Acknowledgments

I am very grateful to my promoter Prof. Dr. Wim Lamotte for presenting me with the opportunity to do an internship with Intel, an opportunity I might not have considered otherwise. I would also like to thank both Prof. Dr. Wim Lamotte and my co-promotor Prof. Dr. Peter Quax for their excellent guidance throughout the work performed during this thesis. I would also like to thank them for reviewing my thesis text and suggesting valuable improvements.

I am also very thankful to the team at Intel which I had the opportunity to be part of from December 2014 to June 2015. They were instrumental in achieving the results presented in this thesis. Being part of the team has been a very educational experience, and I would like to thank the entire team for their support and the time they have taken to answer any questions I had.

Furthermore I would like to thank everyone that was part of the past three exceptional years. Thank you professors, teaching assistants and fellow students for the great time at Hasselt University!

I would also like to thank the Intel team, my father and Jolien Somers for proofreading sections of my thesis text and suggesting improvements.

Finally I would like to thank everyone else who provided support and encouragement during my education.

Contents

Glossary

API	Application Programming Interface
BNG	Broadband Network Gateway
CPE	Customer Premise Equipment
CPU	Central Processing Unit
Cgroup	Control Group
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DPPD	Data Plane Performance Demonstrators
EPT	Extended Page Tables
GRE	Generic Routing Encapsulation
Hugepage	A continuous section of memory that is accessible through a single page entry in the MMU.
IOMMU	Input/Output Memory Management Unit
IP	Internet Protocol
IPC	Inter-Process Communication
IVSHMEM	Inter-Virtual Machine Shared Memory
KVM	Kernel-based Virtual Machine, the hypervisor built into the linux kernel.
MMU	Memory Management Unit
MPLS	Multi-Protocol Label Switching
NFV	Network Function Virtualization
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
PCI BAR	Peripheral Component Interconnect Base Address Registers
PCIe	Peripheral Component Interconnect express
PID	Process ID
QinQ	IEEE 802.1ad, stacking of multiple VLAN tags in a packet
SUT	System Under Test
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UID	User ID
UTS	Unix Time-Sharing
VM	Virtual Machine
VM-exit	Switching the CPU from a virtual machine context to the hypervisor.
VM-resume	Switching the CPU from the hypervisor context to that of a virtual machine.
VNF	Virtualized Network Function
hugetlbfs	A filesystem through which hugepages can be allocated.
root	User with full permissions on a linux system

Chapter 1

Introduction

1.1 Problem statement

Network function virtualization is bringing greater flexibility and easier management to the service provider infrastructure through the usage of software implementations of expensive and inflexible hardware. Virtualized network functions such as routers, content delivery networks and broadband network gateways can be implemented in software and run on traditional computing hardware rather than on special purpose hardware. Since these functions are now available in software they can be virtualized. This will increase flexibility and ease management just like virtualization changed machine management for computing.

Existing research on virtualized network functions has been focused on the use of traditional virtualization using hypervisors, this thesis will explore the benefits and the drawbacks of containers in the Network Function Virtualization setting. Containers allow for application isolation by the operating system such that multiple tenants are unable to influence each other's applications. Our assumptions are that since there is less overhead associated with containers, that performance will resemble that of an application running without any virtualization techniques applied.

Are containers a feasible alternative to hypervisor-based approaches for deploying virtualized network functions? Containers and hypervisors in reality do not have much in common when it comes to the underlying technology and techniques used to achieve a virtualized system. As such, we will be performing a feasibility study in which we will *explore the different network configurations that are actually possible using containers* and on which of these configurations our *assumption that performance will be comparable to native execution* holds. Additionally we will try to *identify significant benefits that come from the usage of containers for use in network function virtualization* over the use of a hypervisor.

This thesis will focus on the use of Docker as it is the most popular tool and has a huge ecosystem around it which includes tools from Microsoft [40], Google [30] and many more [6] [51]. As the competitor we will be using the Kernel-based Virtual Machine (KVM) since there is significant documentation available on the performance and configuration of KVM under network function virtualization loads [81] [44].

1.2 Thesis structure

This thesis is structured in two parts, part I will look into the different possible virtualization techniques and how they work. Part II consists of work and experiments performed when comparing Docker containers with KVM virtual machines.

Chapter 3 will discuss the different types of machine virtualization, where a complete machine is virtualized such that multiple operating system instances can be run inside of these virtual machines. The techniques discussed are traditional *virtual machines* and the *different types of virtual machines* (section 3.1), *unikernels and rump kernels* (section 3.2) and finally the *NoHype architecture* (section 3.3).

The next chapter discusses *containers*, specifically the *components that make up a container* (section 4.1), *different tools to configure and manage containers* (section 4.2.1), *security considerations* (section 4.3) and *issues with containers* (section 4.4).

From here we will continue to discuss how containers can be used and how they perform in practice. We will be comparing containers with the Kernel-based Virtual Machine in chapter 5. In chapters 6 we will be detailing the methods we have used to build container images that are built specifically to run virtualized network functions using the Intel®Data Plane Development Kit to power our packet processing applications.

In addition to these container images we will be detailing the operations that are required to use Docker as the manager for the containers and the actions that have to be taken manually to correct some of Docker's default behaviour.

In chapter 7 we will be able to set up some known virtual network function configurations and test them against their counterparts running on the Kernel-based Virtual Machine.

We end with our conclusions in chapter 8, reflecting on the issues and results of our proof of concept.

Part I

Conceptual

Chapter 2

Context

The goal of this chapter is to provide an overview of the technology that is available to perform virtualization and describe the goals of network function virtualization. Besides the more traditional virtualization using a hypervisor we will discuss other types of virtualization such that we can position the chosen techniques in part II.

2.1 System components

In this chapter we will provide a quick overview of the components that go into a server and how they interact. On most desktop main board the layout of components and interaction is quite simple, there is a single processor which fits in a single socket. The memory that is in the memory slots can be used by the processor without any problems.

A server main board is somewhat more complex, especially those that have multiple processors. For each processor that can be attached to the main board there is a single socket into which the processor can be seated. Each socket has memory banks that are attached locally to that specific socket. Memory attached to the other socket is only accessible through the links that connect the different CPUs, accesses to this memory are slower as they have to traverse the links connecting the CPUs. The same applies for PCIe slots, so we have to be careful where to plug in the cards that we use when we are concerned with the performance of these cards. Figure 2.1 illustrates the layout of the chipset used in our experiments in part II. In linux these sockets are referred to as Non-Uniform Memory Access (NUMA) nodes, each node corresponds to a single socket.

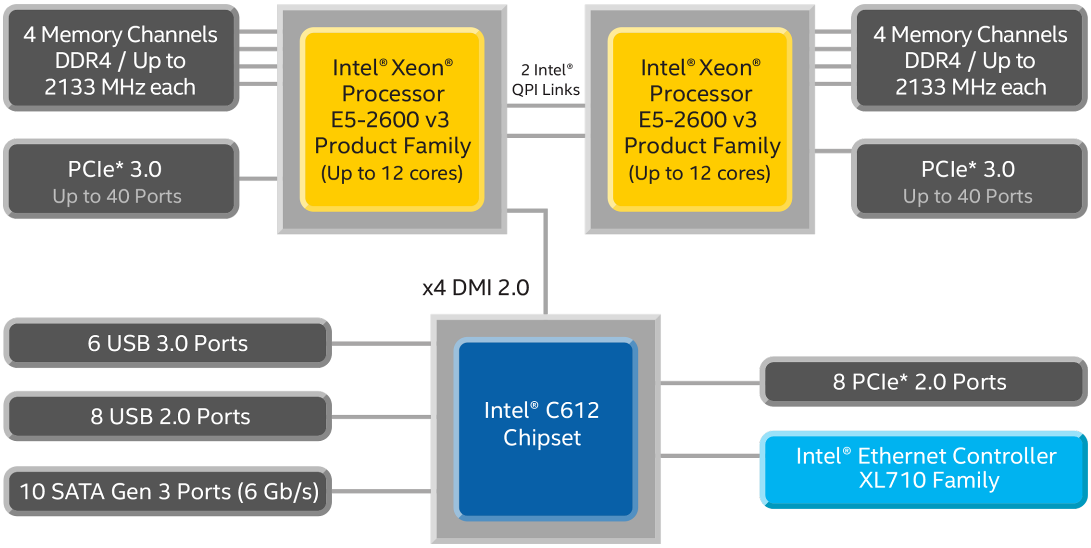

4 Memory Channels DDR4 / Up to 2133 MHz each
PCIe* 3.0 Up to 40 Ports
Intel® Xeon® Processor E5-2600 v3 Product Family (Up to 12 cores)
2 Intel® QPI Links
Intel® Xeon® Processor E5-2600 v3 Product Family (Up to 12 cores)
4 Memory Channels DDR4 / Up to 2133 MHz each
PCIe* 3.0 Up to 40 Ports
x4 DMI 2.0
6 USB 3.0 Ports
8 USB 2.0 Ports
10 SATA Gen 3 Ports (6 Gb/s)
Intel® C612 Chipset
8 PCIe* 2.0 Ports
Intel® Ethernet Controller XL710 Family

Figure 2.1: The Intel®C612 chipset which has been used in this thesis. Source: `http://www.intel.com/content/www/us/en/embedded/products/grantley/specifications.html`

Within the CPU there are different levels of cache, these caches contain partial copies of data that lives in main memory. Caches are much faster and much smaller in size than main memory. The goal of these caches is to reduce the amount of CPU cycles that get wasted when waiting for the data to arrive at the CPU. For example on an Intel i7-4770 the latency introduced by accessing memory that is in Layer 1 is about 4 cycles. Accessing memory that is only in Layer 2 already costs us 12 cycles, and when the memory is only in Layer 3 an access will cost 36 cycles. Whenever memory is not in the caches but still in main memory the latency increases significantly. Since the memory now has to be retrieved off-chip and loaded into the Layer 3 cache, after which it can be read in 36 cycles. [24]

The L3 cache is shared by each processing core on the CPU, the L1 and L2 caches however are only used by a single processing core and the hyper-threaded core that is associated with it. Hyper-threading technology allows a core to be used when it would usually be sitting idle. This can either be because it is waiting for an instruction to complete or memory to enter the CPU. A hyper-threading core has its own architectural state, but the actual execution of instructions is interleaved with the core that it shares.

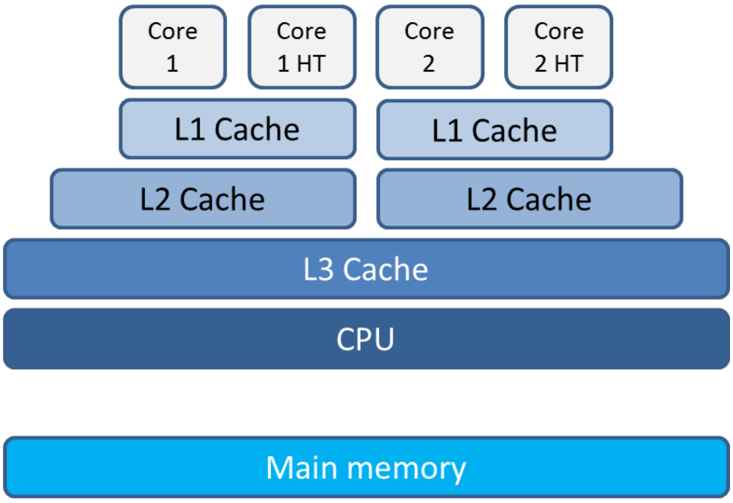


Figure 2.2: How the caches would be structured in a dual core CPU.

2.2 Virtualization

We use virtualization to describe the isolation of applications and entire operating systems from each other while still running them on the same hardware. Many techniques are available to do this isolation each with different use-cases and benefits. Probably the most used and most know is virtualization using a hypervisor. This is however not the only technique; there are many more like containers (2.2.2), unikernels (2.2.1), rump kernels (2.2.1) and the NoHype architecture (2.2.3).

Virtualization is popular when there is a need to run multiple applications in isolation, where the application does not fully utilize all hardware. A user might have a machine running a webserver and another machine that is used as a network attached storage device. Now imagine that none of these two application use more than 20% of CPU during normal operation, wouldn't it be beneficial to run these two applications on a single machine? Often simply running these two applications on the same operating system is not an option due to security risks or different requirements for both. For example a different operating system may be required. Virtualization can solve this problem by dividing the physical hardware into multiple virtual machines which can run completely separate from each other. We are thus now able to consolidate two machines onto one which reduces the cost associated with operating these two applications and the purchase cost of the required hardware.

Since we now have applications running inside of virtual machines maintenance becomes less cumbersome as with physical hardware. Troubleshooting an issue where the guest operating system is unable to start or connect to the network can be done from behind a desk using a virtual terminal. When a physical machine refuses to boot there is no other option than to go down to the datacenter, hook up a monitor and then resolve the issue. It is clear that issues with virtual machines can be resolved much more quickly and efficiently as would be possible when a physical machine fails. But not only servicing the virtual machines becomes less of a cumbersome operation. When a physical machine has to be taken down for maintenance we can move the virtual machines running on that host to a different machine without shutting them down, or even a noticeable interruption of service.

2.2.1 Hypervisors

A hypervisor is a software layer that runs underneath each guest operating system, this software layer emulates hardware devices that represent the machine on which the operating system runs. Network interfaces, harddisks, cd-rom drives, cpu, system memory and many more system resources are allocated to a virtual machine by the hypervisor.

The devices emulated by the hypervisor do not necessarily represent the physical hardware that has been installed, there is a small set of devices emulated by hypervisors which are supported by most operating systems without the need for any additional drivers.

The advantage of using these drivers is that we do not create a dependency on the physical hardware installed in the physical machine. Whenever there is a need to move the virtual machine to a different physical machine we can do so without having to reinstall the virtual machine due to driver breakages or other incompatibilities.

On the other hand, these emulated devices have one major disadvantage, they are very inefficient. The solution to this inefficiency is discussed in a later section on *paravirtualization*(section 3.1.1) which introduces special device drivers that allow for interaction with the hypervisor instead of emulated devices.

Types of hypervisors

Despite the general term 'hypervisors' we have given to the software layer that runs underneath the operating system there still are different types of hypervisors. The difference between the different types can be found in the location where the hypervisor runs.

The *virtual machine monitor*, which manages a single virtual machine, can either live on top of an existing operating system or run directly on top of the hardware inside the hypervisor.

When the hypervisor runs directly on top of the hardware, the hypervisor takes full control of the hardware it will be distributing between the VMs. There is no general purpose operating system running directly on the physical machine. VMware ESXi and Xen are good examples of this type of hypervisor.

For VMware the drivers and services to manage the hypervisor reside in the same layer as the virtual machine monitor. Any other operating system running on the system is a virtual machine which uses the hardware provided by the hypervisor.

Xen on the other hand has a smaller hypervisor footprint. In Xen there is a special virtual machine that is responsible for the physical hardware, communication with physical hardware goes through this management virtual machine.

KVM, Virtualbox and the desktop products of VMware are some examples of hypervisors that run on top of a general purpose operating system. These solutions add modules or extensions to the operating system kernel in order to get the required amount of access to the hardware. However the virtual machine monitor which manages a guest runs outside of kernel space on top of the host operating system.

KVM has support built into the linux kernel and can be compiled in by enabling it at compile time using the *HAVE_KVM* flag. This support is then used by Qemu which has support for the KVM kernel capabilities built in and leverages these to achieve better performance.

Virtual machines

When we talked about *guests* or *virtual machines* in previous sections we were referring to the set of virtual devices (cpu, memory, NIC, ...) that is created and represent a machine in software rather than real physical devices. The difference with real physical hardware is that there might be many of these guests running simultaneous on a single physical machine, these guests require some form of virtualization to achieve this. When talking about virtual machines we use a hypervisor to manage these guests and create the required virtual devices. Without virtualization it is generally not possible to run multiple operating systems on a single physical machine[1].

A virtual machine can be running any operating systems that supports the hardware presented to it by the hypervisor. So it is possible to run an instance of Windows, Linux and FreeBSD all at the same time on the same hardware, each inside their own virtual machine. Figure 2.3 illustrates the configuration of a system running two virtual machines on top of a hypervisor.

[1] In section 2.2.3 we introduce a concept that could make this feasible

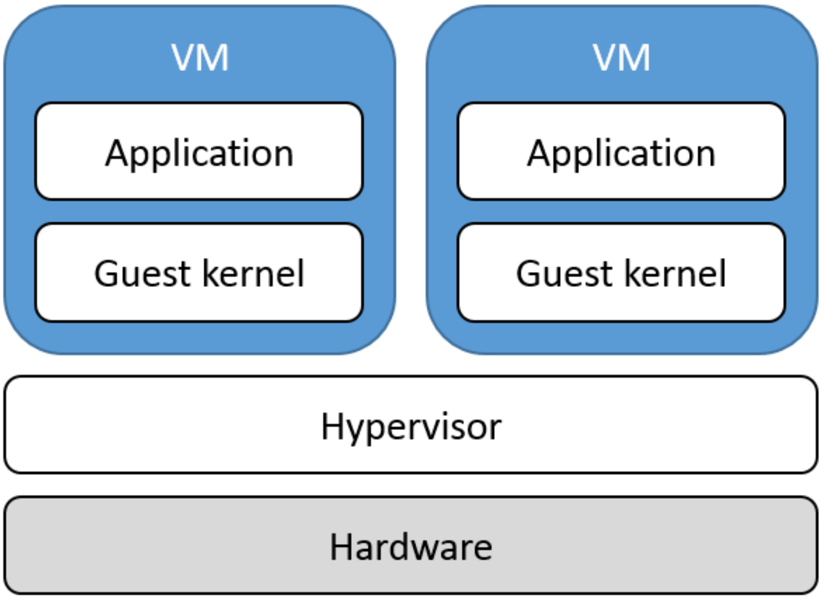


Figure 2.3: A virtual machine runs on top of a hypervisor which manages the hardware. Each virtual machine has its own kernel.

Unikernels

Usually we run general purpose operating systems inside of virtual machines, this is however not an actual requirement. The software running inside a virtual machine can be any code that knows how to work with the virtual hardware provided to it by the hypervisor.

Unikernels are software packages that can be compiled into single binary applications that can run on the hardware provided by the hypervisor (see figure 2.4). This software package does not contain all the code required to run your favorite programs or display a graphical user interface. A unikernel only needs the code to manage the devices it requires and the code it needs to perform the specific function it was built for.

A unikernel is much smaller both in file size and in memory footprint. It is also much more secure since it has a much smaller code base that can be attacked. Many unikernel ecosystems exist for different programming languages, however often they lack drivers for non-standard hardware such as non-emulated NICs.

Unikernels have become more feasible in recent years because the different hypervisors all are able to provide similar, if not the same, virtual devices. Having a known set of devices that will be used to run the unikernel has yielded a set of libraries that can act as device drivers such that the developer does not need to develop a driver for every single physical device that exists.

Rump kernels

While unikernels might sound great, it is often paired with the development of drivers for hardware that is going to be used, especially when needing to talk to the hardware directly. This is far from ideal since this is a lot of work and distracts from the implementation of the actual application.

Rumpkernels are very similar to unikernels in that it compiles application code into a single binary application that can run directly on top of a hypervisor, among other things, inside a virtual machine. Rumpkernels solve the problem of writing driver code by re-using drivers from the NetBSD project. These drivers are developed as standalone drivers that can be plugged into any kernel instead of being tightly integrated into the NetBSD kernel.

Beside being able to re-use existing drivers there is also a version of the standard C library, libc, which allows the use of existing application without (or without many) modifications to the source code, something which is not possible with most unikernels.

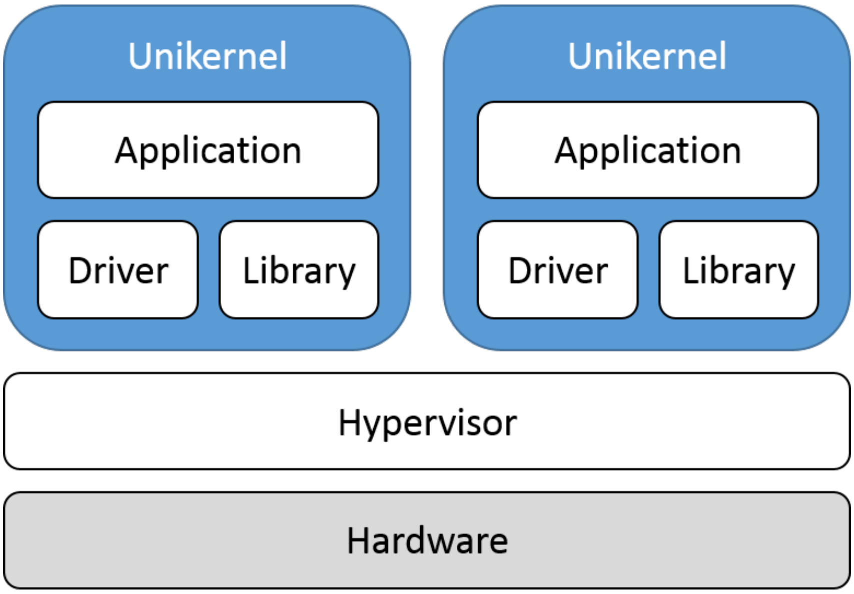


Figure 2.4: A unikernel runs on top of a hypervisor just like a virtual machine, but it only runs the application and its dependencies.

2.2.2 Operating system level virtualization

Up until now we have been talking about the virtualization of hardware, where applications are presented with their own emulated hardware. Operating system level virtualization virtualizes the operating system kernel in such a way that applications running on one part of the kernel are unaware of applications running on another part of the kernel.

The key here is that applications are being isolated from each other instead of entire operating systems. This kind of virtualization thus will not allow multiple different operating systems to run on the same machine, it can however help run multiple applications on the same hardware without them interfering with each other.

On linux this kind of virtualization is called a container, the term is used to describe a single set of processes that are isolated from other processes on the system as is illustrated in figure 2.5.

Operating system level virtualization has many names in different operating systems:

- **FreeBSD**: Jail
- **Solaris**: Zones
- **NetBSD/OpenBSD**: sysjail

For Windows there are proprietary solutions available such as Parallels Virtuozzo [50] and Spoon [68], but since support for operating system level virtualization is not included in the Windows kernel we have not mentioned them here.

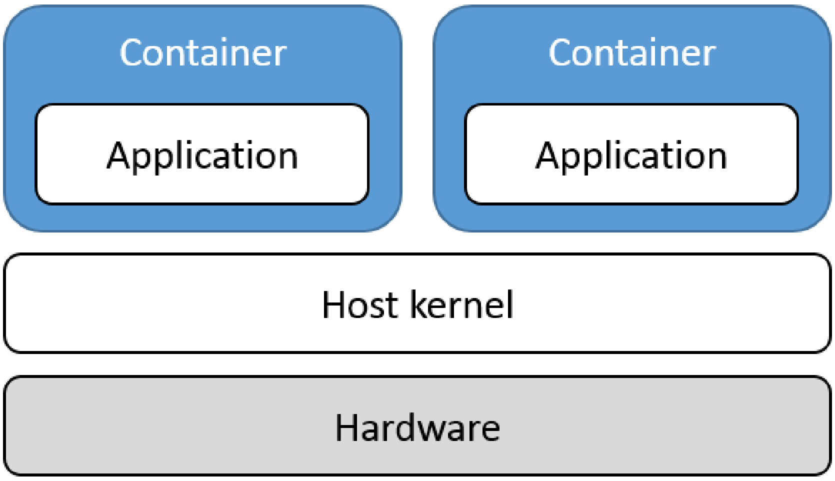


Figure 2.5: A container runs on the kernel that is managing the hardware.

2.2.3 The NoHype Architecture

When virtualizing using a hypervisor there is a piece of software running in the background for as long as the system is up. It ensures the correct functioning of the virtual machine instances. It emulates hardware, traps *privileged instructions*(section 3.1.2) and many more things. But all of these management tasks come with their own overhead and bugs.

The NoHype architecture [85] describes a system that uses only hardware virtualization extensions to achieve a very similar goal. Virtual machines are given a set of processing cores, required devices are associated with an operating system instance using the *IOMMU* (section 3.3.1) and once the operating system is running there are no more interruptions from the virtualization software. The only software running on the hardware that has been set up is the operating system instance we choose to run.

Modern hardware contains many features to assist with the efficient virtualization of a system, these features contain a *Memory Management Unit*, *Memory Management Unit for IO devices*, *special execution levels* for hypervisor software and *device virtualization*. All of these features have come into existence over the years because software solutions were not performing well enough to keep up with the demand of the industry. Many of the functions of a hypervisor can thus be replaced by their dedicated hardware counterparts when absolute control is still required.

Using this technique might create a more secure, better performing platform. This however reduces the overall flexibility gained from traditional virtualization. The NoHype architecture again creates a dependency on the physical hardware and prevents resource sharing since hardware is dedicated to a single guest the entire time it is running.

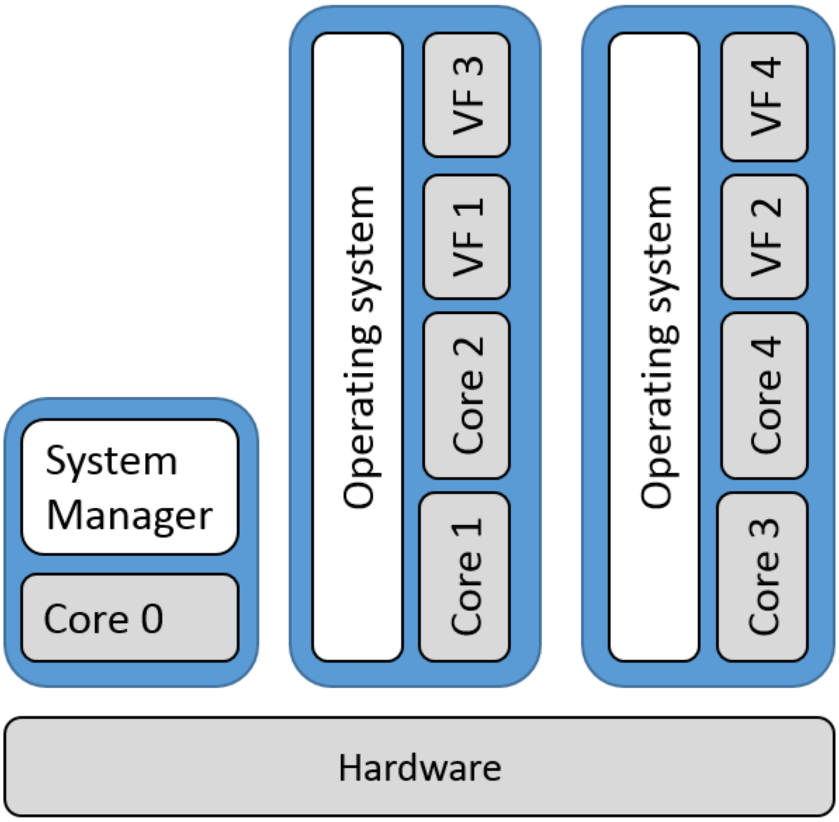
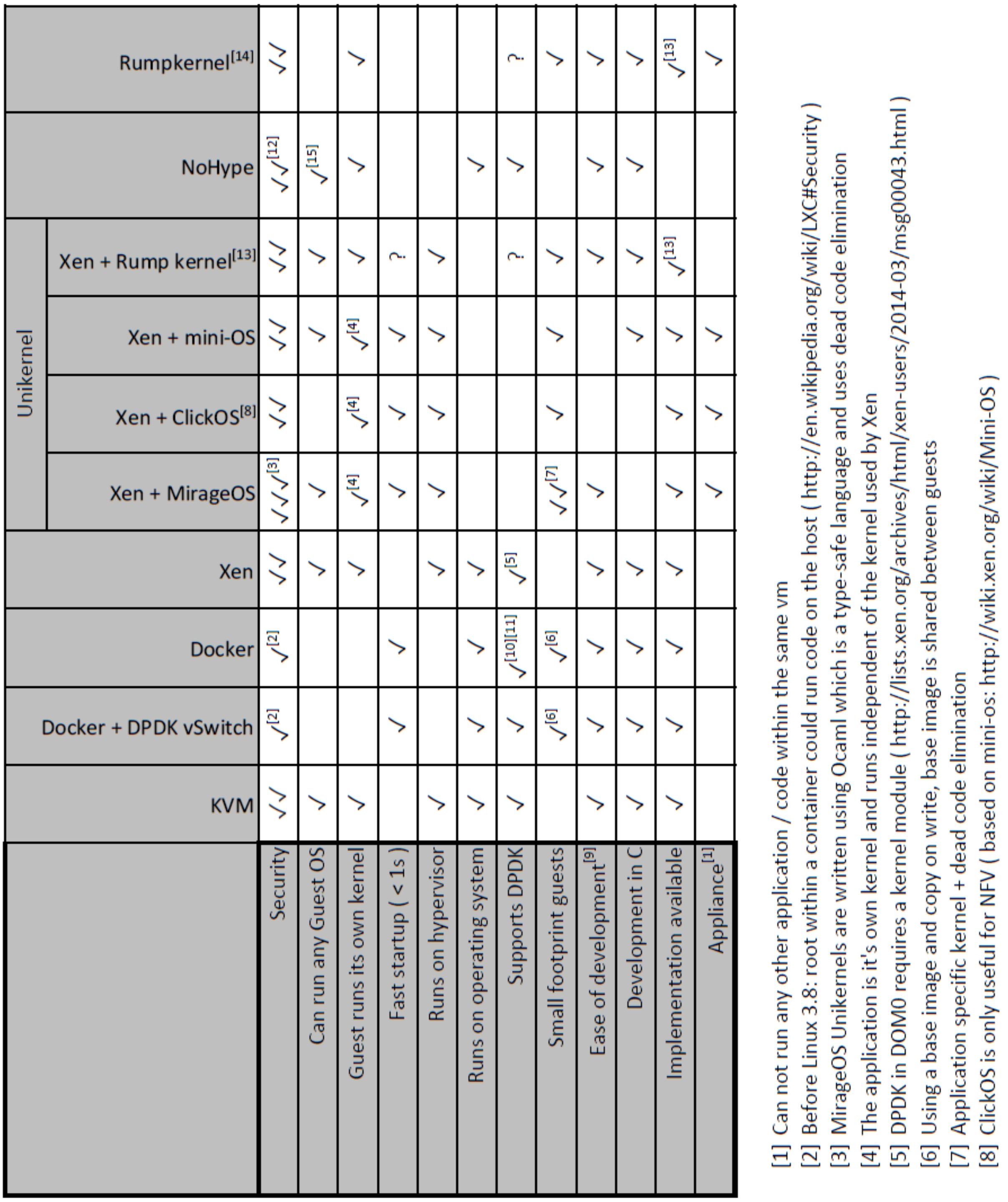


Figure 2.6: Hardware split into multiple segments using the virtualization extensions. Each segment can run a full blown operating system. VF stands for Virtual Function, see section 3.3.1

2.2.4 Overview

We discussed the different techniques that can be used to virtualize a system, each of these has advantages and disadvantages. Below is a table which provides an overview of the differences between the different techniques. We will go into details further in the chapter on machine virtualization (chapter 3) and containers (chapter 4).



	KVM	Docker + DPDK vSwitch	Docker	Xen	Unikernel				NoHype	Rumpkernel[14]
					Xen + MirageOS	Xen + ClickOS[8]	Xen + mini-OS	Xen + Rump kernel[13]		
Security	✓✓	✓[2]	✓[2]	✓✓	✓✓✓[3]	✓✓	✓✓	✓✓	✓✓[12]	✓✓
Can run any Guest OS	✓			✓	✓		✓	✓	✓[15]	
Guest runs its own kernel	✓			✓	✓[4]	✓[4]	✓[4]	✓	✓	✓
Fast startup (< 1s)		✓	✓		✓	✓	✓	?		
Runs on hypervisor	✓			✓	✓	✓	✓	✓		
Runs on operating system	✓	✓	✓	✓					✓	
Supports DPDK	✓	✓	✓[10][11]	✓[5]				?	✓	?
Small footprint guests		✓[6]	✓[6]		✓✓[7]	✓	✓	✓		✓
Ease of development[9]	✓	✓	✓	✓	✓			✓	✓	✓
Development in C	✓	✓	✓	✓			✓	✓	✓	✓
Implementation available	✓	✓	✓	✓	✓	✓	✓	✓[13]		✓[13]
Appliance[1]					✓	✓	✓			✓

[1] Can not run any other application / code within the same vm
[2] Before Linux 3.8: root within a container could run code on the host (http://en.wikipedia.org/wiki/LXC#Security)
[3] MirageOS Unikernels are written using Ocaml which is a type-safe language and uses dead code elimination
[4] The application is it's own kernel and runs independent of the kernel used by Xen
[5] DPDK in DOM0 requires a kernel module (http://lists.xen.org/archives/html/xen-users/2014-03/msg00043.html)
[6] Using a base image and copy on write, base image is shared between guests
[7] Application specific kernel + dead code elimination
[8] ClickOS is only useful for NFV (based on mini-os: http://wiki.xen.org/wiki/Mini-OS)

Figure 2.7: Comparison between features of different virtualization technology

2.3 Network Function Virtualization

While the previous sections were all about making better use of the computing resources we have available and ease the management through the use of virtualization, Network Function Virtualization [12] [77] wants to make the hardware we use to setup our network and network services more flexible through the same mechanisms we have used for traditional computing.

A significant amount of the equipment that can be found in a datacenter is there to provide communication between servers and connect them to the internet. This kind of equipment comes in the form of hardware boxes, called middleboxes. These middleboxes are installed in the network to perform one specific function and do it very efficiently. There is however one problem with this kind of equipment, and that is that they are not very flexible. If we wanted to add a new router to the network we would have to buy a new box and physically install it in the datacenter which might take some days or weeks rather than some hours.

Besides being physical equipment, the middleboxes have another disadvantage. These middleboxes are able to perform their function very efficiently because the hardware that is inside of them has been built to perform a single function, and only that function. The exact implementation of this function is programmed into the circuit boards that make up these devices. Because of this these devices can not be upgraded, or tasked with another function. Most of these hardware implementations have their own management tools and APIs, which in turn are not compatible with the implementations that other vendors might have chosen to use.

The goal of Network Function Virtualization, NFV for short, is to provide the same functionality using software implementations of middleboxes running on commercially available, general purpose hardware. This provides us with more flexibility as we can now deploy applications on machines that are built using the same hardware platform as the compute platform used to run applications and services. We could now instead create a network using virtual machines (VM) where a VM is running the software implementation of the middlebox, allowing us to deploy network functions nearly on demand, reducing the time to deploy such a device to minutes or hours.

In this setup each VM plays a role in the network that is now fully defined by software and runs in a virtual environment to provide us with increased flexibility and consolidate many functions on less hardware, introducing more efficient use of the available hardware.

The network functions we discussed belong in the data plane, they do the heavy liften by processing packets according to the configuration they are specified. The applications performing this configuration are part of the control plane. The control plane does not process high volume packets, rather they configure the devices in the data plane. This is where SDN or Software Defined Networking comes into play. SDN allows open software to configure both hardware middleboxes and virtualized network functions, providing an abstraction layer on top of the physical layout of a network. This allows for rapid reconfiguration and deployment of new network topologies without having to touch the underlying network.

2.3.1 Fast packet processing

In order to achieve the goals of NFV we will be needing much faster processing of network traffic than is available through the generic packet processing mechanisms available on general purpose operating systems[2]. That is why alternative methods of handling network traffic have been developed, either by communicating directly with the network interface cards or by tapping into the kernel memory at the lowest level available.

Intel®Data Plane Development Kit

The Intel®Data Plane Development Kit [25] uses the first approach and talks directly to the network interface cards and provides an entire set of memory and packet management libraries to achieve the highest possible performance. DPDK has been optimized to take advantage of the hardware to the fullest, it uses hugepages to allocate all of its memory and when passing packets around it does so by moving pointers to the actual packets rather than copying the packets themselves. In order to achieve maximum performance on general purpose hardware the use of multi-core systems is going to be required, DPDK provides lockless rings, memory pools and buffer management for efficient communication between cores.

Instead of relying on interrupts from the network cards to signal packet delivery DPDK uses a *Poll Mode Driver* which on a continuous basis ask the network driver if packets are available, which results in faster packet reception and less interrupts being fired. This ensures the application stays responsive and able to keep up with line-rate speeds.

Netmap

Netmap [93] uses the alternate approach of mapping hardware packet buffers into a user-space application using a device independent API. Netmap bypasses the kernel network infrastructure and prevents copies

[2] Improvements are being made, but we are not there yet. `http://netoptimizer.blogspot.dk/2014/10/unlocked-10gbps-tx-wirespeed-smallest.html`

from being made between user-space and kernel-space and then when converting into an mbuf/skbuf in kernel.

To achieve line-rate performance netmap uses some key ideas [93]:

- A shadow copy of the NICs ring supports batching of requests and removes the need for mbufs/skbufs
- Efficient synchronization using *poll()*
- Carefully designed API, event loops need only one syscall per iteration
- Full support for multicore and multiqueue NICs through *setaffinity()*

2.3.2 Virtualized network performance

While virtualization allows us to use system resources to the fullest potential and makes management a lot easier for administrators, it comes at a cost. Virtualization impacts system performance in a number of ways, many of which hardware vendors have tried to mitigate using hardware support for virtualization.

There are many factors that may contribute to the resulting performance of a system such as the workload type, the size of the workload and the placement of virtual machines [88]. Where does network function virtualization fit within these factors and what can be done to optimize the processing of packets using general purpose hardware and virtualization?

To start off with we have to be careful in our choice of hypervisor as not all are created equal and performance differences exist. These differences can be attributed to the implementations of handlers for *privileged instructions* (section 3.1.2), how the hypervisor balances load, which kind of virtualization is used (see chapter 3.1) and other implementation details. The choice of hypervisor might limit us in the way we handle our network traffic. For example: will we be able to retrieve packets without them being copied numerous times? Will it be possible to access the NIC directly without the hypervisor intervening?

Since our goal is to move packets around, and these packets usually are from or for one of our network interfaces we are putting significant strain on the IO interfaces of the system. In order to hit our goal of line-rate packet processing we have to ensure that main memory accesses are reduced to a minimum because this introduces latencies that are not acceptable for network traffic. We will want to keep packet data as close to the executing core as possible, this means that the ideal location for a packet to be is in the Layer 1 cache of the core processing that is going to handle the packet. If data is not readily available to the CPU it will stall on memory loads and sit idle without being able to execute the instructions that are being queued. Because of these constraints we will look at network function virtualization as an IO-intensive workload.

VM placement is of real importance as well when trying to do high throughput packet processing, ensuring that the virtual machine runs on the CPU socket that is attached to the PCIe slot of the NIC for optimal performance. When the hypervisor is allowed to move or choose the placement of this virtual machine it might decide to place it on unfavorable processing cores and reduce the speed with which packets arrive simply because packets have to be fetched from a different socket's memory. CPU caches play an important role here as well since a hypervisor might choose to move a vCPU to a different physical CPU core at any time and all the data that has been allocated in L1 or L2 cache will have to be fetched again on the new core, this creates extra latency on the processing of the packets.

2.4 Virtualized Network Functions in containers

Recently linux containers have gained significant attention from the community with projects like Docker growing significantly in featureset and userbase. Since the introduction of Docker, the developer community has embraced containers as a possible replacement for the hypervisor. Linux containers introduce less overhead in isolating applications from each other because each container interacts with the same linux kernel as the host system instead of communicating with the hypervisor. This reduced overhead is expected to increase performance of virtualized network functions as they rely heavily on the optimization of the platforms used to run them.

This thesis will focus on using Docker to run our containers as most current tools either use or support Docker (section 4.2.1). By using Docker we prevent a technology lock-in since Docker can run on any recent upstream kernel, and all features provided can also be implemented manually or through different tools.

Chapter 3

Machine virtualization

3.1 Virtual machines

Up until now virtualized network functions have mostly been deployed on hypervisors, running inside a virtual machine which in its turn is running a flavor of linux. In the introduction in section 2.2.1 we have already discussed the tasks of the hypervisor, let us now look at what is possible with the virtual machines that run on top of them. We will start off by giving a definition of the different types of virtualized machines and how they are realized.

As we are interested in high performance virtual machines for our use-cases we will only be covering the types of virtual machines that run using techniques that are still being used these days and we will not be discussing technologies that were used in the earlier stages of virtualization such as binary translation.

3.1.1 Paravirtualization

The first type of guest we will be discussing is the paravirtualized guest, this type of guest is fully aware of the virtualization that is being performed. In order to run this type of virtual machine there is a need for cooperation of the guest operating system. Instead of simply running the kernel code paths that would normally be executed on normal hardware it substitutes restricted instructions with a request to the hypervisor to execute the instruction on its behalve (Figure 3.1).

This kind of virtualization allows the hypervisor to transform the instruction and the associated parameters such that they will only have an effect on the guest that is asking to execute it. The hypervisor can thus see to it that a virtual machine is not trying to influence the behavior of a different virtual machine on the system.

This kind of virtualization was considered to be significantly faster than the traditional techniques used when no hardware support for virtualization was available. These techniques usually required the hypervisor to intercept sensitive instructions and then handle them in software which came at a significant cost to performance.

An additional advantage of paravirtualized virtual machines was that drivers could be optimized to interact with the hypervisor in a much more efficient manner as the traditional emulation of hardware. The virtio drivers are a great example of this where communication with the hypervisor happens through a patch of memory that is mapped into the virtual machine's memory. For the virtio-net driver, which provides high throughput performance networking, this block of memory provides the guest access to rings which represent a transmit and receive queue and some memory to store packets in. The drivers in the guest virtual machine can then simply transmit and receive packets by copying the data into this memory and adding a pointer to the packet in the correct ring. This in contrast to the more complex task of emulating virtual hardware where we again run into the problem of handling sensitive instructions which control the (virtual) hardware.

As we mentioned earlier, this kind of virtualization requires the cooperation of the operating system running inside the virtual machine, this means that we are limited to running operating systems that have been updated to work on a specific hypervisor. The hypervisor that uses this feature extensively is the Xen hypervisor. The group of operating systems that are able to run paravirtualized is limited. Windows, OS X and other closed source operating systems have no support for this type of virtualization and require the next type of virtualization that we are going to discuss in section 3.1.2. On the other hand, some open source operating systems have support for paravirtualization. Linux has support for paravirtualization on Xen built in since the release of Linux 3.0 [32], before that a custom kernel had to compiled in order to run linux in a paravirtualized environment. Another operating system that has some form of support for paravirtualization on Xen is FreeBSD [18].

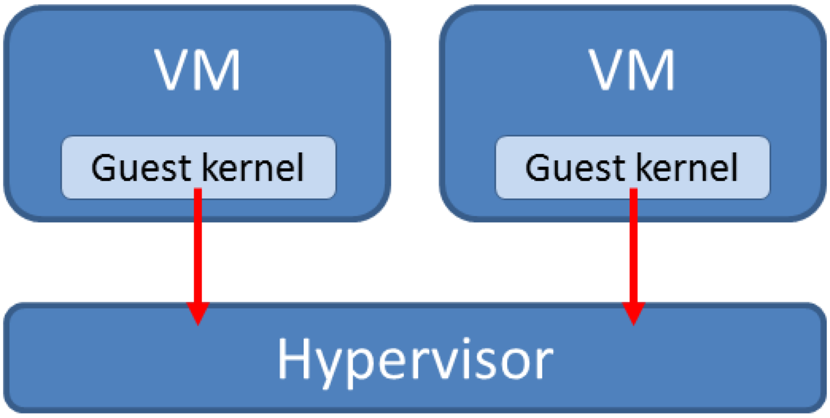
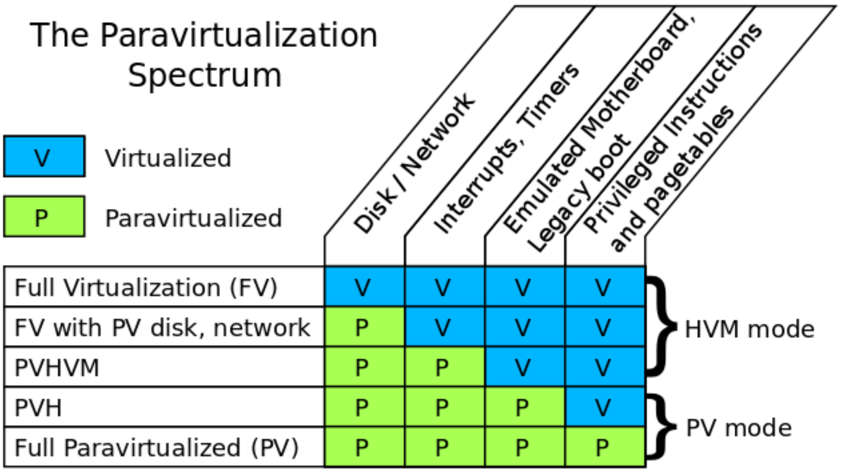


Figure 3.1: Paravirtualized machines will communicate directly and more efficiently with the hypervisor for sensitive instructions and performance critical components.

Paravirtualization can be performed only partially as well, such that specific subsets which are performance critical take advantage of the paravirtualized components while other interactions happen through other mechanisms. An example is the use of virtio network adapters on a Windows machine, the drivers that are able to communicate with the hypervisor can then be installed onto the virtual machine in order for Windows to correctly communicate with the virtual hardware and hypervisor. This type of deployments is common where the virtual machine is sensitive to IO performance, for example when there is a need for high performance disk access. The different degrees of paravirtualized machines can be seen on figure 3.2.


The Paravirtualization Spectrum
V Virtualized
P Paravirtualized

	Disk / Network	Interrupts, Timers	Emulated Motherboard, Legacy boot	Privileged Instructions and pagetables	
Full Virtualization (FV)	V	V	V	V	HVM mode
FV with PV disk, network	P	V	V	V	
PVHVM	P	P	V	V	
PVH	P	P	P	V	PV mode
Full Paravirtualized (PV)	P	P	P	P	

Figure 3.2: The different degrees of paravirtualization in Xen. Source: `https://blog.xenproject.org/2012/10/31/the-paravirtualization-spectrum-part-2-from-poles-to-a-spectrum/`

3.1.2 Full virtualization

Full virtualization in contrast to paravirtualization does not require the operating system to interact directly with the hypervisor. As a result of this a hypervisor that is deploying fully virtualized virtual machines is able to run any operating system, unmodified. Before hardware support had been introduced on x86 processors there were very complex techniques involved in running unmodified operating systems in a privilege level known as Ring 1 on the processor which would prevent the virtual machine from directly running sensitive CPU instruction.

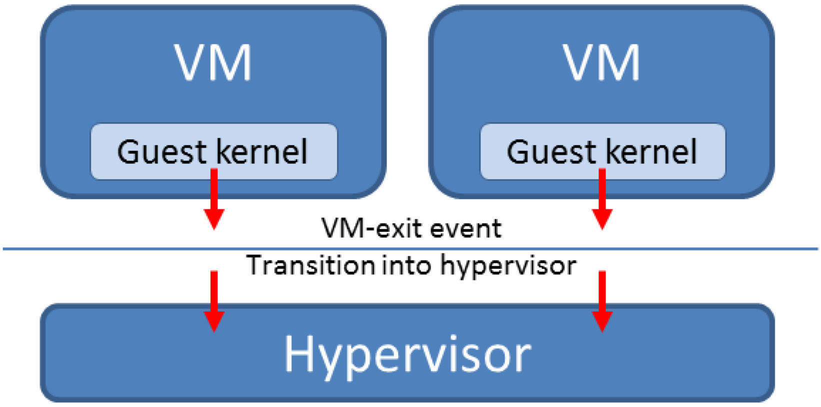


Figure 3.3: With full virtualization the hypervisors defines events on which it should regain control.

Processors with virtualization extensions, AMD-V or Intel VT-x, introduce a new privilege level that is more privileged than the ring 0 in which usually the code with the highest privilege runs. This ring is often called ring -1 or hypervisor mode, it should be no surprise then that this is where the hypervisor will run and thus this allows us to run unmodified guests in ring 0. The hypervisor is then able to define events on which control of the system should return to the hypervisor where the event can be handled [82]. When one of these events occurs, the switch from guest to hypervisor is called a VM-exit and these are very costly. Processor manufacturers have been hard at work to increase the efficiency of these VM-EXIT and VM-RESUME instructions as case be seen in figure 3.4.

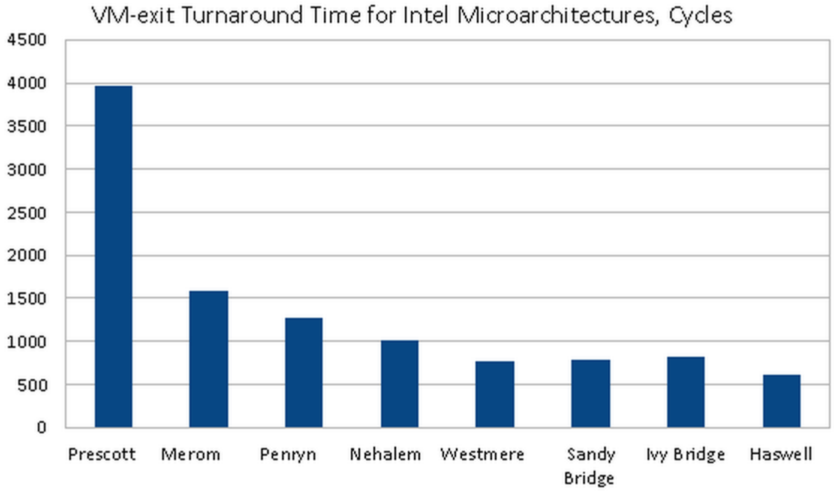


Figure 3.4: The latency introduced by VM-exit and VM-resume on different generations of Intel CPUs. Source: `http://atakua.doesntexist.org/wordpress/2014/09/29/measuring-vmx-root-non-root-transitions/`

These instructions are so costly because they have to save the state of the CPU when a VM-exit event occurs and the state has to be restored as soon as we return control to the virtual machine. The state of the CPU consists of a number of things [82]:

- Information about the cause of the VM exit is recorded in the VM-exit information fields and VM-entry control fields are modified.
- Processor state is saved in the guest-state area.
- MSR[1]s may be saved in the VM-exit MSR-store area.
- Processor state is loaded based in part on the host-state area and some VM-exit controls.

[1] Machine specific registers

- Address-range monitoring is cleared.
- MSRs may be loaded from the VM-exit MSR-load area.

Support for the management of memory has also been added in hardware as Extended Page Tables. Using the EPT extension there is an extra level of depth in the memory management unit which helps looking up real memory locations. The 3 different depths are logical pages, where virtual addresses are stored for use by processes running inside a guest. In the level containing the physical pages are the addresses that the guest sees as physical memory and where during unvirtualized execution the real memory locations would be found. When a hypervisor is active there is an extra level which are called the machine pages and these are the location in RAM where the data will be found. These 3 levels are visualized in figure 3.5.

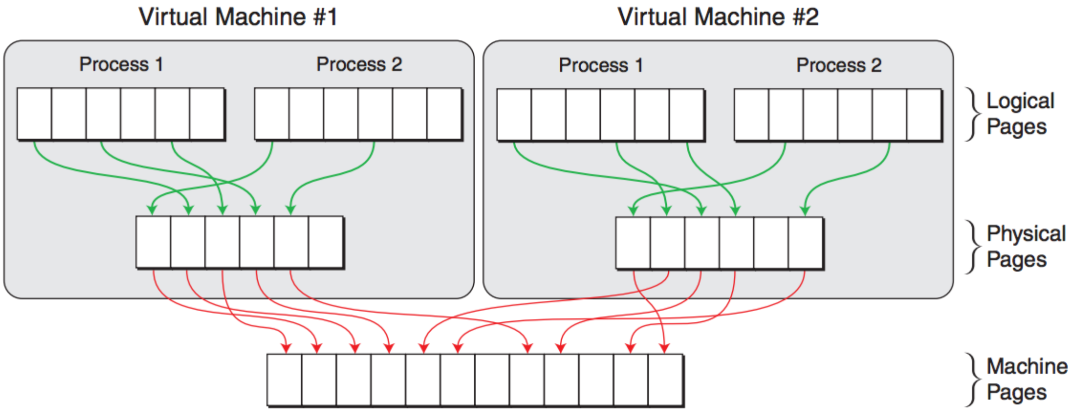


Figure 3.5: Extended Page Tables adds another level in the MMU to translate addresses from physical to machine pages. Source: `https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`

3.1.3 Device passthrough

For some high performance applications such as the VNFs that we will be using in our proof of concept it is required to pass through physical hardware to the virtual machine. This allows the virtual machine direct access to the hardware rather than to communicate through emulated devices. It should not come as a surprise that this provides a real performance advantage to the virtual machine since a level of abstraction is removed.

Device passthrough is commonly done with PCI and PCIe devices, but requires the hardware to assist the hypervisor since we want to prevent the virtual machine from attacking other virtual machines through the PCI devices. This kind of attack is possible because PCI devices can also write to the physical memory of a machine, and it traditionally does so by using the physical memory addresses of the physical machine (figure 3.6).

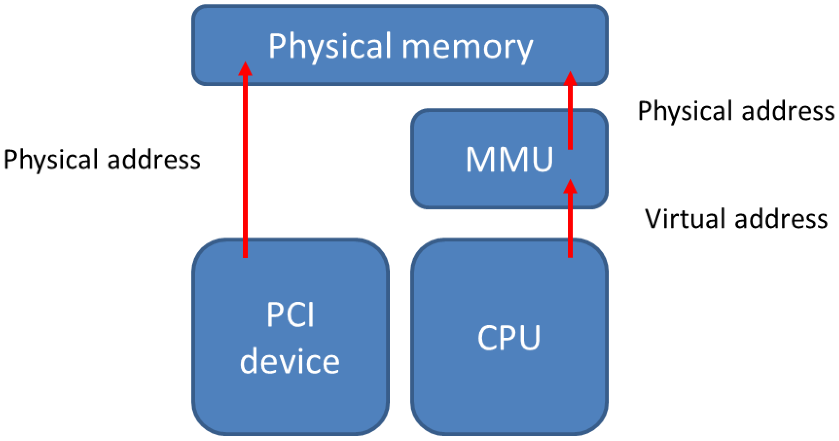


Figure 3.6: The interaction with memory from the perspective of the CPU and a PCI device without a IOMMU

This is where the IOMMU (Input/Output Memory Management Unit) comes into play and provides us with a secure method of assigning devices to virtual machines. The IOMMU will translate the addresses used by the PCI device from addresses that are physical to the virtual machine (similar to the MMU in figure 3.5) and machine addresses that are real memory locations inside the real physical memory. When compared to figure 3.6, where no IOMMU has been used, this works as is displayed in figure 3.7.

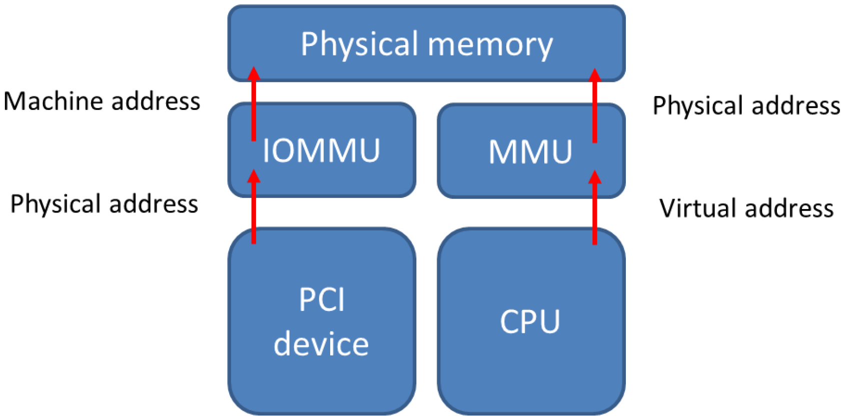


Figure 3.7: The interaction with memory from the perspective of the CPU and a PCI device with a IOMMU

Besides translating addresses the IOMMU unit can also check if an address has been mapped to a certain VM, if it has not been mapped to that specific VM it will cause a trap to the VM. Using the IOMMU it is thus possible to set up the memory regions to which the devices assigned to a VM have access which allows us to safely assign devices to virtual machines.

3.1.4 Hypervisors

Now that we know what virtual machines are we will provide an introduction to some popular open source hypervisors which provide a solid alternative to the commercial alternatives such as VMware or Microsoft's Hyper-V. The location in which hypervisors run within the software stack has traditionally been used to classify the hypervisors into two classes. One class is the Type-1 hypervisor and the second one is the Type-2 hypervisor [92].

A Type-1 hypervisor is a hypervisor that runs directly on top of the hardware, there is no general purpose operating system running on the hardware, instead the hypervisor has full control of the hardware it runs on.

The second type hypervisor, Type-2, has the hypervisor running on top of a general purpose operating system. This involves an extra party in the communication with the hardware which introduces extra overhead.

The distinction between these two types of hypervisor are however not as clear in recent hypervisors. Modern Type-2 hypervisors load drivers or kernel extensions into the kernel in order to provide some of their functionality in the kernel rather than in user-space.

An example where the distinction between Type-1 and Type-2 is not very clear is the **KVM** hypervisor. Kernel-based Virtual Machine implements hypervisor functionality in the linux kernel turning the kernel itself into a hypervisor. Configuring and launching a virtual machine however is not done by KVM but rather by a userspace application that interacts with the KVM API. Through this API a userspace application that has support for KVM can launch virtual machines and configure them as it would like. Qemu is the the userspace used with linux KVM support to launch virtual machines on a linux system that has KVM built into its kernel. Like many current hypervisors KVM requires the processor that is being used to support the virtualization through virtualization extensions, for Intel processors this is VT-x and for AMD processors it is AMD-V. This means that KVM only supports full virtualization, however it can present some paravirtualized devices to the guest.

Another popular hypervisor is **Xen**, which can be classified as a Type-1 hypervisor. The Xen hypervisor itself runs directly on the hardware but does not implement the drivers required to address hardware such as a NIC, instead Xen automatically launches a special purpose virtual machine that is called Domain 0 or Dom0. Dom0 is a paravirtualized instance of linux which is responsible for communication with most of the hardware in the system. Xen has support for both full virtualization, using hardware virtualization extensions, as well as paravirtualization and any combination of these as can be seen in figure 3.2 [75].

The current generation of hypervisors consist of large monolithic code-bases which has an impact on the security of the entire system. An exploit in the hypervisor might put the entire system at risk since the hypervisor is in control of every resource on the system. **Microhypervisors** try to solve this problem by decreasing the size of the hypervisor dramatically and only implement the bare minimum in the hypervisor itself. Drivers and virtual machine monitors all run in a userspace on top of the hypervisor as application instead (figure 3.8 and 3.9). [94]

A concrete implementation of this is the NOVA hypervisor [94] which was able to shrink the trusted code base[2] (TCB) of the hypervisor an order of magnitude. A comparison of hypervisors in lines of code can be seen in figure 3.10, it is clear that the NOVA hypervisor with ≈36000 lines of code is far smaller than any of the other hypervisors.

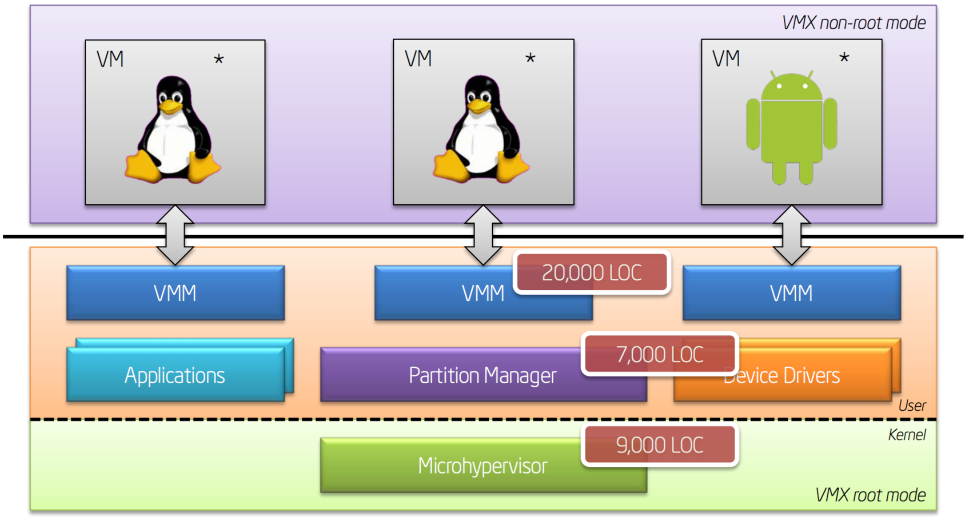


Figure 3.8: The NOVA OS Virtualization Architecture. Source: `http://os.inf.tu-dresden.de/Studium/MkK/SS2012/10_nova.pdf`

[2] The hardware, software, and setup information on which the security of a system depends [87]

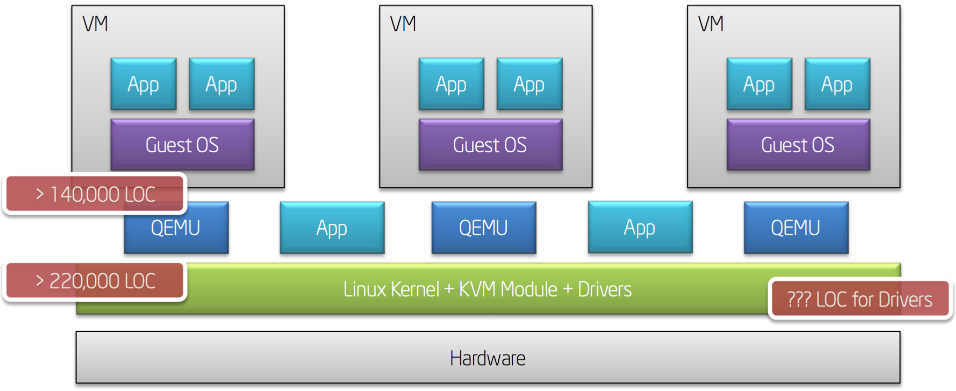


Figure 3.9: The architecture used by KVM. Source: `http://os.inf.tu-dresden.de/Studium/MkK/SS2012/10_nova.pdf`

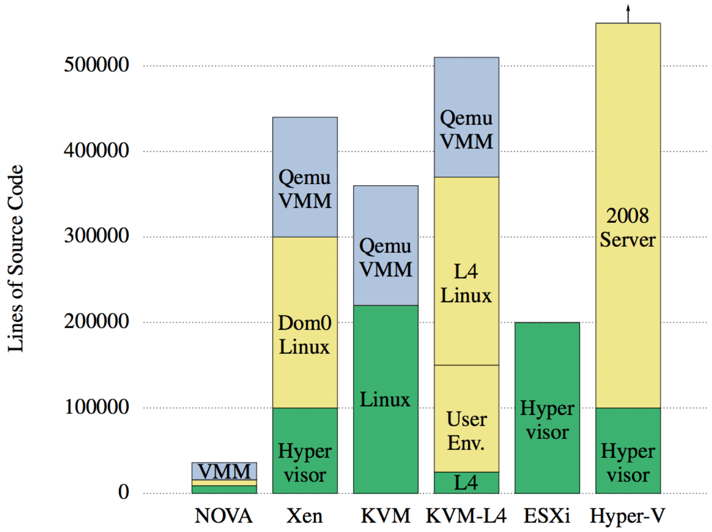


Figure 3.10: A comparison of hypervisor size in lines of code. [94]

3.2 Unikernels

In our discussion on virtual machine we focused on running general purpose operating systems. We are however not required to run any general purpose operating systems in order to execute code on the hardware, and neither are we when running on top of virtual hardware. Developing directly on the hardware is however a time-consuming process since we would have to account for all the different types of hardware and develop the drivers to interact with them.

The virtual hardware that a hypervisor creates however is limited to a much smaller set of devices [74] [73]. Because of this small set of known devices it becomes feasible to develop and maintain drivers only for this small set of virtual devices and re-use these in our applications. Building our applications to run directly on the hypervisor allows us to take advantage of the performance benefits [78] [80] associated with running on bare-metal without having all the drawbacks of running directly on hardware.

Drivers for the small set of known devices are implemented as an application library rather than as an abstraction as is the case for a kernel. Applications that require the driver can reference the library from their own code, at compile time this library will be compiled into the final binary. The final binary is able to run directly on a hypervisor without the need for a general purpose operating system.

Since code and libraries are all compiled into a single binary compilers are able to use nifty optimizations to reduce the binary size, attack surface and code branches [89]. All of these optimizations allow for more efficient execution than would be possible when using an API that abstracted the hardware that is being used. Additional optimizations can be made when running the code, such as a single address space or having the applications run in kernel mode. These remove the overhead that usually is associated with the transition from userspace to the kernel in traditional operating systems.

Because of these optimizations unikernels seem to be an interesting choice for virtual network functions, they provide the developer with unrestricted access to the virtual hardware without having to start from scratch. Using this unrestricted access it is possible to control interrupts, memory layout and scheduling nearly down to the hardware. The downside however is that we would have to develop drivers for physical hardware we would like to access directly without having our data pass through the hypervisor. The DPDK library has developed their own userspace drivers as well to achieve line-rate packet processing with minimal latency [10], so this scenario is not unlikely.

Many other runtimes, programming languages and toolchains exist to build unikernels, we will only explain the ones we found most interesting in our discussion on alternatives to full blown virtual machines for network function virtualization. Some of the other unikernels are HalVM [20], a haskell virtual machine running on MiniOS, OSv [59] is an implementation of a JVM that is able to run directly on top of the hypervisor and the last alternative we will mention here is LING [11] which compiles Erlang code into a unikernel image.

3.2.1 Mirage OS

An implementation that is getting a lot of attention is Mirage OS [42] [89]. Mirage OS allows developers to build applications using the OCaml programming language [49] and compile them into binaries that run on unix or on top of Xen.

Mirage OS employs many techniques to optimize the binaries that are produced by the Mirage OS toolchain. Because application configuration is part of the application it is possible for the compiler to find unused code paths and remove these from the final binary. This optimization not only makes a program more efficient it also reduces the attack surface of the code. Codepaths that contain critical errors might simply not be present in your final binary because your application does not use the functionality containing the error. The use of OCaml means that code is written in a statically types language which prevents the kinds of errors that could occur in more low-level languages such as C where the developer is response for memory management. Buffer overflow errors are an example of errors that are non existent for programs written in OCaml, and if recent problems [21] [76] [45] are any indication this is a great plus.

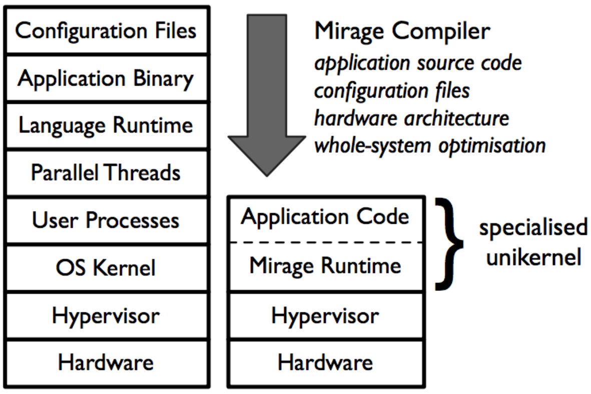


Figure 3.11: A comparison between a virtual machine stack vs. that of a unikernel appliance. [42]

Implementations of protocols such as IP, TCP and HTTP are available to developers using Mirage OS through the OPAM package manager. The OPAM package manager is a package manager that contains libraries for the OCaml programming language, not only for unikernels. Many of the higher level functionality such as a webserver is already available to unikernel developers which makes Mirage OS an interesting choice for cloud appliances.

3.2.2 ClickOS

ClickOS [90] is a software middlebox platform that allows virtual machines on commercial hardware to replace the hardware middleboxes that are common in current networks. ClickOS is a result of the work being done on network function virtualization. The ClickOS virtual machines are very small, fast to spin up and are able to saturate a 10Gb link. The software inside of a ClickOS virtual machine is based on the Click modular router software which already has implementation for many of the network functions that are performed in carrier networks.

ClickOS is based on MiniOS [41], a minimal implementation of the functionality required to run code inside a Xen virtual machine. Since MiniOS does not support symmetric multi processing, it is only possible to run a single Click element inside a VM. This is however no a problem since ClickOS is so small it is possible to scale up the number of VMs rather than the size of each virtual machine. MiniOS uses the Xen netfront driver for network communication, this driver uses shared memory to communicate with the netback driver that resides in the control domain of Xen (dom0) in order to move packets around the system. Signalling the arrival of new packets is done through Xen event channels which are essentially Xen inter-VM interrupts.

ClickOS found that both the netback and netfront drivers did not perform particularly well. In order to achieve better performance from the overal system they redesigned the I/O used by ClickOS and the hypervisor. The redesign involved the removal of the default Open vSwitch that was being used and replace it with a ClickOS switch. The ClickOS switch exposes a ring of packet buffers to the VM. The netfront driver in the VM is then able to map this ring into its memory. The ClickOS switch is based on the VALE switch which uses netmap API (section 2.3.1) to communicate with network devices.

The changes made to the networking stack however break compatibility with existing netfront drivers. The MiniOS implementation was changed to comply with these changes and a new netfront driver for linux has been written to demonstrate performance. These new implementations allow for much greater performance as demonstrated in figure 3.12.

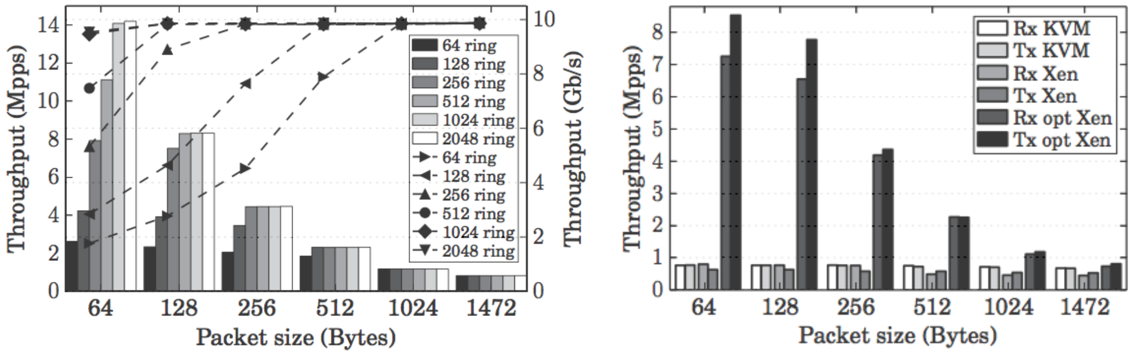


Figure 3.12: Performance of MiniOS using the new network stack with varying ring size (left) and a Linux domU (right) which compares out-of-the-box networking and the new stack (opt Xen). [90]

3.2.3 Rump Kernels

We have thus far discussed unikernels which are meant to be used as virtual instances on top of hypervisors. An interesting alternative which goes by the name of Rump kernels [84] [65] is available as well, like the other unikernels it compiles to a single binary that can be deployed. Unlike any of the other unikernels this goes by the name rump kernel rather than unikernel and it is not only meant to be run on hypervisors.

Rump kernels contain an abstraction from the underlying hardware platform through an interface which is called the *rump kernel hypercall interface*. The rump kernel hypercall interface allows support for other platforms to be build by implementing a known interface. Subsequently application code is able to execute on this platform, many different platforms are already available in many forms online [63] [66].

The next big advantage that rumpkernels have, and this is also what makes it attractive for use on bare-metal without even using an hypervisor, is that they use existing NetBSD drivers to support hardware. These drivers are written without any dependencies into the NetBSD kernel code which means that they can be moved to a different platform quite easily. All of these drivers together with your application code can then be compiled into a binary which can be deployed on an existing unix machine, a virtual machine or even physical hardware.

Besides being able to re-use existing drivers the rump kernels also have the advantage that they can run (nearly) unmodified unix applications since large parts of the kernel code is reused from NetBSD. Rump kernels can be compiled with a modified *libc* library, which allows existing libraries to be re-used and many applications compiled on top of them.

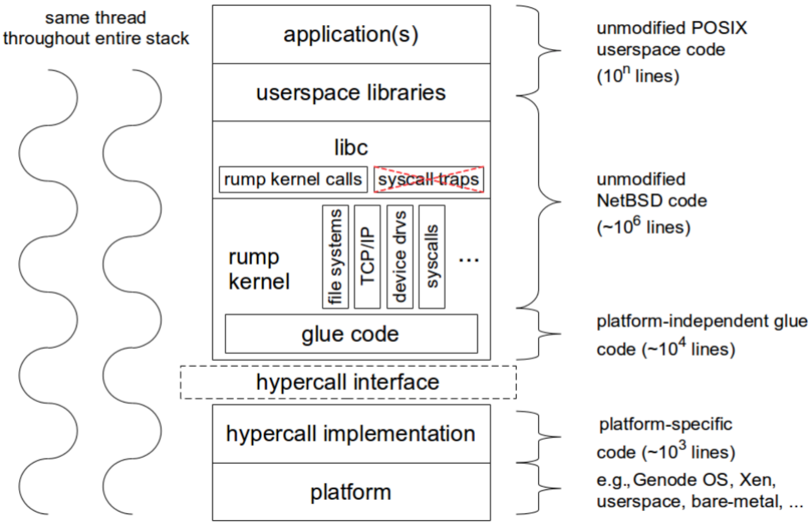

same thread throughout entire stack
application(s)
userspace libraries
libc
rump kernel calls
~~syscall traps~~
rump kernel
file systems
TCP/IP
device drvs
syscalls
...
glue code
hypercall interface
hypercall implementation
platform
unmodified POSIX userspace code (10^n lines)
unmodified NetBSD code (~10^6 lines)
platform-independent glue code (~10^4 lines)
platform-specific code (~10^3 lines)
e.g., Genode OS, Xen, userspace, bare-metal, ...

Figure 3.13: The rump kernel architecture. [84]

3.3 NoHype architecture

The primary goal of the NoHype architecture is to increase security, we however see additional advantages for performance in the architecture described [85]. The NoHype architecture uses only hardware virtualization extensions to run multiple operating systems on a single physical system. Since the NoHype architecture does not intervene while an operating system is executing we do away with the VM-exit's that are required when returning execution to the hypervisor.

The main advantage of the NoHype architecture is that the operating system has full control over the hardware assigned to it. The list of disadvantages is much longer than that of the advantages, the first problem that is paired with the NoHype architecture is that we are working with physical hardware inside our operating system. Physical hardware means that we need to install the driver for that hardware inside of our guest, removing the flexibility to move a guest to another host with different hardware. This is possible with full virtualization because the emulated hardware is not dependent on the physical hardware that is actually present, however using the NoHype architecture we create a dependency on the hardware platform. The nature of the NoHype arhitecture prohibits the hypervisor from regaining control over resources in use by a guest until the guest decides to shut down. This means that it is not possible to multiplex multiple operating systems on the same resource. Once a processor core is assigned to a virtual machine it is no longer possible to assign that core to a different virtual machine. The same goes for memory, memory regions are fenced off when the virtual machine is initialized and cannot be assigned to different guests[3].

3.3.1 Removing the hypervisor

In this section we will be describing how the isolation of guests is achieved in the NoHype architecture. We will go through 3 of the major resources that need to be isolated, these are processing cores, main memory and devices. Configuration of the virtualization extensions needs to happen from the hypervisor mode of the CPU so a small program is active right before the operating system is launched and after the operating system has shut down. This program does not intervene with the execution of the operating system unless it is requested to shut down the guest, similar to the signal an operating system might get when the user pushes the power button on a physical machine.

In order to implement all of the above concepts (section 3.3) we need the help of the virtualization features of our hardware. Modern hardware has many of the **virtualization extensions** in place to make the NoHype architecture a reality (section 3.3.2). We will be discussing the hardware extension that are available and how they assist in the realization of the NoHype architecture.

The **Memory Management Unit** (MMU) maps a virtual address space onto physical memory and ensures a program can only address memory that belongs to it, otherwise it will generate an error. Using

[3] Unless the goal is to create a region of shared memory between two operating system instances

Extended/Nested Page Tables (EPT/NPT) we can add an additonal level of abstraction that performs the same kind of mapping, but now the mapping is performed between the memory that is physical to the VM and the actual physical memory. More information can be found in section 3.1.2.

In order to setup the same kind of memory restrictions from the device side we can use the **Input/Output Memory Management Unit** in a similar fashion as we did with the MMU. The IOMMU will translate addresses used in the DMA memory transfers such that they end up in the correct operating system instance running on the machine. Device interrupts and memory regions can be moved such that only one operating system has access to them.

CPUs traditionally have multiple levels at which code can be executing. Each of these levels represent **privilege levels** and each is associated with a set of instructions that can or can not be executed by the code running in this level.

Traditionally the operating system ran in the lowest level with the highest privilege, known as ring 0. Since the introduction of virtualization, chip makers have introduced a level that is even lower as the ring 0 where operating systems live such that operating systems can still run in ring 0. Ring -1 is the ring used by hypervisors on modern hardware, from this ring it can perform tasks required to manage the system, it can for example interrupt the execution of the operating system.

In the NoHype architecture the application running in Ring -1 is very limited in functionality and is only responsible for setting up the hardware, starting the guest and stopping the guest.

The amount of physical devices that can be installed in a machine is limited to the slots available on the machine. This would limits the amount of virtual machines we can run to the amount of devices that we are able to insert into the physical machine. Luckily there is a solution to this problem provided by a specification from the PCI-SIG [61] consortium. **SR-IOV**, Single Root IO Virtualization, allows PCI devices to expose virtual functions. These functions are serviced by PCI device providing the virtual function but appear to the OS as separate PCI devices which can then be assigned to a guest.

The NoHype architecture has 2 software components that control the operation of the entire system. The first component being the a System Manager which runs on its own core, and at all time for as long as the physical system is up. The system manager is responsible for the management of all the VMs on a system. Additionally outside management components communicate with the system manager to request the execution of concrete commands such as shutdown or launch a virtual machine.

The second component is the core manager which is execute once the virtual machine is initialized by the system manager, it launches in hypervisor mode such that it can perform some initial setup for the virtual machine. The core manager should set up the memory mappings, mapping the network interface cards, initialize the disk controllers and finally execute a VM-exit such that the guest OS can take control.

3.3.2 NoHype prototype

A prototype of the NoHype architecture [95] has been build at Princeton University which implements the NoHype architecture using Xen 4.0 as the startingpoint for their work.

The prototype can not run any operating system of choice because some workarounds for limitations in the hardware virtualization features have to be implemented in the guest OS kernel.

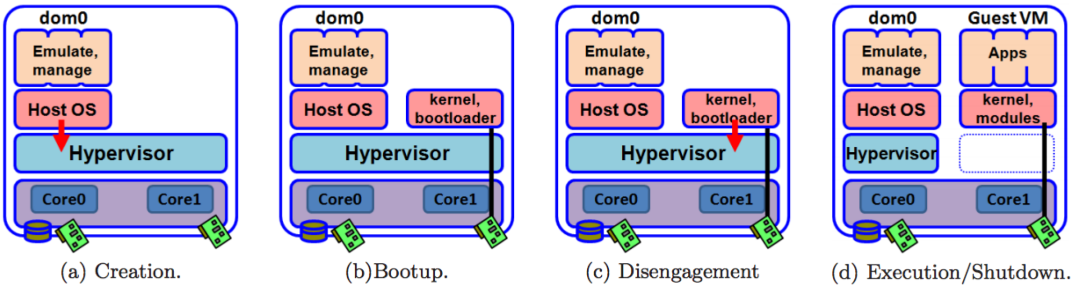
System discovery is one of those functions requiring a workaround in the guest kernel. Some instructions providing information on the system, like CPUID, are not virtualized and will provide information about the entire physical system. To solve this the prototype will run the initial stage of the VM bootup on top of the Xen hypervisor as would normally happen, all of the restricted information that the linux kernel may require is read while Xen is still active and then cached inside the modified kernel. Once all the information is read Xen will perform a VM-exit and allow the operating system to run without interruption as is seen in figure 3.14.

PCI devices cause a similar issue since the detection of a PCI device involves read a known address and checking if the returned value contains the expected result. The prototype performs this check while Xen is still active and Xen is modified to return a response corresponding to ”no device present” for all devices except the network interface card.

Since the hypervisor is still active while the PCI devices are being initialized the addresses in the vector table will be the addresses to the Xen handlers rather than those of the guest. To overcome this issue both Xen and the linux guest OS are modified to make the location of the interrupt vector table configurable.

The next issue to address is **storage**, currently there are no hard disks or RAID cards that support SR-IOV. This limitations means that access to disk are hard to virtualize only using hardware. For the

NoHype prototype use has been made of network booting instead using iPXE which can load all the required data from a network storage device.



(a) Creation. (b)Bootup. (c) Disengagement (d) Execution/Shutdown.

Figure 3.14: The four stages of a VM's lifetime in a NoHype system. [95]

3.3.3 Virtual switching

Communication between virtual machine using the NoHype architecture has to go through the hardware, which is a source of inefficiency in the NoHype architecture. Every message that has to travel from one virtual machine to another has to move over the PCI bus in order to reach the network interface, and a second time when it moves from the network card to the second virtual machine. This adds latency and causes the PCI bus to get saturated [91] by traffic that is local to the machine, which causes other traffic to slow down as well.

In the NoHype architecture the implementation of virtual switching [83] is using a dedicated virtual machine which runs a software switch. These shared memory regions can be setup using the memory management unit, the core managers can set up memory regions for each VM which overlaps with memory that is assigned to the software switch. This allows for communication between the switch virtual machine and the virtual machines seeking to communicate using the software switch. Figure 3.15 illustrates how this looks when running on a system.

The driver that has been written for this type of communication uses a polling mechanism rather than the use of interrupts such that there is no other communication required between the virtual machines.

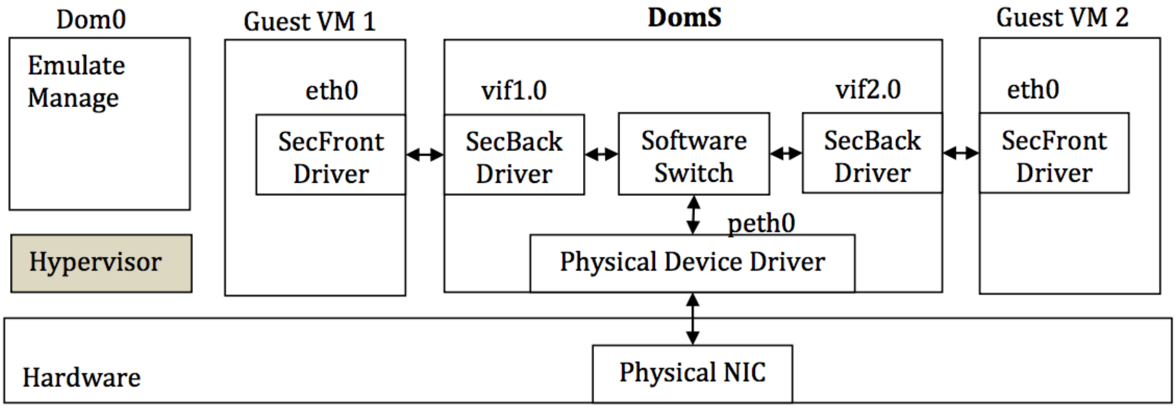


Figure 3.15: A system running the NoHype architecture and using a software switch in a service VM called DomS. [83]

3.4 Summary

This chapter has discussed different methods of virtualizing a physical machine such that we can think of it as multiple virtual machines. The techniques used to achieve this can vary considerably, we have seen paravirtualization, full virtualization and the NoHype architecture.

We described some alternate usages of these virtual machines which allow us to optimize the code being executed in the virtual machine. This increases the control over the virtual hardware by removing the middle-man, the operating system, from the equation. The alternate usages we described were unikernels which execute code directly on Xen and the rumpkernel which produces single binaries from components derived from NetBSD.

Chapter 4

Containers and process isolation

The term container in the context of this thesis is used to describe specific features of the linux kernel to isolate processes from each other using kernel namespaces. Besides namespaces, control groups are often also included when talking about containers. Using these two features we are able to create containers which strongly resemble virtual machines from the user's perspective. Each container is unaware of other containers on the system thanks to the namespaces, while resource usage of these containers can be restricted using control groups.

Compared to virtual machines however, containers use significantly less resources to achieve a similar goal as hypervisors. The overhead associated with containers are only these resources consumed by the kernel control structures. Additionally each container runs on top of the same host kernel, so the kernel is not duplicated in memory by each container instance that is started.

The question now becomes "How do we manage these features?". Of course it is always possible to manage each individual component manually, but this is hardly practical. Luckily there are several tools which build upon the kernel features and deliver a daemon, client or both. In this thesis we will discuss Docker (4.2.1), LXD (4.2.2) and the systemd lightweight containers (4.2.2). When we start our experimentation with network virtualized functions, we will be focussing on Docker. Docker has gained incredible traction in the last few years and has been integrated into a number of other projects as well. We mention LXD here since it is backed by Canonical and it has some very ambitious goals for the future of containers. Systemd lightweight containers on the other hand resembles virtual machines a little bit better and is available in any recent linux distribution that uses it as their init-system [72].

This chapter will try to provide a general idea of how containers work on linux and what some of the challenges might end up being when we start developing the proof of concept implementation of a virtualized network function running inside a Docker container.

4.1 Building blocks

4.1.1 Namespaces

Below are the six types of namespaces [86] that are currently implemented in the linux kernel. These allow us to create groups of processes, each of them with their own mount points, hostnames, inter-process communication, program ids, networking and user ids depending on the type of resource we want to assign to them.

A namespace is a collection of resources to which only the processes in that specific namespace have access. Resources created by processes in one namespace can not be accessed or detected by processes in another namespace. When attaching a new instance of the *UTS namespace*, which we discuss later, to a group of processes those processes will report a different *hostname* as other processes on the system. Namespaces thus isolate system resources rather than control the amount of them that can be used, that kind of limitations are imposed using *control groups* (section 4.1.2).

A process can belong to at most one group (a namespace) in each of the types of namespaces available in the system. Each type of namespace has its own hierarchy of groups that each can contain other groups and processes. Once a process has been assigned to a specific group, it will remain in this group and all of its children will be created in the same group as the parent process.

Launching a process in a specific group for a namespace [43] is done using the *clone* syscall which creates a new process and takes as argument a set of flags which specify the namespaces for which we

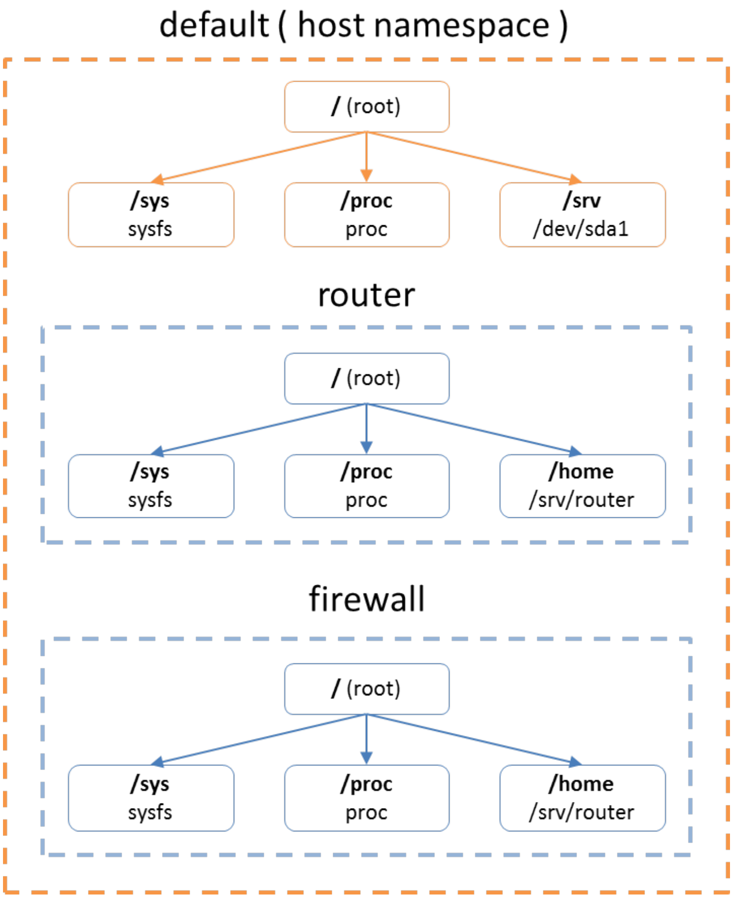


Figure 4.1: A representation of the mount namespace hierarchy and mountpoints in each of the namespaces.

would like to create a new group. For each type of namespace there is a constant with the *CLONE_NEW* prefix, for example *CLONE_NEWNET* which creates this process in a new network namespace. Adding a process to an existing namespace can be done using the *setns* syscall and the *unshare* syscall allows for a process to be isolated into a new namespace.

The first implementation of namespaces to ever appear in the linux kernel (Linux 2.4.19) was the **mount namespaces** which isolated the mount points available to a group of processes. Instead of having *mount()* and *umount()* operate on a global level they now operated only on the namespace from which they were called.

This results in each container only being aware of their own calls to the *mount()* and *umount()* syscalls. Any filesystem mounted inside a container will only be visible to the container and not to any other namespace. An example of namespace hierarchy and the individual mount points inside each namespace can be seen in figure 4.1.

UTS (UNIX Time-Sharing system) namespaces introduced the isolation of domainname and nodename from other namespaces. Here the *uname()* system call has been changed to work on the namespace level instead of global. This feature was introduced in the Linux kernel 2.6.19.

As of linux 2.6.19 inter-process communication has been split into a separate namespace as well, namely the **IPC namespace**. Identifiers used for inter-process communication are now unique to the namespace in which they were created. This means that identifiers can be reused across containers without them interfering with each other. The two ways of inter-process communication that are aware of these namespaces are System-V IPC and POSIX message queues.

PID namespaces were introduced in the linux 2.6.24 kernel. This feature allows the linux kernel to assign multiple process ID's to a single process. Each namespace has its own pool of ID from which an ID can be taken. Each process inside a container has at least two process ID's, one id is assigned on the host and one ID is assigned from the namespace of the container. This enables each container to run it's own init process (process 1) and assign process ID's as it sees fit without creating conflicts. Using the process ID assigned on the host system processes can be controlled and managed by the host system itself.

A process might have more than one ID assigned to it, this kind of scenario may occur when we choose to run another container from within an existing container. In these situations there is an ID assigned for each level in the hierarchy.

The **network namespaces** separate everything related to the operations of the networking of the container. This feature, introduced in linux 2.6.24, gives a container access to its own network devices, ip addresses, routing tables, port numbers and the */proc/net* filesystem. To manage the network stack in a container we can use the *ip* command which can change the configuration of a specific namespace. As with the other namespaces the system calls related to these functions can only affect processes within the same namespace.

Running any command in a namespace other than the current one looks like this for the networking stack. It is important to note that the *ifconfig* command is not namespace aware and we will have to use the *iproute2* suite of tools to configure networking on a system running containers.

```
ip netns exec <ns> ip <remaining command>
```

A feature that had long been missing from the kernel were **user namespaces**. User namespaces, introduced in the linux 3.8 kernel, allow user ID's to be mapped onto different ID's on the host system. This allowed containers to be started from a process other than root, since we now could map user ID 0 onto any ID on the host system, 10 000 for example. From a security point of view this is a major improvement (see 4.3). There are some concerns that there might not be enough ID's available to the host system to map each user ID in containers to a different ID on the host system. Others argue that ID's on the host system can be shared since mechanisms like SELinux (section 4.3) should be employed to limit the ability of processes should they break out of their containers.

4.1.2 Control groups

In namespaces we have discussed how processes are grouped into namespaces, now we wish to allocate a fixed amount of resources to the processes in these groups of isolated processes. This is however not how control groups [5] work, there is specifically no linkage between the groups that are created using namespaces and the groups created using control groups. In order to specify the amount of resources available to a container we have to create a set of processes that matches those in the namespace we wish to limit.

Once processes are divided into their respective control groups we can assign the resources available to that control group and specify the limitations on the use of those resources. There are a many amount of control groups available to the user of this feature, some of the most interesting for this thesis are since they control resources which we wish to control:

- **devices:** Limits the devices that can be used by the apps in a control group.
- **hugetlbfs:** Limits the access to hugepages.
- **cpuset:** Limits the cpu's on which processes can be scheduled and run.

Using the **hugetlb control group** it is possible to set an upper limit on the amount of *hugetlb*[1] memory that can be allocated by that specific control group. Any application in this group trying to allocate more hugetlb memory will report an out of memory error while there might still be hugepages left on the host system. As an example we are imposing a limit of 4GB, which equals 4 hugepages of 1G, on the cgroup with name *test*.

```
mkdir /sys/fs/cgroup/hugetlb/test
echo $((4*1024*1024*1024)) > \
    /sys/fs/cgroup/hugetlb/test/hugetlb.1GB.limit_in_bytes
```

The **devices control group** makes it possible to assign specific rights to device nodes in the system. It is possible to assign *read*, **write** or **make** rights to a device node or a type of a device node. This limits the access to files under the */dev* directory. The below example revokes the default rule that grants access to all device nodes, the second rule then re-enables the device node with the major number 242 and the minor number 2. The processes in the test cgroup can now read (r), write (w) and make (m) this type of device nodes.

[1] Continues sections of equal size memory zones that are accessible using a single entry in the MMU.

```
mkdir /sys/fs/cgroup/devices/test
echo ”a ∗:∗ rwm” > \
    /sys/fs/cgroup/devices/test/devices.deny
echo ”c 242:2 rwm” > \
    /sys/fs/cgroup/devices/test/devices.allow
```

When reserving specific cores in the system for a specific process a mechanism is required to enforce this reservation. The **cpuset control group** allows specific cores to be assigned to a control group such that they can be used to execute any of the processes in that control group. Other cores will be unusable and when trying to affinitize a processes to them it will fail with an error.

Imagine we would like to limit the cores that our current shell can use to logical cores 4, 5, 6 and 7. We would first have to create the group to which we will be adding our shell, we will be calling it *test* like in previous examples. Next we have to specify the NUMA nodes which can be used by this cpuset which is node 0, followed by the cpus that can be used by the group. Lastly we add the PID of our shell ($$) to the *tasks* file, at this point we can only run code on these cores.

```
mkdir /sys/fs/cgroup/cpuset/test
echo 0 > /sys/fs/cgroup/cpuset/test/cpuset.mems
echo 4,5,6,7 > /sys/fs/cgroup/cpuset/test/cpuset.cpus
echo $$ > /sys/fs/cgroup/cpuset/test/tasks
```

If we would now like to verify that the allowed logical cores are enforced we can try to launch an new shell on a different core using *taskset*, which should result in the error message as displayed below. On the other hand, launching a process on a core that has been assigned to us should run fine.

```
> taskset 0x1 bash −c ”echo hello world”
taskset: failed to set pid 0’s affinity: Invalid argument

> taskset 0x16 bash −c ”echo hello world”
hello world
```

4.2 Management platform

4.2.1 Docker

Docker's intention is to provide a way to build, ship and distribute applications without having to worry about the platform it will end up on. This is achieved using two Docker components, the *Docker Engine* and *Docker Hub*.

Docker Engine is the component this thesis will focus on, it is the daemon that manages the containers on a system, the client to interact with the daemon and the packaging tools. Docker Hub is a centralized repository for images that can be used to build upon when creating an application. Lots of pre-build images can be found here in order to spin up a container quickly and without any real work needing to be done.

Figure 4.2 illustrates all of the features inside linux that Docker uses in order to set up a container on a linux system.

The functionality required to create and launch containers is provide by *libcontainer* [31]. It is libcontainer that is used by the Docker binary to provide most of its functionality. The developers behind libcontainer are the same as those driving Docker development, any functionality implemented in Docker using libcontainer is available to other developers as well through this library. The library would allow us to develop our own daemon and client to deploy containers, however this is not the intention of this thesis.

One of the main features of Docker is its UnionFS, the feature that allows us to build a container using layers to compile an image for our container rather than having to build each container from scratch. A full disk image consists of serveral images layered on top of each other such that the final result is an image containing all the required functionality for an application to do its work. UnionFS is the generalized name used by the Docker documentation for all the layering options Docker provides for a disk image. At the time of writing there are four options available in Docker that can be used as the driver for UnionFS: AuFS, btrfs, vfs and DeviceMapper.

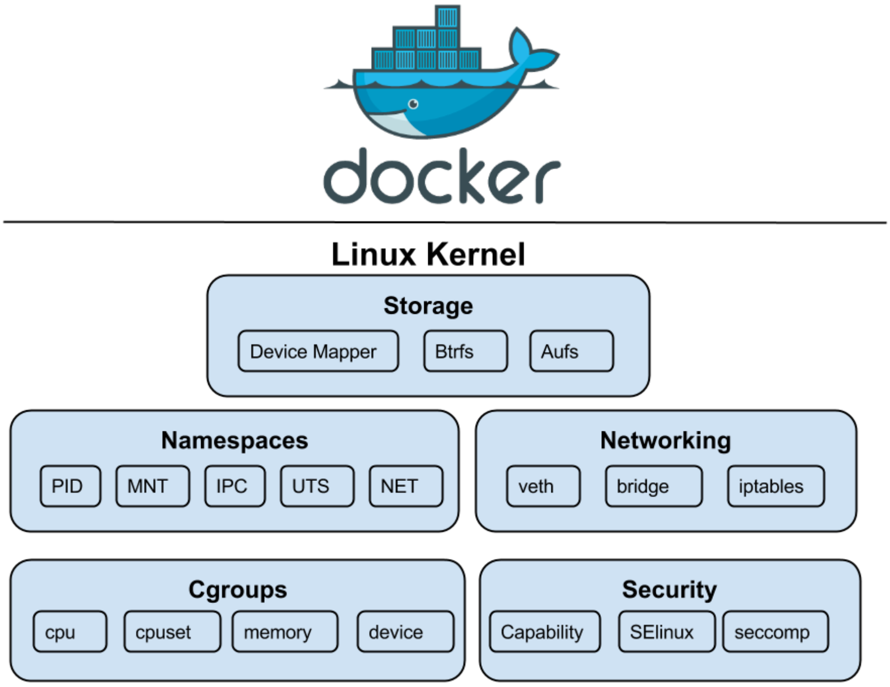


Figure 4.2: Features inside linux that Docker uses. [13]

Using most of these options each layers is a single images, and only the top layer is writable for the container. While the default for Docker is using DeviceMapper since it is available on most linux distribution we will shortly discuss them all below [70].

AuFS [16] is not available on the big linux distributions like Red Hat Enterprise Linux, which is the biggest concern for using AuFS. AuFS is a filesystem implementation that can layer multiple directories on top of each other, where only the top directory is writable. This maps nicely with the requirements that Docker has for its container images.

Btrfs [17] can be used as the storage backend that Docker uses. The layering is implemented using file system level snapshots. Each layer is stored on the btrfs filesystem as a subvolume.

Vfs is the least efficient storage backend that Docker has. Each layer is represented by a directory containing all the files of that layer and all the layers below it. The problem with vfs is that it has no real concept of layers or copy-on-write and thus performs a deep copy of all the lower layers in order to create a new layer based on others.

DeviceMapper is the part of LVM2 that lives in the kernel and we are able to leverage this with the DeviceMapper Thin Provisioning module. The thin provisioning module allows us to use a pool of block devices and create new block devices on top of them. It is then possible using the thin provisioning module to create copy-on-write snapshots that will represent our layers. However, instead of using real block devices Docker uses regular files as the underlying block devices for DeviceMapper such that the user does not have to concern himself with the setup of DeviceMapper.

While disk images use one of the above backends to manage the container filesystem, we can also have storage that survive container shutdown or that can serve as a directory to exchange data between two containers. For this purpose Docker has **disk volumes** [36] which are directories that are not part of the container image that are mounted into the container mount namespace such that it has access to them. This allows for large disks to be mounted inside a container such that it has sufficient space, or as we will do for our virtualized network function mount a *hugetlbfs* filesystem such that our VNF will have access to HugeTLB memory.

On the networking side, the default and only network topology in Docker uses a linux bridge and iptables to setup a private network segment on the host which is connected to the outside world through an iptables *MASQUARADE* rule. This *MASQUARADE* sets up Network Address Translation which is equivalent to what any home router does in order to allow multiple devices to access the internet. Other network configurations can be created using other tools such as *ip netns*(section 4.1.1), however Docker will be fully unaware of its existence and management of such configurations is complicated because of this. Scripts exist to ease the integration of existing network configuration such as Open vSwitch [60]. At the start of this thesis the *docker-net* script [67] was written to create custom configurations using only the networking stack of linux.

This has the disadvantage that without the proper iptables rules the containers will not be accessible over the network from other machines in the physical network. However docker provides the necessary options to configure iptables automatically with the correct rules to forward certain ports to certain containers, much like configuring port forwarding on a home router.

Now that we have a bridge we still need a network interface to assign to the container in order for it to communicate through the bridge. Docker creates a pair of peer interfaces and connects one peer to the bridge and the other is assigned to the container. Traffic that enters one peer will exit the other peer, so any traffic sent by the container will end up in the bridge. Any traffic that enters the bridge and is destined for that specific container will enter the peer interface attached to the bridge and exit the interface that has been assigned to the container.

This however does not allow us to reach the best performance available from our containers since this uses the technique of moving traffic as we discussed in the intro where packets are being copied multiple times while moving through the kernel. While this is the default, and only way to configure a Docker container we will have to investigate how to provide alternative means of communication to our containers.

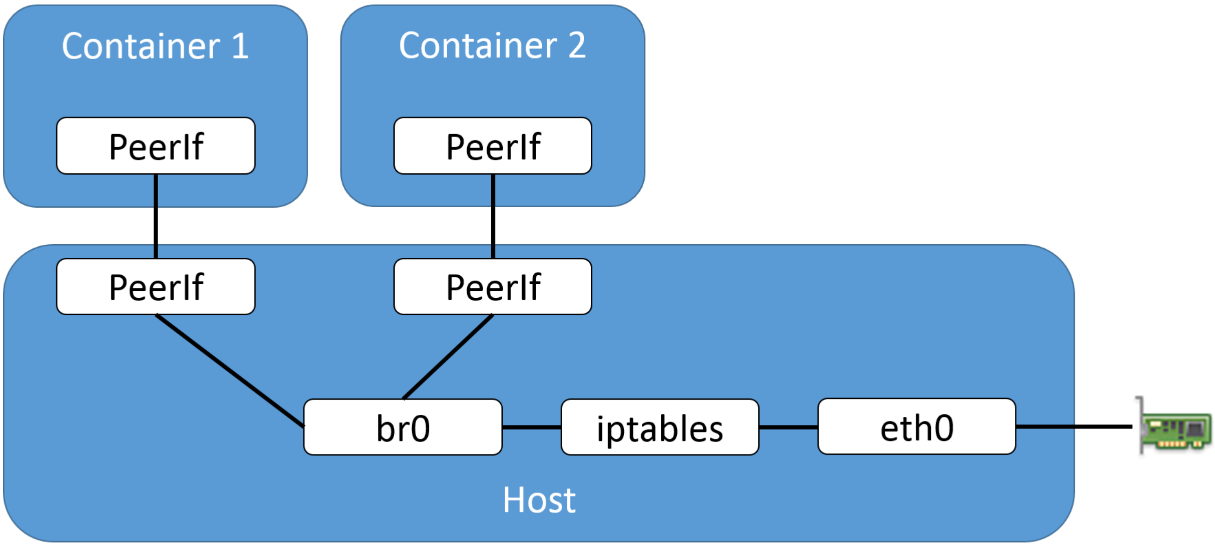


Figure 4.3: Default networking provided by docker using a linux bridge and peer interfaces.

4.2.2 Other container tools

While our focus is going to be on Docker, it is not the only tool available to control containers on linux. Many different tools have been developed for or integrating linux containers.

LXC [35] is one of the older projects that manages containers on linux, it has been around since 2008 when it was first released. It consists of tools, templates and library and language binding for all of the containment features supported by the kernel. These allow container to be easily spun up from templates with minimal effort, these tools however only operate on the local machine only and do not contain the tools to manage a fleet of container hosts.

Late 2014 Ubuntu announced [46] **LXD** ”the next-generation container hypervisor for linux”, which resembles Docker. LXD, pronounced lex-dee, consists of multiple components, just like Docker it has a Daemon that exposes a REST API that can be used to control containers that are managed by that daemon. *lxc* will be used to interact with this REST interface and will be the command line client tool that users will be using when managing containers from the command line. The final component in the LXD project is a plugin for OpenStack Nova, *nova-compute-lxd*, that allows us to use OpenStack to manage large deployments of virtual machines.

Even though LXD looks similar to Docker it has a different philosophy as to what it actually deploys. LXD’s main focus is to deploy full system-like containers which should resemble virtual machines as much as possible. Docker on the other hand is focused on application deployment and building self-contained packages to deploy these application with.

Running containers on a recent linux distribution that comes with systemd as the init-system[2] is quite easy, **systemd lightweight containers** allow the user to launch containers using the *systemd-nspawn* tool to launch containers with incredible ease and without any additional tools. Unlike LXC, even remote management is possible when using systemd to configure and manage containers on a system.

On the fedora wiki [15] the usage of systemd lightweight containers is further detailed and shown how to run containers using a unit file[3]. Using unit files we can deploy containers which are run at system startup, or when they are needed using socket-activation.

OpenVZ [56] has been around for quite some time (2005) [55] and is not really just another tool for linux containers. It used to require a custom kernel to achieve full containerization but it can run on recent mainline linux kernels as well. Traditionally it has been used by many service providers to offer Virtual Private Servers at lower prices than the traditional fully virtualized servers using a real hypervisor. OpenVZ is used by Virtuozzo as a basis to achieve the containerization. OpenVZ has had a lot of time to mature and supports some features which are still lacking in other tools that rely solely on the features provided by the mainline linux kernel. One of these features is for example live migration, which has been available in OpenVZ since April 2006 which is only now starting to make its way into Docker and LXD.

Linux VServer [34], like OpenVZ, is not just a tool but instead requires a patched kernel because many of the isolation features are implemented as patches to the linux kernel and the namespaces and cgroup features are not used by linux VServer.

4.2.3 Containers on non-linux platforms

Containerization is not a phenomena that is exclusive to Linux, it did not even start on the linux kernel at first. Many different operating systems have support for comparable features. Solaris has Zones, FreeBSD has jails, other BSD variants no longer have a jail implementation [71] and Windows does not have operating-system-level virtualization built into the kernel yet.

Solaris Zones [58] implements the same features as linux containers but through different mechanisms. Solaris makes a distinction between two types of zones, global and non-global zones. The global zone has access to all processes and resources on the system, while a non-global zone is comparable to a container on linux. It is unable to access resources outside of its zone and cannot detect other zones or their contents.

An interesting extra feature is what is called a Branded Zone, a branded zone can behave differently to syscalls than is the default implementation. It is for example possible to run older versions of solaris such as Solaris 8 and 9 in a branded zone. On Solaris 10 branded zones were even more surprising as it was capable of running a linux distribution rather than a different Solaris environment, this is however no longer supported in Solaris 11 [57].

On **FreeBSD** the equivalent of a container is a Jail, a jail has its own directory which represents the root inside of the jail, a hostname, IP address and its own set of users. Just like containers resource access is controlled by the kernel and jails can only see their own environment and the processes in them.

Container implementations in other BSD variants such as NetBSD and OpenBSD were inspired by FreeBSD jails but were discontinued on the 3th of March 2009 due to flaws inherent to syscall wrapper-based security. [71] [14]

While **Windows** currently does not yet have support for containers built into the kernel they seem to be catching up [39] [40] to the trend. However it does not means that containerization on Windows was not happening at all, commercial products like Virtuozzo allowed multiple applications to be deployed on a single windows server installation without requiring full fledged virtualization. A different alternative that could be found for virtualization on Windows was Spoon [68], which like Docker has the intention to package Windows applications such that they run on any Windows environment where Spoon is installed.

4.2.4 Cloud orchestration

Many different management platforms for containers are starting to become available, in late 2013 CoreOS made their first release and in October 2014 Kubernetes [30] was released. While these tools specifically for

[2] The initial application running that has the responsibility to launch all the system services.

[3] A configuration file that details how a service should be run by systemd.

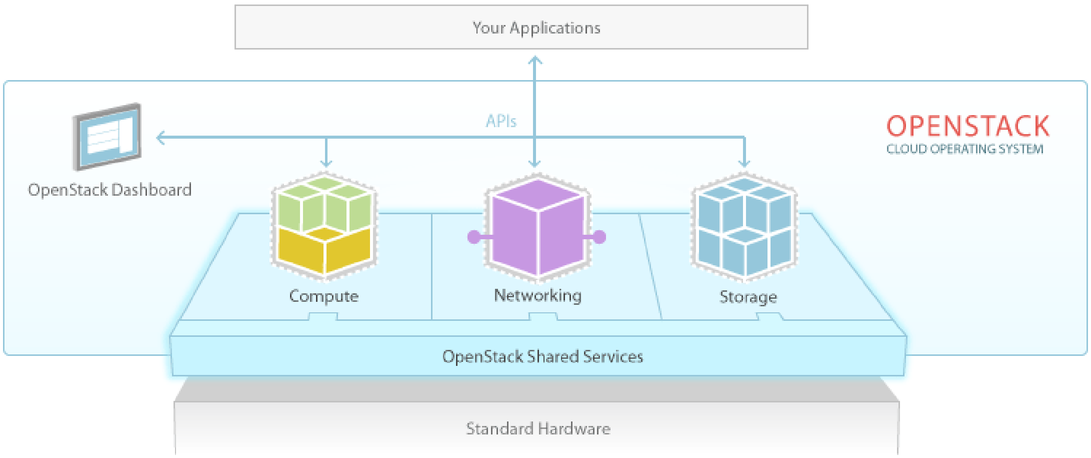


Figure 4.4: The OpenStack software stack. Source: https://www.openstack.org/software/

container deployments come available, existing tools like OpenStack are being integrated with containers. With the availability of the required tools to launch a containers in a cloud setting, cloud service providers are starting to release containers-as-a-service platforms. All of these tools and services are part of the rapid expanding ecosystem around containers of which we will describe only some.

OpenStack [51] allows administrators to manage thousands of machines using the OpenStack Dashboard or API. OpenStack works with many of the more popular virtualization technologies used today in order to provide a flexible platform to manage computing, storage and networking resources. Each of these resources are managed by different components in OpenStack.

Nova [54] is the component of OpenStack that provides the computing resources, commonly represented as virtual machines but these can be containers or physical hosts as well. The supported platforms to run compute resources on can be extended using plugins, for example LXC (nova-compute-lxc), LXD (nova-compute-lxd) and Docker (nova-docker) all have plugins for OpenStack.

The networking component is called Neutron [53] and supports different connectivity options like Open vSwitch or the standard linux bridging.

Storage is managed by the Glance [52] service which manages the storage available for guest placement and in the case of Docker contains the container images.

All of these components interact using a RESTful API and can be integrated into many different tools, but can be managed using the OpenStack Dashboard just as well.

A different option would be CoreOS, the **CoreOS** website [6] describes it as "Linux for Massive Server Deployments". CoreOS is entirely based on containers, using Docker, to provide easy to scale and manage machines. CoreOS differs from other solutions in that it also provides a central configuration repository that allows to change the configuration of a container without ever having to touch your application or the container it runs on.

This central configuration management comes in the form of *etcd*. Etcd provides service discovery and configuration management using RESTful API endpoints. Any application running on a CoreOS cluster can read or write from etcd in order to update its current state.

If an additional webserver would be started to cope with the amount of traffic that is currently flowing to the website, traditionally we would have to update our load balancer configuration to include this new webserver. On CoreOS the additional webserver simply writes an entry in etcd and the loadbalancers can automatically add it to the pool of webservers by looking up the address of the extra server in etcd.

To manage the containers and where they run CoreOS provides *fleet*, fleet uses systemd unit files to start and stop containers. Depending on the properties declared in the unit file fleet will schedule your container on the cluster that is available to CoreOS.

If you are not interested in deploying your own cloud however you can count on the well-established cloud vendors. Some of the large players in the cloud industry have started to provide services based on container technology. Both Amazon and Google have announced the roll-out of their **Container-as-a-Service** offerings. Both of these service provides allow the customer to run container services on top of their virtual machine infrastructure. The customer is billed for each of the virtual machines he uses to run the containers instead of per container. Customers can control the containers running on their virtual infrastructure using the familiar control panels that both of the providers offer.

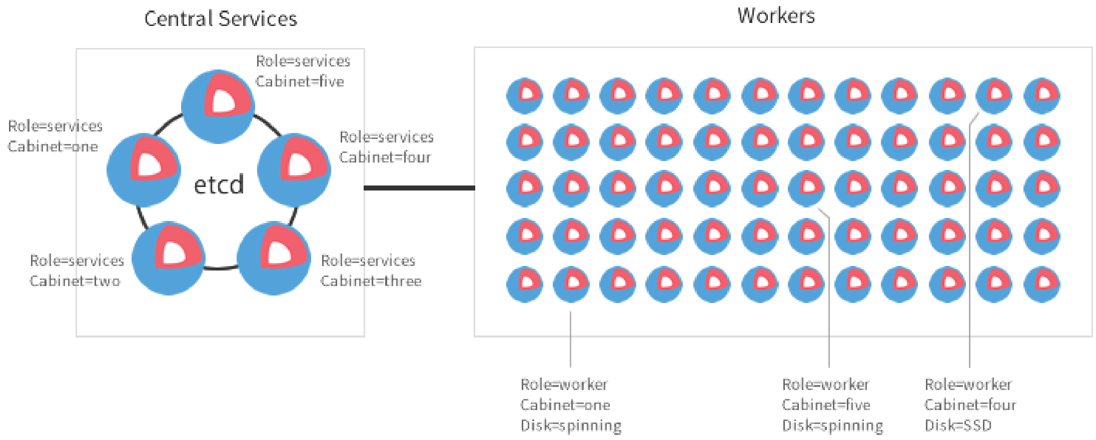


Figure 4.5: A CoreOS cluster with etcd containers and workers which get information from etcd. Source: https://coreos.com/docs/cluster-management/setup/cluster-architectures/

While the goal here is to increase application density, there is no performance benefit gained from this kind of deployment because each application is still dependent on the hypervisor software to service hardware interaction. It is understandable that cloud service providers are taking this approach since containers still have to prove that they are secure and will not interfere with other occupants of the hardware.

Google Container Engine [19] is Google's offering in the Container as a Service business and it runs on top of Google Compute Engine instances. Google has developed Kubernetes [30] to schedule containers, and it is Kubernetes that they run to manage the containers deployed on the Google Compute Engine instances.

EC2 Container Service [1] is what Amazon has to offer and it runs on top of Amazon's Web Services EC2 instances. At this time the management of the containers however still has to happen through the AWS API.

4.3 Security

The community has had some concerns with container security since it does not provide the same level of separation between the guest systems and the host. While traditional virtual machines only share the hypervisor and run their own kernels, containers share the entire kernel with each other. The kernel provides for a much bigger attack surface than a hypervisor does. On top of this a process running as root within the container would also have root privileges outside of the container, until kernel 3.8, should it ever break out. The question has to be raised: how do we protect the host from attackers within a container?

The **user namespaces** allow for greater security in containers since kernel 3.8, user id's can be mapped to a different range of user ID's on the host system. This means that the user or group id within the container can be different from the id seen outside of the container. Specifically this means that containers can be launched using a regular user instead of root so that when a process would break out of its container it would not have root privileges on the host system. This reduces the amount of harm a process can do on the host system significantly.

A common believe is that a root account on a linux machine can do anything. Although this is generally true, it does not have to be. **Capabilities** can be used to specify what a root user can or cannot do. There are in total 32 capabilities available which limit the different actions that a root account can do. An example is *chown*[4], normally a root user can change the ownership of a file regardless of the permissions set on the file, this can be limited by disabling the *CAP_CHOWN* capability. Another example is the capability to create device nodes on the system, this can be controller using *CAP_MKNOD*. [4]

Libseccomp allows dropping system calls from the syscall table that should not be called by a process. For 64 bit containers there are in total 649 system calls in linux, of which 338 [33] are 32bit system calls which can safely be disable on a pure 64 bit system. [79]

[4] Utility on linux to change file ownership.

Limiting the system calls that can be made from a container seriously limits the attack surface of the kernel. After all, these system calls are how processes interact with the kernel and are able to exploit potential bugs in the kernel.

An extra layer of security is integrating **SELinux**, or the similar APPArmor, to contain processes within the filesystem that has been provided. Implementing SELinux security allows us to ensure that even when a process breaks out of its container it will be limited in the damage it can cause. SELinux provides mandatory access control, meaning that access to a resource should be explicitly granted to a *subject*, which can be a process or thread. It allows for much more precise control over the permissions granted to processes than the default Read-Write-Execute model that linux provides.

In the context of containers we can use SELinux to label the filesystems that a container will be using and grant processes in the container only access to resources that have this label. Should a process now break free of the isolation it will have to exploit SELinux before being able to infect other containers or the host operating system [9].

4.4 Unresolved issues

Much of the information about kernel structures is read from the **/proc and /sys pseudo filesystem**. While these are available to containers they present a problem, these structures represent the host kernel data structures. These data structures in turn represent the state of the entire system, and not only the container from which these filesystems are inspected.

Some of the most used userland tools read information from this filesystem instead of making a system call, if there even is a system call. The amount of free memory and memory usage in general is one of these sets of information that is represented by the */proc* filesystem, */proc/meminfo* to be more precise.

One of the most used utilities in linux to inspect memory, *free*, will use the */proc* pseudofilesystem to retrieve the amount of free memory. However, since this file represents the state of the entire system and not the container it will display the total amount of memory free to the host kernel. It does this regardless of the limitations imposed on the container it is being called from. This might result in applications reporting that they are out of memory while the *free* utility still reports the availability of system memory.

Trying to retrieve information about the available CPU's to the system through */proc/cpuinfo* or */sys/devices/system/cpu* will report all the CPU's available to the host kernel, regardless of the *cpuset* imposed on a container. A process or thread might try to set it's affinity to a core it knows is available, but still fail because it is not authorized to access it.

The correct information for a given container is available through the *cgroupfs* mounts but these are not exposed to the container itself for security reasons. To make matters even worse, the cgroupfs file follow a different format as the files under */proc* and thus can not be used as drop-in replacements [38].

Live migration is a feature that none of the container projects that rely on the mainline linux kernel have. The CRIU [7], Checkpoint/Restore In Userspace, project is a live migration implementation that is able to migrate containers from one host to another, but currently there is no integration with any of the tools that manage containers on a system. Because of this we cannot move a running container to a different host system when it is required. Instead it is a more general practice to simply restart the container on a different host since startup time (see 5.2) is considerably less than the startup time of a virtual machine. This however requires the remainder of the infrastructure such as routing to be updated such that requests for the application are routed to the correct container.

While we see the shared kernel as a benefit since it brings better memory efficiency and less overhead it can have a negative impact as well. When an application creates copies of itself endlessly we are talking about a **fork bomb**. When a process fork's we are referring to the *fork* system call which creates a new child process which inherits all of the memory of its parent process. The fork bomb allows a denial of service attack[5] on the host kernel since it is an expensive operation for the kernel to perform, calling this system call in rapid succession will put a high load on the kernel and the system call infrastructure causing the kernel to become unresponsive to other system calls.

Currently there is no real solution to these kinds of security problems inherent to the linux container architecture. There are some solutions to the very specific problem described here, but any system call could cause this kind of attacked when stressed hard enough.

[5] The system is overwhelmed with fake operations such that it will not service legitimate operations anymore

4.5 Summary

Containers are a different breed of virtualization as they do not virtualize a machine. Instead containers virtualize kernel subsystems such that processes running in a container only see the resources from that subsystem that have been assigned to them. Effectively containers isolate processes from each other rather than virtualize the machine. There are 6 subsystems which have support for containers, these are mounts, unix time-sharing system, inter-process communication, process ids, network and user ids. A single group of processes within such a subsystem is called a namespace, one of the building blocks of containers.

Limiting the resource usage of these containers is also the task of the kernel, this is achieved through the use of control groups. Control groups allow us to impose limits on resources such as memory, processing cores and devices. Like a namespace, a control group is a set of processes however these groups have to be defined separately.

Different tools have been discussed among which was Docker. We will be using Docker in our proof of concepts so we went into a some more detail when describing it. We discussed how Docker defines containers and how it sets them up, specifically the default network and filesystem configuration.

We continued the chapter discussing the available mechanisms to secure container deployments since this is still considered a weakness of containers. We have been looking at the user namespaces which allow containers to be launched as non-root users. Capabilities which enable us to strip a root user from his permissions. Libseccomp enables us to harden the kernel by disabling system calls that should not be available. And finally SELinux which helps to contain processes on their own filesystem.

The last section of this chapter discusses some of the problems inherent to containers. A problem we will be seeing again later on is that some parts of the kernel are not yet aware of namespaces, such as the */proc* pseudo filesystem. Another problem that will be a factor in our final conclusion is the missing support for live migration in most container tools that do not integrate *CRIU*.

Part II

Containers in practice

Chapter 5

Comparing linux containers to KVM virtual machines

Although the comparison is often made between virtual machines and containers, they both are very different methods of virtualization. While a hypervisor emulates or passes through hardware, containers will be using the resources given to the linux host without emulating any of the devices it has attached. In this section we will explore the differences between a KVM virtual machine and a container that is created using the features available in the linux kernel.

5.1 Choice of operating system

When running a virtual machine we expect to receive a full emulation of a physical machine, from the bios to the hardware that the virtual machine uses. Having virtual hardware that resembles normal physical hardware allows us to install any general purpose operating system we like. This choice of operating system comes at the cost of having to load multiple instances of the same code into memory, but gives us greater freedom.

When using containers we are a bit more limited in our choice of operating system. Since the isolation of processes happens in the host kernel, and all processes run on the same kernel, there is no duplication of the operating system kernel. The downside of each container using the same kernel is exactly that we cannot change the kernel from inside the container or load any kernel functionality that was not present on the host system. This means that if we require a kernel module to be loaded, it has to be loaded on the host system itself and it will be exposed to every other container running on the same system.

This however does not mean that we are limited to a single operating system, different flavors of operating systems are still possible. We can for example start a container that apart from the kernel behaves like an Ubuntu system, on a host that is actually a CentOS machine. So different flavors of the same operating system are often possible as long as the binaries are compatible with the kernel that the host operating system is running.

5.2 Startup time

The advantage of the virtual machine in the previous section will become the disadvantage in this section. Since a virtual machine emulates a full machine it also goes through the booting of bios and operating system. A container on the other hand does not have to perform any booting ritual, it simply launches a process in the correct namespaces and it is done.

Since launching a container is essentially launching a process it is very beneficial for startup time of a container. Comparing the startup times of a container and a virtual machine produced the results in figure 5.1.

A script was created that would record the start time of the measurement, then launch the container or virtual machine and wait for a message that will be sent from inside the container or VM. Once it receives this message it will record the time again and store this as the time at which the container or VM is ready to be used.

When measuring the time at which a container would be ready to be used we simply started the script that would signal back as the process we wanted to run. When this script runs it means that the runtime

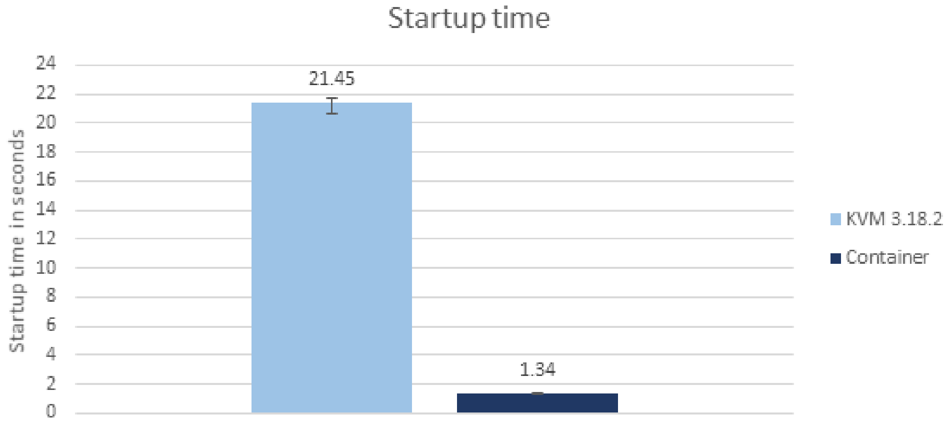


Figure 5.1: Average startup time of a container versus a KVM virtual machine

environment inside the container is ready and the container can be used.

In the VM we attached the script that would signal back to the init script that is launched when the network is up in the VM. At this point the VM is ready to be used and any service running on it could be launched. We choose this point in time of the VM's startup sequence because this is where most services will be launched that need to be accessible from the outside such as webservers, SSH servers and so on.

5.3 Device emulation

When running a virtual machine most of the hardware that is presented by the hypervisor is emulated, it is not really there, it is simply an implementation in software. This kind of device emulation is not very efficient for most hardware but allows for some interesting use-cases such as the usage of shared memory between virtual machines. The shared memory that is used by DPDK to communicate between virtual machines or from virtual machine to the host is called IVSHMEM and it is presented to the virtual machine as a PCI device [28] [27] [64]. What happens behind the scene is that Qemu maps the memory that will be shared into its own virtual address space and then exposes this memory to the virtual machine as a PCI BAR[1]. When writing or reading from this BAR we are actually reading or writing from/to the memory that Qemu mapped for us earlier. All of this is possible in virtual machines because there is a hypervisor to set up the device structures in memory required for the guest operating system to detect a device is present. However, containers do not have a hypervisor that is emulating devices and thus it is not possible to create emulated devices that are not really there. This means that the current technologies like VHost or IVSHMEM will need to be adapted in order to be functional in containers.

5.4 Disk footprint

When running a virtual machine a disk has to be provisioned which the virtual machine can use to install a full operating system on. The operating system install usually includes files and programs which have no use for the chosen configuration or will simply never be used. Containers can use any filesystem that is mountable as a root directory. If the application that is going to run inside the container does not even require write access to the filesystem it is possible to use the host root filesystem in read-only mode.

When using Docker there are highly optimized base images available for most of the popular distributions. These images are sometimes smaller than 500MB while a clean install of most operating systems results in multiple GBs of data. On top of these base images a developer can then layer the tools he requires and the applications he wants to run. These layers only contain the differences between the current layer and the previous one and thus are usually quite small. If any of the layers are re-used by other containers they are simply used as a read-only layer and not duplicated on disk. This ensures that multiple containers running the same base image do not fill up the disk space lineair to the amount of

[1] Base Address Registers, used to configure the PCI device

containers running. Below are the sizes of some of the images available on Docker Hub, to compare this with for example a minimal CentOS 7 install which has a disk size of 1.1GB after a clean install.

Operating System Flavor	Image size
Ubuntu 14.04	188.3 MB
CentOS 7	229.6 MB
Debian 7	84.98 MB
Fedora 21	250.2 MB

Table 5.1: Docker disk image size for popular distributions on Docker Hub

5.5 Memory footprint

The amount of code that is actually loaded into memory is significantly less with containers compared to virtual machines. This is very easy to explain, only the application is loaded into memory when running containers. When running the *sleep* command in a container that we launched with Docker we will lose 17MB of memory in total from the memory available on the host system. This number includes the amount of memory necessary to create the containers as well as the memory allocated by Docker to manage the container. When running a virtual machine it is not only your application that is in need of memory, the operating system and the hypervisor itself will need memory as well which adds significant overhead to the memory requirements of your application. For optimal performance inside our virtual machine we launch Qemu with the –mem-prealloc flag which causes Qemu to allocate all of the memory when the virtual machine is initialized. This means that a virtual machine always uses the total amount of memory allocated to it at minimum. A CentOS 7 virtual machine which has been given 256MB of memory without pre-allocating it will end up using about 125 MB of memory when booted. Which is over 5 times more than the equivalent Docker container uses. When assigning more memory to a virtual machine it will start utilizing more despite it being the same virtual machine and sitting idle.

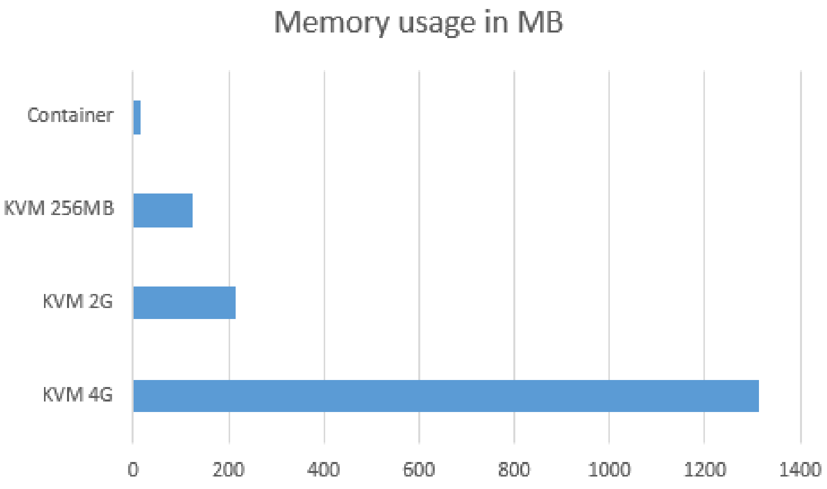


Figure 5.2: The memory used by a container versus the memory used by a KVM virtual machine with varying memory sizes.

5.6 Density

Because containers have a smaller footprint, both in memory and on disk, they allow more of them to be launched on a single system as there could live virtual machines on the same hardware. This improves the utilization of the hardware when a machine's CPU is underutilized but no more guests can be added because of the limitations imposed by the disk or memory that is available. For VNF's using DPDK we are limited more by the number of CPU's available in the system, because for optimal performance we rather not reuse the same cores for different VNF's running at the same time on the same system.

Because of this the memory and disk limitations are less of an issue since we can add more memory to modern systems than we can add cores. As the example below demonstrates we are more likely to run out of processing power than we are to run out of memory, even when we would run multiple VNF's on the same cores. The maximum amount of memory we can add to the board used in our tests [2] is 1536GB while the amount of processors we can add is limited to 2. With the maximum of cores per CPU limited to 18 on an Intel E5-2699v3 CPU [3], this means we get a total of 36 threads per CPU and 72 threads in the system. Resulting in the possibility to allocate 21GB of memory per core which is more than plenty for modern applications.

5.7 Configurability

When deploying a DPDK application in a virtual machine we can use DevOps tools [8] to configure the virtual machine according to a predefined configuration. Or we simply have images laying around that each have different configurations baked in. There is no obvious way to pass a configuration into the virtual machine when launching it. When launching a container we are actually launching a process, this implies that we can also pass configuration down to the container. Applications that we could not flexibly deploy into a virtual machine can now be deployed easily by deploying it in a container. We can supply application configuration simply on the commandline when launching a container, or we could mount a directory containing the configuration of our application inside the container such that the application will transparently pick up the correct configuration. Being able to deploy VNFs, and applications in general, in such a way allows much easier orchestration where the application was not originally created to be orchestrated. Applications can be packaged up as software appliances for which a user should only supply the necessary configuration and not worry about the internals of the appliance. The benefits of this can easily be demonstrated using an image for a webserver, nginx [47] [48], available from Docker hub. This image allows us to deploy multiple webservers without ever having to reconfigure nginx for a different html root directory. We simply specify the directory we want to serve when launching the container.

```
# docker run −v /some/content:/usr/share/nginx/html:ro −d nginx
```

We could similarly deploy the Data Plane Peformance Demonstrators application [22] (DPPD) by passing the location of the configuration file for DPPD on the commandline. For example a BNG:

```
# docker run −i −t dppd:v015 http://192.168.0.2/bng.cfg
```

Or a router

```
# docker run −i −t dppd:v015 http://192.168.0.2/router.cfg
```

5.8 Tuneability

When we discussed the issues in containers a few sections back we mentioned the problematic visibility of the host hardware from inside the container. While it is an issue from a security perspective, it creates transparency that allows a VNF to be tuned without requiring external information. The most obvious advantage is that inside a container we are fully aware of the layout of the cores that have been assigned to our container as seen on figure 5.3.

Using traditional virtualization solutions a virtual machines can only see the virtual CPU's it has been given access to, it cannot determine the physical CPUs that it will end up running on. In order to configure VNFs for optimal performance there is a need for this kind of information and thus we need good communication with the operator of the physical hardware when setting up a VNF. If we wanted two threads to share their L1 and L2 CPU cache we would affinitize one thread to a core and the other thread to the hyperthread core since these share the same L1 and L2 caches. But how do we know which these cores are in our guest? And do we have any guarantee that the operator or orchestrator will not move our virtual machine to different cores? When using a hypervisor we cannot answer these questions unless we have full control over the host configuration. When using a container we have full visibility into the cores received which allows us to tune our applications better according to the core configuration supplied.

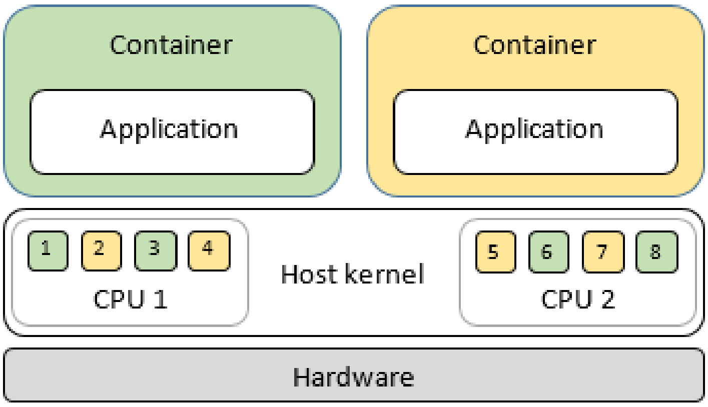


Figure 5.3: Logical core numbers as visible by containers

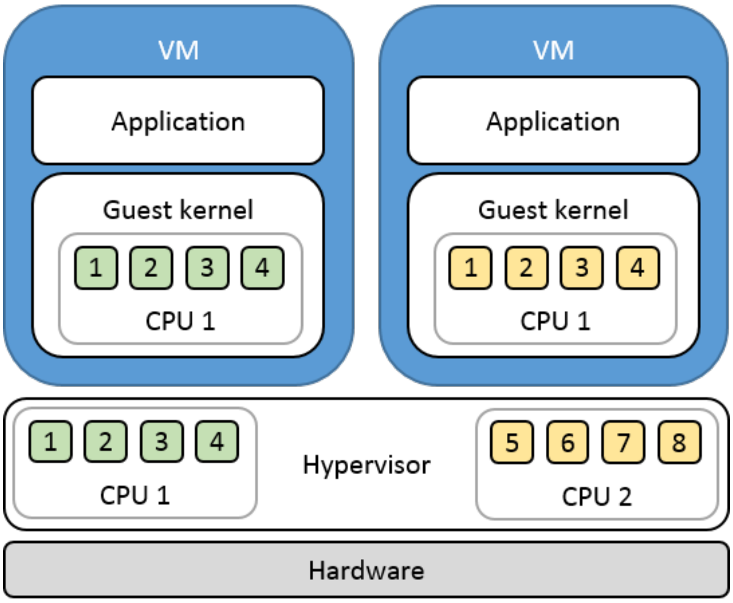


Figure 5.4: Logical core numbers as visible to applications running on guest kernel.

5.9 Summary

Table 5.2 provides an overview of all the topics discussed above to provide an easy comparison of KVM and Docker depending on your requirements. A performance comparison between the two can be found in chapter 7, where we implemented some VNFs on the host, in a virtual machine and in a container.

	Docker	KVM
Choice of operating system	Variant of host OS only	Any
Startup time	$\approx 1.5s$	$\approx 21s$
Device emulation	No	Yes
Disk footprint	Small	Large
Memory footprint	Small	Large
Density	High	Low
Configurability of application	Flexible	Complex
Tuneability from inside guest	Insight into host	Black box machine

Table 5.2: Comparison between Docker containers and KVM

Chapter 6

Setup of a DPDK enabled container

Running DPDK (section 2.3.1) applications in a container requires some configuration before we can actually start running these applications in containers. Here we will discuss the configuration that is required and why it is required. Since DPDK generally assumes full control over the hardware it sees, such as an entire host machine or virtual machine, we need to give it a little bit of assistance when only allowing it to have access to parts of the system.

DPDK uses the **igb_uio driver** to communicate with the NIC, however since this driver is not part of the linux kernel it has to be loaded by the host. Only the host has the capabilities to load a kernel module since it would be a security concern to let any container load a kernel module. We have to remember here that the host and all the containers on the system share the same kernel. Because of this limitation we have to load the igb_uio driver on the host and bind the network interface cards to the driver on the host as well. Any further communication with the NIC can now happen using the */dev/uioX* device node that corresponds with the NIC we would like to use.

Since the release of DPDK 1.7 it is possible to use the **VFIO** driver instead of using the IGB_UIO driver [23]. While IGB_UIO is not available in the upstream kernel, VFIO is. This opens the possibility to use DPDK applications without the need to load application specific kernel modules on the host. Binding the VFIO driver to the network adapters available in the system still requires access to the host system. Giving a container permission to bind adapters to a driver would require full privilege to the specific driver and a container would be able to bind or unbind any device to this driver.

The **KNI kernel module** [29], or Kernel NIC Interfaces, allows a program to create a network interface that is accessible through standard linux network APIs but that is backed by DPDK rather than by the linux networking core. If an application would like to use this kernel module it would have to be loaded by the host as well, but that is not the only problem for the KNI kernel module. When creating a KNI adapter from inside the container, that adapter will show up in the host namespace rather than that of the container. The container will need assistance from the host in order to access the linux interface that has been created by the kernel. The host will have to find out to which container the interface belongs and then assign it to that container.

6.1 Creating a Docker image

Since the philosophy behind Docker is to create images and not log in to each container to configure it we will be creating a set of images here as well. The images that we will describe are base images that allow us to build DPDK applications easily on. Since running inside a container also requires some patches to DPDK we will include them at this stage such that developers do not have to worry about them when deploying their applications.

To create a Docker image we will need to create a Dockerfile that describes the steps required to build the resulting image. The Dockerfile contains commands such as RUN or ADD which each create a new layer in the image hierarchy. We will walk through a skeleton Dockerfile that can be used to generate a DPDK base image from which images containing DPDK applications can be derived.

We start from a CentOS 7 image and will use the */root* directory for our application

```
FROM centos
WORKDIR /root
```

Next we install make, gcc, patch and numactl

```
RUN yum install -y tar make gcc patch numactl && yum clean all
```

Now we will set up some environment variables required by DPDK and by ourselves.

```
ENV RTE_SDK=/root/dpdk
ENV RTE_TARGET=x86_64-native-linuxapp-gcc
ENV RTE_VERSION="1.7.1"
```

Download DPDK, disable the IGB_UIO driver and KNI driver since these are kernel modules which would have to be loaded into the kernel, and this is not possible from inside a container.

```
RUN curl http://dpdk.org/browse/dpdk/snapshot/dpdk-\${RTE_VERSION}.tar.gz \
        | tar -xz && \
        mv dpdk* dpdk && \
        rm -Rf dpdk/app && \
        sed -i 's/ROOTDIRS-y := scripts lib app/ROOTDIRS-y :=  lib/' \
            dpdk/GNUmakefile && \
        sed -i 's/CONFIG_RTE_EAL_IGB_UIO=y/CONFIG_RTE_EAL_IGB_UIO=n/' \
            dpdk/config/common_linuxapp && \
        sed -i 's/CONFIG_RTE_LIBRTE_KNI=y/CONFIG_RTE_LIBRTE_KNI=n/' \
            dpdk/config/common_linuxapp
```

Apply the hugetlbfs patch to detect the allowed size correctly

```
ADD hugetlbfs_statvfs.patch /root/dpdk/hugetlbfs_statvfs.patch
RUN cd dpdk && patch -p1 < hugetlbfs_statvfs.patch
```

Whenever this image is used for a DPDK application we build the DPDK source

```
ONBUILD RUN cd dpdk && make config T=\$RTE_TARGET && \
    make -j && make -j install T=\$RTE_TARGET
```

One of the patches that are required when running in a container helps DPDK detect the size of the hugetlb mountpoint rather than the total amount of hugepages in the system. Whenever a DPDK application launches it detects the amount of free hugepages, then it will try to allocate all of them such that it can select the hugepages that are mapped on the desired CPU socket. However to limit the amount of memory that a single container can allocate we set a maximum to the amount of memory that is allocable from the hugetlbfs. We configure this limit using both the hugetlb cgroup and using the *size* option on the mountpoint. This last option can easily be detected using the *statvfs* [69] function, and that is exactly what this patch implements.

6.2 Launching the container

We will be using Docker to launch and manage our containers, however the default setup used by Docker is not compatible with DPDK applications. We need to mask out any of the PCI devices that we do not want the applications in the container to have access to. We need to provide a volume on which hugepages can be created and finally we will want to mask out the CPU's that are not in the cpuset of that specific container.

Similar to the situation described earlier **PCI devices** are exposed to applications running inside the container even if they do not have access to them. This again causes the DPDK EAL[1] to think that the PCI device is present and usable. However this is not the case, so we will have to hide the devices to which the container should not have access. To overcome this problem we can mask the pci devices on */sys*. The */sys/bus/pci/devices* directory contains only symbolic links to the actual devices structures, we can simply mask the entire directory with an empty directory and then populate this new empty directory only with the symbolic links for the devices that are accessible from inside the container.

Whenever DPDK scans this directory looking for available PCI devices it will only find the devices for which we provided the necessary symbolic links and thus will not access any device to which it has not been granted access. When it comes to assigning network interface cards to the container there is another

[1] DPDK Environment Abstraction Layer

problem, we need to know which device node in the */dev* folder belongs to which PCI device. Luckily this information can be retrieved from the device information under */sys/bus/pci/devices/<pci-dev>/uio* once the device has been bound to the *igb_uio* device driver that is used by DPDK.

The linux kernel exposes the available **CPU's** and their feature through *sysfs* which is mounted on */sys* when running a linux system. On this filesystem we can find a folder for each logical cpu core available on the system under the directory */sys/devices/system/cpu/*. The DPDK EAL uses this directory to detect the logical cores that can be used, specifically it tries to access the */sys/devices/system/cpu/cpuN/topology/core_id* file to see if core N is available. It will try this for each core where N is between 0 and 127.

In order to prevent DPDK from detecting and trying to access cores that are not in the container cpuset we will mount an empty directory over the *cpuN* directories of the cores that should not be available to the application in the container. This will ensure that DPDK is unable to read any file that exposes information about this logical core, including */sys/devices/system/cpu/cpuN/topology/core_id*. This change to our container will not change the behavior of the kernel in any way, but might cause incorrect behavior for other applications that would be run using this setup. The Dataplane Performance Demonstrator application performs a similar check to that of DPDK to find out to which physical core a logical core belongs, but fails to check if the file *topology/core_id* even exists. In such a case we have to patch the application itself to perform a check whether the file exists or else it will crash when attempting to read from that file.

When running a virtual machine it is possible to specify the maximum amount of memory an operating system instance can use. The same is possible for a container, but when using a container that does not include the memory provided by the **hugetlbfs**. Luckily there is a way of specifying the maximum amount of hugepages that a container can retrieve.

In order to achieve this we create a mount point that is unique for each container and mount a new hugetlbfs on each with the *size* option set to the maximum amount of memory that can be allocated from the hugetlbfs. Mounting hugetlbfs with 1GB pages and a maximum of 4GB would look something like the following:

```
# HUGEPAGE_1G=$((1024*1024*1024))
# HUGEPAGE_MEM=$((4*\$HUGEPAGE_1G))
# mount -t hugetlbfs -o pagesize=$HUGEPAGE_1G,size=$HUGEPAGE_MEM \
    none /mnt/container1/hugedir
```

Attaching this mount point to the container will effectively limit the amount of hugepages that each container can allocate. On top of that this approach ensures that containers on a single system cannot see, read or write any of the hugepages of another container. And when sharing is desirable we can simply attach the hugetlb mountpoint of one container to another without endangering any of the other containers on the system. For additional security we will also be setting the HugeTLB cgroup settings such that the memory limit will be enforced when a mmap system call is performed. Enabling this cgroup setting is as simple as creating a new group in the hierarchy, assigning your process ID to the *tasks* file and then setting the limit. Assigning a container to the container1 group and setting a limit of 4GB looks like this:

```
# CGROUP=/sys/fs/cgroup/hugetlb/system.slice/container1
# mkdir -p \$CGROUP
# echo pid-of-container > \$CGROUP/tasks
# echo \$((4*1024*1024*1024)) > \$CGROUP/hugetlb.1GB.limit_in_bytes
```

When all this configuration has been done, DPDK still needs a patch to read the size of the filesystem the hugetlbfs is on. Otherwise it will try to allocate all of the free hugepages in the system, it is more than likely that the amount of hugepages granted to this container is smaller than the amount of free hugepages available on the host system. After the limits have been configured we need a last command to ensure that the memory we receive is allocated on the correct NUMA node of the system. DPDK will refuse to launch if it cannot find enough memory on the socket that it wants to use. It is up to the kernel to choose the NUMA node to allocate memory from, but since we are limiting the amount of memory a container can retrieve we might end up with only pages from the wrong NUMA node. To signal the kernel that memory requested by our application should be allocated on a specific NUMA node we will use numactl to indicate the preferred NUMA node for our application. When launching our DPDK application inside the container we will be running something like this:

```
# numactl -p0 testpmd -c 0xf -n 4
```

However *numactl* is not a package that is included with the docker images of CentOS by default and we would like to set the NUMA node for our container from the host. This can be done by using the *cpuset* cgroup. Setting the *cpuset.mems* will limit the application to memory located on the nodes specified in *cpuset.mems*. Limiting the memory of a container to NUMA node 1 is accomplished with the following command, memory that potentially already resides on NUMA node 0 will be migrated to the new node.

```
# CGROUP=/sys/fs/cgroup/cpuset/system.slice/container1
# echo 1 > $CGROUP/cpuset.mems
```

6.3 Inter virtual machine shared memory

When there is a need for high performance communication between virtual machine we can use shared memory. Shared memory allows us to share packet data and other DPDK structures without copying them between virtual machines. When using KVM the memory is shared through a PCI device that Qemu generates. This device exposes a PCI BAR that represents the memory on the host. The PCI BAR of these virtual devices in different virtual machines map to the same memory on the host machine and thus expose the same section of memory. Containers are unable to use the same approach to expose the memory regions as there is no emulation being performed and due to the use of namespaces they are also unable to communicate the different memory regions with each other like we would be able to do on the host. To support these use-case we need a patch to DPDK that allows us to use the same memory with the least amount of configuration. That is why we added the option to DPDK to read shared memory directly from the hugepages on the hugetlbfs using a patch we wrote. Using this patch we can share a single hugetlbfs mountpoint between different containers and use the pages on this mountpoint to access the shared memory structures. Instead of passing the Qemu options to configure the shared memory to Qemu we now pass it on the commandline of the DPDK application and the IVSHMEM structures will be loaded by DPDK similarly to the approach used when running in a virtual machine.

6.4 Summary

In this chapter we have described the actions that have to be taken when running DPDK applications in a Docker container. Docker containers are not suited for use with DPDK applications by default, so they need some help when setting up.

We described the process of creating a Docker image which is ready for use with DPDK applications. The usage of this images allows DPDK application deployment in Docker with minimal effort from the developer of the application.

Finally we discussed the alterations to the Data Plane Development Kit that allow us to use Inter-Virtual Machine Shared Memory (IVSHMEM). This allows us to use high performance packet communication between different containers or communication between a DPDK enabled software switch and a container.

Chapter 7

Proof of concepts

The goal of these proof of concepts is to illustrate that existing virtualized network functions can be implemented using containers without any loss of functionality, security or performance. The hardware used for these proof of concepts consists out of two machines, one is used for packet generation and the other is used as the system under test. The system under test is the machine that will be running our network virtualized functions while the other machine will be used to generate the packets that will be processed by the system under test. The SUT consists of a dual socket mainboard with two Intel Xeon E5-2690v3 processors with 32GB of memory each, which results in a total of 64GB of system memory. The system is equipped with 2 Intel 82599ES dual port 10Gb ethernet controllers. On this system we will request 50 hugepages of 1GB in size to be used with Qemu and DPDK applications from the kernel. The machine used to generate the packets is a dual socket machine with two Intel Xeon E5-2690 processors each with 16GB of memory. The system is equipped with the same amount of dual port 10Gb ethernet Intel 82599ES controllers. Of the total of 32GB of memory we will be requesting 24 1GB hugepages to be used by the packet generator. In order to exercise the most amount of control over the CPU cores we intend to use for the proof of concepts we add the *isolcpus=* kernel option to the commandline such that no core on socket 0 will be used to schedule user tasks on except core 0. For these tests we will be using the Intel Dataplane Performance Demonstrators v015 [22] on the system under test to perform various network functions. DPPD has many different components which can be combined into a VNF for a specific use-case. On the generation side we will be using Pktgen [62] to generate the traffic that we will be sending to our system under test. All of the scripts used with Pktgen are provided with the DPPD source code.

7.1 System optimizations

When running a VNF on a virtual machine we will optimize the configuration of our virtual machine in order to optimize the performance we gain from the virtual machine. We will for example pin the Qemu threads to a CPU to prevent them from being migrated by the scheduler to a different core. We will have the guests memory backed by hugepages to prevent TLB misses when the virtual machine tries to access large amounts of memory. As containers run on the operating system we investigated if we

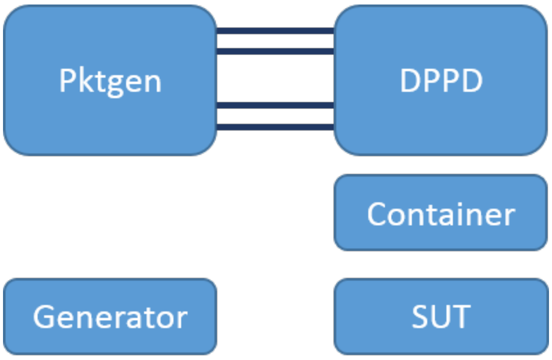


Figure 7.1: Test configuration with containers

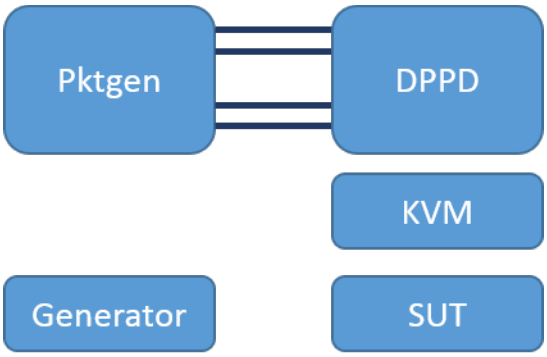


Figure 7.2: Test configuration with KVM

		802.1Q		802.1Q			
Dest MAC	Src MAC	Type	VLAN + flags	Type	VLAN + flags	EtherType	Payload
6 bytes	6 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes	

Figure 7.3: Packet layout for traffic coming from the CPE.

could increase performance of our containers by optimizing the kernel. As we will be running only one application thread per core in the system the full dyntick feature of the kernel seemed like a candidate for improvement. This combined with a slower timer tick (100Hz) could increase the amount of CPU time an application receives because it would not be interrupted by the kernel as much. The full dyntick feature allows a task to monopolize a core when it is the only task running on that core. When this occurs the kernel disables the timer that would invoke the scheduler and thus allow a task to keep executing without being interrupted. This feature is not enabled by default on CentOS 7 so we ended up recompiling our kernel with support for it by enabling the *CONFIG_NO_HZ_FULL=y* flag and friends. The option can be enabled using *make menuconfig* as well and lives under General setup, Timer subsystem.

7.2 Description of tests

For the first test we will be performing **Layer 2 forwarding**, which uses 2 network interfaces to receive packets of 64 bytes in size and 2 different network interfaces to send them back out through. The entire task of forwarding these packets is assigned to a single core of the CPU because more than one core would be limited by the network interface rather than by processing power or memory bandwidth.

Two variations of this test exist, one without modifying the packets and one that updates the mac addresses.

The test without packet modification, “none” named after the mode in DPPD, simply forwards a packet from one interface onto the next without ever touching the packet. This test essentially only moves pointers around in system memory and the results should not differ much between virtualization methods.

The second test, “l2fwd”, will perform almost the same functionality as the “none” test except that it will update the source and destination mac address inside the packet before transmitting it on a different network interface. This causes the application to read and write to the packet and will require packets to be moved into higher level caches which might become a bottleneck.

The third test performing **Layer 3 forwarding** uses the IP information inside the packet to determine the destination network interface to send the packet out through. This requires a lookup of the IP address in a table that resides in memory and thus adds extra overhead on top of the necessary updates to the packet itself.

The final test uses a more realistic testcase and that is the **Broadband Network Gateway**, which moves traffic coming from the customer premise equipment (CPE) onto the core network. Data coming from the CPE is encapsulated using an 802.1ad (QinQ) Ethernet header while the traffic on the core network is using GRE traffic tagged with an MPLS tag as can be seen in figures 7.3 and 7.4.

The BNG VNF consists of tasks performing MPLS untagging, loadbalancing and QinQ encapsulation based on the GRE header for traffic coming from the broadband network. For the traffic coming from

Ethernet	IPv4	GRE	IPv4	UDP	Payload
14 bytes	20 bytes	4 bytes	20 bytes	8 bytes	

Figure 7.4: Packet layout for traffic coming from the core network.

the CPE there are tasks that perform load balancing, QinQ decapsulation and routing. The QinQ decapsulation and QinQ encapsulation tasks are run on only 2 cores in order to create a bottleneck in the system that we hope will perform differently when run inside a virtual machine, container or on the host.

7.3 Results

As the graph in 7.5 suggests there seems to be little performance gain from the use of containers versus virtualization using KVM in most cases. In all the tests the resulting packets per second that the system is able to push through the system is stable and similar. The only test in which we were able to see a degradation of performance for both containers and KVM was the l2fwd test where containers would do worse than the host system and KVM would do worse than the containers. When removing the limit imposed on the amount of hugetlb memory we were able to get the containers up to the same speed as the application running on the host. This suggests that DPDK was not able to obtain the most favorable hugetlb memory in the system. Not limiting the container's use of hugetlb memory is however a security issue so we will be comparing with the limitation in place. The same cause might be affecting the virtual machine performance as well, considering that Qemu allocates the hugetlb memory that it will expose to the virtual machine and not the application inside of the virtual machine. This however is not as easy to check as was possible with the container implementation. We looked into the usage of a full dyntick kernel (section 7.1) versus a normal one in each of these cases and can conclude from the tests we have performed that there is no advantage in terms of throughput for the VNF use-cases we tested for.

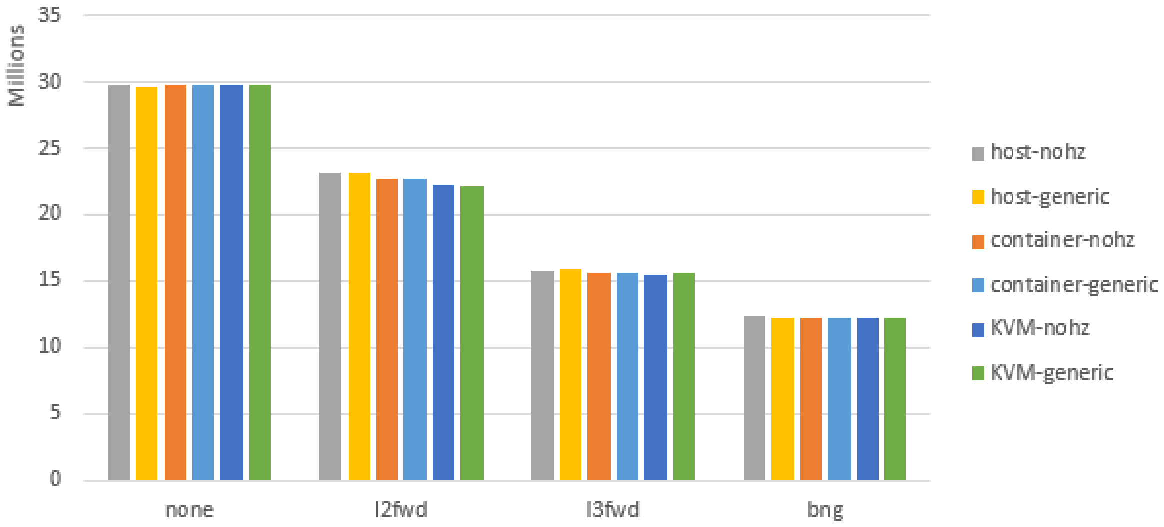


Figure 7.5: Packets per second that a VNF can process in different environments.

When looking at the latency (table 7.1) introduced during the "none" task or the simple forwarding we see that KVM introduces slightly less latency on average. However when we look at the maximum latency introduced, the KVM virtual machine latency jumps up to 457 microseconds which means that there is more jitter when using these VNFs. This difference in maximum latency can be attributed to the additional time spent in the hypervisor when an interrupt is fired. Since timekeeping in linux is done through the hardware timers an interrupt is fired rather regularly, and thus can also impact the latency introduced on packets passing through at the time of the interrupt. We have to remember here that VM-exit and VM-resume operations are still costly on current generation CPUs and they introduce latency as well.

Environment	**Average latency (μs)**	**Maximum latency (μs)**
Host	11.4375	53
Container	11.3955	56
KVM	11.1735	457

Table 7.1: Latency introduced by moving packets from one NIC onto another (none use-case)

When looking at the latency (table 7.2) introduced during the "l2fwd" task we see that containers actually introduce a higher latency on average than KVM or the host do. However we can again see that the virtual machine is less stable and has a much higher maximum latency. The added latency can be attributed to kernel code that is not fired when running in the root control group. For some of the control group functionality the linux kernel branches where it would not when running on the host [37]. This causes more code to be executed and could also confuse the branch prediction logic of the processor.

Environment	**Average latency (μs)**	**Maximum latency (μs)**
Host	493.148	565.441
Container	530.065	599.067
KVM	518.032	974.435

Table 7.2: Latency introduced by layer 2 forwarding (l2fwd use-case)

7.4 Accelerating older hardware

While the results from previous tests on recent hardware yield no significant performance benefits from the use of containers compared to the KVM virtual machines it does not mean that there is no place for containers. The older generation processors, Sandy Bridge and earlier, were far less efficient when using device pass-through to a VNF application due to a limitation in the support for hugepages in the IOMMU TLB [44].

When using containers however we are able to deploy multiple VNFs in isolation without incurring this performance overhead since we are not required to make use of the IOMMU. A comparison between them doing layer 2 forwarding (DPPD mode none) can be seen in following graph. Here, containers are able to achieve line-rate while the KVM virtual machine is unable to keep up until the packet size has increased considerably.

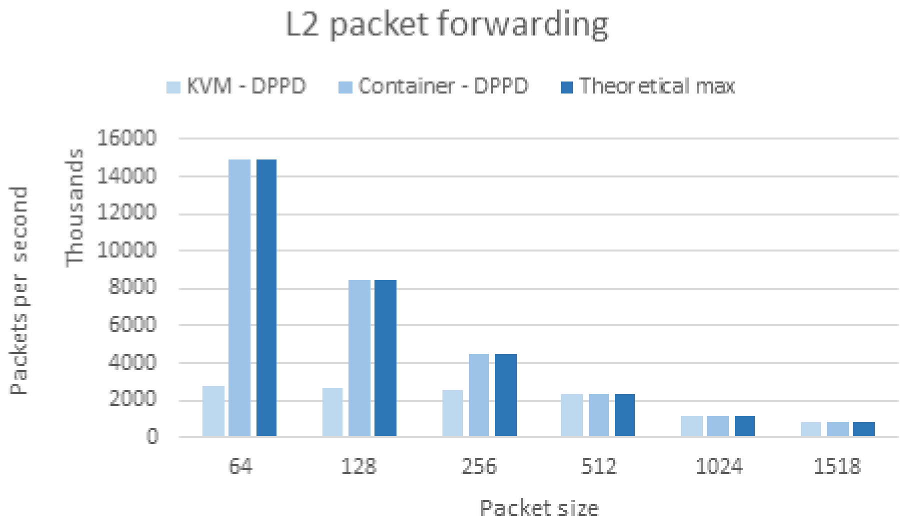


Figure 7.6: L2 forwarding comparison between containers and KVM on a Sandy Bridge processor

Chapter 8

Conclusion

As network function virtualization is taking off, there is a need to investigate every component in the technology stack used to accomplish successful deployments. Other components might result in better performance or easier management. This thesis looked into the benefits that are associated with containers and the benefits that could be gained from them for network function virtualization.

8.1 Disadvantages

Large scale container deployments are only now starting to be used with tools such as kubernetes, CoreOS and the offerings of cloud providers such as Google and Amazon. However, the use-cases of network function virtualization require quite the control over the platforms they are deployed on. This is already a disadvantage for those looking to deploy network functions next to regular applications on a single server platform. Successful deployment of virtualized network functions inside a container requires control over the host system, much more so as is the case with virtualization that uses a hypervisor to accomplish its goal.

Some other problems might arise from the usage of containers which are not tied to the usage of network function virtualization. These problems are related to those parts of the linux kernel that are not yet aware of the operating system level virtualization. The most prominent examples of this are the *sysfs* and *proc* filesystems which simply report host system statistics and thus break tools that rely on it, *free* for example.

While incorrect information is not particularly positive for containers, some information on the other hand is simply not available through practices that are considered safe by the community[1]. Examples of this are the CPU cores that are assigned to the container we are running in or the maximum amount of system memory that a container can use. Obviously virtual machines are far more interesting for these use-cases as the software inside the virtual machine is only aware of itself and the resources assigned to it.

8.2 Advantages

Where containers do shine is when it comes to scaling the amount of applications running on a single server. The amount of system memory required to run a container is almost solely the amount of memory that the applications running inside of the container require. Whereas virtual machines have a need for memory that can be used by the kernel running inside of the virtual machine and the emulated devices that it presents to these virtual machines.

Configuration is also one of those things that containers are able to do better. The configuration for the applications running inside of the container can be passed in at process creation, since containers isolate processes rather than virtualize the system. We gave a few examples of how easy it is to configure a container differently in section 5.7.

As the results of our proof of concepts (chapter 7) show there is almost no difference in the number of packets that can be processed by a network function running in a virtual machine compared to the same function running inside a container. When looking at latency however the story is a little bit different, containers manage to provide a more consistent latency. The latency introduced when running inside a

[1] Control Groups can be exposed to the guest, but this could have security implications

virtual machine is no worse on average, however the hypervisor can cause the latency to peak when it interrupts the VM.

The proof of concepts also illustrate that performance for our use-cases is *comparable to native execution*, something we wanted to investigate as part of this thesis. For the one use-case where we saw a performance degradation (l2fwd) we identified the unfavorable assignment of hugepages by the kernel to be at fault. The same cause also applies to the KVM virtual machine and there we can see that the impact of this is much smaller on the container as is the case for the virtual machine.

8.3 Final conclusion and alternatives

We have now described the *advantages that we were able to identify* during our tests with network functions running inside of containers. We feel that these advantages do not outweigh the disadvantages that we discussed. Deploying VNFs inside containers requires patches to DPDK, workarounds for Docker and possibly patches to the VNFs themselves. This while most of the tools are still maturing the core functionality of containers and are trying to fix issues we discussed in section 4.4.

Chapter 6 and the proof of concepts (chapter 7) show that it is possible to run network functions inside of a container. In section 6.3 we provide a high-level overview of the changes required to use DPDK IVSHMEM to communicate between containers through shared memory. A use-case which we did not include in our proof of concepts but have validated to be working.

A different technology used in VNFs which we did not further investigate is VHost, which can be used with virtual machines to communicate with a software switch. We did not investigate this as it would require more invasive changes to DPDK or the VHost implementation to get working with containers.

Finally we finish by answering the question we asked at the beginning of this text, do we think containers are a feasible alternative to hypervisor-based approaches for deploying virtualized network functions? We feel that the tooling still has some way to go in order to expose the full potential of containers, especially for use-cases with network function virtualization. Not only should the tools improve, the issues we have discussed (section 4.4) apply to any application deployed in a container so these will have to be addressed as well before containers can be taken serious as an alternative to hypervisor-based virtualization. Is it feasible to deploy network functions in a container? The answer would be yes, can it be considered an alternative for hypervisor-based approaches? To that we would have to say no, container orchestration at scale has only been available since late last year and is not yet ready to replace the traditional virtual machine.

When we compared different virtualization techniques in chapter 3.1.2 we mentioned unikernels and rumpkernels, these provide many of the same advantages that containers have. They are lightweight and reduce overhead as they run directly on the hypervisor, eliminating the need for a guest operating system. Unikernels and rumpkernels however can re-use the existing tools for management of virtual machines, which shows promise and is an interesting subject for future work. However the industry is currently more invested in containers as it is in these alternative uses of virtual machines.

Interesting work is being done on *Clear Containers* which provide container functionality using virtualization technology. A first look at Clear containers was published on TWN [26] which details how custom linux kernels can be booted in milliseconds using a paravirtualized linux kernel instead of full machine virtualization. The advantage being that we have a full fledged linux virtual machine at our disposal with the low memory overhead and fast startup speeds associated with containers.

Bibliography

[1] Amazon ec2 container service (ecs). http://aws.amazon.com/blogs/aws/cloud-container-management/. Accessed: 2014-11-14.

[2] Ark — intel server board s2600wtt). http://ark.intel.com/products/82156/. Accessed: 2015-05-21.

[3] Ark — intel xeon processor e5-2699 v3 (45m cache, 2.30 ghz). http://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz. Accessed: 2015-05-21.

[4] Capabilities manual page. http://man7.org/linux/man-pages/man7/capabilities.7.html. Accessed: 2014-10-11.

[5] cgroups. http://en.wikipedia.org/wiki/Cgroups. Accessed: 2014-10-02.

[6] Coreos. https://coreos.com/. Accessed: 2014-09-24.

[7] Criu. http://criu.org/Main_Page. Accessed: 2014-11-18.

[8] Devops tools · github. https://github.com/showcases/devops-tools. Accessed: 2015-05-21.

[9] Docker and selinux by daniel walsh from red hat - youtube. https://www.youtube.com/watch?v=zWGFqMuEHdw. Accessed: 2014-11-6.

[10] Dpdk documentation. http://dpdk.org/doc/nics. Accessed: 2014-09-08.

[11] Erlang on xen - at the heart of super-elastic clouds. http://erlangonxen.org/. Accessed: 2014-10-2.

[12] Etsi - nfv. http://www.etsi.org/technologies-clusters/technologies/nfv. Accessed: 2014-08-23.

[13] Etsukata blog. http://blog.etsukata.com/2014/05/docker-linux-kernel.html. Accessed: 2015-5-15.

[14] Exploiting concurrency vulnerabilities in system call wrappers. http://www.watson.org/~robert/2007woot/. Accessed: 2015-5-16.

[15] Features/systemdlightweightcontainers - fedoraproject. https://fedoraproject.org/wiki/Features/SystemdLightweightContainers. Accessed: 2015-05-31.

[16] Filesystem: aufs.sourceforge.net. http://aufs.sourceforge.net/. Accessed: 2015-5-15.

[17] Filesystem: btrfs wiki. https://btrfs.wiki.kernel.org/index.php/Main_Page. Accessed: 2015-5-15.

[18] Freebsd/xen - freebsd wiki. https://wiki.freebsd.org/FreeBSD/Xen. Accessed: 2015-5-18.

[19] Google container engine. https://cloud.google.com/container-engine/. Accessed: 2014-11-12.

[20] Halvm galois, inc. https://galois.com/project/halvm/. Accessed: 2014-10-2.

[21] Heartbleed bug. http://heartbleed.com/. Accessed: 2015-5-26.

[22] Intel data plane performance demonstrators. https://01.org/intel-data-plane-performance-demonstrators/downloads. Accessed: 2014-08-23.

[23] Intel, dpdk 1.7.0 release notes. http://www.intel.com/content/dam/www/public/us/en/documents/release-notes/intel-dpdk-release-notes.pdf. Accessed: 2014-09-08.

[24] Intel haswell memory latency. http://www.7-cpu.com/cpu/Haswell.html. Accessed: 2015-06-12.

[25] Intel®data plane development kit. http://dpdk.org/doc/intel/dpdk-prog-guide-1.7.0.pdf. Accessed: 2014-08-23.

[26] An introduction to clear containers [lwn.net]. https://lwn.net/Articles/644675/. Accessed: 2015-06-11.

[27] Ivshmem: git.qemu.org git - qemu.git/blob - hw/misc/ivshmem.c. http://git.qemu.org/?p=qemu.git;a=blob;f=hw/misc/ivshmem.c;h=5d272c84e9c7d92f3c17400ab8e41cb9f1622bba;hb=HEAD. Accessed: 2015-05-21.

[28] Ivshmem library - dpdk 2.0.0 documentation. http://dpdk.org/doc/guides/prog_guide/ivshmem_lib.html. Accessed: 2015-05-21.

[29] Kernel nic interface - dpdk 2.0.0 documentation. http://dpdk.org/doc/guides/prog_guide/kernel_nic_interface.html. Accessed: 2015-05-27.

[30] Kubernetes by google. http://kubernetes.io/. Accessed: 2015-1-2.

[31] libcontainer. https://github.com/docker/libcontainer. Accessed: 2015-4-13.

[32] Linux 3.0 how did we get initial domain (dom0) support there? — xen project blog. https://blog.xenproject.org/2011/06/14/linux-3-0-how-did-we-get-initial-domain-dom0-support-there/. Accessed: 2015-05-31.

[33] Linux syscall reference. http://syscalls.kernelgrok.com/. Accessed: 2014-10-11.

[34] Linux-vservers. http://linux-vserver.org/Welcome_to_Linux-VServer.org. Accessed: 2014-12-19.

[35] Lxc - linux containers. https://linuxcontainers.org/. Accessed: 2014-10-02.

[36] Managing data in containers. https://docs.docker.com/userguide/dockervolumes/. Accessed: 2014-10-11.

[37] Memory control group implementation: kernel/git/torvalds/linux.git - linux kernel source tree. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/mm/memcontrol.c?id=refs/tags/v4.1-rc7#n500. Accessed: 2015-06-09.

[38] Memory inside linux containers. http://fabiokung.com/2014/03/13/memory-inside-linux-containers/. Accessed: 2015-4-5.

[39] Microsoft announces nano server for modern apps and cloud. http://blogs.technet.com/b/windowsserver/archive/2015/04/08/microsoft-announces-nano-server-for-modern-apps-and-cloud.aspx. Accessed: 2015-5-16.

[40] Microsoft announces new container technologies for the next generation cloud. http://blogs.technet.com/b/server-cloud/archive/2015/04/08/microsoft-announces-new-container-technologies-for-the-next-generation-cloud.aspx. Accessed: 2015-5-16.

[41] Mini-os - xen. http://wiki.xen.org/wiki/Mini-OS. Accessed: 2015-5-24.

[42] Mirage os. http://openmirage.org. Accessed: 2014-9-30.

[43] namespaces(7) - linux manual page. http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed: 2014-10-22.

[44] Network function virtualization: Virtualized bras with linux. https://networkbuilders.intel.com/docs/Network_Builders_RA_vBRAS_Final.pdf. Accessed: 2014-10-3.

[45] New java 0-day vulnerability being exploited in the wild. http://thenextweb.com/insider/2013/03/01/new-java-vulnerability-is-being-exploited-in-the-wild-disable-the-plugin-or-change-your-security-settings/. Accessed: 2015-5-26.

[46] The next hypervisor: Lxd is fast, secure container management for linux. http://www.ubuntu.com/cloud/tools/lxd. Accessed: 2014-11-05.

[47] nginx news. http://nginx.org/. Accessed: 2015-05-21.

[48] nginx repository — docker hub registry - repositories of docker images. https://registry.hub.docker.com/_/nginx/. Accessed: 2015-05-21.

[49] Ocaml. https://ocaml.org. Accessed: 2014-9-30.

[50] Odin virtuozzo. http://www.odin.com/eu/products/virtuozzo/. Accessed: 2015-5-11.

[51] Openstack. http://www.openstack.org/. Accessed: 2014-09-24.

[52] Openstack: Welcome to glance's developer documentation. http://docs.openstack.org/developer/glance/. Accessed: 2015-5-17.

[53] Openstack: Welcome to neutron's developer documentation. http://docs.openstack.org/developer/neutron/. Accessed: 2015-5-17.

[54] Openstack: Welcome to nova's developer documentation. http://docs.openstack.org/developer/nova/. Accessed: 2015-5-17.

[55] Openvz - wikipedia. http://en.wikipedia.org/wiki/OpenVZ. Accessed: 2014-10-13.

[56] Openvz linux containers wiki. https://openvz.org/Main_Page. Accessed: 2014-10-13.

[57] Oracle solaris 11 zone features - transitioning from oracle solaris 10 to oracle solaris 11. http://docs.oracle.com/cd/E23824_01/html/E24456/glhcg.html. Accessed: 2015-5-16.

[58] Oracle solaris zones. http://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm#OPCUG429. Accessed: 2015-5-16.

[59] Osv - the operating system designed for the cloud. http://osv.io/. Accessed: 2014-10-2.

[60] ovs/ovs-docker at master · openvswitch/ovs. https://github.com/openvswitch/ovs/blob/master/utilities/ovs-docker. Accessed: 2015-06-01.

[61] Pci-sig: Peripheral component interconnect special interest group. https://www.pcisig.com/home. Accessed: 2015-4-4.

[62] pktgen/pktgen-dpdk - github. https://github.com/pktgen/Pktgen-DPDK. Accessed: 2014-08-25.

[63] Platforms - rumpkernel/wiki wiki. https://github.com/rumpkernel/wiki/wiki/Platforms. Accessed: 2015-1-14.

[64] Qemu patch to support ivshmem dpdk driver. https://01.org/sites/default/files/page/qemu-v1.6.2-ivshmem-dpdk.patch. Accessed: 2015-05-21.

[65] Rump kernels. http://rumpkernel.org/. Accessed: 2014-12-19.

[66] Rumpkernels: bare-metal arm. https://www.mail-archive.com/rumpkernel-users@lists.sourceforge.net/msg00562.html. Accessed: 2015-5-25.

[67] Script: Create custom network configurations for docker containers. https://gist.github.com/JorgenEvens/91f81bc7171581b5a1a8. Accessed: 2015-06-15.

[68] Spoon.net. https://spoon.net/. Accessed: 2015-5-11.

[69] statvfs(2): file system statistics - linux man page. http://linux.die.net/man/2/statvfs. Accessed: 2015-06-01.

[70] Supported filesystems - project atomic. http://www.projectatomic.io/docs/filesystems/. Accessed: 2014-11-10.

[71] Sysjail - wikipedia. http://en.wikipedia.org/wiki/Sysjail. Accessed: 2015-5-16.

[72] systemd adoption - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Systemd#Adoption_and_reception. Accessed: 2015-06-09.

[73] Virtual hardware: git.qemu.org git - qemu.git/blob - hw/block/makefile.objs. http://git.qemu.org/?p=qemu.git;a=blob;f=hw/block/Makefile.objs;h=d4c3ab758df8758c290896c7154cdd490afaf400;hb=HEAD. Accessed: 2015-5-24.

[74] Virtual hardware: git.qemu.org git - qemu.git/blob - hw/net/makefile.objs. http://git.qemu.org/?p=qemu.git;a=blob;f=hw/net/Makefile.objs;h=7b91c4e51d0a7cd9750b4839bac7d6ff36c19df8;hb=HEAD. Accessed: 2015-5-24.

[75] Xen project software overview - xen. http://wiki.xen.org/wiki/Xen_Project_Software_Overview#Guest_Types. Accessed: 2015-5-22.

[76] The xen vulnerability that rebooted the public cloud. http://www.eweek.com/cloud/the-xen-vulnerability-that-rebooted-the-public-cloud.html. Accessed: 2015-5-26.

[77] Network functions virtualisation - introductory white paper.

[78] Anderson, T. E. *The case for application-specific operating systems.* University of California, Berkeley, Computer Science Division, 1993.

[79] Chapman, R. A. Linux system call table for x86_64. http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64. Accessed: 2014-10-11.

[80] He, L., Karne, R. K., and Wijesinha, A. L. The design and performance of a bare pc web server. *IJ Comput. Appl. 15*, 2 (2008), 100–112.

[81] Intel. High performance packet processing.

[82] Intel. Intel®64 and ia-32 architectures software developer's manual.

[83] Jin, X., Keller, E., and Rexford, J. Virtual switching without a hypervisor for a more secure cloud. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (2012), USENIX Association, pp. 9–9.

[84] Kantee, A., and Cormack, J. Rump kernels: No os? no problem! *;login: 39*, 5 (2014), 11–17.

[85] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. Nohype: virtualized cloud infrastructure without the virtualization. *ACM SIGARCH Computer Architecture News 38*, 3 (2010), 350–361.

[86] Kerrisk, M. Namespaces in operation, part 1: namespaces overview. http://lwn.net/Articles/531114/, January 2013. Accessed: 2014-10-02.

[87] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS) 10*, 4 (1992), 265–310.

[88] Li, J., Wang, Q., Jayasinghe, D., Park, J., Zhu, T., and Pu, C. Performance overhead among three hypervisors: An experimental study using hadoop benchmarks. In *Big Data (BigData Congress), 2013 IEEE International Congress on* (2013), IEEE, pp. 9–16.

[89] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 461–472.

[90] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *Proc. USENIX NSDI* (2014), pp. 459–473.

[91] Nemeth, B. Network infrastructure optimization.

[92] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

[93] Rizzo, L., and Landi, M. netmap: memory mapped access to network devices. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 422–423.

[94] Steinberg, U., and Kauer, B. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 209–222.

[95] Szefer, J., Keller, E., Lee, R. B., and Rexford, J. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 401–412.

Appendices

Appendix A

Nederlandstalige samenvatting

Virtualisatie heeft al verschillende voordelen betekend voor het beheer van de IT infrastructuur. Netwerk functie virtualisatie wil nu dezelfde technieken toepassen om het beheer van de netwerk infrastructuur te verbeteren. Wat het uitrollen van nieuwe netwerkapparatuur eenvoudiger en flexibeler maakt. Het bestaande onderzoek focust zich op het gebruik van de traditionele virtualisatie met virtuele machines. Deze thesis onderzoekt of containers, een techniek die steeds populairder wordt, een waardig alternatief is voor virtuele machines bij het gebruik van virtuele netwerk functies. Om onze bevindingen aan te tonen zijn er enkele experimenten uitgevoerd die een vergelijking opstellen tussen virtuele machines op basis van de Kernel-based Virtual Machine in linux en Docker linux containers.

A.1 Virtualisatie

Virtualisatie wordt al enkele jaren succesvol gebruikt om de dagelijkse taken van een netwerkbeheerder te vereenvoudigen. Niet alleen maakt virtualisatie het leven van de netwerkbeheerder gemakkelijker, het maakt de IT infrastructuur van een bedrijf ook flexibeler.

Virtualisatie maakt het mogelijk om meerdere besturingssystemen tegelijkertijd te gebruiken op hetzelfde toestel. Op deze manier kunnen toestellen die niet ten volle gebruikt worden benut worden voor meerdere toepassingen tegelijkertijd zonder dat er zich conflicten voor doen tussen de verschillende softwarepakketten op het toestel.

De mogelijke conflicten tussen softwarepakketten worden opgelost door middel van software die direct op de hardware wordt uitgevoerd. Deze software, een hypervisor, bootst het gedrag van een echte machine na voor elk van de verschillende besturingssystemen en zorgt er zo voor dat ons fysiek toestel wordt voorgesteld als meerdere fictieve toestellen. Deze toestellen kunnen doordat ze in software gedefinieerd zijn ook makkelijk overgeplaatst worden naar andere toestellen en het is hierdoor dat de IT infrastructuur flexibeler wordt.

A.1.1 Netwerk functie virtualisatie

Traditioneel bestaat een netwerk uit hardware die een specifieke functionaliteit heeft, zoals een router of een broadband network gateway. Deze hardware wordt een middlebox genoemd en is opgebouwd uit hardware die specifiek ontworpen is om 1 functie zeer goed en snel uit te voeren. Het nadeel aan deze apparatuur is dat de functionaliteit van het toestel geen upgrade kan krijgen of veranderd worden doordat deze ingebouwd is in de circuits van het toestel. Netwerk functie virtualisatie maakt gebruik van software implementaties die dezelfde functionaliteit hebben als de middleboxen. Doordat dit software is kan het gemakkelijker aangepast worden en geïnstaleerd worden in een virtuele machine. Als we een software implementatie van een middle box in een virtuele machine kunnen uitvoeren spreken we over een virtuele netwerk functie. Doordat deze gebruik maakt van virtualisatie kunnen we ook gebruik maken van de voordelen van virtualisatie. Dit zorgt er bijgevolg voor dat het beheren van het netwerk nu veel eenvoudiger is en een grotere flexibiliteit kent.

A.1.2 Virtuele machines

Als er gesproken wordt over virtualisatie, dan bedoelt men meestal virtuele machines. Het opsplitsen van een fysiek toestel in meerdere fictieve, ofwel virtuele, machines. Deze machines kunnen elk hun eigen besturingssysteem uitvoeren zonder dat ze andere gebruikers op het fysieke systeem storen.

Virtuele machines worden gerealiseerd door het nabootsen van bestaande hardware die door elk besturingssysteem ondersteund wordt. Op deze manier kunnen besturingssystemen eenvouding geïnstaleerd worden zonder dat er nood is aan drivers die uniek zijn aan die machine. Doordat deze hardware zo generiek is kan een virtuele machine overgeplaatst worden van 1 fysieke machine naar een andere zonder dat de beheerder een aanpassing moet uitvoeren in de virtuele machine. Dit is zeer voordelig wanneer er onderhoud moet gebeuren aan de fysieke apparatuur waar een virtuele machine op aan het uitvoeren is. De beheerder kan de virtuele machine verplaatsen naar een ander toestel en daarna zijn werkzaamheden uitvoeren zonder de dienstverlening te onderbreken.

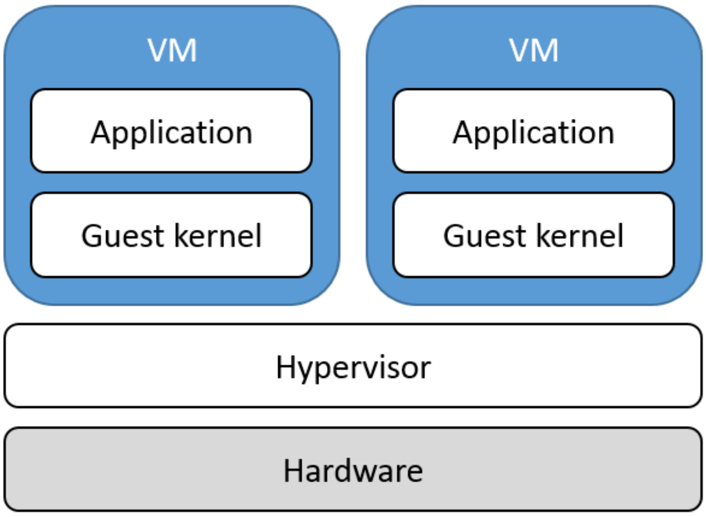


Figure A.1: Voorstelling van twee virtuele machines op een hypervisor.

A.1.3 Unikernels

Doordat er op voorhand bekend is welke kleine set hardware er beschikbaar is in een virtuele machine worden er enkele nieuwe toepassingen voor virtuele machines mogelijk. Normaal gesproken als we software willen schrijven die kan uitgevoerd worden op een toestel zonder een besturingssysteem zijn we verplicht stuurprogramma's te ontwikkelen voor de grote hoeveelheid aan diverse hardware beschikbaar. Wanneer we virtualisatie gebruiken is dit echter niet het geval, dus wordt het haalbaar om software te schrijven die rechtstreeks op de hypervisor werkt.

We hebben dus geen besturingssysteem meer nodig dat de hardware beheert voor ons en hiervan maken unikernels gebruik. Unikernels zijn programma's die enkel de code bevatten die ze nodig hebben om succesvol uit te voeren. Stuurprogramma's voor hardware worden rechtstreeks in de toepassing in gecompileerd wat voor enkele unieke optimalisaties kan zorgen.

Doordat een unikernel enkel bestaat uit de code nodig om zijn functionaliteit uit te voeren is deze veel kleiner, zowel op de harde schijf als in het werkgeheugen.

Unikernels werken in een hardware modus die men ”kernel-mode” noemt, deze is normaal voorbestemd voor het besturingssysteem en enkel je besturingssysteem. Softwarepakketten die je op je besturingssysteem installeert worden uitgevoerd in ”userspace” en telkens als er gewisseld wordt tussen deze twee moet de processor van modus veranderen. Doordat unikernels in kernel-mode werken hoeven ze dus ook niet te wisselen, wat de algemene uitvoer versnelt.

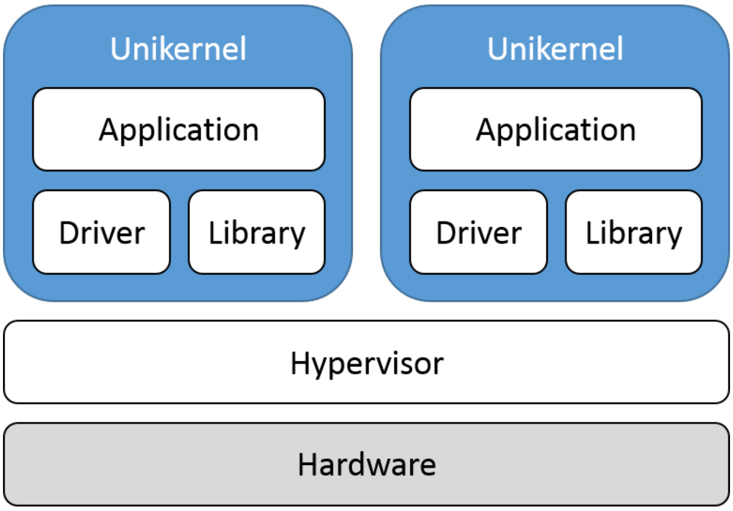


Figure A.2: Voorstelling van twee unikernels bovenop een hypervisor.

A.1.4 Rump kernels

Rumpkernels leunen aan bij de unikernels, ook deze compileren naar één enkele toepassing die rechtstreeks op virtuele hardware kan uitgevoerd worden. In tegenstelling tot unikernels is het voor rumpkernels echter ook mogelijk om uitgevoerd te worden op fysieke hardware.

Rumpkernels bestaan namelijk uit softwarecomponenten uit NetBSD. Zo worden de stuurprogramma's van NetBSD zo ontwikkeld dat ze in eender welk besturingssysteem geplugd kunnen worden. Hiervan kan de ontwikkelaar van de rumpkernel gebruik maken om zo de drivers voor de nodige hardware te gebruiken van NetBSD, en deze tijdens de compilatie mee te compileren in zijn softwarepakket.

Ook componenten als de standard C library kunnen gebruikt worden van NetBSD, dit heeft als gevolg dat bestaande software geschreven in C kan gecompileerd worden naar een rumpkernel die rechtstreeks op de hardware kan uitgevoerd worden. Dit allemaal zonder dat er nood is aan een besturingssysteem om de hardware te besturen.

A.1.5 Operating system level virtualization

Virtualisatie op het niveau van het besturingssysteem verschilt enorm van de voorgaande vormen van virtualisatie. Waar we in voorgaande vormen van virtualisatie hardware nabootsten om zo een fictief toestel te creëren, daar isoleren we nu verschillende groepen toepassingen van elkaar. De toepassingen zijn zich niet bewust van hetgeen dat zich buiten hun groep afspeelt, zodanig dat het lijkt of ze op hun eigen machine draaien. Het besturingssysteem is hier de software die voor de virtuele voorstelling zorgt. Het besturingssysteem is verantwoordelijk om de opgestelde groepen te beheren en ervoor te zorgen dat deze enkel toegang hebben tot wat hun toegekend is.

In linux heet deze technologie containers, terwijl andere besturingssystemen het anders noemen. Er zijn ook besturingssystemen die op dit ogenblik nog geen ondersteuning hebben voor deze vorm van virtualisatie, zoals Microsoft Windows.

Hieronder enkele van de verschillende benamingen in de verschillende besturingssystemen:

- **FreeBSD**: Jail
- **Solaris**: Zones
- **NetBSD/OpenBSD**: sysjail

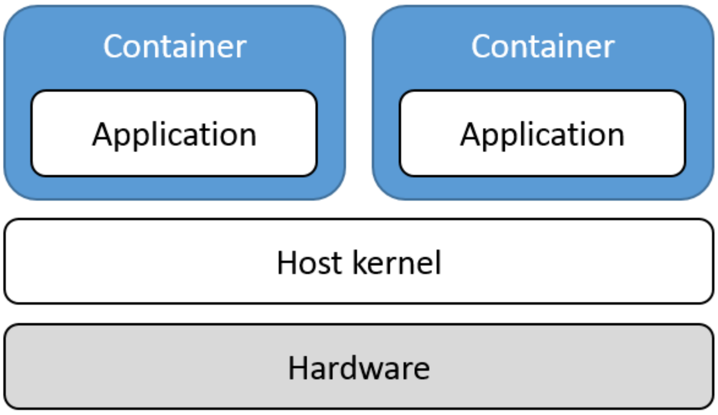


Figure A.3: Voorstelling van twee containers die beheerd worden door de kernel.

A.1.6 NoHype architecture

Door de jaren heen hebben fabrikanten van processoren functionaliteit toegevoegd die het virtualiseren versnellen. Door deze virtualisatie extensies, zoals ze genoemd worden, kunnen gevirtualiseerde machines bijna aan dezelfde snelheid werken als de fysieke hardware zelf. Onder andere bij het beheer van het geheugen, apparatuur in het toestel en onveilige instructies.

De NoHype architectuur maakt gebruik van deze hardware extensies om een virtuele machine op te starten die niet ondersteund wordt door een hypervisor. De virtualisatie wordt in dat geval beheerd door de hardware zelf, zonder de tussenkomst van software die de hardware beheerd.

Deze architectuur werd ontwikkeld met als hoofdgedachte de veiligheid van een virtuele machine. Doordat er geen software is die aangevallen kan worden vanuit de virtuele machine is er ook minder kans dat een kwaadwillige toepassing het gedrag van andere virtuele machines kan beïnvloeden.

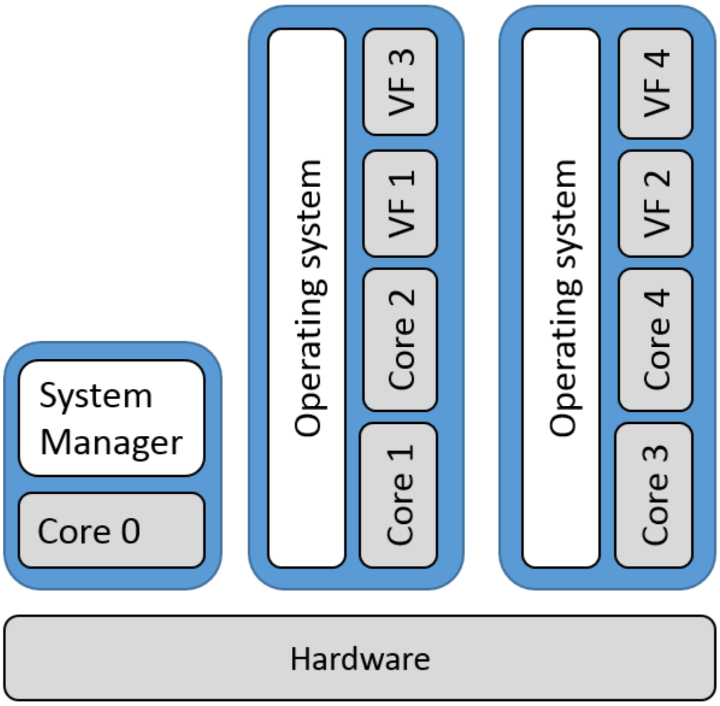


Figure A.4: Voorstelling van een opstelling gebruikmakend van de NoHype architectuur.

Deze architectuur wordt echter nog niet gebruikt en bestaat momenteel enkel uit een prototype dat ontwikkeld werd door onderzoekers. Het prototype maakt nog steeds gebruik van een hypervisor tijdens het opstarten van de virtuele machine, als deze eenmaal is opgestart komt de hypervisor niet meer tussen.

A.2 Containers

Containers, of de linux implementatie van operating system level virtualization, is de virtualisatie techniek die we in deze thesis experimenteel hebben vergeleken met de hypervisor die deel uitmaakt van de linux kernel.

Containers bestaan in linux uit verschillende componenten die aanwezig zijn in linux die het mogelijk maken om containers op te bouwen. Deze twee componenten zijn namespaces en control groups.

A.2.1 Namespaces

Namespaces delen systeem resources op in verschillende groepen zodanig dat elke groep processen een eigen voorstelling krijgt van deze resources. Er zijn namespaces voorzien voor 6 systeem resources in linux. De verschillende types namespaces zijn mount namespaces, Unix Time-sharing namespaces, Inter-Process communication namespaces, Process ID namespaces, User ID namespaces en netwerk namespaces.

- **Mount namespaces**: Zorgt ervoor dat elke container zijn eigen set mountpoints heeft van bestandsystemen waar hij toegang tot heeft.
- **UTS namespaces**: Laat het toe om elke container een eigen hostname te geven.
- **IPC namespaces**: Zorgt ervoor dat alleen processen binnen dezelfde namespace kunnen communiceren met elkaar gebruikmakende van System-V IPC of POSIX message queues.
- **PID namespaces**: Elke container beschikt over zijn eigen set process IDs, zodanig dat elke container onafhankelijk van een andere container IDs kan toekennen.
- **Network namespaces**: Elke container beschikt over zijn eigen netwerk stack (ip routes, netwerkadapters, TCP states, ...).
- **User namespaces**: De nummering van gebruikers in een container is onafhankelijk van het host systeem.

A.2.2 Control groups

Control groups maken het mogelijk om het gebruik van systeem resources te limiteren voor een groep toepassingen. Net zoals bij namespaces moeten we groepen opstellen voor elke type resource die beschikbaar is. Vervolgens kunnen we de kernel configureren om het resourcegebruik van een groep te limiteren.

We kunnen bijvoorbeeld het limiet op het gebruik van werkgeheugen instellen op 512MB, dit zal verhinderen dat er meer geheugen wordt verbruikt door deze groep van processen als is toegelaten.

De drie belangrijkste types control groups die wij zullen nodig hebben in deze thesis zijn:

- **devices:** Beperkt de acties die uitgevoerd kunnen worden op hardware die beschikbaar is via */dev*.
- **hugetlbfs:** Beperkt de beschikbare hoeveelheid hugetlb[1] geheugen.
- **cpuset:** Beperkt de processor cores waarop een process kan worden uitgevoerd.

A.2.3 Management

Deze thesis maakt gebruik van Docker, een tool die enorm populair is geworden in de afgelopen drie jaar. Deze tool heeft ervoor gezorgd dat containers veel aandacht kregen. Oorspronkelijk was Docker gefocussed op ontwikkelaars die ermee hun applicatie konden bundelen met al de afhankelijkheden ervan. Dit heeft als voordeel dat hun applicatie zich altijd hetzelfde gedraagt, ongeacht de computer waarop het wordt uitgevoerd.

Behalve Docker bestaan er ook nog andere alternatieven, deze zijn echter minder populair en hebben een veel kleinere community.

Enkele van deze tools zijn:

- LXD
- Systemd

[1] Aaneengesloten stukken geheugen die beschikbaar zijn via een item in de memory management unit.

- OpenVZ
- Linux VServer

In 2014 werd er ook verder ingezet op containers door middel van tools zoals Kubernetes (door Google) en CoreOS. Deze tools maken het mogelijk om Docker containers te beheren op een schaal die eerder onpraktisch was.

Ook cloud service providers zetten sterk in op het gebruik van containers in de cloud. Zowel Google als Amazon hebben op het einde van 2014 diensten voorgesteld om gebruik te maken van hun platform voor container-as-a-service.

A.2.4 Problemen

Het gebruik van containers bevat echter nog enkele problemen die niet voorkomen bij virtuele machines. Een van de problemen met containers is dat niet alle informatie die containers ter beschikking stellen op de hoogte is van de virtualisatie. Zo zijn de twee bestandssystemen die informatie over het systeem bevatten, */proc* en */sys* niet volledig op de hoogte van de virtualisatie. Deze bestandssystemen gaan bijvoorbeeld nog steeds informatie rapporteren over het gehele fysieke toestel, en niet enkel de informatie over de container.

Veel van de beschikbare tools voor containers beschikken nog niet over de feature "live migration". Bij het gebruik van virtuele machines laat dit een beheerder toe om een virtuele machine naar een andere fysieke machine te verplaatsen zonder dat er een merkbare onderbreking plaatsvindt. Dit is echter een feature die voor containers nog geïntegreerd moet worden, er bestaan echter wel al de nodige tools om dit handmatig te doen.

Het laatste probleem dat we hier bespreken is de overbelasting van de kernel. Doordat elke container op een systeem dezelfde kernel gebruikt is het mogelijk dat een toepassing in een container de kernel overbelast. Door deze overbelasting kan de toepassing een "denial-of-service" attack uitvoeren waardoor andere containers niet meer verder kunnen functioneren.

A.3 Vergelijking tussen linux containers en KVM virtuele machines

In deze thesis hebben we de eigenschappen van containers vergeleken met deze van KVM virtuele machines. Zo hebben we onderzocht wat er wel en niet mogelijk is en welke voordelen elk van de technieken hebben.

Uit deze experimenten kunnen we besluiten dat containers sneller opstarten, minder plaats innemen in het werkgeheugen en op de schijf, makkelijker zijn te herconfigureren en een beter inzicht geven in de hardware waarmee gewerkt wordt.

Virtuele machines laten een bredere keuze besturingssytemen toe en kunnen makkelijker functionaliteit toevoegen aan een specifieke virtuele machine zonder andere virtuele machines te beïnvloeden. Dit is mogelijk door functionaliteit aan te bieden als virtuele apparatuur die echte hardware nabootst.

	Docker	**KVM**
Keuze besturingssysteem	Variant host besturingssysteem	Elk
Duur opstart	$\approx 1.5s$	$\approx 21s$
Apparatuur nabootsen	Nee	Ja
Benodigde plaats op schijf	Klein	Groot
Werkgeheugen verbruik	Klein	Groot
Dichtheid van toepassingen	Hoog	Laag
Configuratie van toepassing	Flexibel	Complex
Afstemming vanuit guest	Inzicht fysieke hardware	Black box

Table A.1: Vergelijking tussen Docker containers en KVM

A.4 Experimenten

In deze thesis hebben een vergelijking gedaan van de performantie van een virtuele netwerk functie rechtstreeks op linux, in een container en in een KVM virtuele machine.

We hebben drie verschillende scenario's gebruikt om zo de invloed van de verschillende workloads te bekijken in de verschillende omgevingen. Hieruit kunnen we nagaan of er performantie valt te winnen bij het gebruik van containers. We bekijken twee factoren die belangrijk zijn voor toepassingen die pakketten op het netwerk verwerken. We kijken zowel naar de throughput en de latency die onze virtuele functie introduceert.

De scenario's die we hebben nagekeken zijn:

- **none**: Hier worden pakketten zonder inspectie doorgestuurd van 1 netwerkpoort onmiddelijk naar een andere.
- **l2fwd**: Hier worden pakketten ook van 1 poort naar de volgende gestuurd, maar de virtuele netwerk functie past de MAC addressen in het pakket aan.
- **l3fwd**: Deze test zoekt op in een tabel naar welke netwerkpoort het pakket gestuurd moet worden. Vervolgens update ook deze de addressen in het pakket.
- **bng**: Deze use-case is de implementatie van een Broadband Network Gateway, een toepassing die zich tussen de internet provider en de modem van de klant bevindt.

A.4.1 Pakketten per seconde

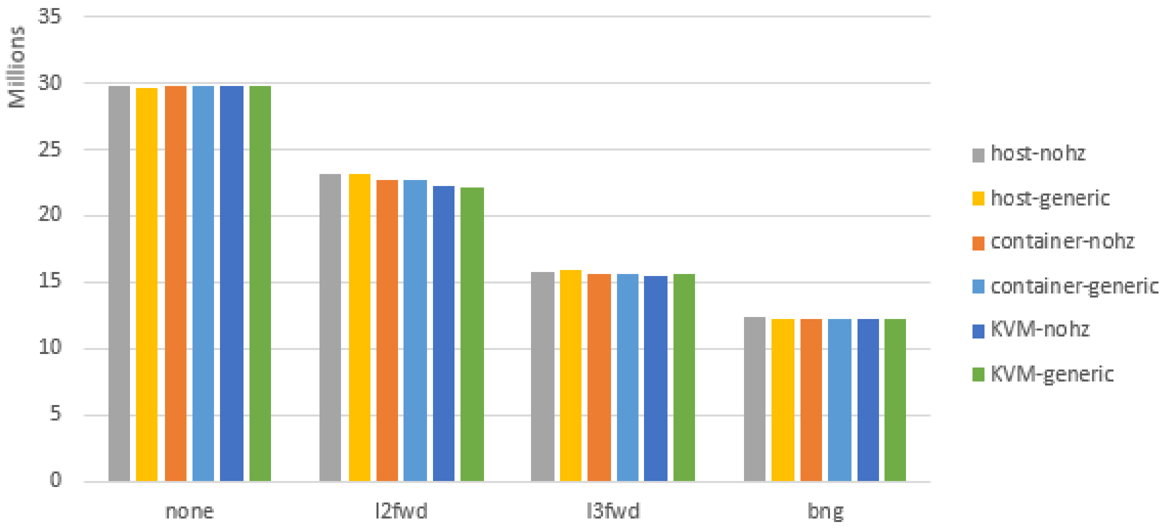


Figure A.5: Pakketten per seconde die de virtuele netwerk functie kan verwerken in de verschillende omgevingen.

We kunnen in deze grafiek zien dat voor bijna alle experimenten containers en virtuele machines hetzelfde aantal pakketten kunnen verwerken per seconde. Het lijkt dus alvast dat throughput niet de belangrijkste factor kan zijn bij de keuze voor containers of virtuele machines. Enkel de l2fwd test toont een verschil tussen de 3 use-cases, we vermoeden dat dit verschil er komt doordat de applicatie niet in staat is het optimale geheugen te kiezen. Bij het gebruik van containers hebben we dit kunnen verifiëren door de beperking op het hugetlb geheugen te verwijderen. Voor virtuele machines was het niet mogelijk om dezelfde test te doen.

A.4.2 Latency

Bij het meten van de latency voor de verschillende use-cases was het duidelijk dat de gemiddelde latency die geïntroduceerd wordt voor elk van de omgevingen hetzelfde bleek te zijn. De maximale latency daarintegen was bij virtuele machines veel hoger. Dit is niet onverwacht, we kunnen deze piek in latency toekennen aan de hypervisor die de virtuele machine onderbreekt. Ondanks de verbeteringen in hardware is het onderbreken van een virtuele machine nog een vrij kostelijke operatie.

Omgeving	Gemiddelde latency (μs)	Maximale latency (μs)
Host	11.4375	53
Container	11.3955	56
KVM	11.1735	457

Table A.2: De latency geïntroduceerd door het doorsturen van een pakket. (none use-case)

A.5 Conclusie

Bij het uitvoeren van de experimenten in deze thesis hebben we heel wat oplossingen moeten zoeken voor problemen die eigen waren aan het gebruik van containers. Sommige van deze oplossingen zijn van toepassing op Docker, andere zijn van toepassing op de virtuele netwerk functies die we gebruikt hebben.

Deze aanpassingen zullen zowel in de virtuele netwerk functies moeten ingevoerd worden vooraleer bijvoorbeeld een operator ermee aan de slag zou kunnen. Bovendien moeten de oplossingen die we hebben toegepast bij het gebruik van Docker geïntegreerd worden met Docker. De projecten die verderbouwen op Docker zullen hier ook rekening mee moeten houden vooraleer deze in staat zullen zijn om netwerk functies op een eenvoudige manier te kunnen uitrollen. Aangezien deze tools nog jong zijn verwachten we dat er op dit ogenblik andere prioriteiten zijn voor deze projecten die de core functionaliteit uitbreiden. Dit terwijl er al volop gewerkt wordt aan de nodige tools om netwerk functies in virtuele machines uit te rollen.

Vinden we dat containers een waardig alternatief is voor virtuele machines? Ja, containers kunnen gebruikt worden als een alternatief voor virtuele machines. Betekent dat dan ook dat containers geschikt zijn voor virtuele netwerk functies? Nee, er zal nog gewerkt moeten worden aan de integratie tussen de twee vooraleer containers een waardig alternatief voor virtuele machines kunnen zijn bij het gebruik van virtuele netwerk functies.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
A comparison of containers and virtual machines for use with NFV

Richting: **master in de informatica-databases**
Jaar: **2015**

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,

Evens, Jorgen

Datum: **2/07/2015**