary to use all available services although some depend on other services to function. For example, Nova depends on Glance to provide images for the creation of virtual machines. Next to

these core services there are also optional services[1] available which can be used to provide additional functionality.

Each device in an Openstack network is referred to as a node. The administrator can assign different roles to different nodes. This role defines the functionality that is provided by that device. The most common types of nodes are the following:

Controller node: this node is used for managing all other nodes in the deployment and it exposes an interface to the users on which they can manage their virtual machines.

Compute node: hosts the virtual machines.

Networking node: acts as a gateway between the virtual instances and the internet and runs network functions like routers, DHCP servers, load balancers, ...

Storage node: provides storage that can be attached to the virtual instances.

The nodes in a Openstack deployment are connected to multiple networks. One of these networks is used to send messages, copy virtual machines, and so on. This network is called the management network and is only used for administrative purposes, virtual machines will never send traffic on this network. Other networks are used to provided connectivity to the internet and between the VMs and so on.

The administrator can limit the amount of virtual machines that can be used by users. This is done by the creation of tenants, also called projects. Each tenant has a certain limit of virtual resources, for example a maximum number of CPUs, RAM, and so on. Multiple users can be part a tenant and a user can be part of multiple tenants.

Chapter 5 describes the potential role of Openstack in the NFV architectural framework. It also describes in how it should be configured to be able handle NFV workloads.

4.2 Nova

To launch a virtual machine, Openstack needs to choose the most suited device and control the hypervisor on that device, this is handled by the service called Nova. Nova supports multiple hypervisors:

KVM: Kernel-based Virtual Machine is a hypervisor that is part of the Linux kernel and makes use of virtualization extensions like Intel VT or AMD-V for near-native performance.

Xen: is a popular open source hypervisor which supports different types of virtualization. Just like KVM it can provide virtualization which uses hardware features to improve performance. It is also able to use a different approach to virtualization where the guest operating system is modified to optimize the communication with the hypervisor. This type of virtualization is called paravirtualization.

VMware vSphere: collection of hardware virtualization products from VMWare including a hypervisor.

Hyper-V: hypervisor developed by Microsoft which uses hardware virtualization support.

LXC: Linux Containers are lightweight virtual machines. Instead of virtualizing an entire operating system we can use isolation techniques to separate part of the operating system. In this case the user sees this isolated part as a single operating system or a virtual instance.

QEMU: Quick EMUlator uses emulation techniques to virtualize hardware. It has a significant lower performance than hardware-assisted hypervisors and is only used for development purpose.

UML: User Mode Linux allows guest operating systems to run as a user space application. Just as QEMU it has a low performance and is mostly used for development purpose.

To be able to boot a virtual machine Nova needs a disk image which contains the OS. These disk images are provided by *Glance*. Whenever Nova wants to boot a VM it requests the correct image. The VMs can be customized at creation using the cloud-init program. This program should already be installed on the image.

[1] `http://www.openstack.org/software/`

The cloud-init program is used by Nova for various reasons. For example, Nova uses it to insert a RSA public key into the VM which allows the user make an SSH connection to the VM without using a password. It is also possible to run a custom script at creation. This can be used to install certain services when the VM is created. For example the installation of an Apache web server.

When a VM is created the user needs to specify the properties it needs. For example the number of CPUs, the amount of RAM and the disk space. These properties are defined by *Nova flavors*. For each VM the user wants to start he has to provide a flavor. Flavors can also be used to request certain features from the host. For example certain capabilities from the CPU.

When a user launches a VM, Nova needs to decide which is the most suitable physical device at that moment. This decision is made by the Nova scheduler. By default the filter scheduler is used which takes 2 steps to launch a virtual machine. First it filters all the hosts that do not meet the criteria, then it will decide the most appropriate candidate using weights (Figure 4.2).

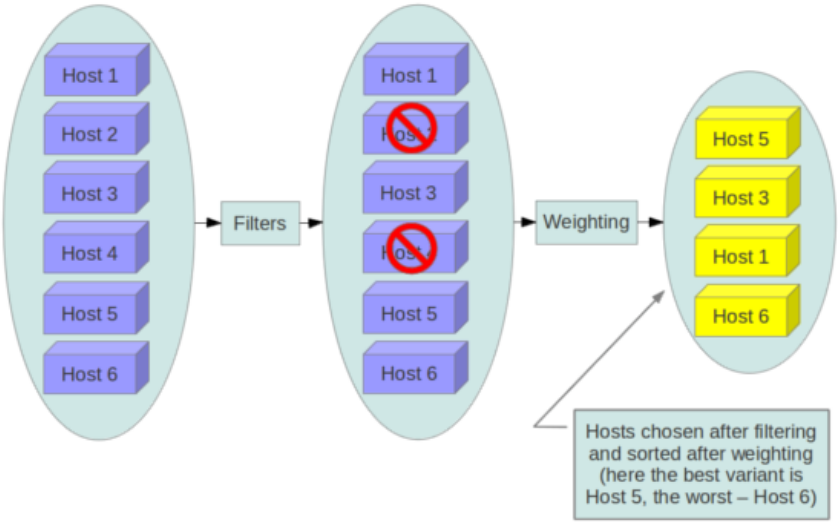

Figure 4.2: Filtering and weighting process when launching a virtual machine using the filter scheduler. (Source: http://docs.openstack.org/kilo/config-reference/content/section_compute-scheduler.html)

The administrator chooses the appropriate filters for his deployment. By default the following are filters are used:

RetryFilter: if the scheduler sends a request to a host and the host fails to respond, this filter will prevent the scheduler from retrying a request to this host.

AvailabilityZoneFilter: it is possible to group physical hosts in availability zones, this filter filters out all the hosts that do not belong to the requested availability zone.

RamFilter: this filter prevents the scheduler from launching virtual machines on hosts that already have all their physical RAM consumed.

ComputeFilter: selects all hosts that are enabled and available.

ComputeCapabilitiesFilter: if the hosts makes a request for certain CPU properties, this filter will select all hosts that satisfy these properties.

ImagePropertiesFilter: cloud images can have properties defined like architecture, hypervisor type, hypervisor version, and virtual machine mode. This filter passes all the hosts that can support these properties.

ServerGroupAffinityFilter: it is possible to group virtual instances together as a server group. This filter will try to schedule the virtual machines that belong to the same group on the same host.

ServerGroupAntiAffinityFilter: in some cases it is preferable that virtual machines are launched on different hosts. This filter will select only hosts that do not run an instance belonging to the same server group.

These are just the default filters that are enabled in Openstack, there are more filters available and it is possible to create your own filters. All hosts that pass these filters are weighted to be able to select the most suitable host. Every time an instances is launched on a host it will affect its weight.

4.3 Neutron

Openstack allows user to create a logical network topology on top of the physical infrastructure. To create this topology Neutron allows you to create virtual networks and network functions. Neutron can be used to create 2 types of networks: provider networks and tenant networks (Figure 4.3). Provider networks map to the physical network in the data center, this allows Openstack objects to communicate with the physical devices like routers to provide internet access. Tenant networks allow network connectivity between virtual machines and network functions that belong to the same virtual networks. Openstack supports a number of network virtualization techniques to create virtual networks, In this way, the networks belonging to different tenants are completely isolated.

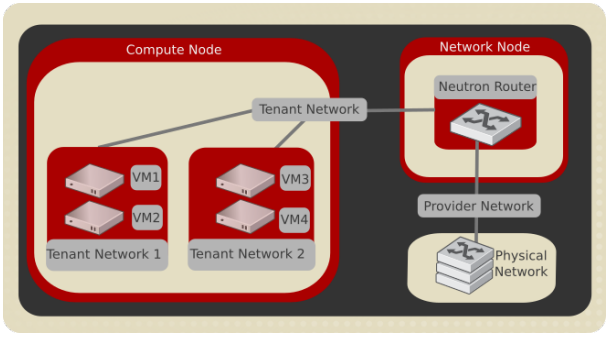

Figure 4.3: Different network types in an Openstack deployment. The tenant networks provide connectivity between VMs and virtual routers. The provider network is connected to internet. (Source: http://docs.openstack.org/liberty/networking-guide/intro-os-networking-overview.html)

Neutron provides the following network types and virtualization techniques:

Flat: when using a flat network type there is no network virtualization used, this is often the case for the provider network which is connected to the internet.

VLAN: when using VLANs, each virtual network is associated with a VLAN ID. When this networking type used, each node is connected with the switch using a VLAN trunk connection where all traffic is tagged with a 802.1Q header. In this way multiple virtual networks can be created using a single connection with a node. It also allows virtual machines to communicate with physical devices that are connected on a port belonging to the same VLAN.

GRE and VXLAN: GRE and VXLAN are encapsulation protocols that encapsulate each packet inside a new UDP packet. The connection between 2 instances is called a network tunnel. This requires a router that is also connected with a provider network in order to communicate with physical devices.

Openstack has many different possible ways of deployment. The most basic scenario uses a network node to provide connectivity to the external network for the virtual machines. This external network can only be configured by the network administrator. Users are able to create tenant networks to provide connectivity between virtual machines, routers and all kind of network services. Each virtual machine is connected to a software switch. The default configuration uses Open vSwitch[2]. Users are able to connect these virtual machines on a tenant network using private IP addresses. Neutron refers to these private IP addresses as fixed addresses.

[2] `http://openvswitch.org/`

To provide internet connectivity on the virtual machines administrators can create virtual routers on the network node. These routers use SNAT to provide internet connectivity using the routers external IP address. They also use DNAT to be able to bind external IP address to virtual instances. Data center providers usually have a range of public IP addresses available, these IP addresses can be used by the users to provide external connectivity to virtual instances. These external IP addresses are referred to as floating IP addresses. Besides connectivity to the external network, these routers also forward traffic between different tenant networks. Neutron also provides other network services like DHCP, firewalls, load balancers, ...

Figure 4.4 shows a scenario where a virtual machine is connected to the internet. In this scenario it is possible to create tenant networks using one of the 2 available virtualization techniques, either VLANs or VXLAN tunnels. Each compute node has 2 NICs which are used for traffic on the tenant networks. The network node has 3 NICs, 2 for the tenant networks and 1 for the external network. The tenant networks are connected to the external network using a virtual router in the network node.

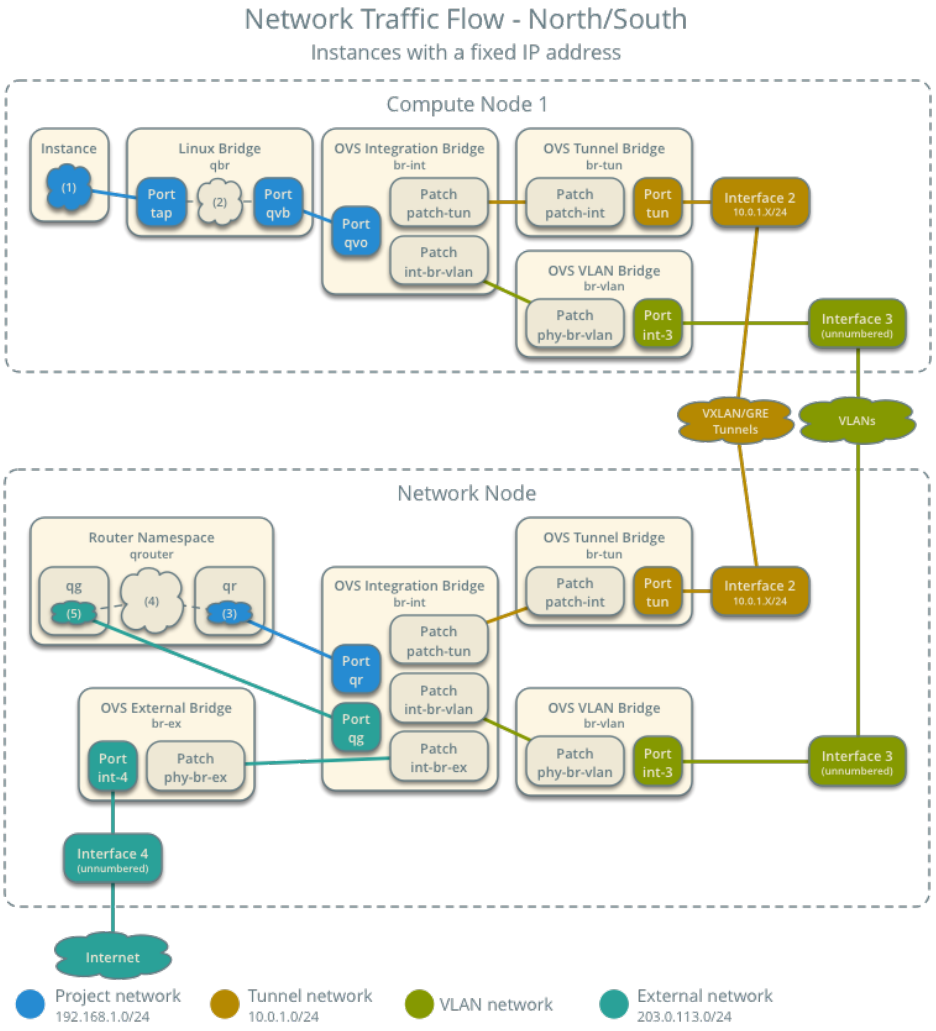

Figure 4.4: Traffic flow from a virtual machine (instance) to the internet. (Source: http://docs.openstack.org/liberty/networking-guide/scenario-classic-ovs.html)

All the traffic from the virtual instance to the internet follows these steps on the compute node:

1. the virtual machine forwards the traffic to the Linux bridge qbr. This bridge is connected to a firewall which filters all the traffic that is sent and received by the virtual machine.

2. the bridge qbr forwards the traffic to the OVS Integration bridge br-int, this bridge knows to which tenant network the port is connected to. Each virtual network is associated with a internal VLAN

tag, depending on the type of network, the traffic is either forwarded to the Tunnel bridge or the VLAN bridge.

3. For the tunnel network:
   (a) The integration bridge forwards the traffic to the Tunnel bridge.
   (b) The traffic is encapsulated in a UPD packet and tagged with the ID of the tenant network.
   (c) The encapsulated traffic is sent into the physical network to the network node.

   For the VLAN network:
   (a) The integration bridge forwards the traffic to the VLAN bridge.
   (b) The traffic is tagged with ID of the tenant network using a 802.1Q header.
   (c) The tagged traffic is sent into the physical network to the network node. Using this type of network each node needs to be connected to the same sub-net and the physical infrastructure needs to be able to handle VLAN traffic.

When the traffic arrives on the network node, these steps are followed:

1. For the tunnel network:
   (a) The encapsulated traffic arrives on the Tunnel bridge.
   (b) The Tunnel bridge replaces the VXLAN tag with an internal VLAN tag.
   (c) The traffic is forwarded to the integration bridge.

   For the VLAN network:
   (a) The tagged traffic arrives on the VLAN bridge.
   (b) The VLAN tag is replaced with the internal VLAN tag.
   (c) The traffic is forwarded to the internal bridge.
2. The internal bridge forwards the traffic to the Router Namespace, this represents a virtual router using the Linux network stack. The traffic is translated to the external IP address using SNAT.
3. The traffic is forwarded tot the integration bridge which forwards it further to the External bridge where it eventually is forwarded to the internet.

This is the most basic deployment scenario which can be used to provide network connectivity. The major downside of this scenario is that all the traffic that is sent to a different network has to pass the network node, this limits the scalability of the network and it creates a single point of failure. To overcome these issues, more complex deployment scenarios are possible.

Distribute Virtual Routing is an extension of this basic scenario where part of the router functionality will be provided directly on the compute nodes. This requires an extra NIC port on each compute node. In this scenario, a big part of the traffic is forwarded directly to the correct compute node without passing the network node.

Another possible extension of the basic scenario is a scenario using the *Neutron Virtual Router Redundancy Protocol*. The major goal of this scenario is to distribute network functionality on multiple network nodes to avoid a single point of failure.

It is also possible to use the physical infrastructure for all the network functionality including tenant networks. This scenario maps the virtual networks to the physical networks which are referred to as provider networks, In this case the external connectivity is provided by a physical router.

4.4 Other services

Nova and Neutron are the most important Openstack services which we will be using in this thesis. Openstack has a lot of services which provide a certain functionality. In the following sections we will describe some services which we will use. Besides these there are still much more services.

Openstack provides a web interface called *Horizon* in order to configure the platform and for users to manage their available resources. Figure 4.5 shows the overview page where the usage the resources is

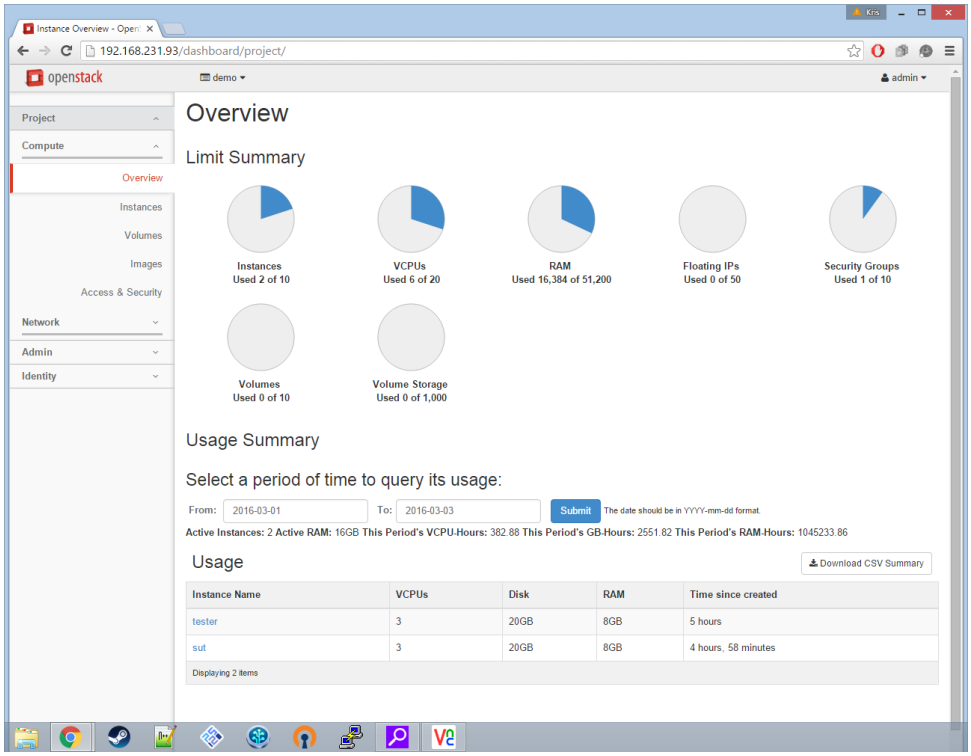

Figure 4.5: Example of a web interface provided by Horizon. This view shows the current number of virtual resources allocated by a tenant.

shown. This interface allows users to a launch new virtual instances, create virtual networks and many more.

Openstack uses its own authentication system. This system is provided by the Openstack service called *Keystone*. The administrator can add users to the Keystone database. Whenever a user wants to execute a request it has to provide its credentials.

A user needs to be part of tenant if he wants to run VMs. A tenant, also called project, is a group which holds a certain quota for launching VMs or creating virtual networks. For example, the administrator can limit the maximum number of CPUs that can be used to launch VMs.

If a user wants to launch a VM there are often additional task which need to be executed before the VM has the desired functionality. These tasks can be executed manually but that would be tedious if the user wants to create a complex cloud application.

Openstack provides a service called *Heat* which allows the user to describe the different components and their relation in a template file. This template file holds virtual networks, the different VMs and their relation. Heat has its own template format but is also supports the AWS CloudFormation template format[3]. Heat is responsible for the entire life-cycle of the cloud application. For example, after the application is launched it can be updated by providing an updated template file. It can also integrate with telemetry services which allow autoscaling.

4.5 Openstack deployment

An Openstack deployment can use thousands of servers to host VMs. One way of installing the platform on all the nodes is to install the operating systems manually. Then each node can be prepared by installing the necessary packages and prepare a database. Finally the Openstack packages can be installed and the platform can be configured.

It would be very tedious to perform all these steps manually on a lot of nodes. Therefore it is recommended to use a deployment tool which executes these steps automatically on all the machines. There are multiple deployment tools available, these include Fuel, Compass, TripleO, etc. Each tool has its own features and no tool can be called the best tool. The difference between these tools is mostly the

[3] `https://aws.amazon.com/cloudformation/`

OS used on the nodes and the deployment automation method. The Openstack platform itself is almost identical. They all use the same Openstack code base but some deployment tools add extra features.

Openstack Fuel provides a system that allows administrators to install Openstack automatically on many machines. It uses Ubuntu as the OS on the nodes and puppet[4] as the deployment automation method. Fuel is provided as a Linux image which can be installed on a separate node. The Fuel node will act as a PXE server on a separate network which is connected to all the Openstack nodes (figure 4.6).

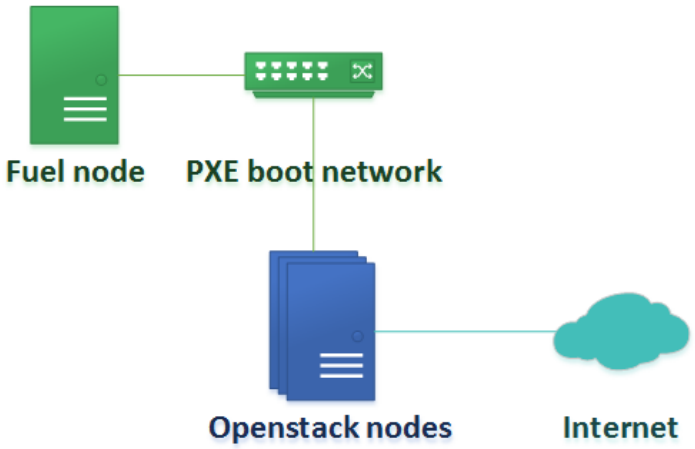

Figure 4.6: Network architecture using Fuel to deploy Openstack. The Fuel node is connected to all the other nodes through a PXE network. This network is used to deploy the OS and Openstack components. The Openstack nodes are also interconnected through other networks and with the internet.

When an Openstack node is assembled and connected to the PXE boot network it will boot an operating system provided by Fuel. When the node is booted it becomes available in a web interface which is accessible on the Fuel node. After that the administrator can assign a role to the new node and deploy an Openstack environment.

4.6 Conclusion

The Openstack platform provides an way to manage a lot of servers which can be used to deploy VMs and provide network services. It can therefore be classified as a Infrastructure as a Service cloud platform. The main components, called services in Openstack terminology, are Nova and Neutron. Nova deploys and manages the VMs. To do this it keeps a database with all the nodes and selects the most suitable when a VM needs to be launched.

Neutron provides network services like routing, firewalls, etc. This allows the users to provide network access between the VMs among each other and to the internet. Openstack contains other services which can be used. It is not necessary to use all services although a service can depend on another service.

To deploy Openstack it is recommended to use a deployment tool. This allows the installation of the platform in may servers in an automated way. The tool used in this thesis is called Fuel.

The way Openstack is explained it this chapter represents the configuration which is used in *normal* cloud applications. These include Web hosting, eCommerce, big data, etc. In the context of NFV there is a different requirement for the balance between storage, compute and networking. Storage is not discussed in this thesis, but both the compute part and the networking part require modification. These requirements are focused on data plane networking applications. Control plane applications can be applicable for the default configuration.

The platform allows various cloud application to run on the same environment. If we assume a public cloud model it is possible that different use-cases or even different companies share the same environment to run applications. If Openstack is used as part of the NFV architectural framework it will not be shared with other applications. In this case the platform should be interpreted as a specialized *NFV data center* which is solely used by telecommunications companies to host network services.

[4] `https://puppet.com/`

Chapter 5 explains how Openstack can be used as a part of the NFV architectural framework. In this way the underlying hardware specifics are abstracted for components that are concerned with deploying network services.

Chapter 5

Openstack as NFV virtual infrastructure manager

5.1 Introduction

NFV makes use of multiple virtualised hardware resources to run networking functions. Openstack can be used to manage these resources and fulfill the role of virtual infrastructure manager (Figure 5.1). At first sight it seems as simple as just taking Openstack and use it to launch VNFs. But the overhead that is involved when using virtualization has much more impact in a NFV context. Therefore it is necessary to tune certain aspects of the platform for better performance.

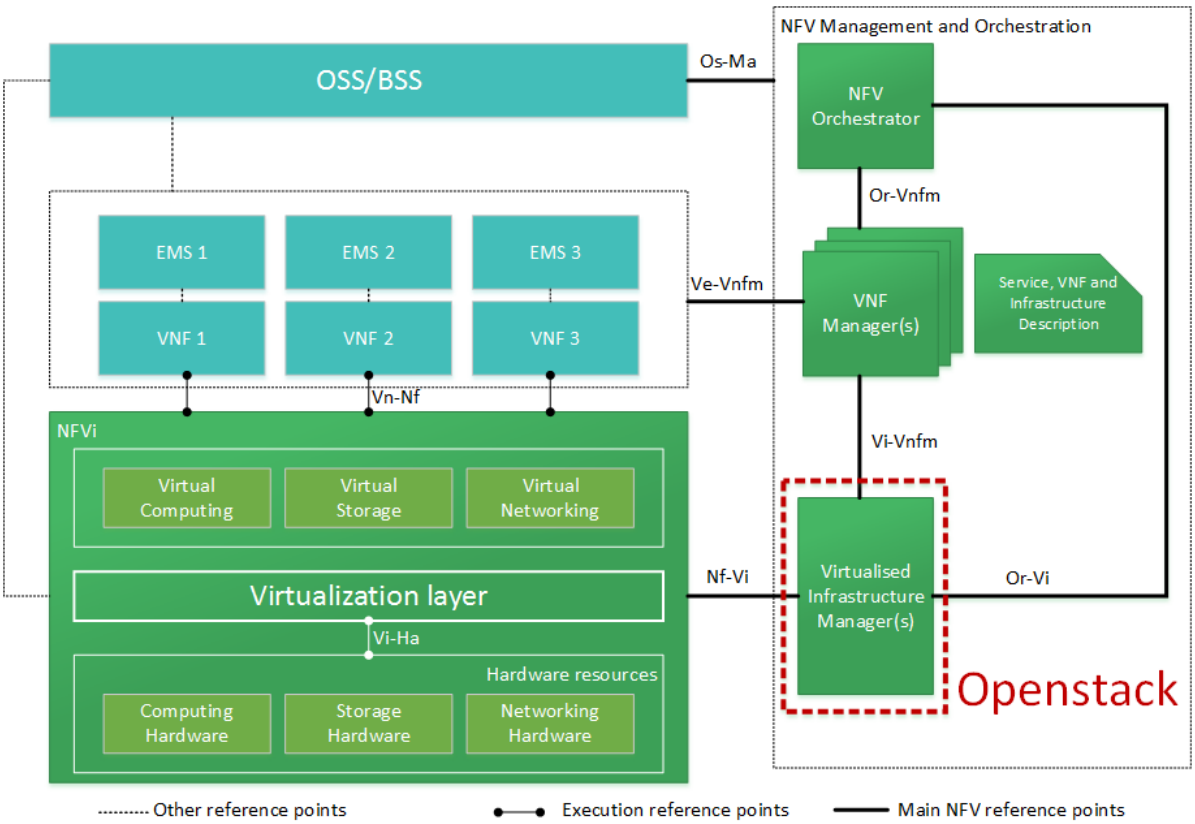

Figure 5.1: Role of Openstack in the NFV architectural framework. The platform provides an interface which can be used by the VNF manager and NFV orchestrator to run network services.

Since the loss of a single packet can have a significant impact in the delay of the upper lying protocols it is useful prevent packet loss at all time. When sending 64 bytes packets at line rate on a 10Gbit/s

NIC, each packet needs to be processed in 67.2 microseconds. Therefore even very small interruptions can cause a packet to be dropped.

When using general purpose hardware it becomes much more challenging to prevent packet loss. Most operating systems are designed to run multiple tasks and are therefore able to handle interrupts. It becomes even more complex when running the network functions in a VM. In this case both the host and the guest OS can get interrupts which are able to disrupt the packet processing application. These properties can mostly be overcome by configuration.

To minimize the chance for packet loss it is necessary to configure the system to handle interrupts on CPU cores which are not assigned for packet processing. It is also necessary to configure the hypervisor in a way that the VM can request certain performance optimization features.

5.2 Hypervisor configuration

By default Openstack uses QEMU as a hypervisor. To get the maximum performance the hypervisor needs to be able to use hardware virtualization extensions like Intel VT-x. Therefore it is recommended to use the KVM hypervisor. KVM is a fork of the QEMU project with the goal of getting the best possible performance. It requires a CPU with hardware virtualization extension, otherwise it falls back on QEMU. Both hypervisors are supported by Openstack and it is up to the administrator to choose the most suitable.

In the case of NFV it is highly recommended to use KVM. Since the cost of losing packets is very high it is necessary to expose certain performance optimization features to the VMs. These features can be requested in 2 ways, either by adding properties to the Nova flavor used to launch a VM or adding properties to the Glance image.

When a VM is launched, by default the CPU topology is not exposed to the VM. For example, when 6 cores are requested these are chosen at random in the host and these are presented to the VM as individual CPU sockets. It is recommended to expose the CPU topology in the VM. For example, a VM can be launched with the following properties: *"hw:cpu_sockets=2"*, *"hw:cpu_cores=4"* and *"hw:cpu_threads=2"*. This exposes a total of 16 cores which are exposed as 2 CPU sockets with 4 hyperthreaded cores to the VM. The VNF may use this information to improve performance[26].

When running a DPDK application in a VM it needs huge pages to achieve the best possible performance. The compute node can be configured to assign part of the memory as 1GB huge pages. When huge pages are available they can be requested by the user by setting the *"hw:mem_page_size": "1048576"* property.

By default a virtual CPU core is dynamically assigned to a physical CPU core. This means that the hypervisor can decide to move the virtual core to a different physical core at any time. The migration to a different core can not be predicted and causes packet loss. It is also possible to assign more virtual cores on a compute node than the number of physical cores. In this case physical cores are shared by virtual cores. By setting the *"hw:cpu_policy": "dedicated"* property the hypervisor will map each virtual core to a single physical core. This prevents the virtual core to be migrated and physical cores cannot be shared.

Some server boards have multiple CPU sockets. With most of these architectures PCI devices and RAM memory have a different access time from each CPU socket. Each PCI bus and memory channel is connected to one socket. When a CPU tries to access a device attached to a different socket it has to use a special connection between these sockets. One example of such a connection is Intel QuickPath Interconnect (QPI). This asymmetric design is called Non-uniform memory access (NUMA) where each socket with its devices is called a NUMA node. DPDK based applications can take this architecture into account to get optimal performance.

By default Openstack assigns any available core or memory region to a VM. The VM doesn't know to which NUMA node each CPU, memory region or PCI device belongs. The user can configure the NUMA location of each CPU core location and memory region. By setting the *"hw:numa_nodes": "2"* property a certain number of NUMA nodes can be requested. This requires a compute node with at least this amount of NUMA nodes. Next the amount of memory for each node can be configured by setting *"hw:numa_mem.0": "4096"* and *"hw:numa_mem.1": "2048"* properties, in this case 4GB of memory is requested for node 0 and 2GB of memory for node 1. Also the NUMA node for each virtual core can be configured by setting the *"hw:numa_cpus.0": "0-1"* and *"hw:numa_cpus.1": "2-5"* property, in this case the first 2 cores are requested for node 0 and the last 4 cores for node 1.

5.3 QEMU iothread interrupts

When a VM is launched using Openstack it creates a new QEMU process based on the given configuration. The architecture of this process uses a multi-threaded model to benefit from the multi-core CPU (figure 5.2). There is distinction between vcpu threads and iothreads. For each CPU core in the guest a vcpu thread is spawned, the host can schedule each thread on a different CPU core to maximize parallelism.

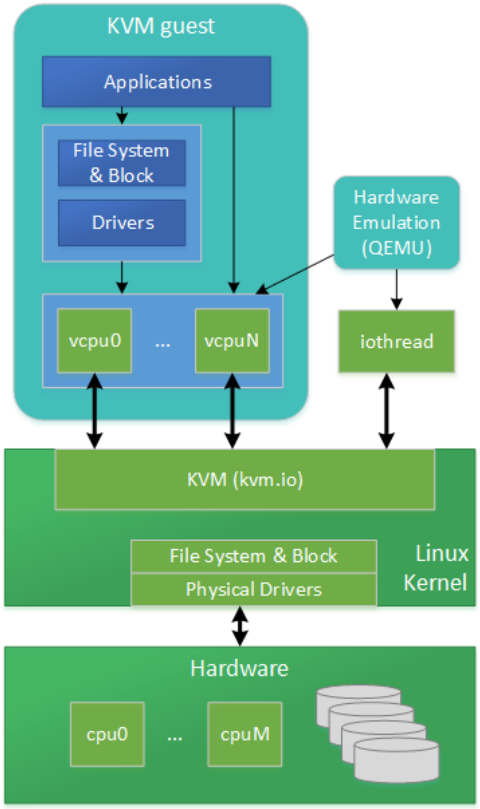

Figure 5.2: High level architecture of KVM. The hypervisor uses iothreads to emulate hardware. (Source: `http://events.linuxfoundation.org/sites/events/files/slides/CloudOpen2013_Khoa_Huynh_v3.pdf`)

Besides the vcpu threads there is also an iothread which handles an event loop[1]. This event loops waits for events generated by file descriptors and timers which expire. In some cases the events loops can spawn a worker thread for specific tasks.

If the VM is configured to use core pinning, then the vpcu threads and the iothreads have a different core affinity. Each vcpu thread is affinitized to a different physical CPU, but the iothreads and the workers are affinitized to use all the cores used by the vcpu threads. In other words the core mask of the iothread is the same as that of the core masks of all the vcpu threads combined.

If there is no activity on the iothread, this has no influence on the vcpu threads. But during the execution of the tests there was some activity on the iothreads which interrupted a vcpu thread for an exceptional long time. This behavior has not been observed when a VM was launched using the QEMU command line tool directly.

It is highly undesirable to have large interrupts in the guest OS. If the guest OS is processing packets of 64 bytes at 10Gbit/s it can theoretically process 14880952 packets per second. If the system is interrupted for 1 microsecond this would mean that it has to buffer at least 15 packets to prevent packet loss.

When running test cases which use a lot of cores for packet processing there was a lot of variation in the results after the tests where repeated. After further investigation it became clear that one core was interrupted for an unusual long amount of time. To prevent these interrupts from occurring on the vcpu threads we used the *taskset* command to affinitize the iothreads on different cores.

[1] http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html

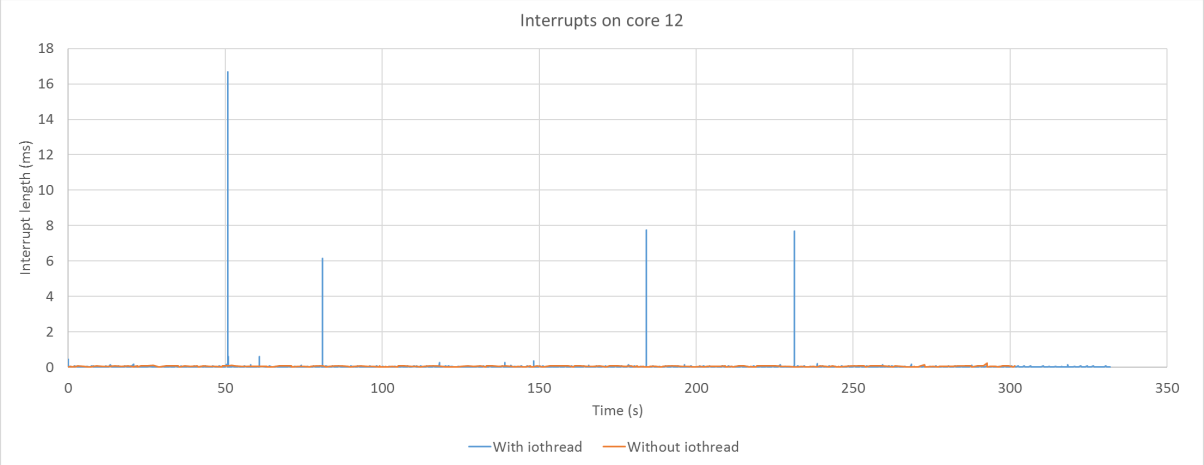

Figure 5.3: Duration of all the interrupts on core 12 in the guest OS. Core 12 has been identified as the core where the iothread is scheduled, this is an arbitrary number and can be different in another configuration.

Figure 5.3 shows the duration of all the interrupts for a period of more than 6 minutes when the iothreads are scheduled on the same care as vcpu 12 and when the iothreads are scheduled on a different core. With the iothread and vcpu thread on the same care we observe 4 interrupts with a length of more than 5 milliseconds.

	avg (s)	**largest (s)**
With iothread	45,7	16672
Without iothread	18,0	229

Table 5.1: Average and maximum length of the interrupts measured in figure 5.3.

Table 5.1 shows the average and maximum duration of the interrupts in figure 5.3. The largest interrupts is more than 16 milliseconds, when processing packets at 10Gbit/s this would require a buffer of 248095 packets to avoid packet loss. The buffers in a DPDK based application achieve better performance when they are small[25]. Therefore it is better to affinitize the iothreads to cores which are not used for packet processing.

Nova uses a tool called *libvirt* to launch and manage VMs. Libvirt translates the given configuration to a qemu command line which runs the VM. Although there is a configuration parameter available in libvirt to pin iothreads to certain CPUs on the host[2], this is not exposed to Nova. At the time this thesis is written there is no indication that it will be exposed in the near future. The reason is probably to keep generality since other hypervisors may not have similar configuration options.

5.4 Integration with OpenMANO

The goal of a platform like Openstack in a NFV context is to manage the resources and expose an API that hides low level details. The API that is exposed can be used by a NFV Orchestrator to create end-to-end network services. One example of such an NFV Orchestrator is OpenMANO.

OpenMANO is a Management and Orchestration implementation created by Telefonica which has been published under an open source license. The architecture is based on the NFV Architectural Framework[16] and consists of 3 main components:

- openvim
- openmano
- openmano-gui

[2]https://libvirt.org/formatdomain.html#elementsCPUTuning

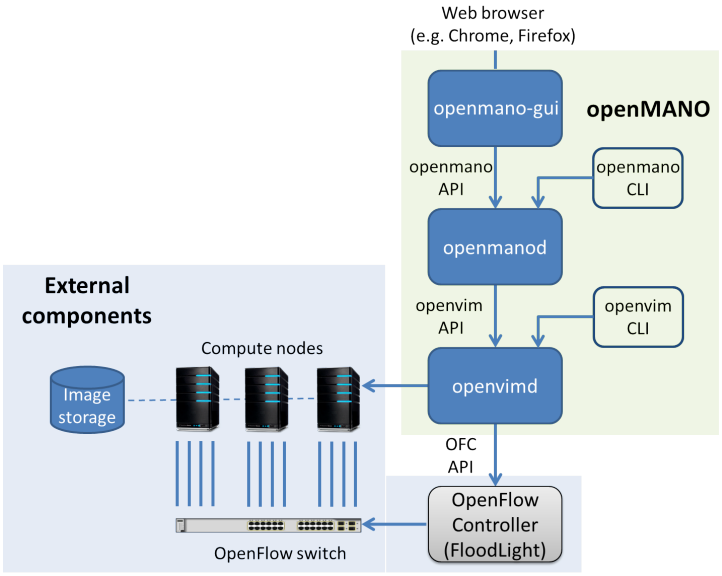

Figure 5.4: High level overview of the interaction between the different OpenMANO components. The OpenMANO components are based on the MANO part of the NFV reference architecture. The *External components* part represent the NFVi.(Source: `https://github.com/nfvlabs/openmano/blob/master/images/openmano-architecture.png`)

Figure 5.4 shows how these components interact with each other. It is not necessary to all the components, for example openmano-gui is not necessary and openvim can be replaced by Openstack. It is also possible to use multiple VIMs which can be used to create end-to-end network services that go through more than 1 VIM.

To use Openstack it is necessary to add a plugin which allows Neutron to interact with the OpenFlow switch which is used by the dataplane interfaces. It is also necessary to expose the hypervisor configuration features described in section 5.2.

The openmano component provides an API based on REST which can be used to interact. This API allows the deployment and management of VNF- and NS descriptors in the form of templates which can be used to instantiate network services. The openmano-gui component interacts with this API, but it is also possible to integrate into an OSS/BSS.

5.4.1 VNF Descriptors

A VNF descriptor describes a software based network function which can be deployed as part of a network service. The descriptor is stored by the openmano component in the form of a template written in YAML. A VNF descriptor contains the following components:

name Unique name of the VNF

description High level description

external-connections List of connections which can be used by other VNFs or networks.

internal-connections List of connections which can be used by VNF components to interconnect. This component is only required if the VNF consists of multiple VMs.

VNF Components List of VMs which are part of the VNF.

```
vnf:
    name:       dataplaneVNF1
    description: "Example of a dataplane VNF consisting of a single VM for data plane workloads with high I/O performance requirements: 14 HW threads, 16 GB hugepages and 4 10G interfaces"
    external-connections:
    -   name:              mgmt
        type:              mgmt      # "mgmt"(autoconnect to management net)
        VNFC:              dataplaneVNF1-VM
        local_iface_name:  eth0
        description:       Management interface for general use
    -   name:              xe0
        type:              data
        VNFC:              dataplaneVNF1-VM
        local_iface_name:  xe0
        description:       Dataplane interface 1
    -   name:              xe1
        type:              data
        VNFC:              dataplaneVNF1-VM
        local_iface_name:  xe1
        description:       Dataplane interface 2
    -   name:              xe2
        type:              data
        VNFC:              dataplaneVNF1-VM
        local_iface_name:  xe2
        description:       Dataplane interface 3
    -   name:              xe3
        type:              data
        VNFC:              dataplaneVNF1-VM
        local_iface_name:  xe3
        description:       Dataplane interface 4
    VNFC:
    -   name:        dataplaneVNF1-VM
        description: "Dataplane VM with high I/O performance requirements: 14 HW threads, 16 GB hugepages and 4 10G interfaces"
        #Copy the image to a compute path and edit this path
        VNFC image:  /path/to/imagefolder/dataplaneVNF1.qcow2
        numas:
        -   paired-threads: 7          # "cores", "paired-threads", "threads"
            paired-threads-id: [ [0,1], [2,3], [4,5], [6,7], [8,9], [10,11], [12, 13] ]
            memory: 16                 # GBytes
            interfaces:
            -   name:      xe0
                vpci:      "0000:00:11.0"
                dedicated: "yes"         # "yes"(passthrough), "no"(sriov with vlan tags), "yes:sriov"(sriovi, but exclusive and without vlan tag)
                bandwidth: 10 Gbps
            -   name:      xe1
                vpci:      "0000:00:12.0"
                dedicated: "yes"
                bandwidth: 10 Gbps
            -   name:      xe2
                vpci:      "0000:00:13.0"
                dedicated: "yes"
                bandwidth: 10 Gbps
            -   name:      xe3
                vpci:      "0000:00:14.0"
                dedicated: "yes"
                bandwidth: 10 Gbps
        bridge-ifaces:
        -   name:      eth0
            vpci:      "0000:00:09.0"
            bandwidth: 1 Mbps          # Optional, informative only
```

Figure 5.5: Example of a VNF descriptor in openmano. The descriptor describes the connections and hardware requirements for each VNF component.

Figure 5.5 shows an example of a VNF with 1 VM, 1 management interface and 4 data plane interfaces. This VNF is very similar to the VNF used to compare the data path in this thesis. The VNFC section shows a single VM which is used to process packets. This section contains a list of NUMA nodes and for each NUMA node the number of cores and the amount of memory which is requested. For this descriptor it is necessary to have compute nodes available which have exposed the hypervisor configuration features described in section 5.2.

The connections to this VNF are visible in figure 5.6. The 4 connections named xe0-3 are used for packet processing. The management interface name eth0 can be connected to a VNF manager which handles FCAPS functionality.

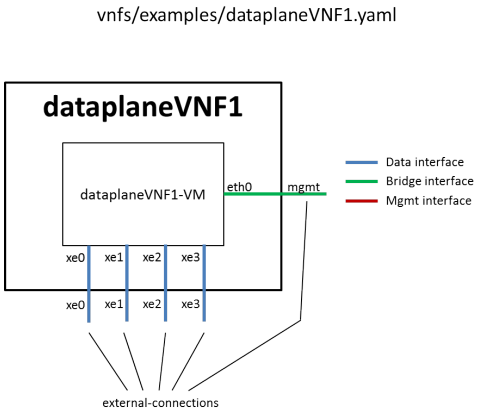

Figure 5.6: Visual representation of the VNF descriptor in figure 5.5. The blue connections represent data plane traffic which require PCI passthrough and/or SR-IOV. The green connection is a control plane connection which can be done using an unmodified virtual switch. (Source: https://github.com/nfvlabs/openmano/blob/master/images/descriptors/example-dataplanevnf1.png)

5.4.2 NS Descriptors

Figure 5.7 shows an example of a Network Service (NS) descriptor. It has a name and description which are similar to the VNF descriptor. It also has a topology part where it describes the necessary nodes which are VNFs and networks that are already stored in the database.

```
name:           insert a name for the scenario
description:    insert a description for the scenario
topology:
  nodes:
    vnf1:                   # vnf/net name in the scenario
      type:     VNF         # VNF, network, external_network (if it is a datacenter network)
      VNF model: vnf_model1   # VNF name as introduced in OPENMANO DB
      # vnf_id:    519f03ee-8ab6-11e4-ab4c-52540056c317   # Optionally, instead of the VNF name, the VNF id in Openmano DB can be used
    vnf2:
      type:     VNF
      VNF model: vnf_model2
      # Optional information for display in the openmano-gui: graphical position of the node and its interfaces
      graph:    {"y":399,"x":632,"ifaces":{"left":[["xe0","d"],["xe1","d"]],"bottom":[["eth0","v"],["eth1","m"]]}}
    bridge1:                # Bridge networks must be declared in this section if we want to interconnect VNFs using a Linux bridge
      type:     network
      model:    bridge_net   # 'bridge_net' or 'dataplane_net' for 'network' type
    default:                # External networks (datacenter nets) must be declared in this section if we want to interconnect VNFs to them
      type:     external_network
      model:    default      # datacenter net name, as introduced in OPENMANO DB
  connections:              # In this section, connections between VNFs and networks are explicited
    datanet:                # name
      # Data plane connections do not need to include a bridge since they are built through the Openflow Controller
      nodes:                # nodes that will be connected: one or several vnfs, and optionally one additional network declared in nodes section
      -  vnf1: xe0          # First node and its interface to be connected (interfaces must match to one in the VNF descriptor)
      -  vnf2: xe0          # Second node and its interface
    control net:
      # Control plane connections must include a bridge network in the list of nodes
      nodes:
      -   bridge1: null      # Bridge networks must be included if we want to interconnect the nodes to that network
      -   vnf1: eth1
      -   vnf2: eth1
    external net:
      # Connections based on external networks (datacenter nets) must include the external network in the list of nodes
      nodes:
      -   default: null      # Datacenter networks (external networks) must be included if we want to interconnect the nodes to that network
      -   vnf1: eth0
      -   vnf1: eth0
```

Figure 5.7: Example of a NS descriptor in openmano. This descriptor describes the topology of the VNFs in used by the network service.

5.5 Conclusion

Openstack can fulfill the role of the virtual infrastructure manager in the NFV architectural framework. This allows the VNF manager and NFV orchestrator to interact with an API that abstracts low level details which can be different on various environments. NFV applications are often very performance sensitive, therefore it is necessary to expose certain features from the hypervisor. These features include exposing the underlying CPU topology, enabling huge pages, allowing to pin vcpus and exposing the NUMA topology.

Even with these features enabled there is still a chance of getting interrupts from QEMU iothreads. To avoid these interrupts it is necessary to affinitize the iothreads to CPU cores on the host which are not used by the VM for packet processing. This cannot be done with Openstack at the moment.

At the time when this thesis is being written Openstack is already being used in MANO platforms like OpenMANO. OpenMANO has its own VIM implementation but can also use Openstack. It requires PCI passthrough, if necessary with SR-IOV, to run data plane traffic. The network infrastructure described in chapter 4 is still used for control plane traffic.

Although PCI passthrough improves the performance a lot, it is not aligned with the cloud model. The cloud principle states that the VM is totally unaware of the underlying hardware. This is not the case since the guest OS requires the driver of the assigned device. This and other issues are described in chapter 6.

Chapter 6

NFV data path

6.1 Introduction

In the context of NFV there needs to be a data path between the NIC and the VM for data plane packet processing. This data path needs high performance which can be achieved using either a optimized virtual switch or direct hardware assignment.

The first approach runs a switch in software that is attached to the physical NIC. When a VM is launched it is attached to the switch. This allows traffic to move between the NIC and the VM, but also between VMs. The second approach creates a direct data path between the VM and the NIC using PCI passthrough or/and SR-IOV.

6.2 PCI passthrough

When virtualization technologies became popular there was initially no support for virtualization of PCI devices, therefore I/O devices where emulated. Device emulation has certain advantages, for example it allows physical devices to be shared among multiple VMs, but it also comes with a huge performance cost. Therefore hardware manufacturers added support in their products to allow virtualization of I/O devices through a Input/Output Memory Management Unit (IOMMU). One example of this is Intel VT-d[5].

With hardware support it is possible to assign a PCI device to the VM (figure 6.1), this technique is also called PCI passthrough. This allows I/O data traffic without any intervention of the host. One of the features provided by VT-d is DMA remapping. This allows the creation of multiple DMA protection domains. Each of these domains represent an isolated environment containing part of the hosts physical memory. In the context of virtualization, the hypervisor can threat each VM as such a domain to prevent a VM from accessing devices that are not assigned to that domain.

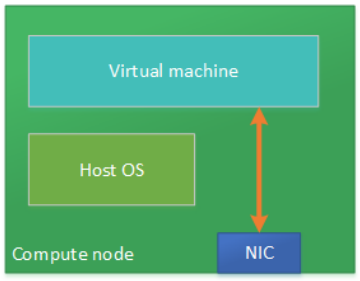

Figure 6.1: High level overview of a VM with PCI passthrough.

When a hypervisor launches a VM it provides a address space to it which is seen by the VM as its physical address space, also called Guest Physical Address (GPA). This may different from the Hosts Physical Address (HPA). The DMA capable device needs HPA to transfer data to the physical memory. But when a device is assigned to a VM, the driver uses GPA. Therefore DMA remapping can be used to translate GPA to HPA. Since the hypervisor provides GPA to the VM it can also configure the DMA

remapping hardware. Intel VT-d also allows interrupt remapping which completely avoids the hypervisor to interact when the VM and the device communicate.

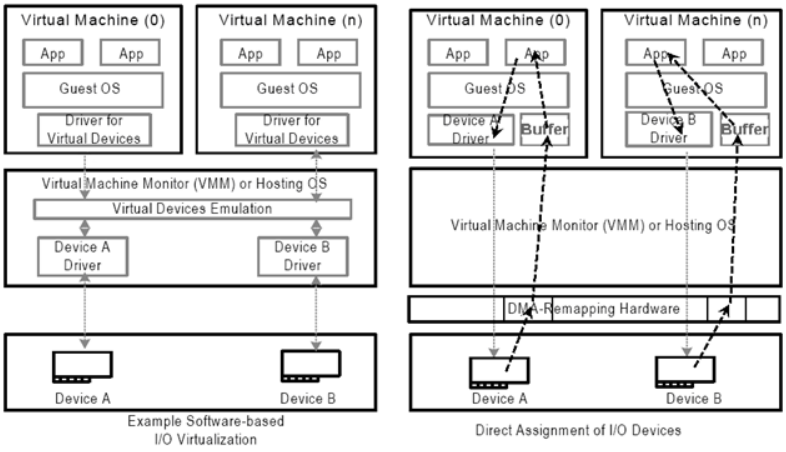

Figure 6.2: I/O using device emulation and I/O through DMA remapping. The example on the left shows the classic approach with device emulation. The example on the right shows an example with hardware support to enable direct assignment. (Source: https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices)

Although PCI passthrough greatly improves the performance, there are some problems with live migration [40]. Migrating a VM from one machine to another involves copying the entire state of the machine. This includes virtual devices, the physical memory and the processor state. When a device is emulated copying the state is not a problem since the device is maintained by the hypervisor. But in the case of direct assignment there are some issues. First of all the target platform needs to have the same device, but if we assume this is the case there is also the problem of accessing and writing the state of the device. It is possible that the state of the device is not completely accessible. Even if this is the case it still requires device specific knowledge from the hypervisor. Finally some parts of the state are impossible to migrate since they are unique to a specific device, for example the MAC address of a NIC.

Because of these issues live migration with PCI passthrough devices need to use a bit different approach. One approach could be to stop using the device before live migration and restoring it when the VM is migrated. This could be done in combination with a PCI hotplug mechanism[40]. Another approach is to switch to a virtual device during the migration. This would be useful for networking devices to keep connectivity.

PCI passthrough is supported by a number of hypervisors. KVM allows the user to launch VM with PCI devices directly assigned. This requires hardware supported such as Intel VT-d. Intel VT-d needs to be enabled in the BIOS and it needs be supported by the OS. Linux supports VT-d by adding *intel_iommu=on* to the kernel boot parameters.

6.3 SR-IOV

When a device is assigned to a VM the performance is almost the same as when the device is used in a non virtualized environment. But the device *cannot* be shared with different VMs. Therefore the number of VMs, assuming they all need a device, is limited to the number of devices. This causes a number of issues.

In the context of NFV there needs to be a data path with the NIC. When using PCI passthrough the number of VNFs is limited since they need connectivity with a NIC. This prevents optimal usage of a NIC for 2 reasons. A VNF can be too slow to process data at line rate, therefore the usage of the NIC is limited by the performance of the VNF. It is also possible that a VNF doesn't process packets at a constant rate, so the NIC is not constantly used. In this case periods that the NIC is not used a lot there is room for other VNFs using the same NIC.

Single Root Input/Output Virtualization (SR-IOV)[11] allows the server to expose a single PCIe device as multiple virtual PCIe devices. The entire PCIe device is called the Physical Function (PF) and the virtual PCIe devices are called Virtual Functions (VF). This allows for example a NIC to be exposed as multiple network interfaces which can be attached to VMs (figure 6.3). The NIC will act as a layer 2 switch to route the packets to the correct VF. At the moment there are NICs which are able to offer up to 128 VFs [9].

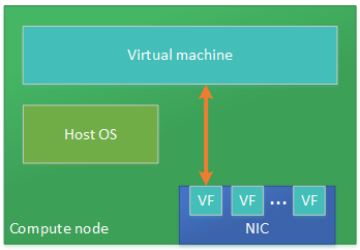

Figure 6.3: High level overview of a VM with PCI passthrough of a VF.

The PF is used to create VFs, in Linux this can be done by executing the following command: *echo 16 >/sys/class/net/ens785f0/device/sriov_numvfs*. This creates 16 VFs for the PCIe device associated with the network interface named ens785f0. Figure 6.4 shows an example of VFs created from this command.

```
root@node-3:~# lspci | grep Ethernet
02:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01)
02:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01)
02:10.0 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:10.2 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:10.4 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:10.6 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:11.0 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:11.2 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:11.4 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:11.6 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:12.0 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:12.2 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:12.4 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:12.6 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:13.0 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:13.2 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:13.4 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
02:13.6 Ethernet controller: Intel Corporation 82599 Ethernet Controller Virtual Function (rev 01)
```

Figure 6.4: This terminal output shows the virtual functions which are created on a NIC. Each virtual function is visible to the operating system as a separate network interface.

The NIC used in this thesis is based on the Intel 82599 10 Gigabit Ethernet Controller[4]. This Controller has a total of 128 receive and 128 transmit queues. These queues can be assigned to a pool which belongs to a VF. The amount of pools can be configured to be 16, 32 or 64. Depending on the number of pools each VF will receive its share of queues. For example with 64 pools, each pools contains 2 transmit and 2 receive queues.

To decide where a packet should go there is a Layer 2 classifier. When a packet is received on the controller this classifier will examine the L2 header and decide in which pool the packet belongs. This could based on the destination MAC address or the VLAN ID.

Figure 6.5 shows an example of a packet arriving on the NIC. This involves the following steps:

1. The packet arrives on the NICs Ethernet connection.
2. The packet is sent to the L2 classifier/sorter.
3. The classifier decides to which pool it should be sent based on the destination MAC address.
4. The VF starts the DMA operation.
5. The Intel VT-d chipset translates the virtual address (GPA) provided by the driver to a physical address (HPA) and the packet is copied to the buffer in memory.

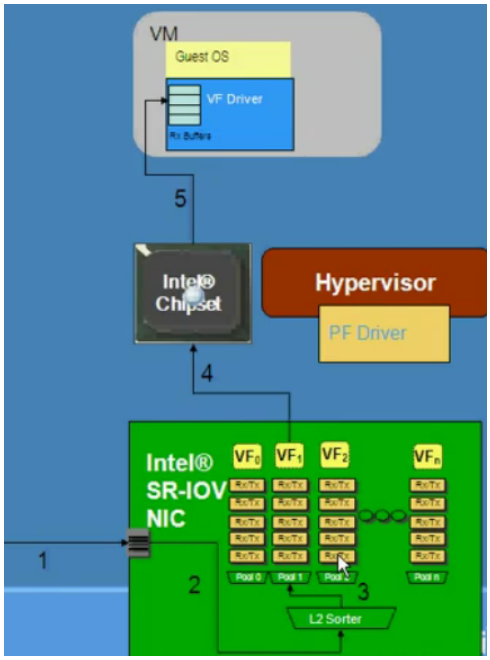

Figure 6.5: Flow of a packet arriving on a NIC with SR-IOV enabled. (Source https://communities.intel.com/community/wired/blog/2010/09/07/sr-iov-explained)

6.4 Open vSwitch

The movement of servers into virtual machines created an additional demand for virtualization of the underlying network. For example it is possible that 2 VMs run on a single physical machine, but belong to 2 completely separated networks. First attempts of creating virtual networks on servers have been done using existing networking components of the OS. For example, Openstack can be configured to create virtual networks using Linux bridges[1].

The increasing demand for virtual networking has led to more specialized software implementations which provide much more functionality than the OS networking components. Open vSwitch provides a virtual switch which supports a number of features which are heavily focused on virtualizing the network. One of these features is OpenFlow which will be discussed more in section 6.4.1.

The major components of Open vSwitch are visible in figure 6.6[29]. The ovs-vswitchd component is the same for all operating systems. This components holds all the actions that should occur for the incoming traffic. These actions can be cached in the kernel datapath module to improve performance. The ovsdb-server component keeps all the configurations which need to be stored persistently.

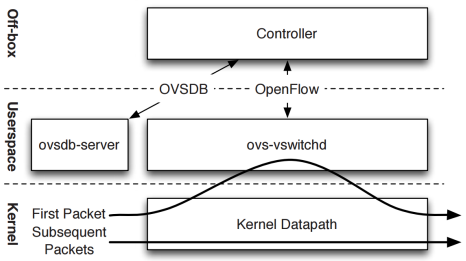

Figure 6.6: Components of Open vSwitch[29]. The main components are ovs-vswitchd and the Kernel Datapath which are both used for packet forwarding.

[1] http://docs.openstack.org/mitaka/networking-guide/scenario-classic-lb.html

6.4.1 Software Defined Networking

The goal of SDN is to increase programmability of existing networking devices by decoupling the data plane (or forwarding plane) and the control plane [28]. The data plane defines the part of a networking device that processes each individual packet that arrives at an interface. This part often exists in the form of Application-specific integrated circuits (ASIC) in order achieve high performance on a large number of ports. The control plane makes decisions on where the traffic should be sent, this part is concerned with the network topology and doesn't require specialized hardware.

Traditional networks use routers and switches which have vendor-specific control interfaces where each device makes its own decision about where traffic should be sent. By decoupling the control- and data plane, we can create a centralized controller which can run in software and controls multiple networking devices. This requires an open protocol like OpenFlow to allow communication [24] between the controller and the forwarding devices.

At first sight it seems that NFV and SDN are two competing technologies, they both try to solve the same problem using a different approach. In reality this is not the case, SDN solutions like OpenFlow allow some degree of programmability but they are not very suitable for more complex network applications like a router or a statefull firewall. Both technologies have certain shortcomings which can be resolved by combining them into one complete solution. One example of this is the deployment of network services. This is done by deploying VNFs as VMs which are interconnected using SDN.

OpenFlow

As mentioned before, OpenFlow defines an open communication protocol to program the data plane in switches and routers. The protocol needs to be supported by the device in order to work, these devices are called OpenFlow switches. Figure 6.7 shows an example of an OpenFlow switch.

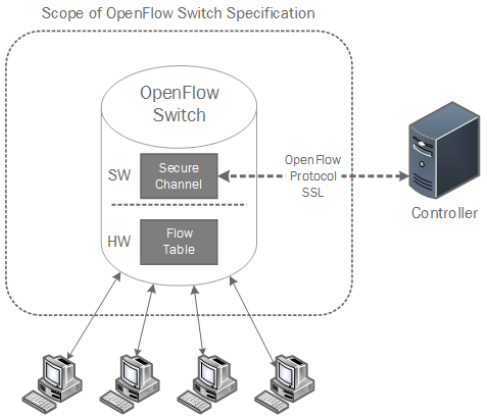

Figure 6.7: Example of an OpenFlow switch. The flow table is updated by the controller through the secure channel. The controller can be connected to multiple OpenFlow switches.

OpenFlow is by far the most supported SDN technology and a lot of vendors of networking devices added this protocol to their products. The forwarding decisions in an OpenFlow switch are based on one or multiple flow tables which contain flow entries associated with an action. The flow tables are updated by the controller through a secure channel using the OpenFlow protocol. Each flow-entry is associated with a simple action, the 3 basic ones are:

- Forward the packet to a certain port. It is possible to modify the packet headers like adding a VLAN tag.
- Encapsulate and forward this packet to the controller through the secure channel. The controller can choose to insert a new flow-entry to handle all packets belonging to this flow, in this case only the first packet is sent to the controller.
- Drop packets that match this flow-entry. This can be used for to avoid security threats like denial-of-service attacks.

Switches and routers can also keep their main functionality and implement OpenFlow as an additional feature. In this case a fourth action can be added to forward the packet to the devices normal processing pipeline.

6.4.2 virtio

When a VM runs on top of a full virtualization hypervisor, the guest OS is unaware of being virtualized. This requires the emulation of I/O devices or direct assignment. An alternative for full virtualization is paravirtualization, in this case the guest OS is modified to be more suitable in a virtual environment. For I/O devices this means that the guest OS contains a paravirtualized driver which avoids hardware emulation.

Virtio[21] is one example of such a paravirtualized driver for the KVM hypervisor. It provides an API to the guest OS which provides an abstraction for a set of common emulated devices. This could be seen as follows, the guest sees the front-end of the driver which is always the same. But the hypervisor provides a back-end which doesn't need to be the same in different hypervisors.

The data transport between the host and the guest goes through a queue. The Open vSwitch used for in the test setup in chapter 7 uses 1 queue. Although it is possible to use multiple queues[2], this is not supported by Openstack at the moment of writing this thesis. The transfer of a packet between the host and the guest also involves copying the packet between the different memory spaces. Although zero-copy is possible[33], it is not implemented with the current implementation used by Openstack.

6.4.3 DPDK acceleration

The architecture of Open vSwitch makes use of the OS networking stack. Chapter 3 describes the shortcomings of this stack and provides frameworks which improve the performance. To use open vSwitch in a NFV context it is necessary to avoid the OS networking stack. Therefore it is possible to configure Open vSwitch to use a DPDK application instead of the kernel data path (figure 6.8)[7].

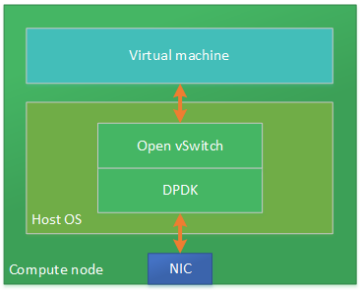

Figure 6.8: High level overview of Open vSwitch using DPDK to boost the fast data path.

The ovs-vswitchd component stays the same but the kernel data is replaced. The use of DPDK requires huge pages to be allocated and it requires certain CPU cores to be used for the poll mode driver. The amount of memory and the specific cores can be configured. The process will spawn a thread for each receiving port and affinitize it to core which has the least threads already. To achieve best performance it is recommended to isolate the cores used by the Open vSwitch process. This avoids interruption of the poll mode driver threads.

6.5 Feature comparison

Table 6.1 shows a comparison of the features of PCI passthrough, SR-IOV and Open vSwitch (with DPDK). These are features relevant in the NFV context. Offload features can be used to improve performance, for example the calculation of the IP checksum can be done by certain NICs. OpenFlow is an important feature to interconnect VNFs.

By using a virtio driver in the VM, the VM is abstracted from the physical NIC. For PCI passthrough and SR-IOV the physical drivers have to be present in the VM. This can become a problem if new hardware is used from which the drivers are not yet present in the guest OS.

[2] `http://www.linux-kvm.org/page/Multiqueue`

	PCI passthrough	SR-IOV	Open vSwitch
Max. virtual interfaces	1/port	64/port[3]	Unlimited
Offload features	yes	yes	no
OpenFlow support	no	no	yes
Virtio driver	no	no	yes
Live migration	no	no	yes

Table 6.1: Comparison of the features which are relevant in the NFV context.

6.6 Conclusion

There are 2 approaches available for creating a data path used for data plane packet processing in a virtualized environment. Either by direct hardware assignment or a virtual switch.

Direct hardware assignment, in this case PCI passthrough, can be achieved with the support of an IOMMU like Intel VT-d. In this case the VM has full control over the PCI device. Since the device cannot be shared there is a possibility to use SR-IOV which allows the creation of virtual functions on a single NIC. Each virtual function can be assigned to a different VM.

Using a virtual switch, like Open vSwitch, enables certain features which are very useful in the context of NFV. OpenFlow creates more programmability in the network interconnecting VNFs. Virtio allows more generality and abstraction between the guest OS and the underlying hardware. If live migration is available it is possible to orchestrate VNFs in a way which makes more optimal use of the infrastructure.

Plain Open vSwitch suffers from the performance limitations explained in chapter 3. This has been the motivation to create an alternative configuration which uses DPDK to accelerate the data path. The answer for the *first research question* is as follows: since Open vSwitch uses the OS networking stack it is not suitable for data-plane packet processing.

The features which are available using Open vSwitch align much more with the cloud model. The VM is completely abstracted from the underlying hardware and there are much more orchestration and management capabilities. This answers the *second research question* since there is a demand for these features which are not available using PCI passthrough and/or SR-IOV. Therefore a virtual switch can provide an alternative.

Chapter 7

NFV data path test setup

7.1 Introduction

All tests are performed using the Open Platform for NFV (OPNFV)[1] Brahmaputra. This platform adds capabilities to Openstack which make it more suitable for NFV use cases. The focus of this thesis will be on the OVSNFV project which upgrades Open vSwitch (OVS) to use DPDK to accelerate the data path. The choice for using OPNVF is mainly focused on the tests which use Open vSwitch with DPDK. Since Openstack supports PCI passthrough and SR-IOV by default these can be done with a standard deployment.

Besides the OVSNFV project there are other projects which add features and capabilities to the platform. These features can be enabled for a certain target scenario, but it is not necessary to use all of them.

OPNFV can be deployed using multiple deployment tools, but they do not all support the same features. The administrator can choose the desired deployment tool based on the supported features and certain properties of the platform. For example, some tools us Ubuntu as the host OS and other tools use CentOS. The tool used for these test is called Fuel since this is the only tool that supports the OVSNFV plugin.

7.2 Hardware components

All tests are executed on a dual socket Intel server board with 2 Xeon E5 processors. Table 7.1 shows more detailed information about the components.

Item	Description
Processor	2x Intel Xeon CPU E5-2697 v3 @ 2.60GHz
Platform	Intel Server Board S2600WTT
Network interface	2x dual port Intel 82599ES 10-Gigabit SFI/SFP+
Memory	8x 8GB DDR4
BIOS	SE5C610.86B.01.01.0016.033120161139

Table 7.1: Hardware properties of the Compute node.

7.3 Software components

7.3.1 Host

Table 7.2 shows the software components used by the Openstack compute nodes. This node is used to host the VMs that run the tests. The software components are mostly the same except for QEMU and Libvirt. The OPNFV plugin that is used to install Open vSwitch with DPDK also installs a predefined version of QEMU and Libvirt to avoid dependency issues. Also, the Open vSwitch with DPDK version is not applicable for PCI passthrough and SR-IOV.

[1] `https://www.opnfv.org/`

Item	Description
Cloud platform	Mirantis 8.0 OpenStack Liberty
Operating system	Ubuntu 14.04 kernel 3.13.0-85-generic x86_64
QEMU-KVM	*SR-IOV:* libvirt 1.2.9 qemu 2.4.0 *Open vSwitch + DPDK:* libvirt 1.2.12 qemu 2.2.1
Open vSwitch with DPDK	Open vSwitch 2.4.90 DPDK 2.0.0

Table 7.2: Software properties of the Compute node.

7.3.2 Guest

The VM is running a CentOS distribution. Since the VM can be any OS the choice has been made to use a familiar system. Table 7.3 shows more detailed information about the VM.

Item	Description
Operating system	CentOS 7.2 kernel 3.10.0-327.el7.x86_64 x86_64
DPDK	DPDK 2.2.0
PROX	PROX 0.31
DATS	DATS 0.31

Table 7.3: Software properties of the VM.

7.4 Test metrics

To compare networking devices of multiple vendors, it is necessary to execute tests that generate comparable data. This report is focused on measuring the throughput, this is defined in RFC 1242 by the number of packets that can be processed without any loss. The tests are run using packet sizes of 64, 128, 256, 512, 1024, 1280 and 1518 bytes. These are include the smallest and largest possible packet sizes together with a range of sizes in between with a finer precision for smaller sizes.

Some tests change the length of the packet, in this case the reported packet size is the size on the part where the packet is the largest. For example, if the test case adds a 4-byte MPLS tag, then the smallest packet size will be 68 bytes on the tagged side and the 64 bytes on the untagged side. For the next packet size it will be 128 on the tagged side and 124 on the untagged side. The reported packet sizes are the sizes on the part where the frame is the largest (68, 128 ... 1518).

Finding the exact point where packets can be processed without loss requires a searching algorithm that reduces the speed when a test run fails and increases the speed when a test run is successful. The results in this report are based on a binary search algorithm. In this case the first test run is done at 100% of the speed. If there are no packets lost after running the test for 5 seconds it is considered a success. Otherwise the test is failure and repeated at half the rate until the difference between 2 runs is less than 1%.

The device that is being tested is referred to as the System Under Test (SUT). The ideal setup would use a tester which both transmits and receives the traffic. The generated traffic consists of UDP packets. In the context of NFV the SUT will not be a physical device but a Virtual Network Function (VNF). The VNF will run as a VM on an Openstack compute node.

By running multiple use cases which represent a certain category of network functions we can determine the characteristics of the configuration. Each of the selected use cases generates a unique workload which is more sensitive to different parts of the system.

As mentioned before, the throughput is measured in packets per second. We will compare the measured results with the theoretical maximum of a 10Gbit/s link. To calculate this maximum we need

Figure 7.1: Logical view of the test setup.

divide the speed of the link with the size of a packet with the inter-frame gap. Equation 7.1 shows the calculation for 64 byte packets and table 7.4 shows the maximum values for different packet sizes.

$$\frac{\frac{10*10^9}{8} bytes/s}{(64+20) bytes} = 14880952 pps \approx 14.88 Mpps \tag{7.1}$$

Packet size (bytes)	Maximum throughput (Mpps)
64	14.88
128	8.45
256	4.53
512	2.35
1024	1.20
1280	0.96
1518	0.81

Table 7.4: Maximum theoretical throughput for different packet sizes on a 10Gbit/s link.

To run these tests we need a tester which is able to generate traffic at 10Gbit/s. This can be done by a dedicated appliance or software running on a commodity server. When traffic is generated at the maximum speed, also called line rate, the gap between each packet is minimized. When traffic is generated at a lower rate this gap will increase, ideally the gap between each packet would be the same. Unfortunately this is very hard to achieve using an application which runs on a commodity server. Therefore dedicated appliances are available which can guarantee a much higher degree of precision of when each packet is sent.

The choice between a software- and a hardware based solution should not be taken lightly. Some companies have trusted hardware which has been validated to work correctly. Therefore it is easy to reproduce test results using the same hardware. A software based generator is dependent on the platform it runs on. The chosen hardware, the OS and the configuration of the generator can all influence the test results. In a sense it also measures the testers performance. On the other hand it allows for more flexibility when setting up the test configuration.

7.5 Test cases

This test setup uses a DPDK software application which both generates and receives the traffic. We will be using a tool called PROX which can be used as the tester and the SUT. At the tester side it will be used to generate and analyze the traffic. At the side of the SUT, PROX can be used to create DPDK applications which execute certain network functions.

To compare different configurations we will run 7 different test cases on the SUT. The first 5 test cases are synthetic test cases. The last 2 test cases represent realistic test scenarios that can be part of a network. To run all these test cases a tool called Dataplane Automated Testing System (DATS) is used which is available on Github[2]. This tool runs on a system with 4 times 10Gbit/s NICs which generates a maximum of 40Gbit/s. The first 3 test cases have the following architecture:

The implementation possibilities of a DPDK application are dependent on the environment where it runs. It can be possible that an implementation which works in a certain environment doesn't work in another environment. This is also the case with these test cases. First the buffer size of the memory region which can be accessed by the NIC, also called descriptors, can be different for different scenario's. When using PCI passthrough the size can be around 256 descriptors. The NIC itself is able to buffer

[2] `https://github.com/nvf-crucio/DATS`

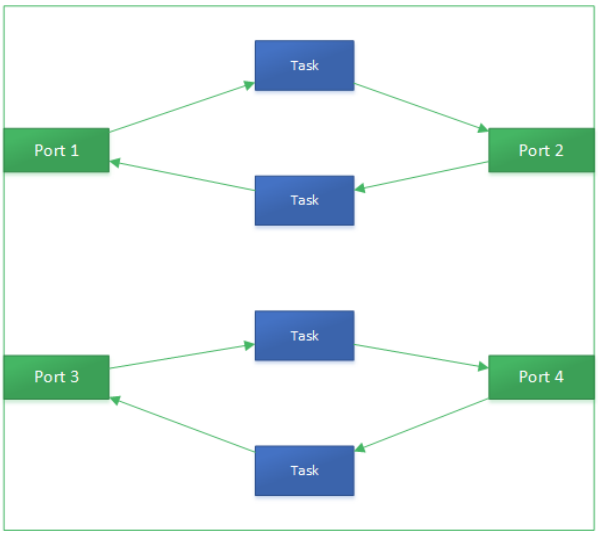

Figure 7.2: Architecture of the first 3 test cases. This picture visualizes the architecture of the DPDK application running in the guest OS. Each task runs on a different core.

packets which avoids losing packets. In the case of SR-IOV the NIC doesn't buffer packets. It it is therefore necessary to allocate a larger memory region. Therefore these test cases are configured using 2048 descriptors.

In some scenario's it is possible to send packets from multiple tasks to a single TX port. This is the case for PCI passthrough, but for SR-IOV and Open vSwitch only 1 task can send to a port. The test cases are therefore configured to aggregate packets before sending them to a TX port.

The test results which are generated in this thesis are executed with the same configuration. In theory it is possible to create optimized configurations for certain environments but this would create less comparable test results.

7.5.1 Layer-2 forwarding

The first test case handles packets from a layer 2 perspective. The application doesnt do any processing except for setting a preconfigured source and destination MAC address. The CPU and memory usage is minimal, therefore the performance will mostly be affected by the receive and transmit channel.

7.5.2 Tag/Untag

This test case changes the size of each packet that is received by the SUT. 2 of the 4 task add a 4-byte MPLS tag to each packet and the other 2 strip a 4-byte MPLS tag. This tag is inserted after the Ethernet header, it is therefore necessary to move this header.

7.5.3 ACL

The third test case will match the traffic against an Access Control List (ACL). The SUT matches 7 header fields against a preloaded list of 32K rules. The generated traffic randomizes these fields to make sure that all the rules are hit. For each packet the algorithm looks for the "best" matching rule. If a packet matches multiple rules it will take the rule with the highest priority.

7.5.4 Load distributor

This use-case focuses on identifying flows and making a forwarding decision. A 5-tuple is used to decide where the load should be distributed. This tuple consists of the fields that identify a flow so that every packet that belongs to the same flow is sent to the same port. The load is uniformly distributed over all the output ports using a hash table of 8M entries. Figure 7.3 shows the architecture of this test case.

Each load distributor stores its own hash table, so there are 4 hash tables with 8M entries. Each has entry is 16 bytes long, therefore the total amount of memory used for all the hash entries is 512MB.

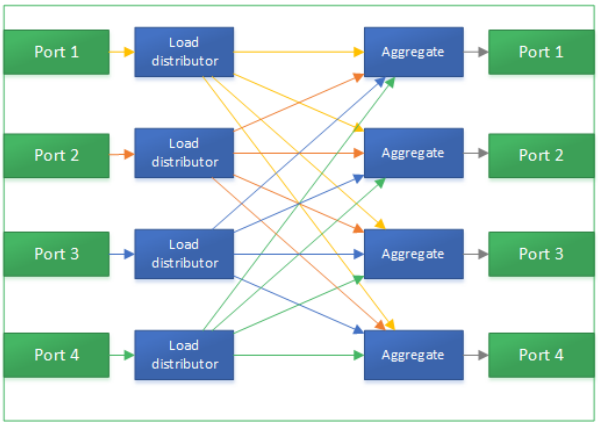

Figure 7.3: Architecture of the load distributor test cases. The test case only uses 4 ports, the figure shows 8 ports for better readability.

7.5.5 Buffering

The buffering test cases measures the impact of the condition when packets get buffered, thus they stay in memory for the extended period of time. This can be part of network to enhance the quality of streams. For example, when a CDN is used for multimedia streams there are often multiple layers of servers which are used to distribute all the stream data. The servers on the edge of the network typically buffer stream data to compensate for packet loss. In contrast to the other test cases it only receives packets on a single port (Figure 7.4).

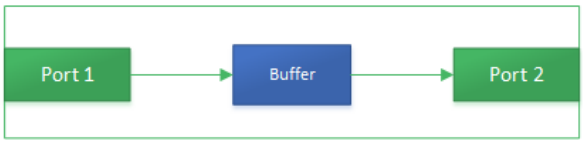

Figure 7.4: Architecture of the buffering test case.

The packets are buffered for 125ms before they are forwarded. This requires enough memory to hold all the packets before forwarding them. The amount of memory required for each packet is independent of the size of the packets. Since it would the cost of allocating and deallocating memory is very high all packets are stored in a data structure with the same size. The amount of memory necessary to buffer packets sent at line rate is approximately 3.3GB.

7.5.6 BNG

The last test cases represent a more realistic scenario called a Border Network Gateway (BNG) which is the access point for subscribers to the internet (Figure 7.5).

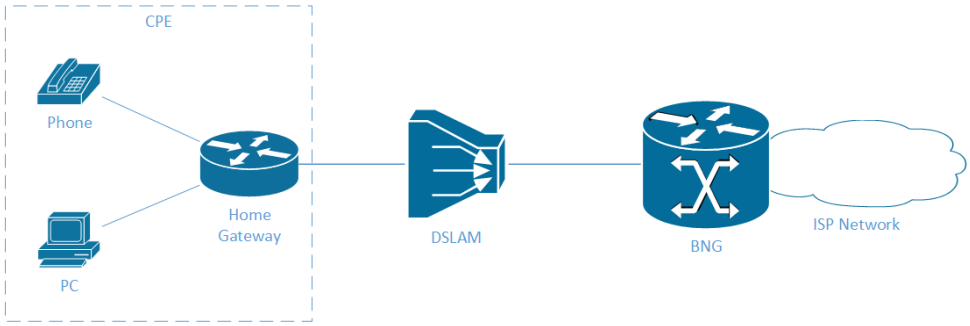

Figure 7.5: Basic BNG architecture. The image visualizes the role of the BNG application in a network service provided by an ISP.

In a real scenario multiple users would be connected to a digital subscriber line access multiplexer (DSLAM) which would send the traffic on a single Ethernet connection to the BNG router. At this point the different subscribers are identified using two VLAN tags. The BNG router translates the traffic to an appropriate format which can be routed on the Internet side. Our test scenario has 64K subscribers.

The traffic that arrives at a NIC first goes to a load balancer. Then the load balancers distribute the traffic uniformly to a number of worker tasks. These worker tasks map the internet traffic to CPE traffic and the other way around. Each worker thread has a lockless queue configured where it receives traffic. When a load balancer has decided to which worker the packet should go it will transfer the packet to this queue (Figure 7.6).

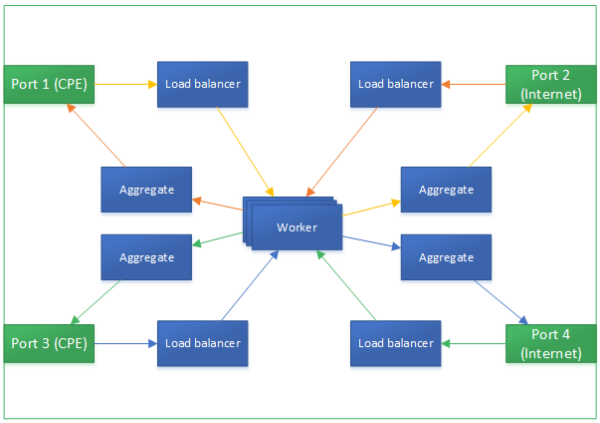

Figure 7.6: Architecture of the BNG test case.

7.5.7 BNG+QOS

The last test case adds Quality of Service (QOS) functionality to the BNG router[25]. Without QOS there is no guarantee about the traffic rate for each user. QOS policies determine how the traffic flow for each user influences each other. This is achieved by managing when packets should be transmitted or dropped using queues (Figure 7.7)

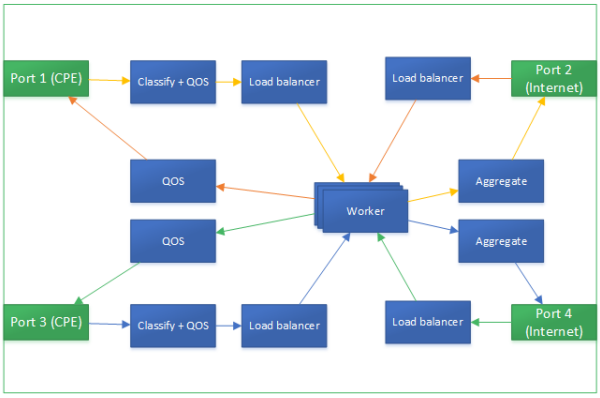

Figure 7.7: Architecture of the BNG+QOS test case.

7.6 Physical setup

As mentioned before all test setups have been configured using the platform but with a different configuration. The SUT has been deployed using Fuel as the deployment tool which is delivered by OPNFV Brahmaputra. An Openstack environment requires at least 1 controller node and 1 compute node. The diagrams only show the compute nodes since they are the ones that do the data processing.

To optimize the compute node for packet processing it is necessary to do some manual configuration after the installation is done. This is not how it is supposed to be used in an environment with a lot of nodes. This could be overcome by creating a Fuel plugin which does the necessary configuration.

The tester is not part of the Openstack environment, but it is directly connected to the compute node using 4x 10Gbit/s connections. Both the tester and SUT have a communication channel with a DATS controller which sends commands to both system which run the tests.

7.6.1 PCI passthrough setup

The PCI passthrough setup (Figure 7.8) attaches the NIC ports directly to the VM which bypasses the host OS. The Open vSwitch data path is still used to create a management interface. This management interface is used by DATS to launch the correct PROX application. The tester runs a PROX application which is used to generate traffic, this application is also managed by DATS.

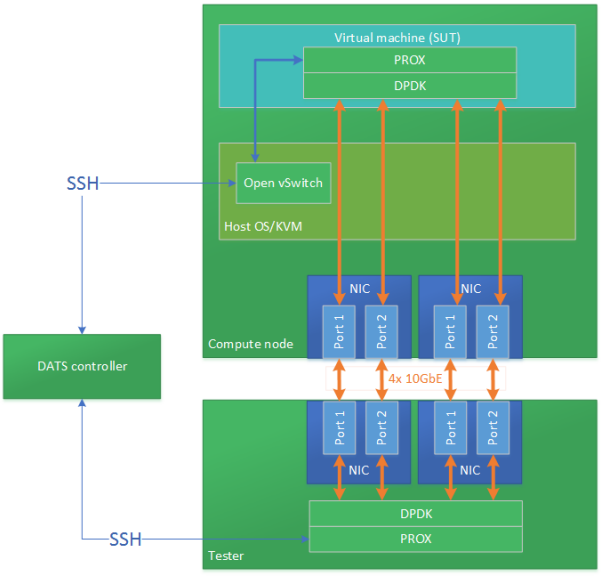

Figure 7.8: PCI passthrough test setup. Each port of the NICs in the compute node is directly assigned to the VM. The Tester runs a traffic generator/analyzer in bare-metal.

7.6.2 SR-IOV setup

The SR-IOV setup is similar to the PCI passthrough setup (Figure 7.9). Instead of assigning a complete port to the VM the port is first divided in Virtual Functions (VF). Instead of assigning the entire NIC port (PF), a VF of each port is attached to the VM using PCI passthrough.

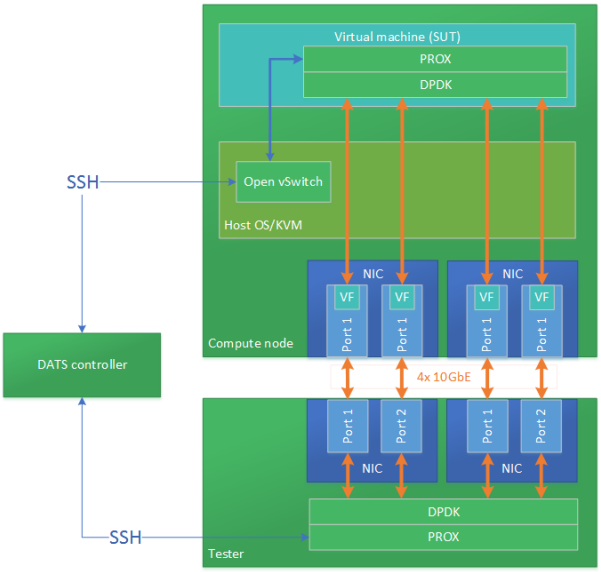

Figure 7.9: SR-IOV test setup with PCI passthrough of the VFs.

Although SR-IOV can be used without PCI passthrough there is no real benefit if doing this. Therefore the convention in this thesis is that whenever SR-IOV is mentioned it means that PCI passthough of the VF is used.

7.6.3 Open vSwitch with DPDK setup

Using the OVSNFV plugin we can accelerate Open vSwitch using DPDK. The plugin is still in an experimental phase and is not ready to use in production environments, but it can be used for experimental purposes. Using this setup the data plane traffic and the management traffic are both sent on the same virtual switch (Figure 7.10). Openstack uses Open vSwitch bridges to decide to which port the traffic should be forwarded. The VM is configured to have 5 virtual interfaces, each interface is mapped to a single port. The first port is used to manage PROX and the other 4 ports are used to send data.

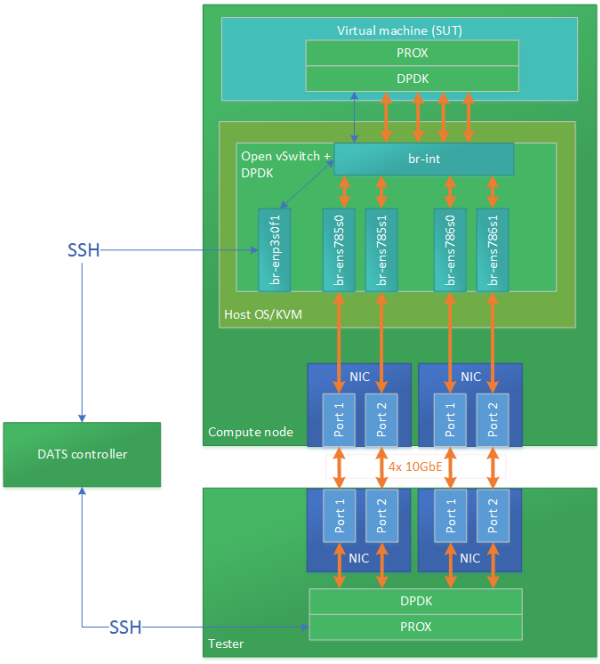

Figure 7.10: Open vSwitch with DPDK setup. The NIC ports are connected to the virtual switch which exposes virtio interfaces to the VM.

When using this setup it is necessary to assign certain cores to Open vSwitch. 1 core is necessary for the ovs-vswitchd process and there has to be at least 1 core for packet processing. In this setup Open vSwitch has been configured to use 5 hyper threaded cores for packet processing.

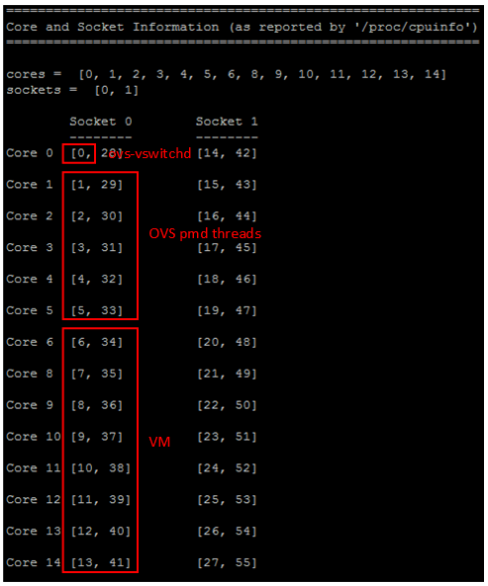

Figure 7.11: Distribution of cores among Open vSwitch and the VM.

Figure 7.11 shows the CPU numbering of a dual socket server board. ID 0-13 belong to the cores on socket 0 and ID 28-41 are their hypertreaded siblings. For example core 1 and 29 are siblings. Socket 1 uses ID 14-27 and 42-55, but this socket is not used for packet processing. Core 0 is used for the ovs-vswitchd process. Core 1-5 with their hypertreads are used for the pmd threads of Open vSwitch. The remaining 8 hyperthreaded cores are used by the VM.

7.7 Conclusion

This chapter describes the components used for testing different data paths between the NIC and the VM. The different test setups are installed using the same platform called OPNFV. By configuring each test setup in a different way it is possible to create a different data path.

To compare the different test setups we are using 7 different test cases with distinctive characteristics. In this way it is possible to make a more accurate conclusion of the system without digging into the implementation details of the components. Chapter 8 shows performance results using these test cases.

Chapter 8

NFV data path test results

8.1 Introduction

In this chapter the performance results of the different technologies are presented and compared. First we present the test results for throughput measurements. These measurements are created for 7 different test cases which are described in section 7.5.

Secondly the throughput results are compared in 2 ways. PCI passthrough and SR-IOV are compared to see if there is a difference between assigning a PF to the VM and a VF to the VM. Also SR-IOV is compared to Open vSwitch with DPDK, this comparison is the main subject for this thesis. SR-IOV has already been proven to achieve decent performance but Open vSwitch with DPDK is still in an experimental phase when this thesis is written. The goal is to investigate if Open vSwitch with DPDK is a valid alternative and in which use cases.

Finally the latency between sending and receiving packets on the tester is compared between all 3 test setups. This comparison has only been done on the L2 forwarding test case.

8.2 Throughput results

The test cases are executed using the smallest possible packet size and packet sizes of 128, 256, 512, 1024, 1280 and 1518 bytes. The smallest possible packet size depends on the test case, table 8.1 shows the smallest packet size for each test case. The results are grouped by test case where each color represents a different packet size.

Test case	**Smallest packet size (bytes)**
L2 forwarding	64
Tag/Untag	68
ACL	64
Load distributor	64
Buffering	64
BNG	78
BNG+QOS	78

Table 8.1: Smallest packet size for each test case.

The results are originally produced in packets per second. This is calculated by counting the amount of packets received in a certain amount of time and dividing this amount by the time passed. The results are presented in percentage of line rate for better readability. This means that 100% is equal to 40 Gbit/s for all test cases except for the buffering test case. For the buffing test case 100% corresponds to 10Gbit/s.

8.2.1 PCI passthrough throughput

Figure 8.1 shows the result for the PCI passthough setup. The L2 forwarding test case shows a throughput of 100% for packet sizes greater or equal to 128 bytes. The maximum throughput for 64 bytes packets

is around 80%. This is a result of the PCI Express 2.0 transfer rate which acts as a bottleneck[25].

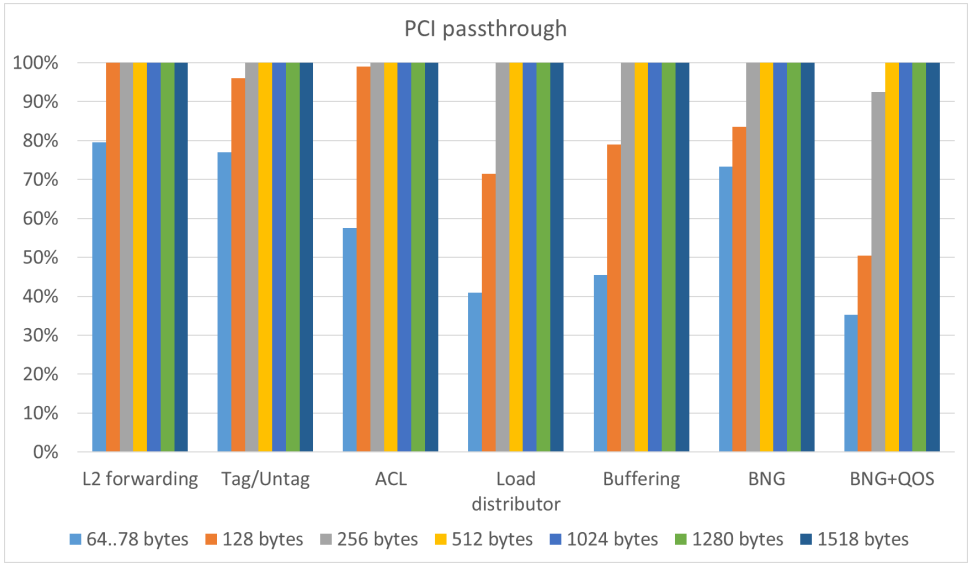

Figure 8.1: PCI passthrough throughput results. The throughput is displayed as percentage of line-rate. Each color represents a different packet size.

The Tag/Untag and ACL test cases have a lower throughput for the smallest packet size but they almost reach the maximum for the 128 bytes packets. These test cases are a bit more complex than the first one and require more processing power.

The Load distributor and buffering test cases show a lower throughput for both the smallest packet size and the 128 bytes packets. Both test cases use more memory than there is available in the cache which results in cache invalidation.

Even though the BNG test case is much more complex than other test cases, by using load balancers and enough worker threads it still shows a decent performance. With QOS it shows a low performance for the smallest packet sizes. Each receiving port has a single QOS task which acts as a bottleneck. Nevertheless the throughput reaches more than 90% for 256 bytes packets and 100% for packets from 512 bytes and larger.

The PCI passthrough setup shows a similar behavior for the different test cases. The smallest packet sizes show a decreased throughput but from 256 bytes and larger they almost all reach line rate.

8.2.2 SR-IOV throughput

Figure 8.2 shows the results for the SR-IOV setup. The results look very similar to the results of the PCI passthrough setup. Section 8.3.1 shows a detailed comparison with the PCI passthrough setup.

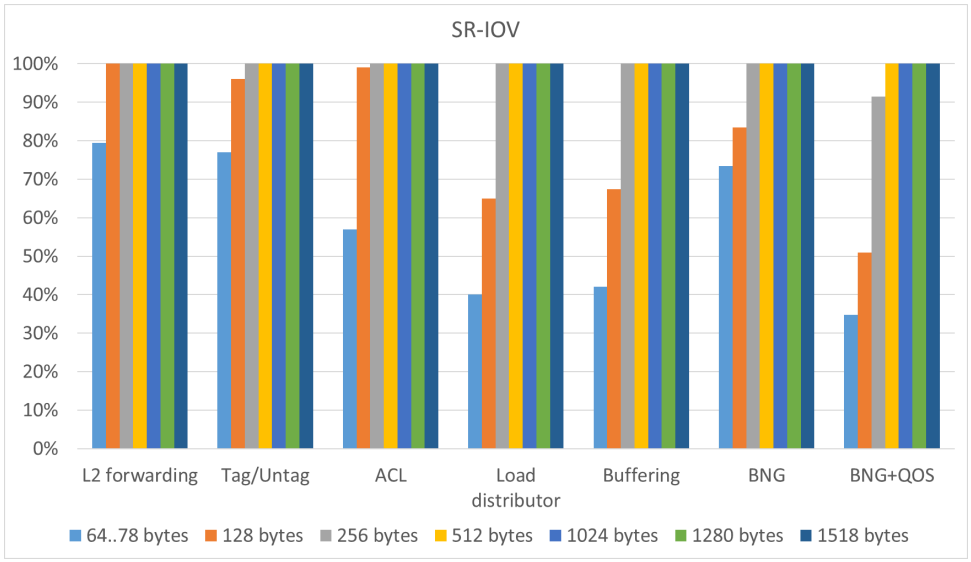

Figure 8.2: SR-IOV throughput results. The throughput is displayed as percentage of line-rate. Each color represents a different packet size.

8.2.3 Open vSwitch + DPDK throughput

The throughput results of the Open vSwitch with DPDK are shown in figure 8.3. The throughput for the smallest packet sizes are much lower than the other test setups. For the smallest packet size the throughput never reaches 25% or higher.

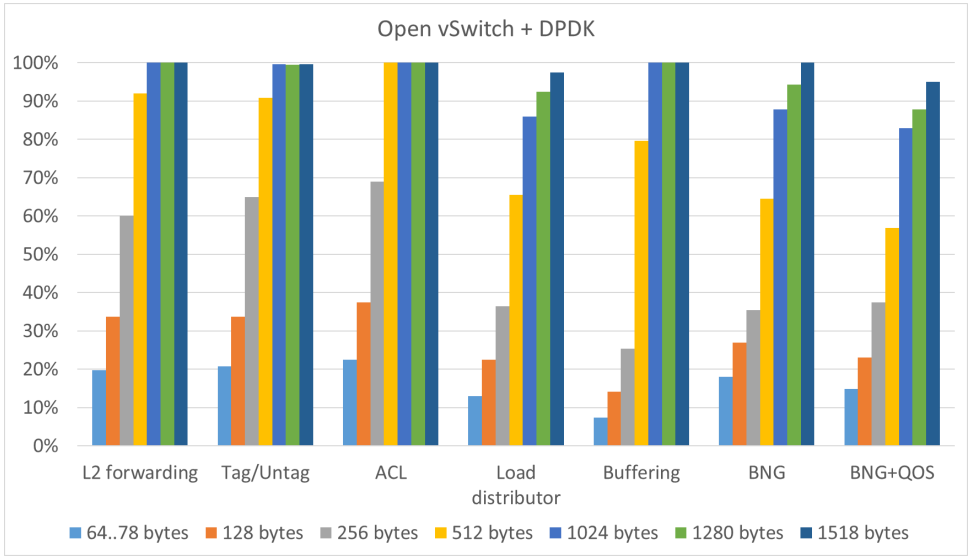

Figure 8.3: Open vSwitch with DPDK throughput results. The throughput is displayed as percentage of line-rate. Each color represents a different packet size.

The first 3 test cases show a similar behavior. The smallest packet size is around 20%, the 128 bytes packets are around 35%, the 256 bytes packets are around 65% and packets from 512 bytes almost reach line rate. It is remarkable to see that the ACL test case shows a better performance than the first 2 test cases. We will discuss this further in section 8.2.3.

The last 4 test cases show a lower performance for almost all packet sizes. All of these test cases use a lot of memory when they process packets. Open vSwitch also uses memory when processing packets, this means that the Last Level Cache (LLC) is shared between the VM and Open vSwitch. When the VM uses a small amount of memory there is little interference between the VM and Open vSwitch. But when the VM uses a lot of memory, the VM causes cache invalidation for Open vSwitch, therefore they

act as *noisy neighbors*.

ACL performance

The ACL test case shows a better performance than the L2 forwarding test case. This behavior is unexpected since the ACL is computational more expensive. A possible explanation could be that both test cases handle packets in different bulk sizes.

A DPDK application processes packets in bulks instead of one by one. The size of this bulk is dependent on the performance of the application and the configuration. When an application is processing packets very fast it will have small bulks, if it is slow it will use large bulks.

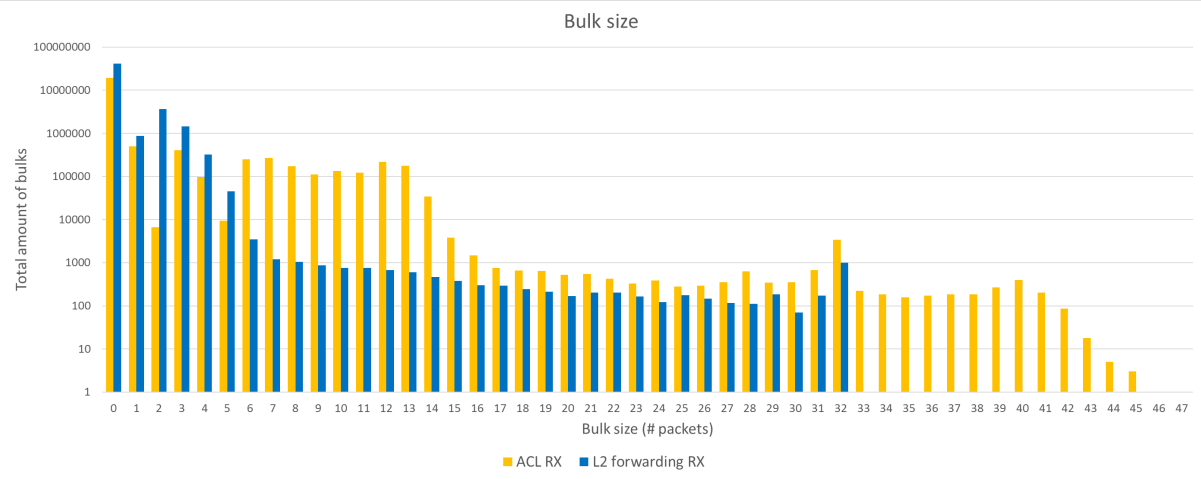

Figure 8.4: Bulk size distribution of the L2 forwarding and ACL test cases. The Y-axis has a logarithmic scale which allows to see the details more clearly.

Figure 8.4 shows distribution of bulk sizes in a certain amount of time. The yellow bars show the bulk sizes of the ACL test case, the blue bars show the L2 forwarding bulk sizes. The L2 forwarding test case has a lot of small bulks, the amount of bulks show a decrease when the bulk size becomes higher. There is a small increase in amount of bulks at bulk sizes of 32 packets, but there are no bulks with more packets.

The ACL test handles a lot more packets in bulks with sizes up to 12 packets. It is important to notice that a logarithmic scale has been used, for example the number of bulks with size 12 is more than 300 times higher for the ACL test case than it is for the L2 forwarding test case. This indicates that the packets are more evenly distributed among larger bulk sizes which allow a higher throughput.

The cause of this difference has not been identified, it could be a result of configuration. It is important to notice that not only the complexity matters but also implementation details. A comparison between different test cases is therefore not justified. This is also the reason why the choice has been made to use the exact same implementation for each test case on all the test setups.

8.3 Throughput comparison

In this section the throughput is compared between different setups. The data is presented in a way that shows what the increase or decrease is when using one setup instead of another. For example, if setup A has a throughput of 100% and setup B has a throughput of 80%, then system B is 20% slower than system A. The following formula is used for this calculation:

$$\text{difference in percentage} = \frac{\text{throughput of setup B}}{\text{throughput of setup A}} - 1 \tag{8.1}$$

The first comparison is between PCI passthrough and SR-IOV. The motivation behind this comparison is to see if there is a performance penalty between using the PF and the VF of a NIC. Therefore the comparison is focused on the virtualization capabilities of the NIC.

The second comparison is between SR-IOV and Open vSwitch with DPDK. This comparison is the focus of this thesis since both represent a very different approach of solving the same problem. These are also the more flexible solutions which make the system more orchestratable.

8.3.1 PCI passthrough and SR-IOV

The throughput measurements are mostly the same between PCI passthrough and SR-IOV (Figure 8.5). The Load distributor and Buffering test case show a decrease of throughput for the smallest packet sizes between 1% and 15%.

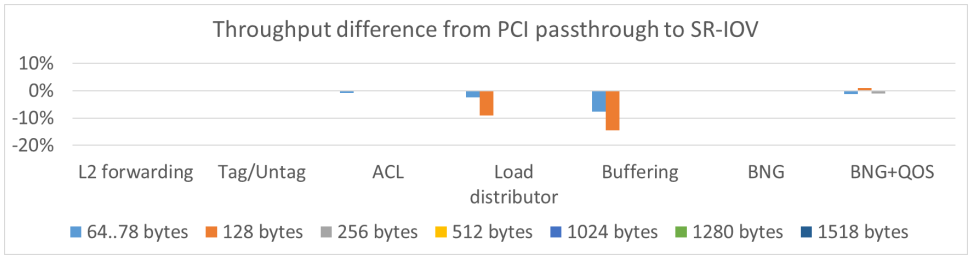

Figure 8.5: Throughput difference between PCI passthrough and SR-IOV.

Since both test setups show almost the same performance it can be assumed that the resources on the NIC are shared in a best effort way. This allows the most optimal use of the NICs resources since a single VF has can use the maximum performance as long as there is no other activity. But it can also cause unpredictable behavior when multiple VMs are connected to VFs on a single NIC port.

8.3.2 SR-IOV and Open vSwitch + DPDK

SR-IOV and Open vSwitch represent 2 very different approaches to providing a data path to the NIC. SR-IOV has direct hardware access and therefore is not using CPU or memory for this transfer. Open vSwitch runs in software on the host system and uses CPU cores and memory.

For the smallest packet sizes there is an overall reduction of throughput between 60% and 85%. For packet sizes of 1024 bytes and bigger the difference is never higher than 20%.

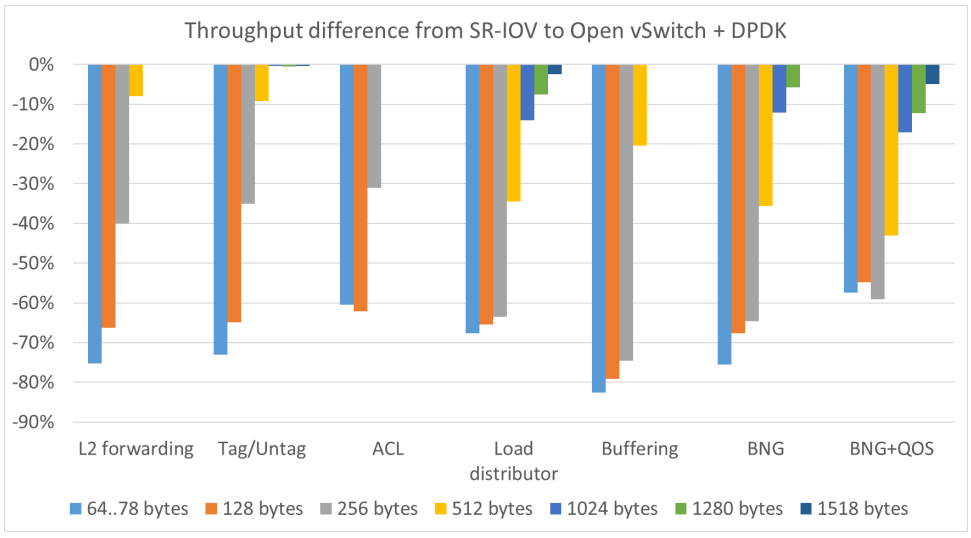

Figure 8.6: Throughput difference between SR-IOV and Open vSwitch with DPDK.

The conclusion can be made that depending on the traffic mix of the use case the performance difference becomes negligible. If the traffic mix mostly consists of packets which are equal or larger than 1024 bytes, then the performance penalty is less then 20% for all use cases. For some test cases the throughput of the different test setups are equal to each other for these packet sizes.

On the other hand there when using Open vSwitch some of the system resources need to be sacrificed. For example, part of the CPU cores are allocated to Open vSwitch. In this test setup the system is using a CPU with 14 hyperthreaded cores. 6 cores are assigned to Open vSwitch which leaves 8 cores which can be used by the VM. This results in a reduction of available CPU cores for the VM of almost 43%.

8.4 Latency comparison

The average latency was measured for the layer 2 forwarding test case with a packet size of 64 bytes using PROX as traffic analyzer. The latency is measured as the time difference between generating a packet and receiving this packet at the tester. Graph 8.7 shows measurements at different speeds. The measurements start at 1% of line rate and increase to 100% with steps of 1%.

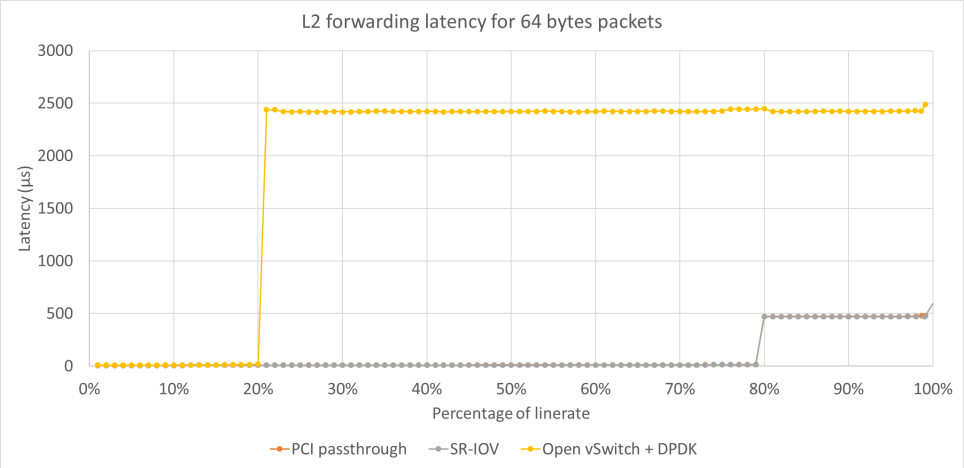

Figure 8.7: Latency measurements for traffic generated between 1% and 100% with intervals of 1%.

The results for the PCI passthrough and SR-IOV test setups overlap each other which makes the measurements for PCI passthrough barely visible. This implies that both test setups behave similar under normal conditions and in overloaded conditions. The results for the Open vSwitch with DPDK test setups show a much larger increase when the system is overloaded.

When the system is overloaded there is a very large increase of latency which makes the data under normal conditions barely visible. The latency increase under overloaded conditions is unacceptable for all the test setups. Therefore it is very desirable to avoid these conditions. Figure 8.8 only shows the data when the system is not overloaded.

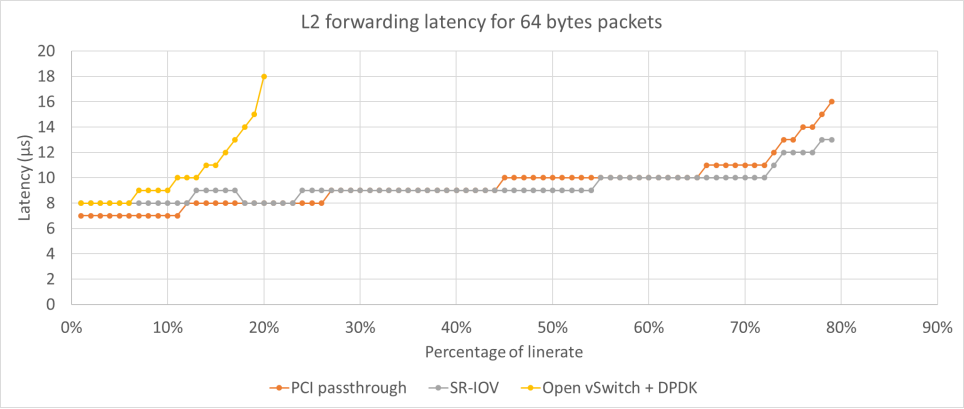

Figure 8.8: Latency measurements before the system is overloaded.

When traffic is generated below 10% of line rate all the test cases show a very similar latency. Between 10% and 20% the Open vSwitch with DPDK test setup starts reaching its maximum throughput which

causes the latency to increase. The PCI passthrough and SR-IOV test setups show a very similar behavior for all the throughput values.

To use the system at its full potential it is desirable to run it at the maximum throughput. Table 8.2 shows the minimum, average and maximum latency at this point. It shows some remarkable results. The SR-IOV test setup has a lower average and maximum than the PCI passthrough test setup. Also Open vSwitch with DPDK has a lower maximum than the other test setups.

	min (μs)	**avg (μs)**	**max (μs)**
PCI passthrough	8	16	94
SR-IOV	8	13	85
Open vSwitch + DPDK	14	18	43

Table 8.2: Latency at the point each system reaches the maximum throughput.

8.5 Conclusion

In this chapter the performance of 3 different approaches for creating a NFV data path between the NIC and the VM have been compared. The first 2 approaches use a similar technique called PCI passthrough, first without SR-IOV and then with the use of SR-IOV. These techniques are already used by telecommunication operators.

The third approach uses a software based data path where the NIC and VM are connected to a virtual switch. This technique is still experimental but it supports certain features which make it an attractive alternative.

With PCI passthrough there is no big difference between when SR-IOV is used or not. The throughput shows only a small (less than 15%) difference under a few conditions. Most of the test cases show the exact same throughput for all the packet sizes. Also there is no big difference between the latency measurements. When SR-IOV is used, there even is a small improvement.

When a virtual switch is used there is big difference compared with SR-IOV. The comparison is done using *Open vSwitch with DPDK* as the virtual switch. For the smallest packets sizes there is a reduction of throughput between 60 and 85%. When using larger packet sizes there is a much smaller reduction of throughput. For packet sizes of *1024 bytes and larger* the reduction is less than 20%. Therefore Open vSwitch with DPDK is mostly useful for use cases with a traffic mix where most of the packets are 1024 bytes or larger. On the other hand, the use of a virtual switch requires CPU cores and memory. In this case almost 43% of the CPU cores are assigned to Open vSwitch.

There is no big difference between the latency measurements of SR-IOV and Open vSwitch with DPDK. Therefore the main consideration for choosing the most appropriate technology is the maximum throughput.

The answer to the *third research question* is as follows: Open vSwitch with DPDK can be used instead of SR-IOV. But it has limitations which include a reduced performance especially for packets smaller than 1024 bytes. It also requires resources on the host which reduces the potential complexity of the VNFs and the amount of VNFs on a single host.

Chapter 9

Conclusion and discussion

9.1 Open vSwitch performance evaluation

Plain Open vSwitch uses the native networking stack of the OS. This results in a throughput below 1 million packets per second for all packet sizes[14]. For packets with a size of 64 bytes this would mean that the throughput is less than 2%, for the packets with a size of 1518 bytes that would be no more than 30%. This means that the system is never reaching line rate even for the largest packet sizes. Using Open vSwitch with DPDK as the fast data path allows an increase of throughput of more than 15 times[14].

The performance evaluation is based on the results in chapter 8 which are about the throughput and latency of a VNF running in a VM on a compute node. This includes the data path from the NIC to the VM, the VM itself and the data path back to the NIC. The results from using PCI passthrough without SR-IOV and with SR-IOV are in most cases the same. Only a few workloads show a difference which is still less than 15%. Therefore based on these measurements there is no real performance penalty when using SR-IOV .

Open vSwitch with DPDK on the other hand shows a big difference in throughput compared to SR-IOV. The smallest packet sizes show a decrease of more than 60% in most cases, but packet sizes of 1024 bytes and larger never show a difference bigger than 20%. Therefore Open vSwitch with DPDK can best be considered as an alternative in use cases where the traffic mix mostly consists of packets of 1024 bytes and larger.

There is a small difference between the latency when Open vSwitch with DPDK is used and SR-IOV under normal conditions but both are in the same magnitude. Under overload conditions Open vSwitch with DPDK shows a much larger latency than SR-IOV but this should be avoided at all times.

Another consideration to be made when using Open vSwitch with DPDK is that resources need to be assigned to the virtual switch which is not the case with SR-IOV. One of these resources is a number of CPU cores on the host. In the test setup used for this thesis that was 6 cores, leaving 8 cores for the VM. This is a reduction of almost 43% which limits the amount of VNFs and the complexity of each VNF. Although the throughput decrease is more acceptable for packets of 1024 bytes and larger, this should also be considered when choosing the most suitable technology.

Since resources need to be allocated to the virtual switch the question can be posed, can functionality from the virtual switch be offloaded to the NIC. This can partly be done with an Intel Ethernet X710 Controller[9] which allows network virtualization offloads like VXLAN. Our prediction is that NFV use cases will require more offloads features, like OpenFlow, which provide improved performance and accelerate the adoption of virtual switches and VNFs. These offload features would be very useful in data centers which are specifically designed for service providers, which can also be called *NFV data centers*. These kind of data centers are exclusively used for deploying VNFs which create network services.

9.2 Research questions

The research question is divided into 3 specific questions and 1 general question. The 3 specific questions relate to the current issues with the data path between the NIC and the VM. The answer to these questions are the following:

1. *Why is it necessary to optimize the data path between the NIC and the VM?*
   The default data path used by Openstack uses the network stack of the OS. This network stack is

not well suited for environments that require a high performance. Using 4x 10Gbit/s interfaces, the performance of the throughput varies between 1% and 30% of line rate depending on the packet size.

2. *Why is the current technology, SR-IOV with PCI passthrough, insufficient? Is an alternative really necessary?*
   SR-IOV and/or PCI passthrough have certain shortcomings. The most important ones are the lack of support for *live migration*, the requirement for the hardware driver which breaks generality and no support for OpenFlow. OpenFlow can be an important factor in interconnecting VNFs. At this moment the SR-IOV acts as a L2 switch which is bound to Ethernet traffic and has very limited control capabilities.

3. *Is Open vSwitch with DPDK a valid alternative? If so, what are the shortcomings?*
   Open vSwitch with DPDK is an alternative but the following considerations should be made. The throughput is much lower when processing small packet sizes. The difference becomes negligible with a traffic mix which consists of packets with a size of 1024 bytes and larger. Also the operator should consider the loss of resources on the compute node. In the configuration used in this thesis 6 of the 14 CPU cores where used by the virtual switch leaving 8 left for the VNF. This both reduces the maximal amount of VNFs on a single node and the potential complexity of the VNF.

Then as a final conclusion the following general question is answered: *How can a virtualized environment (based on Openstack) be used by telecommunication operators to deploy their network services?*
A cloud platform like Openstack, which heavily depends on virtualization, can be used by telecommunications operators but it requires some modifications in order to handle the performance requirements. The hypervisor needs to expose hardware features to the VM, these include CPU pinning, NUMA awareness and huge pages. CPU pinning is a way to prevent applications from disturbing each other. But NUMA awareness and huge pages are specifically aimed at VNFs that are based on the DPDK library since this is mostly used in the industry. Also the data path needs to be optimized, at this moment PCI passthrough or/and SR-IOV are the most used solutions. These properties are self-evident in the context of NFV but they heavily break with the cloud model. The cloud model states that hardware needs to be fully abstracted from the VM. Therefore Openstack initially didn't expose these features. Because of the increasing demand and the release of NFV example architectures, the Openstack community started to implement these features which are marketed as Enhanced Platform Awareness (EPA)[10]. It is likely that eventually Openstack will also support Open vSwitch with DPDK.

The work in this thesis is mainly focused on the Virtualization of compute resources. Section 6.4.1 describes that SDN technology, like OpenFlow, can be used to interconnect VNFs. This works the best if the entire data path between all the VNFs is programmable, including networking resources. Therefore, from a network perspective there is also the need for OpenFlow enabled ToR switches to be able to deploy network services. By default Openstack doesn't require OpenFlow enabled ToR switches.

9.3 NFV data center

Traditional cloud applications are focused on delivering application level services. These services are focused on the content of the data and not the packet headers. An application level request is usually handled by a single machine. Therefore the performance cost is isolated to a single VM.

An NFV data center is concerned with the forwarding part of the data, for an internet service provider this is the L2, L3 and L4 part of the networking stack. In a NFV data center it is the goal to replace hardware appliances by VMs. This transition is aimed to have as little performance degrade as possible, therefore the virtual infrastructure has special requirements.

Traffic going through a VNF is often part of a chain with multiple VNFs. These chains are identified by ETSI as forwarding graphs. Instead of one VM handling a request there are many VMs which process data, therefore the performance cost is being aggregated. For example, if a single VNF drops data this means that all the previous VNFs have done useless processing.

This could be seen as follows: traditional data centers *own* the data while NFV data centers transport the data from point A to point B. The architecture for traditional data centers therefore includes replication from data to prevent loss. This is not a requirement for NFV data centers since they don't own the data but provide end-to-end connections.

The required time for handling data is significantly different in telecommunication environments. For example, a traditional data center could be used to host a web server. The time needed to respond to a

web request varies a lot but is measured in the order of milliseconds or seconds. In general, a respond time of 0.1 second is considered an instantaneous reaction[27]. The expected delay of an networking appliance is much lower. A hardware assisted switch is expected to have a latency of around 25 microseconds[3].

9.4 NFV evolution

NFV is emerging as an alternative for dedicated hardware appliances. A number of use cases have already been identified as applicable for using NFV technology. Virtual customer edge and mobile packet core are technologies which are already moving into cloud environments[1]. One example is a PoC for virtual CPE technology by Telefonica[39]. Figure 7.5 shows the role of such a CPE in an ISP network. Other examples include virtual IMS[2], virtual EPC[3], virtual Gi-LAN[13].

Each use case is addressed individually since the throughput, latency and orchestration requirements can be different. By studying new use cases telecommunication operators can identify potential for SDN and NFV to design architectures. There needs to be a tight collaboration between telecommunication operators and hardware vendors to explore how COTS hardware can be used in these environment. And maybe even add features to the hardware to create better support.

[1] https://www.sdxcentral.com/sdn-nfv-use-cases/
[2] https://www.ericsson.com/ourportfolio/products/ims?nav=productcategory002
[3] https://networks.nokia.com/solutions/packet-core

Appendix A

Nederlandstalige samenvatting

A.1 NFV achtergrond

Door de evolutie van technologie moeten telecomoperatoren constant hun infrastructuur veranderen om netwerkdiensten te voorzien. Zo is bijvoorbeeld de vraag naar mobiel internet drastisch gestegen sinds smartphones populair zijn geworden. Hiervoor is nieuwe apparatuur nodig die de nodige functionaliteit ondersteunt. Met andere woorden, om nieuwe netwerkdiensten uit te rollen is nieuwe hardware nodig.

Deze evolutie verloopt steeds sneller waardoor telecomoperatoren op zoek zijn gegaan naar alternatieven op het steeds aankopen en installeren van nieuwe hardware. Sinds enkele jaren is er veel aandacht gegaan naar cloud technologie in data centers. Hierbij wordt gebruik gemaakt van virtualisatie technologieën om hardware los te koppelen van software. De voordelen van een gevirtualiseerde omgeving voor telecomoperatoren zijn de volgende:

Sneler uitrollen van netwerkdiensten Bij traditionele netwerkdiensten is de functionaliteit deel van de hardware. In cloud omgevingen wordt de functionaliteit in software voorzien. Hierdoor is het gemakkelijker om nieuwe diensten uit te rollen.

Dynamisch schalen Met behulp van virtualisatie kan de capaciteit van een netwerkdienst snel en dynamisch aangepast worden.

Flexibele plaatsing van netwerkfuncties De fysieke plaats van traditionele netwerkfuncties is vast. Met behulp van virtualisatie kan een netwerkfunctie op eender welke machine geplaatst worden.

Uitbereiding van de infrastructuur Doordat de netwerkfuncties gemakkelijk verplaatst kunnen worden is het ook gemakkelijker om de infrastructuur uit te breiden.

De evolutie van traditionele hardware naar gevirtualiseerde omgevingen is ook onder de aandacht gevallen van het European Telecommunications Standards Institute (ETSI) in 2012. ETSI is een organisatie die standaarden voor telecommunicatie technologie voorziet en heeft een aparte sectie over Network Functions Virtualization (NFV).

Figuur A.1 toont het NFV architectural framework[16] zoals het beschreven staat door ETSI. Het doel is de elementen identificeren die deel uitmaken van een virtuele infrastructuur voor netwerkdiensten. De virtualisatielaag zorgt er voor dat Virtuele NetwerkFuncties (VNF) op eender welke machine gehost kunnen worden. Dit zorgt ervoor dat er ook nieuwe uitdagen zijn rond management en orchestratie. De elementen die hier deel van uit maken staan rechts op de figuur.

A.2 Data Plane Development Kit

De meeste besturingssystemen hebben een eigen netwerkstack die bepaalde netwerkfuncties voorziet. Door de opkomst van netwerkkaarten van 10Gbit/s en zelfs 40Gbit/s is deze netwerkstack ontoereikend. Het verwerken van pakketten vereist meerdere kopieën en het interrupt mechanisme is onderhevig aan het interrupt livelock fenomeen[15].

De Data Plane Development Kit (DPDK) bestaat uit een set van libraries die het onder andere mogelijk maken om de netwerk stack van het besturingssysteem te omzeilen. Het kopiëren van pakketten

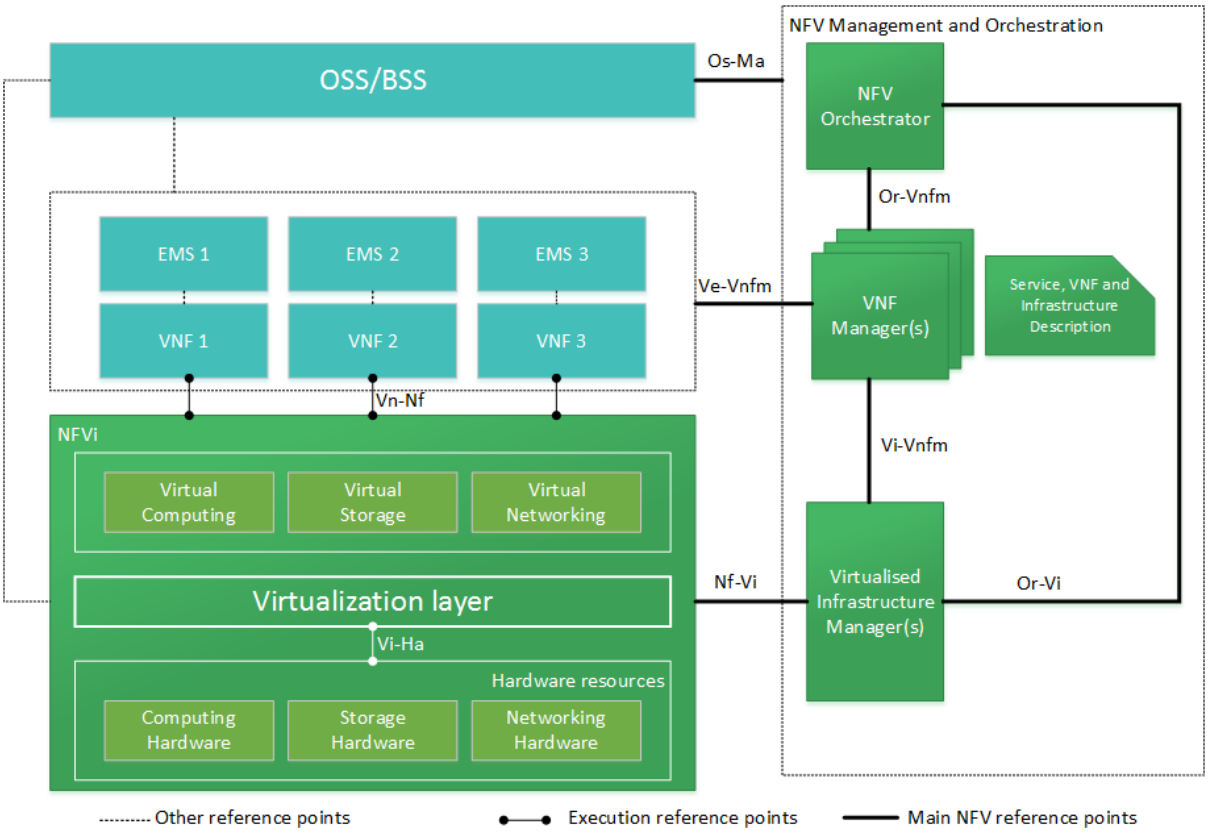

Figure A.1: NFV reference architecture die door ETSI beschreven wordt[16]. Het NFV Management and Orchestration gedeelte is uniek voor het beheren van een gevirtualiseerde infrastructuur.

wordt vermeden door pointers door te geven naar de verschillende taken die het pakket verwerken. Interrupts worden vermeden door gebruik te maken van een poll mode driver die via een polling mechanisme pakketten in bulks verwerkt. Daarnaast wordt geheugen op voorhand gealloceerd in huge pages, op deze manier worden Translation Lookaside Buffer (TLB) misses voorkomen. De communicatie tussen de NIC en de verschillende taken onderling gebeurt via lockless rings om de overhead van locks te vermijden.

Doordat DPDK gebruik maakt van deze technieken om een zo goed mogelijke performance te krijgen, wordt het in deze thesis gebruikt om de NFV infrastructuur te evalueren.

A.3 Openstack

Data centers bestaan vaak uit honderden of duizenden servers. Met behulp van virtualisatie is het mogelijk om een Virtuele Machine (VM) te lanceren op 1 van deze servers. Met behulp van Openstack kan de meest geschikte machine geselecteerd worden om een VM op te hosten. Daarnaast kunnen met behulp van Openstack ook virtuele netwerken gemaakt worden om enerzijds verbinding tussen VM's te voorzien en anderzijds verbinding met het internet en verbinding van buitenaf.

Openstack heeft zichzelf als 1 van de meest geschikte platformen voor NFV gepresenteerd. Het platform kan de rol van Virtual Infrastructure Manager (VIM) vervullen. Aangezien de meeste NFV use cases heel gevoelig zijn voor performantie, is het nodig om bepaalde delen van het platform te optimaliseren. De delen die de performance het meeste beïnvloeden zijn het data path tussen de NIC en de VM, en de hypervisor. Voor een optimale performantie moet de hypervisor de volgende hardware eigenschappen onthullen aan de VM:

CPU topologie Door de onderliggende topologie van CPU sockets, CPU cores en hyperthreads te onthullen aan de VM, kan ze gebruik maken van deze informatie om de performantie te verbeteren[26].

CPU pinning Normaal gezien worden virtuele CPU cores dynamisch gekoppeld aan een fysieke CPU core. Hierdoor kan het zijn dat een virtuele CPU core gedeeld kan worden met een andere virtuele

CPU core. Door virtuele CPU cores te pinnen aan fysieke CPU cores, wordt voorkomen dat ze gedeeld worden.

huge pages Wanneer in de VM een DPDK applicatie wordt gebruikt, is het aangewezen om gebruik te maken van huge pages. Deze moeten eerst gealloceerd worden op de compute node en kunnen vervolgens onthuld worden aan de VM.

NUMA awareness Sommige moederborden hebben meerdere CPU sockets. Voor deze moederborden kan de performantie verschillen van PCI devices en RAM geheugen voor de verschillende CPU's. Ieder PCI device, geheugenkanaal en CPU socket behoort tot een Non-uniform memory access (NUMA) node. DPDK applicaties kunnen deze informatie gebruiken en zoveel mogelijk proberen om PCI devices en geheugenkanalen te gebruiken die bij de NUMA node van een CPU behoren.

A.4 NFV data path

Wanneer een VM wordt gelanceerd op een compute node, dan zijn er 2 methodes om een data path te creëren met de NIC: Met behulp van een virtuele switch of door middel van directe toekenning van hardware. De virtuele switch is een software applicatie die aan NIC staat gekoppeld en aan de VM. Hierdoor is het mogelijk om data tussen de VM en de NIC te versturen. Directe toekenning van hardware creëert een direct data path tussen de VM en de NIC met behulp van PCI passthrough of/en SR-IOV.

A.4.1 PCI passthrough

Om de beperkingen van een virtuele switch te vermijden is het mogelijk om de NIC rechtstreeks aan de VM te koppelen (Figuur A.2), dit wordt PCI passthrough genoemd. Hierdoor heeft de VM de volledige controle over de NIC en worden de performantieproblemen met een virtuele switch vermeden. Anderzijds is het niet mogelijk om een NIC te delen met meerdere VM's. Met andere woorden, als er meerdere VM's zijn die ieder een direct data path naar een NIC willen hebben, dan moet voor iedere VM een aparte NIC aanwezig zijn.

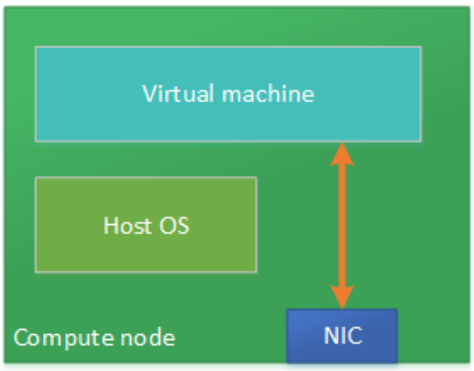

Figure A.2: Visuele weergave van een VM met PCI passthrough. De VM kan het PCI device gebruiken zonder dat de host moet ingrijpen.

A.4.2 SR-IOV

Single Root I/O Virtualization (SR-IOV) is een hardware eigenschap op het PCI device dat ervoor zorgt dat 1 PCI device als meerdere virtuele PCI devices verschijnt. Deze virtuele PCI devices worden Virtual Functions (VF) genoemd. Het normale PCI device wordt de Physical Function (PF) genoemd. Een PCI device kan meerdere VF's hebben en maar 1 PF. Op deze manier kan een NIC gedeeld worden met meerdere VM's door iedere VM een VF te geven (Figuur A.3). De NIC gedraagt zich vervolgens als een L2 switch om te bepalen waar het dataverkeer naartoe moet.

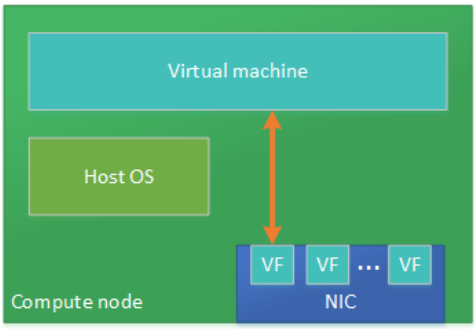

Figure A.3: Visuele weergave van een VM met PCI passthrough van een VF. Door meerdere virtuele functies te maken is het mogelijk om een PCI device met meerdere VM's te delen.

A.4.3 Open vSwitch

Openstack maakt gebruikt van een virtuele switch om VM's met elkaar te verbinden en met de NIC binnen een compute node. Open vSwitch is één van de meest populaire opensource switch beschikbaar. Ze ondersteunt vele technologieën die nuttig zijn voor het virtualiseren van netwerken, zoals bijvoorbeeld OpenFlow.

Normaal gebruikt Open vSwitch de netwerkstack van het besturingssysteem om pakketten te verwerken. In traditionele cloud omgevingen is dit voldoende maar voor de meeste NFV use cases is de performantie ontoereikend. Daarom is het mogelijk om de netwerkstack van het besturingssysteem te vervangen door een DPDK applicatie (Figuur A.4).

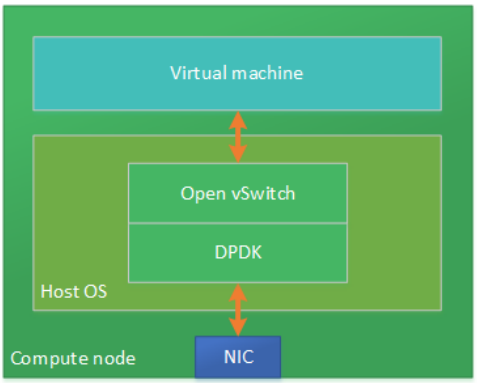

Figure A.4: Visuele weergave van Open vSwitch met DPDK om het data path te accelereren.

A.5 NFV data path test setup

Om de verschillende test setups met elkaar te kunnen vergelijken is de throughtput gemeten. Deze is gemeten met verschillende pakketgroottes: 64, 128, 256, 512, 1024 en 1518 bytes. Throughput wordt gedefinieerd als het maximaal aantal pakketten dat verwerkt kan worden zonder verlies. Daarom is een zoekalgoritme nodig om dit punt te vinden. De testen in deze thesis zijn uitgevoerd met een binair zoekalgoritme. De tester begint met pakketten te versturen aan 100% van line-rate. Indien dit lukt eindigt de test, anders stuurt de tester aan de helft van de snelheid. Vervolgens blijft de tester halveren tot het verschil van 2 uitvoeringen minder dan 1% is.

De throughput is gemeten voor 7 verschillende test cases die ieder een bepaalde categorie van netwerkfuncties representeren. Iedere test case genereert een unieke belasting van het systeem, waardoor verschillende componenten worden gemeten. De 7 test cases zijn de volgende:

Layer-2 forwarding Deze test case vervangt enkel het MAC adres van de doelbestemming.

Tag/Untag Bij deze test case wordt voor 2 taken een MPLS tag van 4 bytes toegevoegd aan de header. Voor de andere 2 taken wordt een MPLS tag gestript.

ACL De pakketten in deze test case worden opgezocht in een ACL tabel.

Load distributor Deze test case maakt gebruik van een hash tabel om pakketten te distribueren over de verschillende uitgaande poorten. Deze hash tabel is groter dan de cache ruimte en zorgt daarom voor cache misses.

Buffering De pakketten worden 125ms gebufferd in deze test case.

BNG De laatste 2 test cases representeren een meer realistisch scenario genaamd Border Network Gateway (BNG).

BNG+QOS De laatste test case voegt Quality of Service (QOS) aan de vorige test case.

A.6 NFV data path testresultaten

De test cases zijn uitgevoerd met de kleinst mogelijke pakketgrootte en pakketgroottes van 128, 256, 512, 1024, 1280 en 1518 bytes De kleinst mogelijke pakketgrootte hangt af van de test case, tabel A.1 toont deze voor iedere test case. De testresultaten zijn gegroepeerd volgens test case waarbij iedere kleur een verschillende pakketgrootte representeert. De test cases zijn allemaal uitgevoerd met 4x 10Gbit/s netwerkpoorten behalve de buffering test case. De buffering test case is uitgevoerd met 1x 10Gbit/s netwerkpoort.

Test case	**kleinst mogelijke pakketgrootte (bytes)**
L2 forwarding	64
Tag/Untag	68
ACL	64
Load distributor	64
Buffering	64
BNG	78
BNG+QOS	78

Table A.1: Kleinst mogelijke pakketgrootte voor de verschillende test cases.

A.6.1 PCI passthrough throughput

Figuur A.5 toont de throughput voor de PCI passthrough test setup. De meeste test cases tonen een lagere throughput voor de kleinere pakketgroottes, maar vanaf pakketten van 256 bytes en groter bereiken ze bijna allemaal 100% van line rate.

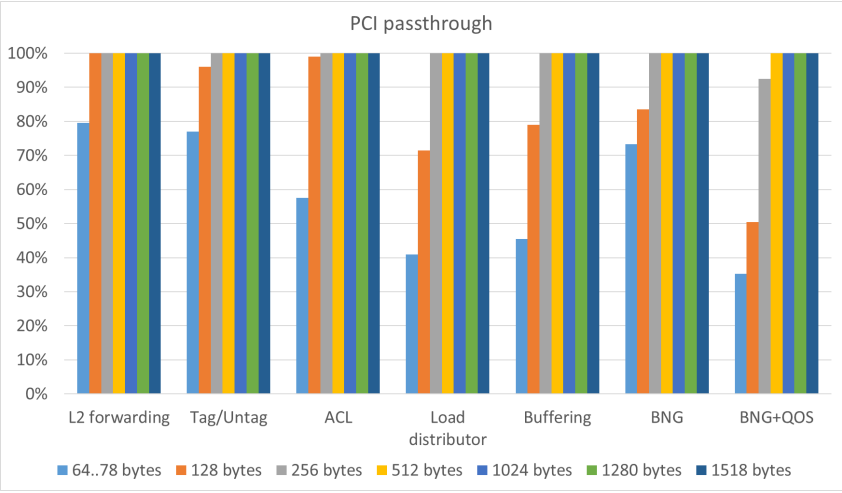

Figure A.5: PCI passthrough testresultaten. De throughput wordt weergegeven in percentage van line-rate. Iedere kleur representeert een verschillende pakketgrootte.

A.6.2 SR-IOV throughput

Figuur A.6 toont de throughput voor de SR-IOV setup. De resultaten zien er heel gelijkaardig aan de PCI passthrough resultaten uit. Sectie A.6.4 toont een gedetailleerde vergelijking.

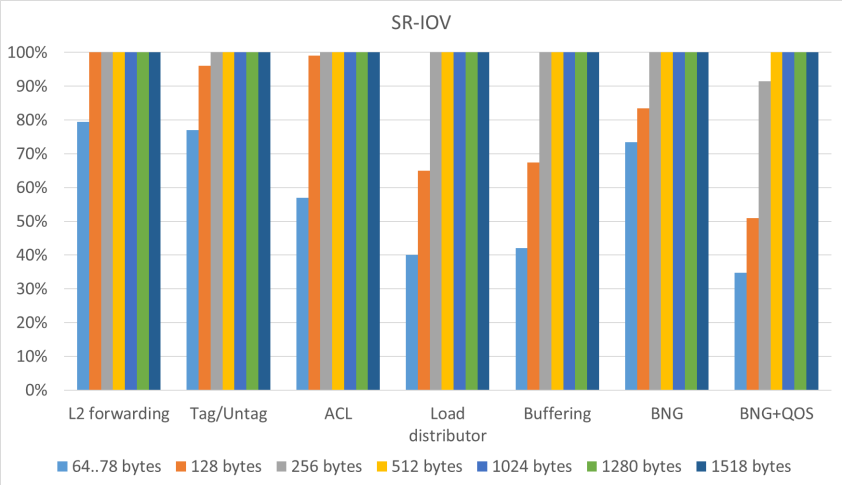

Figure A.6: SR-IOV testresultaten. De throughput wordt weergegeven in percentage van line-rate. Iedere kleur representeert een verschillende pakketgrootte.

A.6.3 Open vSwitch met DPDK throughput

De resultaten voor Open vSwitch met DPDK zijn zichtbaar in figuur A.7. De throughput voor pakketgroottes kleiner dan 256 bytes komen nooit boven 40%. De resultaten van 256 bytes en 512 bytes tonen ook een lage throughput, zeker voor de laatste 4 test cases. Vanaf pakketgroottes van 1024 bytes en groter is de throughput voor alle test cases meer dan 80% van line-rate.

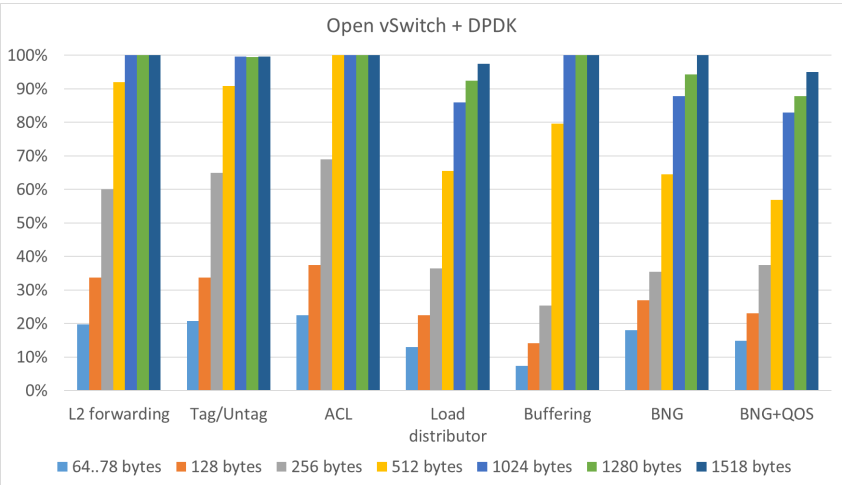

Figure A.7: Open vSwitch met DPDK testresultaten. De throughput wordt weergegeven in percentage van line-rate. Iedere kleur representeert een verschillende pakketgrootte.

A.6.4 Throughput vergelijking

De vergelijking is gemaakt door de groei of vermindering van throughput wanneer systeem B in plaats van systeem A wordt gebruikt. De volgende formule wordt hiervoor gebruikt:

$$\text{verschil in percentage} = \frac{\text{throughput van setup B}}{\text{throughput van setup A}} - 1 \tag{A.1}$$

De throughput metingen tussen PCI passthrough en SR-IOV zijn bijna altijd hetzelfde. Voor een paar gevallen is er een lichte daling van 15% maar over het algemeen is er geen echt verschil.

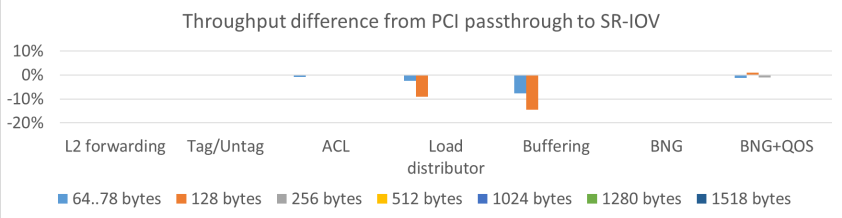

Figure A.8: Throughput verschil tussen PCI passthrough en SR-IOV.

Wanneer Open vSwitch met DPDK wordt gebruikt in plaats van SR-IOV is er wel een grote vermindering. Voor pakketten met een grootte kleiner dan 1024 bytes is het verschil het grootste. Voor pakketten van 1024 bytes en groter is het verschil nooit groter dan 20%. Hieruit kunnen we concluderen dat Open vSwitch met DPDK vooral een alternatief vormt voor een traffic mix met vooral pakketten van 1024 bytes en groter.

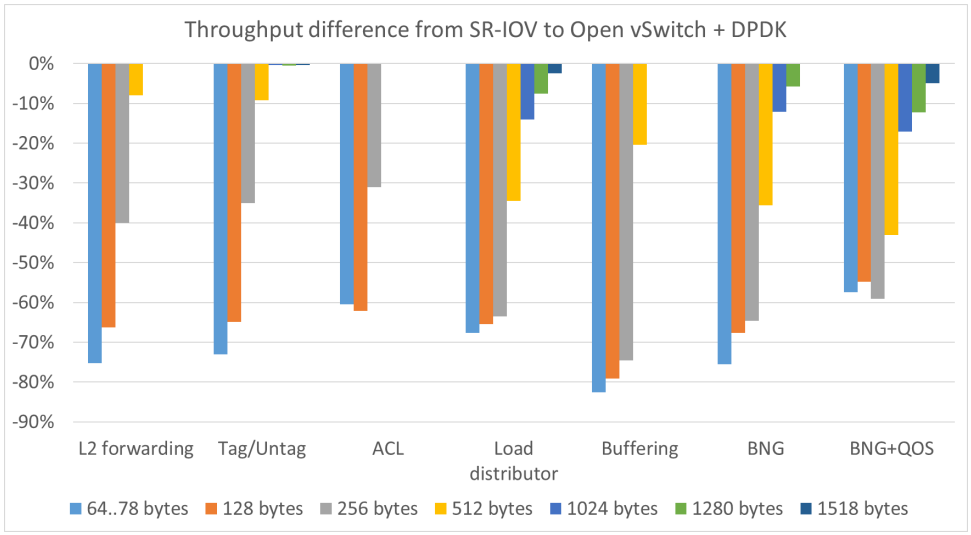

Figure A.9: Throughput verschil tussen SR-IOV en Open vSwitch met DPDK.

A.6.5 Latency vergelijking

The gemiddelde latency werd gemeten voor de L2 forwarding test case met een pakketgrootte van 64 bytes. The metingen starten op 1% van line rate en verhogen tot 100% met intervallen ven 1%. Figuur A.10 toont de metingen. De resultaten voor PCI passthrough en SR-IOV overlappen elkaar waardoor de resultaten van PCI passthrough amper zichtbaar zijn. Open vSwitch met DPDK toont vooral een veel grotere latency wanneer het systeem overbelast wordt.

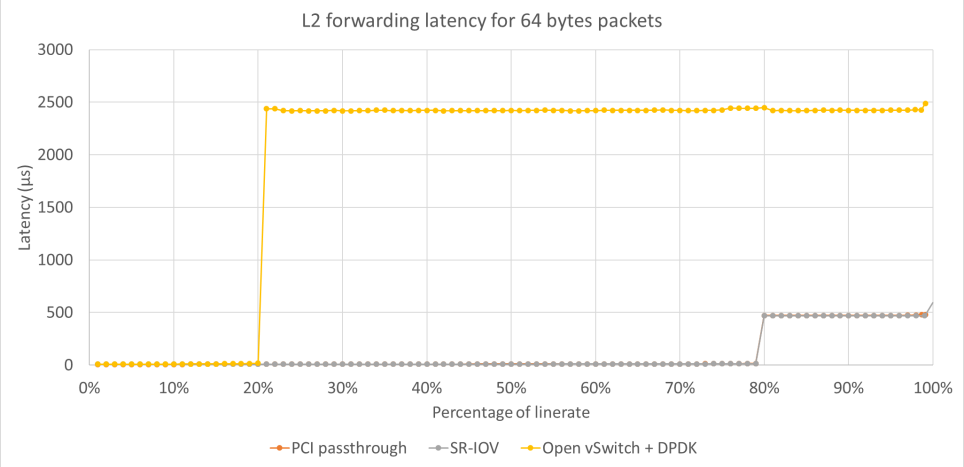

Figure A.10: Latency metingen voor data gegenereerd tussen 1% en 100% met intervallen van 1%.

Wanneer het systeem overbelast wordt, dan is de latency onaanvaardbaar voor alle test cases. Daarom is aan te raden om deze condities te vermijden. Figuur A.11 toont de latency voordat het systeem overbelast wordt voor iedere test setup. De latency op het moment voordat het systeem overbelast wordt, is voor alle test setups vergelijkbaar. Het voornaamste criterium voor de keuze tussen deze technologieën is dus de throughput.

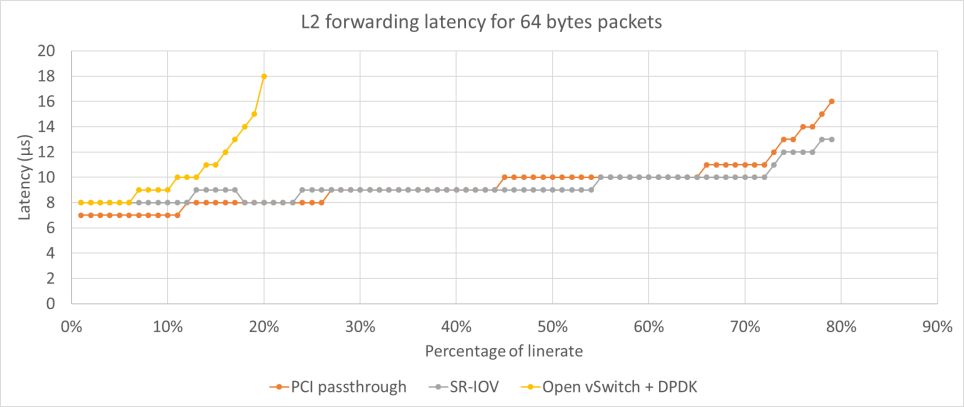

Figure A.11: Latency metingen voordat het systeem overbelast wordt.

A.7 Conclusie

Een platform zoals Openstack kan gebruikt worden door telecomoperatoren indien enkele aanpassingen worden gedaan om de performantie condities aan te kunnen. De hypervisor moet bepaalde hardware eigenschappen onthullen aan de VM zoals CPU pinning, huge pages en NUMA awareness. Door CPU cores te pinnen, wordt vermeden dat VM's elkaar storen. Huge pages en NUMA awareness zijn technieken die vooral nuttig zijn voor VNF's die gebruik maken van de DPDK library. Daarnaast is het nodig om het data path tussen de VM en de NIC te optimaliseren. Op dit moment zijn PCI passthrough en SR-IOV de meest gebruikte oplossingen. Deze eigenschappen zijn vanzelfsprekend in de context van NFV maar staan loodrecht op het cloud paradigma dat zegt dat hardware volledig geabstraheerd moet zijn van de VM. Omwille van de groeiende vraag van de telecom sector heeft de Openstack gemeenschap deze eigenschappen toegevoegd aan het platform.

Open vSwitch met DPDK is een virtuele switch die OpenFlow ondersteunt, Live migration toelaat en virtio drivers ondersteunt. Deze eigenschappen maken het een aantrekkelijk alternatief op PCI passthrough en SR-IOV. Op dit moment wordt Open vSwitch met DPDK nog als een experimentele technologie beschouwd en is het nog niet toegevoegd door de Openstack gemeenschap. Uit de testen kan geconcludeerd worden dat Open vSwitch met DPDK vooral een aantrekkelijk alternatief is voor een traffic mix met pakketgroottes van 1024 bytes en groter.

Appendix B

Bibliography

[1] Sophia Antipolis. New etsi group develops open source for nfv. 2016.

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[3] Inc. Cisco Systems. Design best practices for latency optimization, 2007.

[4] Intel Corporation. Intel 82599 sr-iov driver companion guide, 2010.

[5] Intel Corporation. Intel virtualization technology for directed i/o, 2010.

[6] Intel Corporation. Impressive packet processing performance enables greater workload consolidation, 2012.

[7] Intel Corporation. Open vswitch* enables sdn and nfv transformation, 2014.

[8] Intel Corporation. Intel ethernet converged network adapter xl710 10/40 gbe, 2015.

[9] Intel Corporation. Intel ethernet converged network adapters x710 10/40 gbe, 2015.

[10] Intel Corporation. A path to line-rate-capable nfv deployments with intel architecture and the openstack* kilo release, 2015.

[11] Intel Corporation. Pci-sig sr-iov primer: An introduction to sr-iov technology, 2015.

[12] Intel Corporation. End-to-end service instantiation using open-source management and orchestration components, 2016.

[13] Intel Corporation. Evaluating dynamic service function chaining for the gi-lan, 2016.

[14] Intel Corporation. Intel open network platform release 2.1 performance test report, 2016.

[15] Luca Deri et al. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, volume 2004, pages 85–93. Amsterdam, Netherlands, 2004.

[16] ETSI. Network functions virtualisation (nfv); architectural framework, 2014.

[17] ETSI. Network functions virtualisation (nfv); management and orchestration, 2014.

[18] ETSI. Network functions virtualisation (nfv); infrastructure overview, 2015.

[19] José Luis García-Dorado, Felipe Mata, Javier Ramos, Pedro M. Santiago del Río, Victor Moreno, and Javier Aracil. *Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience*, chapter High-Performance Network Traffic Processing Systems Using Commodity Hardware, pages 3–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[21] M Tim Jones. Virtio: An i/o virtualization framework for linux. *IBM White Paper*, 2010.

[22] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[23] Diego R. Lopez. The dataplane ready open source nfv mano stack, 2015.

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[25] Balazs Nemeth. Network infrastructure optimization. Master's thesis, UHasselt, 2014.

[26] Balazs Nemeth, Xavier Simonart, Neal Oliver, and Wim Lamotte. The limits of architectural abstraction in network function virtualization. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 633–639. IEEE, 2015.

[27] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[28] Bruno AA Nunes, Manoel Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys & Tutorials, IEEE*, 16(3):1617–1634, 2014.

[29] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[30] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[31] Luigi Rizzo. Revisiting network i/o apis: The netmap framework. *Queue*, 10(1):30:30–30:39, January 2012.

[32] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 gbit/s line rate packet processing using commodity hardware: Survey and new proposals, 2012.

[33] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[34] Jamal Hadi Salim. When napi comes to town. In *Linux 2005 Conf*, 2005.

[35] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ALS '01, pages 18–18, Berkeley, CA, USA, 2001. USENIX Association.

[36] Jia Song and Jim Alves-Foss. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS)*, 6(4):256, 2012.

[37] Konstantina Tsiamoura. A survey of trends in fast packet processing. *Proceedings zu den Seminaren Future Internet (FI) und Innovative Internet Technologien und Mobilkommunikation (IITM)*, 41, 2014.

[38] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando Martins, Andrew V Anderson, Steven M Bennett, Alain Kägi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[39] Business Wire. Telefnica announces successful proof of concept for sdn/nfv with nec and netcracker. 2015.

[40] Edwin Zhai, Gregory D Cummings, and Yaozu Dong. Live migration with pass-through device for linux vm. In *OLS08: The 2008 Ottawa Linux Symposium*, pages 261–268, 2008.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Performance Evaluation of Data Path Improvements in NFV Openstack Deployments

Richting: **master in de informatica-multimedia**
Jaar: **2016**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

De Langhe, Kris

Datum: **15/09/2016**