Nesting Virtual Environments

Jani Kuutvuori

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 17.5.2017

Thesis supervisor:

Prof. Risto Wichman

Thesis advisor:

M.Sc. Jan Zizka

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Jani Kuutvuori

Title: Nesting Virtual Environments

Date: 17.5.2017 Language: English Number of pages: 7+57

Master's Programme in Computer, Communication and Information Sciences

Professorship: Signal Processing

Supervisor: Prof. Risto Wichman

Advisor: M.Sc. Jan Zizka

Virtual Machines have been a common computation platform in areas of cloud computing for some time now. VMs offer a decent amount of isolation for security and system resources, and from application perspective they behave much like native environments. Software containers are gaining popularity, as a new application delivery technology. Just like VMs, applications started inside containers are running in isolated environments but without the performance overhead caused by virtualization of system resources. This makes containers seem like a more effient option for VMs.
In this thesis, different combinations of containers and VMs are benchmarked. For each benchmark, host environment is also measured, to understand the overhead caused by the underlying virtuel environment technology. Benchmarks used include storage and network access benchmarks, and also an application benchmark of compiling Linux kernel.
As another part of the thesis, a CPU intensive workload is run on the virtualization host server. Then the benchmarks are repeated, in order to determine how much the given workload effects the benchmark score, and also if this effect can be observed from the virtualization guest side by measuring CPU steal time.
Results show that containers are slightly slower in the application benchmark than the host. The main difference is expected to come from the way docker handles storage accesses. With default network configuration, the container is losing in terms of performance to the host.
In every benchmark we did, VMs always lost to host and containers in performance.

Keywords: Virtualization, Software Containers

AALTO-YLIOPISTO SÄHKÖTEKNIIKAN KORKEAKOULU	DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Jani Kuutvuori

Työn nimi: Sisäkkäiset virtuaaliympäristöt

Päivämäärä: 17.5.2017 Kieli: Englanti Sivumäärä: 7+57

Signaalinkäsittelyn ja akustiikan laitos

Professuuri: Signaalinkäsittely

Työn valvoja: Prof. Risto Wichman

Työn ohjaaja: Dipl. Ins. Jan Zizka

Virtuaalikoneista on tullut yleinen laskentaalusta pilvitietokoneille. Ne eristävät virtuaaliympäristön muista palveluista samalla fyysisellä koneella ja sovellusten näkökulmasta ne toimivat lähes samalla tavalla kuin natiivit ympäristöt. Ohjelmistokontit ovat nousseet suosioon tehokkaana sovellusten toimitusteknologiana. Molemmat, sekä virtuaalikoneet, että ohjelmistokontit tarjoavat niiden sisällä suoritettaville sovelluksille eristetyn virtuaaliympäristön. Ohjelmistokontit eivät pyri virtualisoimaan kaikkia järjestelmän resursseja vaan käyttävät alla olevaa käyttöjärjestelmän ydintä hyväkseen. Tämä tekee ohjelmistokonteista houkuttelevan vaihtoehdon virtuaalikoneille.
Tässä diplomityössä suoritettiin erillaisia suorituskykymittauksia ohjelmistokonttien ja virtuaalikoneiden avulla luoduissa ympäristöissä. Myös alla olevan isäntäkoneen natiivisuorituskyky mitattiin, josta saatiin hyvä arvo erilaisten virtuaaliympäristöjen vertailuun. Mittasimme pysyvän muistin, verkon ja sovelluksen suorituskyvyn. Sovelluksena toimi Linux:in kääntäminen lähdekoodista toimivaksi käyttöjärjestelmäksi.
Tuloksemme osoittavat että sovellussuorituskykytestissä kontit häviävät natiivijärjestelmän suorituskyvylle vain vähän. Eron oletetaan johtuvan tavasta, jolla valitsemamme konttiteknologia hoitaa pysyvän muistin lukemisen ja kirjoittamisen. Oletusverkkoasetuksilla, kontit hävisivät natiivijärjestelmälle myös.
Kaikissa tekemissämme suorituskykymittauksissa virtuaalikoneet hävisivät natiivijärjestelmälle, sekä ohjelmistokonteille.

Avainsanat: Virtualisointi, Ohjelmistokontit

Contents

Symbols and abbreviations

Abbreviations

APIC	Advanced Programmable Interrup Controller
CDN	Content Delivery Network
COTS	Commercial Off The Shelf
CPU	Central Processing Unit (also known as Processor)
DMA	Direct Memory Access
EPT	Extended Page Tables
FIB	Forwarding Information Base
IAAS	Infrastruture As A Service
IETF	Internet Engineering Task Force
IPC	Inter Process Communication
KVM	Kernel-based Virtual Machine
NFV	Network Function Virtualization
NIC	Network Interface Controller
ONF	Open Networking Foundation
PID	Process Indentifier
RIB	Routing Information Base
SDN	Software Defined Networking
SF	Service Function
SFC	Service Function Chaining
SR-IOV	Single Root I/O Virtualization
TLB	Translation Lookaside Buffer
UTS	Unix Timesharing System
VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Monitor (also known as Hypervisor)

1 Introduction

This chapter is written in order to describe the structure of this thesis, what kind of questions it is seeking to answer and also tries to highlight the underlying motivation behind the work.

1.1 Motivation

Virtualized mobile network functions are subject to performance overhead caused by the given virtualization environment. New software delivery methods and paradigms include isolating the software in a software container. Some container technologies such as Docker add additional overhead when the isolated process accesses disk or network interfaces. When different virtual environments are nested, the overheads become cumulative.

Virtualization of computing resources is not a new topic. Use cases for it have been around perhaps as long as digital computers have existed. Goldberg (1974) [44] describes an early use case, in which a developer might need to write a program for machine architecture, which has been specified but not been implemented yet. In this case, the system can be virtualized first to enable developers work, and when the actual hardware is implemented, the software can be moved from the virtual environment to the actual physical environment. This type of virtual environment is called a Virtual Machine (VM).

1.1.1 Virtualized Network Functions

Network Function Virtualization (NFV) paradigm is later on introduced in the text. As a general idea, NFV aims to run network functions inside virtual environments, instead of dedicated hardware, and more specifically as Virtual Machines (VMs). Usually these VMs are deployed in private clouds. However, VM is not the only form of virtual environment, by which applications can be delivered and executed on. Software containers, for example, provide a virtual execution environment and smaller virtualization overhead when comparing to VMs. It is possible, that if a network element vendor wanted to move their legacy software from VM to a container, they might first run some parts of the program inside containers, and start this container inside the VM. This leads to one of the nested virtual environments inspected in this work: container inside VM.

1.1.2 Nested virtual machines

Another nested virtual environment that we are measuring is a traditional nested virtualization environment: VM inside VM. This kind of virtualization has many use cases. Zhang et al (2011) [35] introduces a use case for improving VM security, through a nested VM ordeal.

Hypervisor is a program used to create VMs, and is mainly in charge of virtualizing the system resources that a VM requests to use. Oracle [3] is offering a hypervisor designed for nested virtualization. This kind of hypervisor can be deployed on any public cloud service (such as Amazon's EC2) in order to standardize the environment for the VM. The claim here is that cloud computing environments with different data centers, hypervisors and VM images are not exactly same.

Operating Systems (OSs) are gaining hypervisor functionality for example, newer Windows OSs running older ones through a hypervisor. These OSs need nested virtualization support from the environment, should they be deployed on top of a VM.

1.1.3 Software containers

Another kind of virtual environment is provided by software containers. Software containers do not seek to virtualize the underlying system resources but rather to isolate a group of processes.

Docker is a platform used to create software containers. It is mainly designed to support the needs of software continuous delivery paradigm but as it provides a virtual environment, other uses are also proposed. Chamberlain et al (2014) [16] describe a way to use Docker in scientific research, in order to create easily reproducable environments. They list Docker having several benefits in this area, some examples are: Unlike VM images, Docker images are natively versioned and Docker container filesystem is every time created from the original image, providing unchanged computational environment.

1.2 Research questions

This thesis tries to answer two research questions, first being:

How big effect a given virtualization environment has to a given performance benchmark? In other words, a virtualization overhead is to be determined.

Second research question tries to estimate a state of computing environment, through benchmark results and by measuring CPU steal time. The question is:

What is the effect of competing host loads to those performance benchmarks, that were run in order to answer the first research question. Simple CPU and memory based stress processes are started to compete for hardware resources with the virtualization software. We will also limit the CPU time of those stress processes, in order to try to define some kind of threshold, after which the software running in virtual environment would be greatly affected.

We ask these questions from perspective of a general cloud applicaton. This kind of cloud application is unaware of the underlying host state. The host may be dedicated to the guest only, or may be overcommissioned to run multiple VMs at the same time. Also this kind of application is hardly a strict realtime application, as it is unaware of what kind of scheduling policy the host is applying to it.

1.3 Thesis structure

Chapters 2, 3 and 4 discuss technological background of the thesis. Chapter 2 aims to describe how network function virtualization is following other trends, emerging from data centers, such as SDN. Chapter 3 presents general terminology and concepts in virtualization, VMs and software containers are more closely inspected. Chapter 4 is presenting important opensource software used in creating and running virtual environments.

Chapter 5 describes benchmarking related details. Used hardware and software are presented. Used virtualization software and its configuration is discussed. Chosen benchmarks are reviewed. And also how, and what kind of statistics were measured, both from virtulization host side, and virtulized guest's side.

Chapters 6 and 7 demonstrate results obtained from measurements. Chapter 8 attempts to draw conclusions based on these results.

Chapter 9 aims to make some conclusions about nested virtualization, and how different kinds of virtual environments could be taken into use in future.

2 Path to network virtualization

This chapter talks about how increasing use of cloud services and data centers lead to a computer networking paradigm called Software Defined Networking. Eventually SDN was followed by another paradigm called Network Function Virtualization (NFV). In NFV, network services that were previously running on dedicated hardware were moved to data centers and to Virtual Machine (VM) environments. In mobile networks side, this lead first to virtualization of the network core elements, and eventually virtualizating the network radio elements. [74, 33]

Virtualizing radio elements is expected to carry many benefits. They may increase network capacity by allowing more light weight deployment of new Base Band Units, energy effiency is expected to increase and cost savings can be achieved on baseband resources. [36]

2.1 Data centers and cloud

Netscape was started at 1994, Amazon and eBay at 1995. These were early internet sites selling services to customers. At 1999 salesforce.com started selling cloud services to businesses. Amazon started their Amazon Web Services at 2006. It was the first time cloud computing resources were available to everyone, not just big businesses and organizations. Business services were now sold and operated over web browsers. Some estimates say that in 2015 Amazon hosted 1,5 million servers and Google and Microsoft 1 million each. [51, Chap. 5]

With cloud services becoming more and more common, latency quickly became an issue. One reason to this is, that theoretical throughput of TCP is limited by latency, as seen in (1). It was partly solved by building additional cloud sites around the world. Another way to tackle this issue is to use a Content Delivery Network (CDN) companies, such as Akamai. CDNs allow streaming services such as Netflix to stream digital content to subscribers by using local caches. Akamai operated in 2014 in 102 different countries. Google and Netflix both also have their own CDN operations. Netflix's one is called Netflix Open Connect Network and Google's one is called Google Global Cache. Both of these operations aim to put these so called 'network caches' directly into Internet Service Providers networks. [51, Chap. 5]

$$\text{Maximum throughput} = \frac{\text{TCP window size}}{\text{Round Trip Time}} \tag{1}$$

Cisco (2016) [63] has predicted some numbers relating to growth of cloud service usage. The amount of large scale public cloud datacenters will grow by 53% during time from 2015 to 2020 and at the end of 2020, they will add up to 47% of all data centers. Data center IP Traffic, that consists of to, from and inside of data centers will grow from 4.7 ZettaByte (ZB, 10^{21}) in 2015, to 15.3 ZB by year 2020. At 2015, 77% of the data center IP traffic stays within the data center, and this is expected to stay steady atleast until year 2020. Rack-local traffic is exluded from these numbers.

Modern data centers consist of densely packed racks. Racks contain so called *blades* which usually contain a single physical server. Figure 1 shows 5 data center cabinets, each hosting tens of physical servers. Single server could be a host for example 20 Virtual Machines. If a data center contains 120 000 blades, it might have to provide networking for 2 400 000 Virtual Machines. Each rack communicates to the rest of the network through a Top Of the Rack (TOR) switch. As this setup is mainly static, it sets different kind of requirements for networking, than the prior networks. Data center traffic has different topology, traffic patterns and scale, as traditional networks. Software Defined Networking (SDN) is a technology that was designed to answer the shortcomings of traditional internet traffic routing. [59, 1.3]

Research conducted by Facebook [11] indicates that most of the traffic happens inside cluster (57,5%). 12,9% happens inside rack and 11,9% happens inside the datacenter. The rest is traffic that is going outside of datacenter (17,7%). The research was measuring network traffic inside 3 sets of data centers, i.e. clusters, running different applications.

From these examples, it is clear that data center IP traffic will continue to grow, and create new kind of routing challenges and possibilities to data center operators.

2.2 Software Defined Networking

Separation of data plane and control plane, as well as the use of OpenFlow form basis of SDN. Also the management of network distribution state, regardles of chosen centralization method, is important part of SDN. [60, pp. 71]

Management of network distribution state also defines one of the fundamental research questions in SDN. Network states range from *Strictly centralized* to *Fully distributed* control plane. Modern SDN controllers typically adopt *logically-centralized* control plane, which is somewhere in between of the two. Fully distributed approach is proven to handle failures very well but may encounter some convergence difficulties. Convergence difficulties in this case, may mean for example infinite loops in network. [60, pp. 9-10] Strictly centralized distribution state is not a feasible option because of three major weaknesses: It has slow responsiveness, because of single point of failure it is not reliable and it is difficult to scale. [45, pp. 1]

SDN packet routing is done in data plane, guided by rules received from control plane. These rules are put into a forwarding table. According to the forwarding table entries, data plane may perform one or more actions on a given packet. Some examples of actions are: forward, drop, re-mark, count and queue. Data plane may also provide Access Control List and QoS -services. [60, pp. 16-17]

Packet being on fast path, means that it already has entry in a Forwarding Information Base (FIB) and the data plane may just forward it according to the given rule. Sometimes an incoming packet may have address, which is unknown to the data plane. In these situations the data plane needs to send the packet to the control plane. Control plane then looks its Routing Information Base (RIB) to create correct rule for these type of packets. In general, RIB contains information about the network topology and when it is found to be stable, FIB is created based on it. A control plane packet may deliver new information about the Network, in which case the RIB may possibly be updated. [60, pp. 11-12, 16]

Figure 1: Data center cabinets, serving as physical mounting points for tens of servers.

2.3 OpenFlow

OpenFlow defines protocol to be used between controller and a network router, and also the expected behavior of the router. It first emerged in computer networking scene at 2008. First it was used by research groups and network vendors, for trying out new protocols in existing networks. About three years later, SDN started to have impact on the networking industry. [59, 3.3]

OpenFlow only defines set of protocols and programmable interface. So a controller which is implementing OpenFlow, always needs also an application program to provide the actual functionality.

The protocols are divided into two groups. A wire protocol sets up a control session, defines message structure when changing flow modifications, and defines fundamental structure of switch ports and tables. Configuration and management protocol allocates physical switch ports, defines high availability and behavior of controller failure. [60, pp. 49]

2.4 Open Networking Foundation

Open Networking Foundation (ONF) is a nonprofit organization that was found in 2011 by Deutsche Telekom, Facebook, Google, Microsoft, Verizon and Yahoo. Its purpose is to openly define new SDN standards and solutions, and bring them to market. OpenFlow was developed to commercial viability by ONF. ONF has published OpenFlow Switch Specifications and also OpenFlow Configuration and Management Protocol. [71]

ONF has defined five features for SDN Architecture. First network control functions can be programmed, because they are no longer tied to forwarding functions. Second the network wide traffic flow can be changed by system administrators. Third network is logically placed around SDN controllers that have whole view of the network. Fourth, system administrators can adjust network parameters with opensource SDN programs. Fifth, network is simpler as it is implemented with open standards and as vendor-neutral. [33]

2.5 Network Function Virtualization

Ultimately OpenFlow and SDN have led to Network Function Virtualization (NFV), where network vendors have been able to move network functions that were previously running on dedicated hardware into virtual environments.

According to Gray, Nadeau (2016) [52, Chap. 1] NFV consists of three components: SDN controller development, VM orchestration, and Commerial Off The Shelf (COTS) component evolution. Since 2013, opensource software has grown its potential in networking and compute environments. Many orchestration components and SDN controllers are based on opensource software.

Others sources [64] have also defined NFV in a similar manner. However, this work emphasizes how SDN is not an irreplaceable part of NFV, but admits that NFV may benefit from SDN. It also says that opensource software and declining costs of server hardware components make important part of NFV.

Service Function Chaining (SFC) was added to NFV concept later. Internet Engineering Task Force (IETF) standard RFC 7665 defines SFC architecture. Service Function (SF) is a function, that has a predefined way of handling packets. It may be located in one of many OSI-layers, and it can be running in either virtual environment or dedicated hardware. Same network element may have many SFs. By definition, SFC means multiple SFs in an ordered chain. SFC also consists of order constrains that are applied to the packets arriving at the SFC. [75, pp. 6]

Adding SFC to NFV plans allowed true service overlay. Also from original plans, Operations Support Systems (OSS) were missing. OSS is used for operating and managing the NFV networks. [52, Chap. 1]

Historically speaking a single network service used to locate on a single device. Some rare expections to this are for example DNS and DHCP services. When services residing on dedicated hardware, were chained together, a physical cabling or internal connection was required. This kind of requirement lead to services being tied to a specific slot in some architecture and also, even to a certain physical location. When comparing to network virtualization, this kind of setup is referred to as tighly integrated service. [52, Chap. 2]

A step from tighly integrated network services towards virtualization of network functions, is loose integration. Gray and Naudeau (2016) [52, Chap. 2] explain that there are three kinds of problems that need to be answered when creating loosely integrated network services. First is, how the required configuration data is passed to the elements. This means, that all of the SFs are required to have similar configuration interface, otherwise the configuration task becomes unnecessary complicated. Second is, how to pass the so called *metadata* between different SFs. Metadata consists of for example subscriber or session ID. After being virtualized, the SFs sending and receiving this data, may be located on different machines and need to connect to each other by using standard protocols. Third problem is the distribution of the user data classification work to different SFs.

Gray and Naudeau (2016) summarize that the combination of advances in OSS, in COTS to support virtualization and high speed network forwarding, and the latest trends in opensource network components form the NFV. [52, Chap. 1]

2.6 Cloud operating systems

Managing cloud resources, such as deploying virtual machines for computational purposes or storage volumes for saving files, relies heavily on cloud operating systems such as Open Stack.

Open Stack is used to control virtualized hardware. The hardware may provide for example computing, storage and networking resources. Common feature in cloud operating systems is a web browser based dashboard, which allows cloud users to take cloud resources into use by e.g., enabling a deployment of server instances. Open Stack consists of dozens of services. Services are divided into groups of core services and optional services. Some examples of core services are Nova, which takes care of a lifecycle of a computing instance. Core service Cinder allows creating a block device for storing data, and attaching it to a computing instance. [29]

Another opensource cloud operating system is called *Eucalyptus*. It offers framework to control computational and storage infrastructure. Big benefit in using Eucalyptus is that its interface is compatible with Amazon EC2 -interface. This way for example a research setup working with eucalyptus private cloud, can easily be ported into commercial public cloud environment. [37, pp. 2-3]

On high level, eucalyptus consists of 4 components. There is one Node Controller on each host and it is used to control VM instances. Cluster Controller is using Node Controllers and also manages virtual networks. Storage Controller provides interface for accessing storage data, for example VM images. Finally, Cloud Controller offers primary interface for users and administrators. [37, pp. 2-3]

2.7 Summary

This chapter was mainly focussing on Software Defined Networks. After reading this chapter the reader should have a brief overview how the increased use of data centers created new demands for computer networking, and how SDN is supposed answer those demands. Network Function Virtualization then was accepted as a paradigm to move network functions from dedicated hardware into virtual environments, running on Common Off-The Shelf components. SDN is considered as a important part of NFV.

Also two cloud operating systems were briefly reviewed. Anybody using cloud services from server administration perspective should easily find themselves facing one of these two, or another kind of cloud operating system.

3 Virtualization concepts

In this chapter, two different kinds of technologies for creating virtual environments are discussed: Hypervisors and Software Containers. Also hardware assisted virtualization is introduced. In hardware side mainly Intel technologies are reviewed.

3.1 Hypervisors (Virtual Machine Monitors)

Popek, Goldberg (1974) [43] defined Virtual Machine (VM) and whether a given architecture can be virtualized. This definition has been widely accepted and referenced in literature. They also defined an idea of Virtual Machine Monitor (VMM) and gave it three characteristics. The three characterstics are: First, VMM should create environment which is identical to the original machine. Only exceptions allowed are available system resources and timing dependencies. Second characteristic is: Resource performance should be *almost native*. It is required that most of the processor instructions are handled without VMM interference. Third characteristic is defined as: VMM must be in control of all hardware resources. This means that a program running under VMM may only access resources, which are directly allocated to it and also, that the VMM should be able to regain control of these resources when necessary. In this characteristic definition, the processor may be excluded from the hardware resources. Popek and Goldberg conclude that 'Virtual Machine is the environment created by the Virtual Machine Monitor'.

In literature and this thesis, the system running VMM is sometimes referred to as *host* and the VM is referred to as *guest*.

VMMs are commonly divided into two types, type 1 and type 2. Type 1 VMM is also known as *bare-metal VMM* and it is run directly on hardware. Some examples of type 1 VMMs are VMware ESX, Microsoft Hyper-V and Xen. [53, Chap. 2]

Type 2 VMMs are run as applications of an operating system. Some examples of type 2 VMMs are VMware Player, VMware Workstation, and Microsoft Virtual Server. [53, Chap. 2]

Type 1 VMMs are considered to be more efficient and secure, because they are not run as a part of an operating system. Type 2 VMMs are considered to be easier to install and deploy, and to be able to support more different kinds of hardware. [53, Chap. 2]

3.2 Hardware assisted virtualization

Basic operation principle of desktop, server, embedded and mobile computers has been revolving around a few simple steps for years. Processor reads a program from persistent storage (typically hard disk or Flash memory) into main memory (typically RAM). It then reads instructions from the program residing in memory and executes them one at a time, sometimes reading and writing data to main memory, persistent storage or other devices such as network controller. Hardware assistance for virtualization allows a virtual machine to take control of this processor, with the hypervisor still being able to take the control back, if the virtual machine is about to break its virtual boundaries. This section mainly discusses hardware assisted virtualization on Intel processors. AMD also has its own hardware assisted virtualization technologies, but they are not discussed here. Intel offers hardware assistance for virtualization of instruction set, memory page tables and device emulation.

3.2.1 Instruction set

Processor architecture x86 is commonly used in the processors of cloud infrastructures. However, fully virtualizing it, has been challenging. In order for hypervisor to maintain control of CPU, the CPU needs to generate a trap when its privileged state is exposed and this is not happening with every instruction in x86. Also hiding privilege state from guest is difficult. [40, pp. 225]

An instruction is said to *trap*, if after executing it, the processor state is restored into the state it was before executing the trapping instruction, and any storage is left untouched. [43, pp. 414]

To overcome the problems in x86 virtualization, AMD and Intel implemented their virtualization extensions. In hardware assisted instruction set virtualization, some guest instructions are executed directly on the processor without VMM intervention. Often this is considered faster than software virtualization, as the VMM does not have to read or modify those instructions. Downside is that those instructions which do need VMM intervention, require VMM to do additional procedures when executing them.

In order to support hardware assisted virtualization Intel processors have introduced two new operating modes: VMX root operation and VMX non-root operation. Also two new transitions are defined. Transition where processor changes mode into VMX non-root, is called *VM entry.* The transition back to VMX root mode is called *VM exit.* These modes and transitions are illustrated in 2. VM is supposed to run in VMX non-root operation as this mode has limited amount of instructions and some registers have been made unaccessible. Also in this mode, some instructions are modified to cause VM exit causing processor control being handled back to VMM. VMM is supposed to run in VMX root operation mode, as in this mode the processor has no VMX-related restrictions. After VM exit, VMM may do any necessary actions it deems, and then resume VM execution with a VM entry. [61, Chap 23-1] Under Linux systems, cpu flag *vmx* indicates this capability.

Necessary data to handle transitions between VMX root operation and VMX non-root operation is stored in Virtual-Machine Control Structures (VMCS). VMM may use different VMCS for every VM it has created and may also use different VMCS for every virtual processor. The host processor allocates a memory region for every VMCS. [62, Chap 24-1]

Data contained in VMCS can be categorized into three groups. Guest state, which stores virtualized guest CPU registers. Host state holds host CPU registers, to be used when CPU is switched back to root mode. Control data, through which the VMM may for example send interrupts to guest machine. Control data also allows the VMM to define VM exit reasons and allows VMM to see after VM has exited, which VM exit reason caused the exit. [46, pp. 4]

Intel has also implemented Shadow VMCS in order to decrease performance overhead caused by nesting VMs. When using Shadow VMCS, the host VMM creates a Shadow VMCS in processor memory. Then the guest VMM can access this specific VMCS directly, without getting interrupted by the host VMM. The host VMM still needs to keep a copy of this shadowed VMCS in system memory, which requires additional copying of data. However, turns out that often 90% of data in VMCS is not read, and 95% of data is never written to. VMCS are edited only with instructions called *VMREAD* and *VMWRITE* and Shadow VMCS feature allows host VMM to define bitmaps for these instructions, in order to restrict access to the shadowed VMCS data fields. This reduces time it takes to copy the shadowed VMCS from processor memory into system memory by up to 15 times. [70, pp. 5]

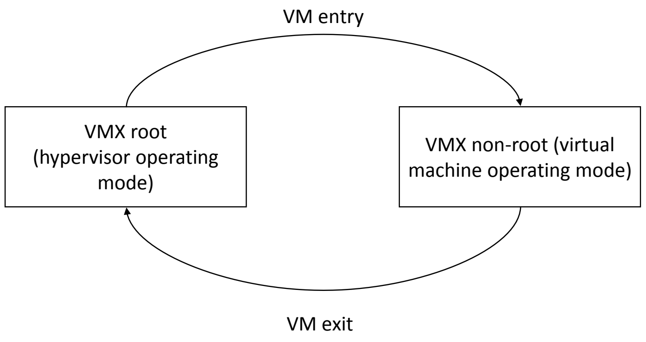
VM entry
VMX root (hypervisor operating mode)
VMX non-root (virtual machine operating mode)
VM exit

Figure 2: Figure illustrating VMX operation modes and transitions of a processor. In VMX non-root operation mode, not all instructions are available and some instructions and memory areas are configured to cause a VM exit.

3.2.2 Memory page tables

Memory management of VMs also has its own hardware level support. Before trying to explain it, a basic memory management of Linux is quickly reviewed.

In order to use memory and CPU more efficiently, Linux implements a basic memory management technique called *virtual memory.* The idea of virtual memory is to divide process memory into small units called *pages*. Then also main memory is divided into *page frames* which are of the same size as pages. Now the kernel maintains a record called *page tables*, through which it can map each page into a page frame. [56, pp. 118-120] x86-architecture processors support memory paging. The memory page table address is stored in Control Register 3. Thus meaning, that in this architecture the address translation is done on hardware level. [62, Chap. 25-2, 3-4]

A step towards full virtualization of memory management is *shadow page tables*. These page tables are maintained by hypervisor and map guest virtual addresses directly into host physical addresses. Maintaining shadow page tables creates some virtualization overhead however, because whenever the guest updates its own page tables, the hypervisor needs to update the changes to the shadow tables. [40, pp. 227]

Intel's technology for hardware assisted virtualization of memory page tables is called Extended Page Tables (EPT). The basic concept of EPT is to translate a guest physical address into an actual physical memory address, along with the host physical addresses. This is done by using a set of *EPT paging structures*. For every incoming access to a memory address, the processor checks page tables of both guest and host. [61, Chap. 25-2]

A maximum of 4 EPT paging-structure entries are read while translating an address. EPT paging structure, and how the addresses are resolved is fairly complicated to explain in the scope of this thesis, but still an example of an address resolving path is given: Address of EPT Page-Directory-Pointer-Table Entry (PDPTE) is used to access EPT Page Directory Entry (PDE). PDE is then used to read EPT page table. And from this page table, a physical address is actually discovered. [61, Chap. 25-2]

Several sources have measured EPT to be superior, when comparing it to shadow page tables. VMWare (2008) [72] claims EPT to deliver 48% better performance in MMU stressing benchmarks and up to 600% better performance in MMU stressing microbenchmarks. Chamarthy (2013) [41] reports a kernel compilation benchmark to be 6 times faster with EPT than with shadow page tables.

3.2.3 Device emulation

In hardware assisted device emulation, VMM can assign I/O-resource directly to a VM. This way the VM may perform Direct Memory Access (DMA) transfers on the device and access its device generated interrupts. However, this ordeal prevents other VMs from accessing the I/O-device. [27]

Intel processors such as *Intel Xeon E5-2600 v2* have capability to emulate guest interrupts in the hardware.[65] This reduces required VM exits and re-entries when guest is handling interrupts or accessing Advanced Programmable Interrup Controller. The processor is able to access guest APIC using virtual-APIC page, address of which is stored in VMCS. [62, Chap 29-1]

Commonly packets arriving from network to a Network Interface Controller (NIC), cause the NIC to send an interrupt request to a CPU, in order to let the CPU know, that a packet has arrived. When the packet is actually dedicated to a VM, the host CPU needs to send another interrupt to the virtual CPU, which decreases the performance of VM packet processing. This lead to Intel creating Virtual Machine Device Queues (VMDQ), which allowed hypervisor to assign a specific packet queue for each VM, that is accessing the physical NIC. This enabled the NIC to send interrupts directly to each VM. [5]

After VMDQ, SR-IOV was developed in order to more enhance the VM packet processing capability. SR-IOV is a standard in order to enable VMs to share devices natively. A device implementing SR-IOV provides interrupts, memory area and DMA streams for each VM that is accessing it. This physical device may support many Virtual Functions and appear as multiple physical devices. From VMs perspective, those additional functions appear in PCI configuration space, each having own Base Address Register. [68, pp. 11-13]

3.3 Software Containers

In this section, the basic idea behind software containers is explained. Also one type of software container technology is more closely inspected: Docker. Other software container technologies also exist, such as LXC, but they are not discussed here.

In terms of virtualization, software containers are usually referred to as Operating-system-level virtualization. This is a virtualization concept where underlaying kernel is used by the virtual environments. Unlike VMs they do not have a hypervisor restricting access to some system resources. In Docker default configuration, reading and writing files to disk is done through storage driver. Also the filesystem is limited to the container image, which is explained later.

Grey, Nadeau (2016) [52, Chap. 7] list several issues in the usage of hypervisors: Hypervisor creates overhead of full Operating System and introduces nondeterministic behavior to the system through scheduling of virtual processors and traffic shaping on virtual NICs. Industry has moved towards direct access of processors and memory, so including hypervisor in the system may introduce testing permutation problem when supporting and validation is performed. Containers can answer to these problems and also often outperform VMs.

But containers have also their own drawbacks. According to Weldon (2016) [51, Chap. 5], VMs isolate their work loads better than containers. This means that they keep to their virtual boundaries better than containers, in terms of system resources and security. Another key point is that containers need to be run on the same OS, as the OS which the container was created for. VMs do not have this weakness, as their image always contains the kernel program.

3.3.1 Docker

Docker is a software container system that is designed for application delivery and process isolation. It consists of docker images, running containers, registries and services. Docker is written with programming language called Go and runs on Linux systems. Docker used to be based on LXC but is now using libcontainer. Docker Engine forms a core of the Docker technology. It consists of in-host daemon and Docker client. Outside of Docker Engine is the Docker Hub, which is a platform for storing public and private docker images, among other things. [10]

Docker Engine uses Linux namespaces to isolate the processes that are running inside a container. Following namespaces are used: Process ID (pid), Networking (net), Interprocess Communication (ipc), Mount (mnt) and Unix Timesharing System (uts). To limit the available hardware resources such as memory to the process, docker uses Linux control groups. [6] Namespaces and control groups are more thoroughly inspected in the next chapter of this thesis.

Docker images have a layered filesystem and this provides several benefits when comparing to traditional filesystem images like *qcow*. In software development, a new image update is just a filesystem layer on top of the old image. When customer has downloaded an application image, downloading each following update to the image is significantly faster, when comparing for example, to downloading a new whole qcow image for every update. [52, Chap. 7]

When creating a docker process, a new thin layer is created on top of the container image. Inside docker process, every write is stored on top of this newly created layer. Every other underlying filesystem layer stays the same. This means that multiple docker processes can use the same image, and all the writes they make to the image, are only stored in their own 'local' layer. This kind of layer-sharing makes docker startup really fast, as no new data needs to be written to disk when creating new docker process. [8]

Docker offers many possible storage drivers for handling disk accesses and creating image layers of a docker process: OverlayFS, AUFS, Btrfs, Devicemapper, VFS and ZFS. Devicemapper is discussed here, as it was used later on, when doing performance benchmarks inside Docker environment. Devicemapper was developed by Red Hat developers to replace AUFS used in debian Docker. It relates to Linux framework called Device Mapper, which is also developed by Red Hat, but it is not the same thing. [7]

Three steps are performed, when docker allocates disk space for new container image. First, devicemapper creates a thin pool from block device or loop mounted file. Second, a base device is created, which is a snapshot with a filesystem. Third, new image is created by taking a *thin provisioned* copy-on-write snapshot of the base device. [7]

Thin provisioning is a technique for taking snapshots of data volumes. It offers two benefits over previous methods of takings snapshots: Multiple virtual devices can be saved on same data volume and any number of recursive snapshots is allowed. Previous implementation caused performance to degrade steadily with the increased number of snapshot depth. Another characteristic of thin provisioning is that it allows saving metadata separate from the actual data. This way for example, performance of metadata could be improved by storing it on a faster permanent storage. [20][1]

Disk read operations from applications are done through pointers. The devicemapper will find contents of the file from correct layer, and store it to the application memory. When writing, two different kind of operations may be performed: *copy-on-write*, when overwriting existing data, and *allocate-on-demand* when writing new data. Write operations are done in block level instead of file level. These operations may introduce additional overhead and latency for the applications disk I/O performance.

[1]Commit e4c78e210daea17f82f12037005df225e22189b9, `Documentation/device-mapper/thin-provisioning.txt`

[7] Our benchmarks later on in the thesis aim to measure this overhead, when running docker natively and also inside a VM.

Docker also allows binding parts of filesystem directly to the container's filesystem. When using these bound filesystem parts, there is no any additional overhead, and the disk I/O performance is same as on the host. [7]

According to Docker documentation, Docker is not designed to be a virtualization technology, but instead it is supposed to be an application delivery technology. Key difference is for example that if one wishes to update Docker container, they are expected to shutdown the old container, update the image and then just start a new container. VMs on the otherhand can be updated like normal computers, just running the update program and maybe rebooting the VM. [9]

3.4 Summary

This chapter introduced two ways of creating virtual environments: Through hypervisors which seek to emulate all devices required by the guest operating system and through software containers which merely isolate a process. After reading this chapter the reader should also have an idea about hardware assisted virtualization and basic services offered by Docker.

4 Linux virtualization components

This section discusses important opensource software that is mainly developed for Linux environments, has commercial significance and is used for virtualization or to increase its performance.

Also some Linux kernel features that have effect on, or enable virtualization are reviewed.

4.1 Qemu

Qemu is a VMM, that can emulate many unmodified guest Operating Systems (e.g. Windows and Linux), on many host OS (e.g. Windows, Linux and Mac OS X). Inners of Qemu can be categorized to 6 subsystems: CPU emulator, Emulated devices, Generic devices, Machine descriptions, Debugger and User interface. The CPU emulator can emulate many CPU architectures such as x86, PowerPC, ARM and Sparc. [48, pp. 41]

The emulation of CPU architecture is done by dynamically translating every guest CPU instruction into a host CPU instruction. In short, this procedure has couple of steps. First, each guest CPU instruction is splitted into few more simple instructions, called *micro operations*. These small pieces of code are compiled into an object file by GNUs C Compiler. During compilation, a program called *dyngen* is used to generate a dynamic translator from the object file. Then at the guest CPU runtime, this translator translates guest instructions, modifies them if necessary, and executes them on host CPU. [48, pp. 42]

Performance of Qemu was benchmarked using BYTEmark benchmark using native and emulated x86 CPU-architecture. On code doing integer computations, guest was roughly 4 times slower than host. When doing floating-point computations, the difference grew to guest being 10 times slower than host. When emulating whole system, Qemu virtualized Memory Management Unit is about 2 times slower than native MMU. [48, pp. 45]

4.2 Kernel-based Virtual Machine

KVM is a kernel module, which together with Qemu works as a VMM. KVM is used through a device node inside linux filesystem: */dev/kvm*. This device allows for example: creating new virtual machines, allocating memory to virtual machines, and running virtual processors. KVM provides virtualization for Memory Management Unit, processor, and I/O. It also supports hardware assisted virtualization of modern Intel and AMD processors. [40]

IBM's Turtles project was merged into Linux kernel upstream during 2011. [20][2] Theoretically the Turtles project enables starting arbitrary amount of VMs, each one being launched on top of the next one. This has hardly any uses cases whatsoever, so we are discussing nested VMs here, which refers to a VM running inside a VM. Running nested VMs was made possible for Intel processors with VMX-capability and

[2]Commit 801d342432190947928e18f893f073fd87cd8bdf

it allowed host VMM to create guests, which also seemed to have the VMX-capability. However, the guest VMX-instructions are actually emulated by the host hypervisor. [46, pp. 3-4]

Timekeeping of VM may be trivial, as the VM has no real means to discover, if the hypervisor is slowing it down. A basic way to keep VMs clock synchronized with host's time, is sending interrupts to the VM, with fixed length time between every interrupt. The time is then calculated from the interrupts but it goes easily wrong, if for example, the host can not send the interrupt on time. [18]

In order to tackle with this problem, KVM offers to guests a para-virtualization clock device. The time information is then passed from host to guest through a memory page allocated by the guest. [18] This way the guest can update its clock any time it wishes and always get same time as the host.

4.3 Para-virtualization driver for I/O: Virtio

Here the basic principle of Linux para-virtualization driver *virtio* is explained. Other hypervisors such as Xen and WMWare have also their own para-virtualization solutions but here the focus is on virtio. In para-virtualization, guest operating system is running as a modified VM. This provides some benefits: It allows using a VMM in situations where hardware cannot be fully virtualized, performance is increased, and interface between guest OS and virtualized device becomes more simple. [54]

By 2008, Linux kernel supported atleast 8 different virtualization platforms. Virtio was created in order to have one set of para-virtualization drivers that would work on all of these virtualization solutions. This would reduce the amount of work needed to optimize and maintain all virtualization drivers needed by those platforms. Also, when KVM was introduced in 2006, it had no para-virtualization drivers, an issue that virtio would answer. [34, pp. 95]

Virtio contains backend drivers that are located on the host side inside hypervisor and frontend drivers which are located on the guest side. Frontend drivers are located in the kernel source. Five types of frontend drivers are used: block devices, network devices, PCI emulation, balloon drive and console driver. Virtio relies on virtual queues, and each separate driver may have their own desired amount of these queues. After getting connected with the virtio queue interface, each frontend driver may reach their corresponding backend drivers. [26, pp. 2-3]

Originally virtio used Qemu inside user space. This lead to memory copying and context switches when packets travelled from KVM in kernel space to Qemu in user space. Later this was changed and the network stack and virtual queues were simply moved to kernel space, an effort aiming to reduce unnecessary overhead caused by Qemu. [52, Chap. 7]

In a benchmark with KVM and virtio used for networking, TCP bandwidth was increased roughly 40 times, from 7.41 MB/sec to 303.35 MB/sec, when comparing to emulating RTL-8029 NIC with KVM. [19]

4.4 Data Plane Development Kit

Data Plane Development Kit (DPDK) is a framework for packet processing in data plane applications. It has been created and maintained by Intel as opensource software. The DPDK can be compiled under Linux systems. [67, pp. 3]

DPDK handles packets using a Run to completion or pipeline model. Run to completion is a multicore packet processing model, where a packet being processed is handled by a single thread only. Opposite of it, but also a multicore model, is Request based model where different stages required by packet processing are handled by different threads. Main benefit of Request based model is that the number of packets being currently processed, can exceed the number of threads dedicated to processing. Main benefit of Run to completion model is that the thread can store packet descriptor in thread's local memory instead of shared memory. [66, pp. 15-16]

In pipeline model, only a single core polls the NIC for packets.

DPDK has 5 libraries that form its core components. Memory Manager is used for allocating memory. It allocates memory from hugepages instead of heap. The aim here is to reduce cache misses in processors Translation Lookaside Buffer. Ring Manager provides a lockless First In First Out queue. Memory Pool Manager handles pools of objects in memory. It uses rings provided by Ring Manager to store them. Network Packet Buffer Management API is used to allocate and free memory buffers that are used to store messages. Timer Manager provides a timer service, in order to allow asynchronous function exection. [67, pp. 4-6]

Important part of DPDK are the Poll Mode Drivers (PMDs). PMDs drivers access the NIC packet descpriptors without waiting for interrupts. [67, pp. 43] This kind of setup increases potential maximum throughput, but the downside is that the host hast to dedicate a whole CPU core to constantly poll the NIC.

Redhat [31] claims that they have measured bare-metal like performance when running DPDK inside a software container. They used DPDK sample applications, like *l2fwd*, for packet forwarding.

4.5 Open vSwitch

OVS is opensource network switch software that implements OpenFlow. According to tests performed by Emmerich, Raumer, et al (2015) Open vSwitch provided highest packer per second (pps) rate amongst all techniques compared using Linux kernel network stack. With using Intel physical NIC, OVS delivered throughput of 1,88 Mpps, when for example Linux IP forwarding gave throughput of 1,58 Mpps. By using same test setups DPDK applications were much faster, as DPDK vSwitch was delivering throughput of 11,31 Mpps. They conclude that major bottleneck for packet forwarding with Linux is the kernel network stack. Among things listed as kernels weaknesses, are a spinlock when queueing for packets from NIC and unnecessary memory management. When a system is dedicated to perform only packet forwarding, these features give very little benefits. [47]

4.6 Namespaces and control groups

This section discusses Linux kernel features, that are important for software isolation, and which some software container technologies, such as Docker, heavily rely on: Namespaces and Control Groups.

Linux provides namespace interface for processes to allow them joining existing namespaces, creating new ones or leaving their current namespace. Following namespaces are provided: PID, User, Mount, Cgroup, IPC, Network, and UTS [21]. In this section, each namespace is described.

Process ID namespace isolates process identifiers (PID)s. This way same PID can occur multiple times in same operating system, as long as they are located in different PID namespaces. The PIDs in a new namespace start always from 1 and the process with PID of 1 has a special role. It is called "Namespace init process" and if it is terminated, the kernel will terminate all of the processes in that namespace. Every PID namespace has a parent namespace. A process is visible to other processes in its ancestor PID namespace. However, processes in parent namespace are not visible to the processes in a child namespace. Visibility of PID means that a process is able to use a visible PID as argument for system calls that require it. Processes have a PID in every namespace, in which they are visible. [23]

User namespace allows isolation of user IDs, group IDs, root directory, kernel keys and kernel capabilities. Kernel capabilities are separate attributes that may be granted for normal user, in order to perform tasks that are usually allowed only for a root user [17]. Just like PID namespaces, user namespaces can be nested. Since version 3.11 of Linux Kernel, maximum of 32 levels of nested user namespaces are allowed. A process gains all of the capabilities which the user namespace had, the process joined to. If the process created a new user namespace, it gains full set of capabilities in that namespace. [24]

Mount namespace isolates the list of mount points. Creating a new mount namespace creates a copy of the callers mount point list. Whenever using systemcall *mount*, a mount event propagation type may be defined. The possible options are: Shared, private, slave, and unbindable. These allow for example to share mount events with other mount namespaces or totally isolate a mount point. In the latter case, the new mount point would only be visible in the current mount namespace. [25]

Network namespaces virtualize network related resources, such as network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewalls, and port numbers. As one physical network device can only exist in one network namespace, a virtual network device can be used to connect different network namespaces. [21]

Control groups is a Linux feature that allows grouping of processes and then performing tasks on those groups. Control groups were required to gather processes in a way, that when a process forks, its child process becomes also a member of the same control group. Control groups also allow hierarchies, in such a way that each process is mentioned once in each hierarchy. [28]

Practical example of Control group is *cpuset*. Cpuset allows user to allocate specific CPUs and memory nodes to a specific set of processes. Then, a process is only able to see those CPUs which are defined in its control group. [4]

4.7 Process scheduling

When running VMs and Hypervisors inside Linux, Linux scheduler has big effect on their performance. It decides which processor cores they run on and how much CPU time they get. Looking from virtualization point of view, scheduler is important part of operating system, as it is in charge of how much execution time the hypervisor process receives. Thus this section tries to explain basic principles of scheduler and reviews one scheduler implementation more closely: Completely Fair Scheduler (CFS). Also some common parallel computing technologies such as Hyper-Threading and NUMA are discussed, since they affect scheduling decisions.

Scheduler goals can be divided into two groups: Goals from user's perspective and goals from system's perspective. Goals from user's perspective include minimizing the waiting time the process has to spend waiting for other processes and minimizing response times of user interface actions. Also being predictable can be included in these goals, as predictability makes operating the system easier and more comfortable. Important goals from system's perspective include maximizing process completion rate, i.e. throughput and balancing the utilization of CPU and I/O resources. [58, Chap. 6.6]

Hyper-Threading is a Simultaneous Multi-Threading (SMT) technology from Intel. This allows multiple threads to use a single CPU core at the same time. The goal is to reduce the processor idle time, which is caused, when an instruction loads something from the memory, and the processor has to wait for the data to arrive. In SMT system, during this wait time, the processor can execute instructions from another thread. [38]

Non-Uniform Memory Access systems have multiple processors, each having own local memory and being considered as a single NUMA-node. All of the NUMA-nodes have also access to the common system memory. [57, Chap. 8]

A Linux feature called *scheduling domains* was created in order to help scheduler make more intelligent decisions, when balancing loads between CPUs. A single scheduling domain contains a set of CPUs with common properties and scheduling policies. Under every scheduling domain, there are one or more *CPU groups*. The CPU load of a domain is tried to be kept even among all of the CPU groups. When the scheduler is doing its load balancing duties, it consideres each CPU on multiple levels. On Hyper-Threading level, moving process from one CPU to another is relatively light operation, and may occur every 1 or 2 milliseconds. On physical processor level, moving process from one CPU to another is more costly. On this level, the inbalance between CPUs is expected to be higher than on Hyper-Threading level, before balancing commences. On NUMA-node level, the cost is highest and scheduling policy acknowledges this. [32]

Current scheduler in Linux kernel is CFS. It was added to the kernel source code in version 2.6.23. CFS tries to model ideal and precise multi-tasking CPU on physical processor. Through introducing a concept of *virtual runtime*, it aims to meet this goal. Virtual runtime is calculated by dividing the real time process has ran on CPU by total amount of processes. Virtual runtime should be almost the same between all of the processes, and to ensure this, CFS always executes the process with lowest virtual runtime. [20][3]

The process information is stored in a Red-Black Tree, ordered by the virtual runtime of a process. Red-Black Tree is a binary search tree, with each tree node being colored either red or black. The node coloring has to satisfy couple of properties: Every tree leaf has to be black, every child of a red tree node has to be black and all of the tree leaves have to have equal black depth. [50, Chap. 4.3] CFS always executes the process most far on left. This way, the virtual runtime of that task is increased, and it is more likely to end up moving more right on the tree.

4.8 Summary

This chapter presented Linux based opensource software and also parts of the kernel. From software the most important points are that qemu and KVM are commonly used together as a hypervisor, and virtio is a paravirtualization driver seeking to improve VM performance, especially in areas of disk and network accesses. From kernel side, it was told how namespaces and control groups enable process isolation in different areas. Also it should be understood, that every process running in modern multitasking operating system is often competing for resources with other processes, and that the scheduler is distributing those resources.

[3]Commit 09c3bcce7c3f640b560df148a3f47d4a3a13dc5e,
`Documentation/scheduler/sched-design-CFS.txt`

5 Benchmark setups

This section describes how virtual environments were built and which benchmarks were used to measure them.

5.1 Previous research

Recently a lot of papers have been released about benchmarking VM and software container performance. Here two are briefly introduced.

Felter et al. (2015) [49] have done several benchmarks to compare peformance overhead of containers and VMs. They noticed that containers and VMs have close to nothing overhead when performing CPU and memory intensive tasks. However, when doing MySQL application benchmark, an overhead can be seen. In VMs this is expected but also Docker AUFS image system, with Docker NAT networking are measured to have reduced performance when comparing to the same benchmark being run natively on host. In disk I/O benchmarks the host mount is used with Docker, which bypasses Docker storage driver and thus introduces no overhead.

Morabit et al. (2015) [39] compared Docker, KVM, LXC and host environment. They observed that when using *y-cruncher* as benchmark, multicore CPU performance is similar between the two container solutions and the host environment, but the KVM is slightly slower. *NBENCH* benchmark resulted in similar single threaded CPU performance between all of the environments. Only in the memory benchmark of NBENCH, KVM lost to other environments by resulting in over 30% worse benchmark score. When measuring disk I/O performance with *Bonnie++*, Docker and LXC are discovered to be slightly slower than the host. However, KVM sequential write speed is measured to be one third of that of the host, and read speed is one fifth of that of the host. After doing other disk I/O related benchmarks as well, such as *Sysbench*, *dd*, *IOZone*, the authors note that the results have some kind of fluctuations and conclude that it may be challenging to measure performance of disk I/O.

5.2 Test environments

Our benchmarks were run in 5 different virtual environments and on the host. Three of the virtual environments were so called 'nested virtual environments'. The nested environments consist of: VM on VM, VM on docker, and docker on VM. In other words, 'docker on VM' environment means a docker isolated process started inside a VM. Sometimes, we drop out the 'on', when referring to these environments and in this case, 'docker on VM' becomes just 'docker-VM'.

VM and docker environments were also benchmarked alone, and are later referred to as 'Native VM' and 'Native docker' or sometimes just 'VM' and 'docker'. Figure 3 demonstrates hierarchy inside the total of 6 test environments.

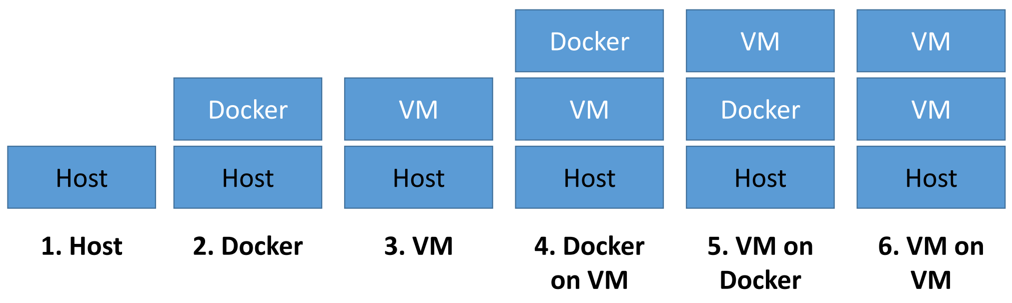

Figure 3: Benchmarked environments and their names.

5.3 Environment setups

This section describes briefly how test environments were configured and what kind of scripts were created for running the tests and recording the results.

5.3.1 Platform software

In all of the environments, Fedora 25 Linux distribution was used as operating system. Particularly, Xfce version of it. Fedora Xface is a lightweight version of the Fedora desktop. Using desktop was necessary for installing the virtual machines, and during other times Command Line Interface (CLI) was preferred, because of its sophisticated automation capabilities. The underlying Linux kernel version was 4.8.6.

Fedora was chosen because the scope of the thesis is on general cloud applications, and thus a realtime operating system is not required. This kind of cloud application can respond to realtime-like requirements by for example, dedicating CPU cores to a certain process using Linux control groups. Fedora has a packet manager and repository from which most of the virtualization and benchmarking software was retrieved. Also its configuration is well documented [12].

As hypervisor KVM was used together with qemu. Other hypervisor options were narrowed out of the scope of this thesis. Qemu version was 2.7.1.

Docker was chosen as the software container technology. The Docker version used was 1.12.6.

These versions are summarized in Table 1.

Table 1: Summary of platform software versions.

	Distribution	Kernel	Qemu	Docker
`Host`	Fedora 25	4.8.6	2.7.1	1.12.6
`VM`	Fedora 25	4.8.6	2.7.1	1.12.6
`Nested VM`	Fedora 25	4.8.6	-	-

5.3.2 Automated scripts

A set of scripts was created to automate redundant testing. To connect to running virtual environments, these scripts used SSH-commands. In order to make it more simple to connect to nested virtual environments, a port redirection was created from the host to target environment. Docker offers a CLI option for this, but with libvirt we had to resort to SSH command's tunneling feature. Important part of these scripts was also to name test result files correctly, in order to make it easy to organize and recognize measured data and results, and also do some post processing of the measured data, to summarize most interesting results.

The automated scripts were also responsible for starting host loads, when measuring guest steal time. Host loads were mainly stressing CPU and memory. A single host load process was launched before running the actual benchmarks, for every CPU core the host system had. Then these loads were distributed evenly among the cores using Linux kernel feature: cpusets. The loads were also given CPU limits using program *cpulimit*.

When starting tens of background processes and their CPU limiting counterparts with a script, some issues were encountered. Sometimes some CPU limiting processes would exit, thus making the load run with all of the CPU time it can get from the scheduler. A bash feature called *disown* was used but the problem could not be fully solved. Then instead of starting the host loads from a measurement script, a systemd service was created to start the loads when requested. This solved the problem, and the host loads were always getting the CPU time that was desired.

A perl script was created to measure and log performance related statistics from the host. The script stores different types of CPU times every second from Linux kernel file: */proc/stat*. Packet throughput was recorded from kernel file: */proc/net/dev*. It also stores Disk read and write statistics with the help of a program: *iostat*. During every test, this script was running to see what actually happens on the host system, when tests were performed inside virtual environments. The script was also used inside virtual environments in order to see mainly guest CPU statistics. In order to minimize its effect on the measurement results, the script was sleeping between sampling times.

5.3.3 Virtual machines

Virtual machines were created using virtualization library *libvirt*. Libvirt was creating VMs using Qemu together with KVM as hypervisor. CPU was set to 'host-passthrough', which enables KVM to provide the VM with identical CPU as the host has. Virtio was used as a virtual NIC. Inside docker environment, VM was installed and started by directly calling qemu with CLI, as running libvirt inside container proved to be challenging because of missing systemd. VMs were connected to network through libvirt default bridge, except for case 'VM on docker', in which qemu user mode networking was used.

In order to make file accesses faster, operating systems maintain *page cache*. Recently accessed files are stored in this cache, which resides in main memory. This makes frequent accesses of same files faster. [14]

When creating a VM with KVM, both the host and guest may create and update their own page caches. It is considered as a good practice, to only resort to one of these caches, and complete ignore the other [14]. When creating VMs, we set them to bypass hosts page cache. Virtio -driver was used as a hard disk driver and VM image format was *raw*.

5.3.4 Software containers

Dockerfile was used to build the docker image. This image was based on Fedora and was built to contain the necessary tools for benchmarks. The image was configured to start SSH-server when it launched, to make it easy for automated scripts to connect to the environment. Devicemapper was used as a storage driver and network settings were left to default, which makes docker to add the process to Docker-bridge network. This type of networking gives the docker process a visible IP-address inside its host environment. Docker storage backend was configured to use direct LVM instead of loopback mount, as advised on their manual. [7] Also trying to setup 124 GB file with fio, inside docker environment running with the loopback configuration, ended up with a kernel I/O error. This error was no longer happening with the direct LVM configuration.

Backing filesystem of Docker's direct LVM was *xfs* while the host filesystem was *ext4*. They are both journalling filesystems, which means that the filesystem keeps a journal of each disk operation performed. Now if a power failure occurs, the next time disk is booted, operating system does not have to go through whole disk to look for corrupted data. Instead it needs only to check recently accessed files for inconsistencies. [55, Chap. 6]

5.4 Hardware specifications

A single data center capable blade was used as the benchmarking environment host. The physical blade, surrounded by other server racks and mounted in a data center cabinet, is illustrated in Figure 4. The physical blade with cover removed, exposing standard hardware components, such as Intel processor can be seen in Figure 5.

As a processor, it had two Intel Xeon E5-2630 v3s. This kind of processor has 8 cores with each having 2 Hyper-Threads. The base frequency is located at 2.3GHz and the processor is based on Intel's Haswell microarchitecture. [15] This processor has many of the hardware virtualization capabilities that were introduced earlier. Some examples of capabilities are mentioned here: VMX, EPT, VMCS, Shadow VMCS, and APICV.

Hard disk was Seagate's Constellation ES3. This is a RAID capable, SATA-inferface connected hard disk and it is implemented with a traditional rotating disk technique [69]. Table 2 summarizes the hardware specifications.

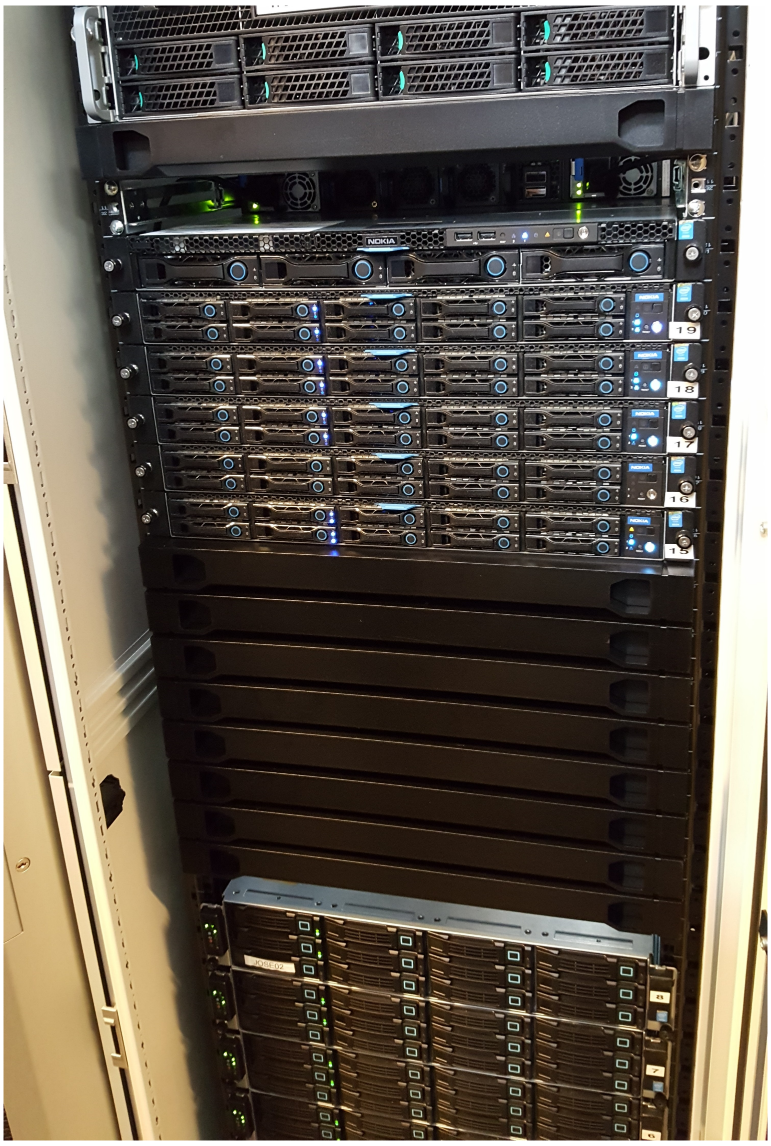

Figure 4: Server rack mounted inside data center cabinet. At least 8 different physical servers can be seen in this picture, as well as some empty slots.

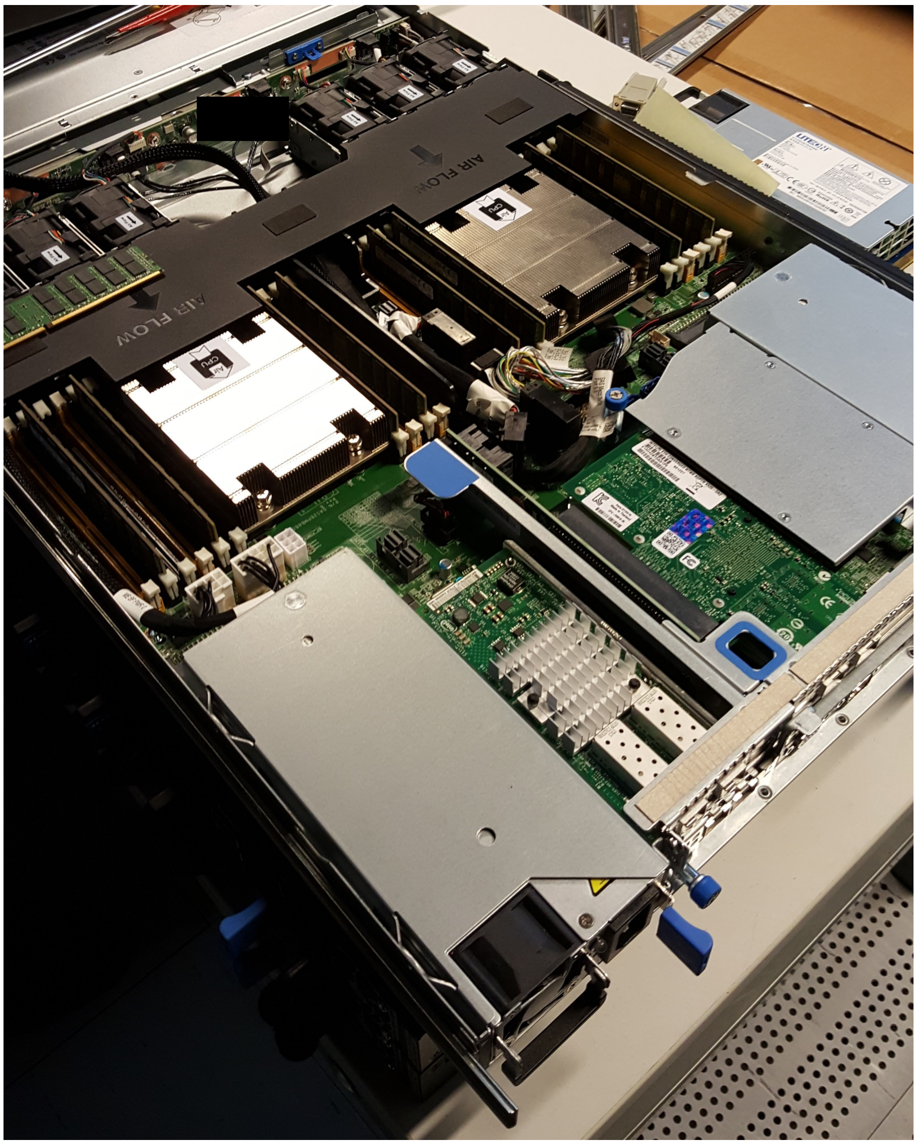
[redacted]

Figure 5: Server rack with top cover removed. In the middle of the picture two processor heat sinks can be seen. At bottom middle is first power unit, and second one is at right edge. Between the two power units, are located two network interface controllers. Hard disks, which are located at top-left, are barely visible. Main memory *combs* are surrounding the processor heat sinks.

5.5 Performance Benchmarks

Here the benchmarks used to measure virtualization performance are presented. General points of each benchmark are discussed and also the actual program versions, as well as command line calls are shown.

Single core CPU performance was also briefly tested, but the results are not reported here. In this case VM, docker and host benchmark scores should be very close, if not equal.

Table 2: Table describing Bare metal host specifications.

	CPU model	CPU cores	Memory	Disk	NIC
`Host`	E5-2630	32	62 GB	1 TB	82599ES
`VM`	E5-2630	17	20 GB	600 GB	virtio
`Nested VM`	E5-2630	16	10 GB	200 GB	virtio

5.5.1 Compilation benchmark

General multi-core CPU performance is tested by timing compilation of Linux kernel. It also stresses memory and disk I/O. Compilation of Linux also represents a real application load, one which could be actually run for commercial or academic purposes inside a virtual environment. Linux was chosen to be the compilation benchmark, because the same benchmark has been used elsewhere also. Although the results are not directly comparable because of different hardware and compilation configuration were used, the previous research [41] [42] gives us a hint what the virtualization overhead could be. Basically, instead of Linux, any other program could have been compiled, and it should yield same kind of results.

The compilation is configured in a host environment and then the same configuration is used on all of the test environments. In every environment the compilation was performed 10 times, then cleaned, and the average of these compilations is reported. 15 jobs, as understood by GNU make command, were used for each compilation.

5.5.2 Disk I/O

Linux program *fio* was used to test disk I/O performance. Previous research that we reviewed also used the same benchmark [49]. Other choices for this benchmark would have been *Bonnie++* and *sysbench*, use of which has also been documented [39].

Fio was created by a maintainer of Linux's block layer in order to simplify kernel debugging [13].

Fio was configured to do sequential read and writes on a 124 GB sized file, which is the double of the host's RAM size. I/O was performed in native Linux asynchronous way and was set to access disk directly, bypassing the disk cache. Fio version was 2.12. Following listing shows the actual command line command to start the benchmark:

```
$ fio --minimal --rw=rw --size=124g --ioengine=libaio \
  --direct=1 --ramp_time=4 --file=FILENAME
```

5.5.3 Disk latency

Linux program *ioping* was used to test disk operation latency. It tests how long disk request took to complete.

Ioping version was 0.9. Measurement was repeated 10 times. Upper and lower bounds of accepted results were also set, in order to avoid results that would heavily differ from others. The command used for measuring was:

```
$ ioping . -c 10 -t 100us -T 2ms -q
```

5.5.4 Network throughput

Network performance was tested with *iperf*. It was configured to transmit for 60 seconds, and bandwidth statistics were collected for every 10 second period. Version of iperf was 2.0.8. The client side iperf command was:

```
$ iperf -c SERVER-IP -y c -i 10 -t 60
```

5.5.5 Network latency

Network latency was measured using *ping*. Ping is based on sending Echo Request packets according to ICMP and measures packet travel time from given environment to target environment and back.

The command used was simply:

```
$  ping -w 10 SERVER-IP
```

5.6 Measuring guest steal time

When application is running inside a VM, it is difficult and may be even impossible to try to debug problems caused by host hypervisor. However, one measurable quantity is CPU steal time.

When running in virtualized environment with virtual processor, CPU Steal Time is the time, in which the guest requested CPU time but did not get it from the hypervisor. Measuring steal time may be important when using IAAS and not having access to the underlying machine, as it may be cause for decreased performance inside the virtual environment.

Under KVM hypervisor, steal time indicates the time when virtual CPU was not running, and it is not including idle time. [20][4] The information is actually retrieved from a scheduler data structure, which tells how much a given process was waiting on scheduler runqueue for other processes, in order to use the CPU. [20][5]

Netflix has defined a threshold of steal time. The application running inside VM is measuring CPU steal time, and if the threshold gets crossed, then the VM is shutdown and another one is started on a different physical host. Netflix claims that in their case, the problem of increased CPU steal time in some physical machine usually does not go away by starting a new VM, which is why they prefer to start the new VM in different physical host. [22]

[4]Commit 5924bbecd0267d87c24110cbe2041b5075173a25, `Documentation/virtual/kvm/msr.txt`, Line 231

[5]Commit 2beb6dad2e8f95d710159d5befb390e4f62ab5cf, `arch/x86/kvm/x86.c`, Line 2130

Steal time is not only caused by hypervisor being genuinely busy with other VMs or competing for host resources with other load. In Amazon Web Services, steal time is used to indicate throttling down of CPUs, to prevent users from getting more powerful CPU, or more computation time than what they are paying for. When user is starting a VM instance in AWS, a choice is given where the user has to define the instance type. This chosen type, such as *m1.large*, actually maps to a EC2 Compute Unit (ECU), instead of mapping to actual physical hardware or virtual CPU. [1] Amazon started using ECUs to normalize the differences between hardwares. When user is renting ECUs, they can expect to always get around the same computation power, despite what the underlying hardware is. [2]

6 Results: Initial benchmarks

This chapter presents initial benchmark results. That is, for the case in which no intentional host load was running, except the benchmark. Results of five benchmarks are presented and then discussed. For kernel compilation, in addition to results, also disk accesses and CPU times as measured from the host, are shown. For disk access bandwidth, total amount of disk accesses are also displayed. For the rest of the benchmarks, only results are presented.

6.1 Kernel compilation benchmark

Figure 6 shows the results of this benchmark as graphs. Table 3 shows the results in numbers. For each virtual environment, average compilation time is reported and also proportion of standard deviation of the average time is shown. Compilation times are normalized to host results in order to demonstrate the overhead caused by virtualization.

Average CPU times during the benchmark are shown in Table 4. Basically 100 units of CPU time account for one processor core for one second. They show that kernel compilation is a proper benchmark for multicore CPU performance. Most of the time CPU is executing something, in either User or System mode. Also it can be seen from 'Sum' column that, in all cases roughly 14 cores out of 15 are constantly used. Only with Docker-VM this does not hold, and 13 cores are mostly used out of 15.

Docker and VM-docker have high IOWait times, when comparing to other environments. One reason for this could be that docker environment has faster CPU instruction execution time and memory than VM, so it is spending proportionally more time waiting for I/O operations than VMs. On the other hand, the host is expected to have faster I/O than docker, so the host is spending relatively little time waiting for disk operations to complete.

VMs use less system CPU time and more user CPU time than other environments. This is because qemu runs these processes in user space instead of kernel space.

Host's standard deviation suffers because during first compilation, all of the source files are read from the disk. This compilation lasted for about 10% longer, than the others. On consequtive compilations, because of disk caching, kernel source files are read from RAM. This leads to compilations done after first one being faster and also to have less standard deviation between the results.

Native docker's average times also suffer from high standard deviation, for unexplained reason. When compiling inside docker environment, often the compilaton was about 5% slower than on host. However, for 2 times of 5 compilations, the process took 40% longer than on host. Compilation benchmark inside native docker environment was repeated but the results were very similiar.

Docker is not expected to lose much to native host environment. The difference is partially explained by docker performing more disk accesses, as demonstrated by Table 5. This table also shows how all of the virtual environments perform more disk accesses than host.

Docker is expectedly faster than VM. Docker on VM is the slowest environment. One reason for this, could be that emulating devicemapper -storage driver could be slower than emulating virtio of VM.

This benchmark was redone with different settings. The significance of configuring docker to use thinpool storage system instead of loopback mount, was realized only at late parts of the thesis. It seems like virtualizing this loopback setup is rather costly, when comparing to thinpool. The redone benchmark results are illustrated in Figure 7. In these results, the performance of docker-VM is much closer on native VM.

In these redone measurements, 4 things were intentionally changed. Same kernel configuration was copied to every environment. Docker storage configuration was changed from loopback to thinpool. VM on docker -environment's VM cache configuration was changed from default to 'none'. The compiled results were synced to disk after every compilation and host as well as guest operation systems were ordered to drop disk caches after every compilation. This more resembles the real use case of compilation application; the compilation needs to be repeated very seldom.

The redone measurements are probably more succesful, as the three VM environments are very close in resuls: VM, docker-VM, and VM-docker. This is in line with our idea, that the docker performance penalty should be very small. The VM-VM environment is now very slow, presumambly because of disk cache dropping; the nested VM setup needs to read a lot of files while also being slowed down by virtualization.

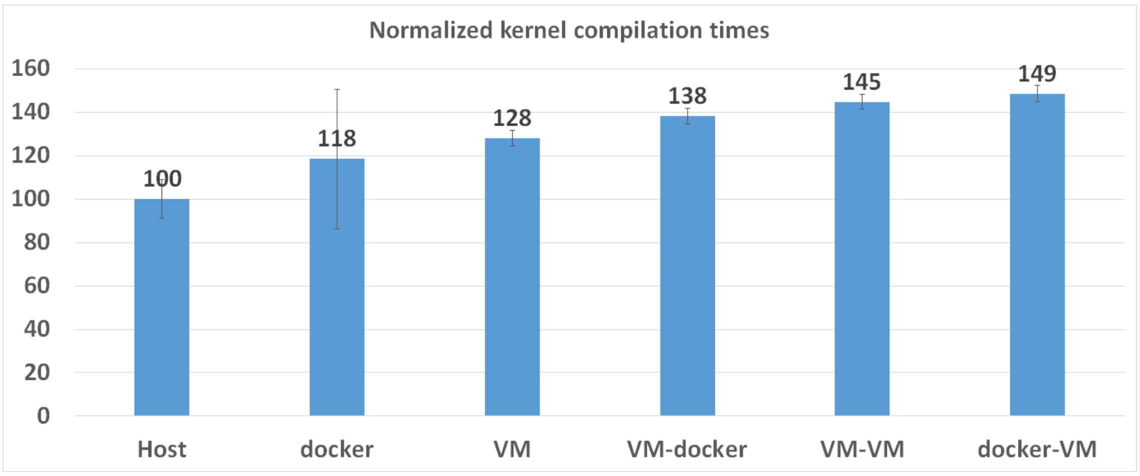

Figure 6: Graph demonstrating virtulization overhead by compiling Linux kernel in different environments.

6.2 Disk I/O bandwidth

Figure 8 shows results of this benchmark. Interestingly, in this benchmark docker is faster than the host, altough it is showing higher standard deviation. The reason why native docker was faster than the host, may relating to performance differences

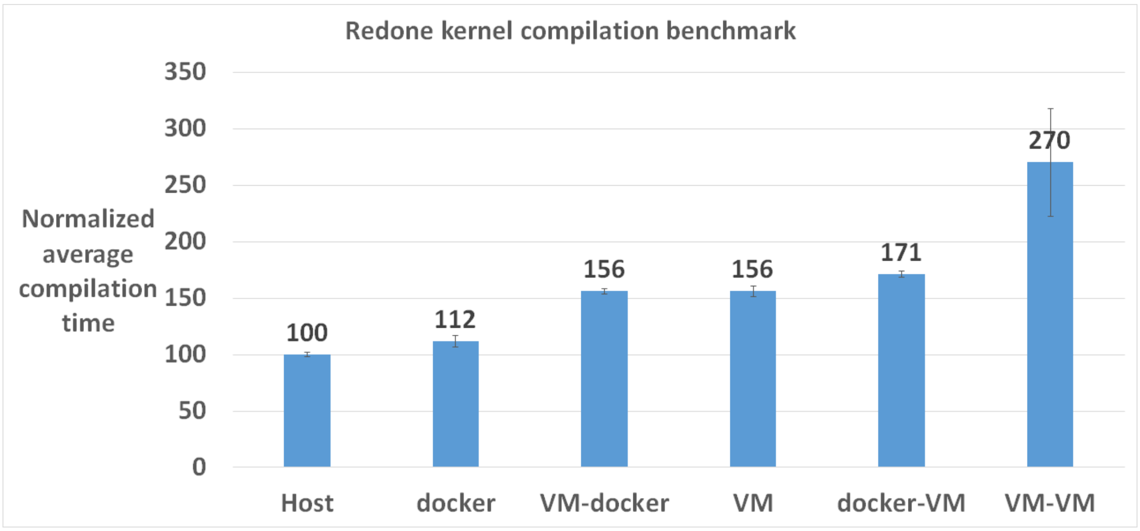

Figure 7: Results of redone compilation benchmarks with different configurations.

Table 3: Table showing results of kernel compilation benchmark as absolute values. 'Avg' means 'Average time in seconds'.

	Avg	Std dev	Guest Steal	Norm. avg	Prop. std dev
`Host`	418	37	0	100	9
`Docker`	495	159	0	118	32
`VM`	535	20	34	128	4
`VM on Docker`	578	21	15	138	4
`VM on VM`	605	21	4	145	3
`Docker on VM`	621	23	29	149	4

between different filesystems. One is that docker was configured to use xfs as backing filesystem, while host's filesystem was ext4.

In the results, VM is significantly slower than docker and host, and running docker on top of the VM adds only very little overhead. VM-docker is suprisingly much slower than docker-VM. It is surprising because on the VM-docker -case, docker is running natively, while on the docker-VM case, docker related processor instructions are more or less emulated by hypervisor.

Table 6 shows total amount of disk accesses during these benchmarks. It is added here to indicate that during this benchmark, all of the environements read and wrote same amount of data to the actual physical disk. Although there are some small differences in the amounts, they do not correlate with the results and can be ignored.

Table 4: Average CPU times on host side during compilation benchmark. Sum -column shows the sum of all CPU times with Idle being excluded from it. Nice and Steal times are not shown, as their values are constantly 0.

Environment	User	System	Idle	IOWait	Interrupt	Soft-Interrupt	Sum
Host	1238	151	1811	14	5	3	1411
Docker	1155	153	1850	64	6	6	1384
VM	1386	19	1820	11	7	4	1427
VM-Docker	1314	37	1836	60	8	6	1425
VM-VM	1305	147	1757	11	9	4	1476
Docker-VM	1210	35	1976	19	7	5	1276

Table 5: Total megabytes read and written to disk during all of the compilation benchmarks.

Environment	Read	Write
Host	749	49129
Docker	996	65247
VM	804	58332
VM-Docker	1059	63228
VM-VM	723	63066
Docker-VM	1032	66912

6.3 Disk operation latency

Figure 9 shows results of this benchmark. The order of environments is expected. Docker adds little latency to disk operations when comparing to the host environment, and VM adds a lot. Docker on VM loses to its counterpart VM on docker. Nested VM is again significantly slower than other environments.

The results of this benchmark seem to correlate well, with the perception of virtualization overhead of the different environments. Again, host is fastest with docker being only a little bit slower, because of the storage driver performance penalty. Next comes VM. Slower than VM, is VM running inside docker environment. Docker-VM is second last in the results, as virtualizing the storage driver is expected to decrease its performance. Even with virtio, the nested VM is almost 6 times slower than the host, with also being the slowest environment of all.

6.4 Network throughput benchmark

Figure 10 shows the results of network throughput benchmark. The absolute values are shown in Table 7. VM on Docker had very low throughput, because qemu user

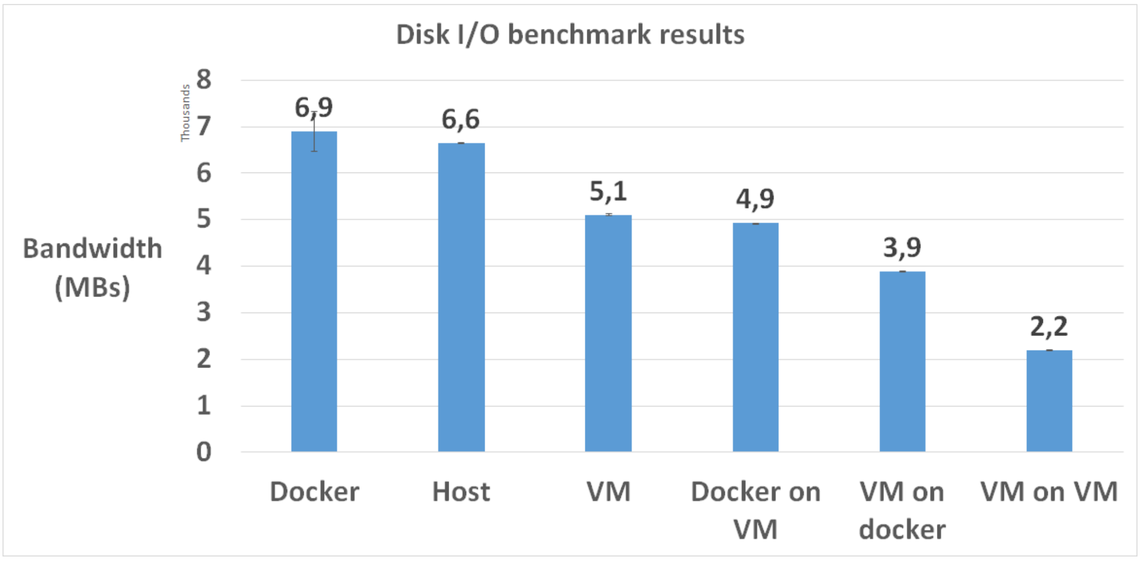

Figure 8: Chart demonstrating disk I/O benchmark results

Table 6: Total amount of disk accesses in megabytes during disk bandwidth benchmarks.

Environment	Read	Write
Docker	130504	260985
Host	130491	261035
VM	132344	264295
Docker-VM	130801	261803
VM-Docker	130846	260905
VM-VM	130866	261588

mode network was used instead of libvirt network bridge.

Docker is again faster than VM. One reason for this could be that they both use similar linux network bridge with Network Address Translation, but the VM has to also emulate NIC behavior, even when using virtio.

This benchmark was conducted to measure throughput from virtual environment to host but measuring throughput to external server would have provided insight on more realistic use-case. However, because of the high transfer speeds, a sufficient network connection was unavailable. In benchmark environment we used 1 Gbps link but even a single 10 Gbps link would not be sufficient to support these transfer speeds, and thus would be the bottleneck of this benchmark.

6.5 Network latency benchmark

Figure 11 shows results of network latency benchmark. VM on docker is missing from results as qemu usermode network does not support ICMP packets, which are

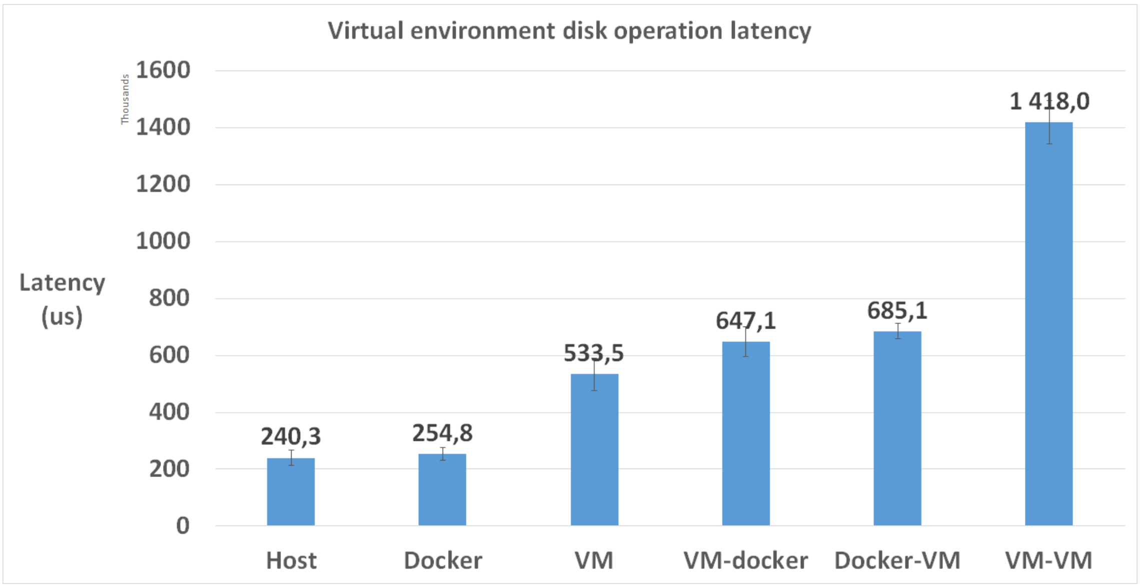

Figure 9: Results of disk operation latency benchmark

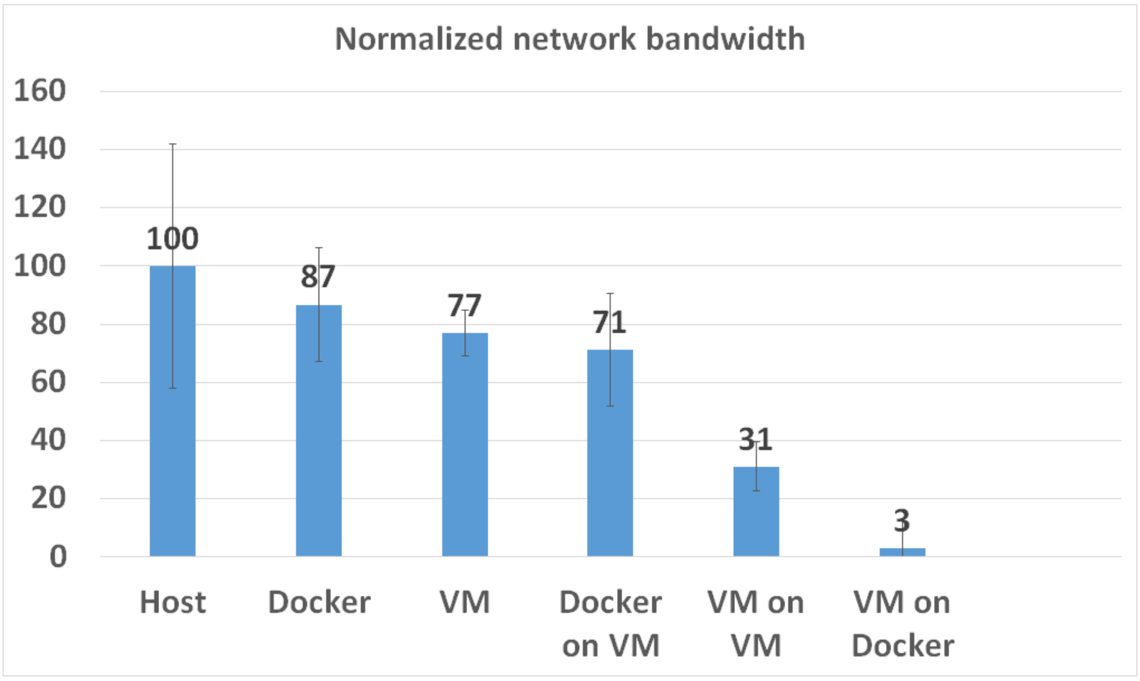

Figure 10: Graph demonstrating overhead in networking, from virtual environment to host.

required by our measuring tool ping. The target server was located on the next rack, and was connected to our test environment through a network switch. Standard deviations are not reported as their values are less than 1/1000th of results in all of the cases.

The results show how docker networking adds roughly a 50 ms network latency when comparing to host environment. The same latency can be seen in two docker

Table 7: Absolute values of network bandwidth benchmarks. The unit is megabytes and it shows the throughput from given environment to host. The host performance was measured through a network loop-interface.

name	avg	std dev
Host	28274	11832
Docker	24521	4813
VM	21747	1695
Docker-VM	20125	3892
VM-VM	8802	739
VM-Docker	819	78

cases: Native docker and docker on VM.

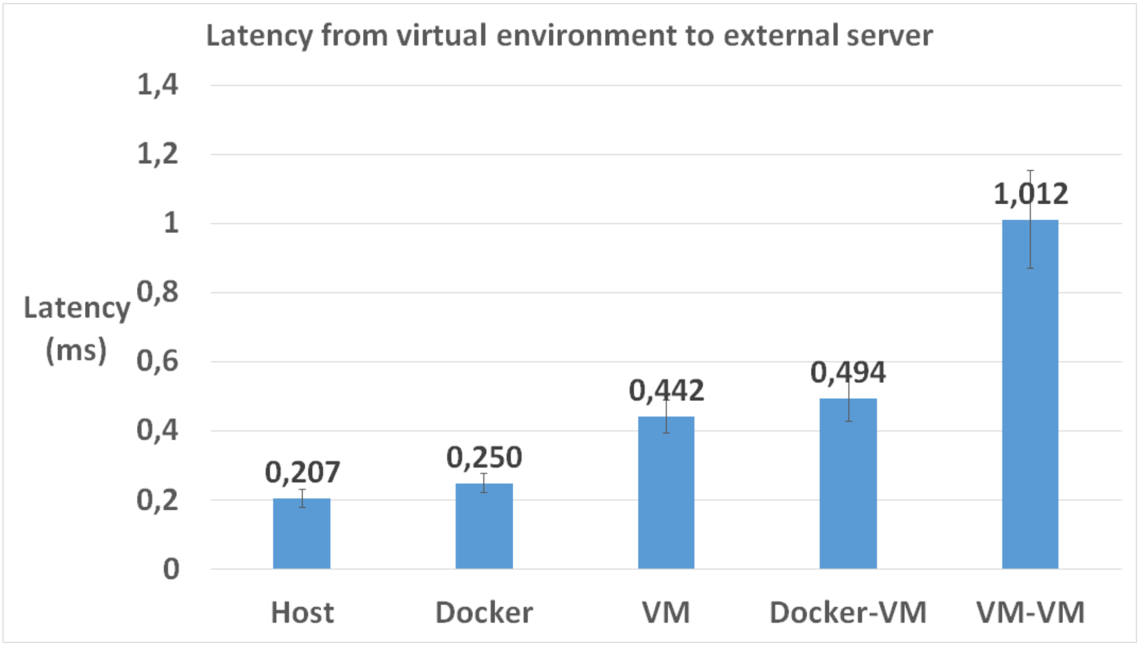

Figure 11: Results of network latency benchmark.

7 Results: Steal time benchmarks

This chapter presents results for compilation, network, and disk benchmarks, when running under additional host load. 10 different levels of host loads were simulated. These loads were ranging from 0 to 100, and were directly given as argument to cpulimit. The compilation benchmarks and network throughput were only done for 4 of the 6 environments used before. In order to keep the measurement times feasible, each measurement was only done twice. For other benchmarks, only one virtual environment was used: docker-VM.

7.1 Compilation benchmark

Figure 12 shows results of compilation benchmark under host load measurements. Table 8 shows the normalized values. VM on docker was mostly effected by the host load and it is not shown in the picture, as it is almost on a different magnitude than other environments. One compilation benchmark under 100 host load took approximately 7 hours, when without host load it takes around 9 minutes.

VM on VM is not much effected by steal time, probably because this kind of environment performs a lot of disk accesses, and thus needs to spend a lot of its CPU time waiting for disk operations to complete.

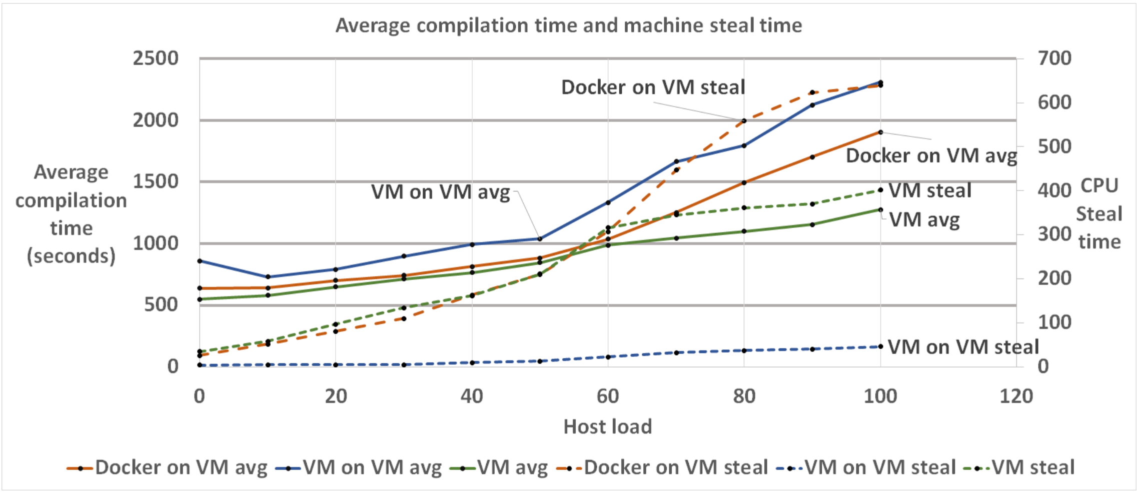

Figure 12: Graph demonstrating how measured steal time correlates with increased compilation time.

7.2 Disk I/O bandwidth

Only after host load is set to 50, we start to see its effect on disk bandwidth benchmark. At this point, the benchmark result has dropped roughly 17% from its initial value. This is illustrated in Figure 13. From host load of 50 to 80 the benchmark result steadily decreases and we can see similar trend on guest steal time. After host load of 80 the benchmark result drops rapidly and steal time rises fast.

Table 8: Table describing normalized compilation benchmark average of each environment, and how it is effected by increased host load. For each environment, first normalized average is described and then CPU steal time.

Load	VM	Steal	Docker-VM	Steal	VM-Docker	Steal	VM-VM	Steal
0	100	35	100	26	100	16	100	4
10	105	58	100	52	111	50	85	5
20	118	97	110	81	119	99	92	5
30	130	134	116	110	135	191	104	5
40	140	162	127	163	160	322	116	10
50	154	211	138	210	212	516	121	13
60	180	316	162	307	288	710	155	23
70	190	345	196	447	436	949	194	32
80	201	361	234	559	737	1246	209	37
90	211	370	267	623	1509	1663	247	40
100	232	402	299	639	-	-	269	46

From these measurements we can conclude, that a measured steal time of 5 can be expected to have a slight worsening effect on disk access speed. And performance of disk accesses is greatly hindered, if steal time measured is over 10.

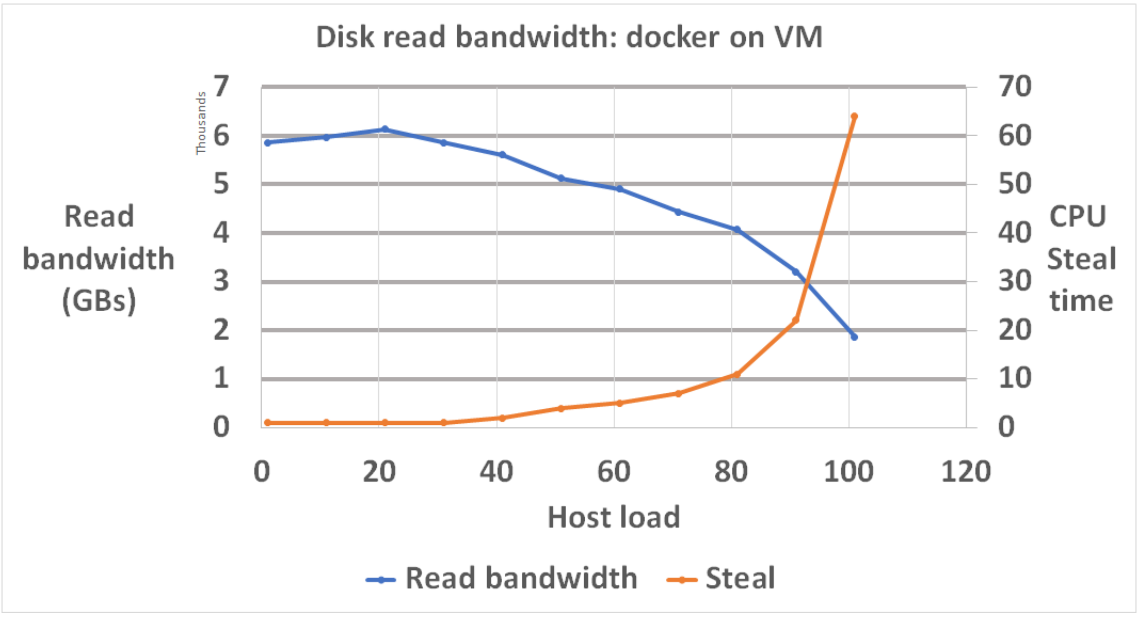

Figure 13: Results of disk bandwidth benchmark when it is run under additional host load.

7.3 Disk latency

Disk latency benchmark results fluctuate strongly. Some kind of unexpected trend of decreasing latency can still be seen. This may hint that benchmark requires some extensive warmup measures, and that results would keep chancing for some time after starting the benchmark runs. 14 shows results of these measurements. Since the latency is decreasing as the measured steal time is increasing, it is safe to assume that host load does not have much effect on disk operation latency.

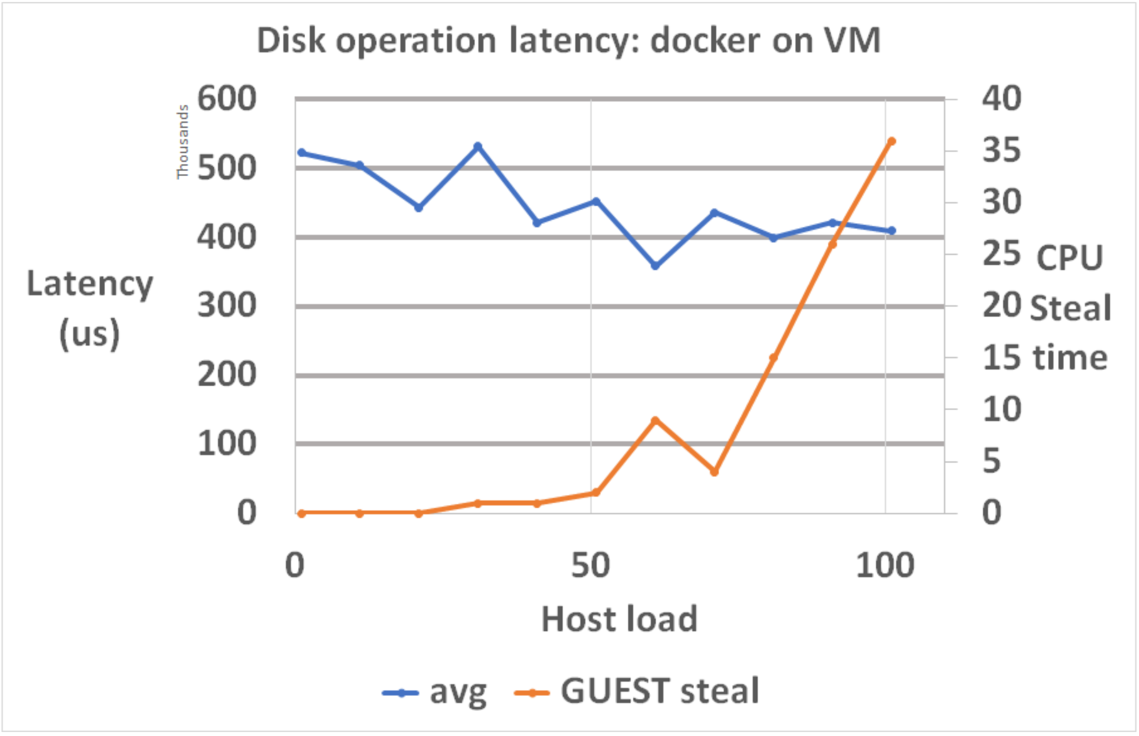

Figure 14: Results of disk operation latency benchmark when it is run under additional host load.

7.4 Network bandwidth

Maximum possible network throughput is strongly effected by other host load. From even the lowest host load the throughput is effected. Figure 15 illustrates these results. Table 9 shows these values normalized.

However, this benchmark has couple of things to consider. First, in this *virtual environment to host* setup the host server is also affected by the host load. This could or could not make the server a bottleneck and whether or not this affects the results, was not confirmed. Second, these transfer speeds are hardly realizable when transferring data between two physical machines.

This test was repeated using 1 Gbps link to another server from docker-VM environment. During these measurements, the host load is barely seen to have any effect on throughput. Even at 100 host load, the performance has decreased only

12% of the maximum throughput. This is illustrated in image 16. The CPU steal time starts to show around 50 host load and significantly increases after 80 host load.

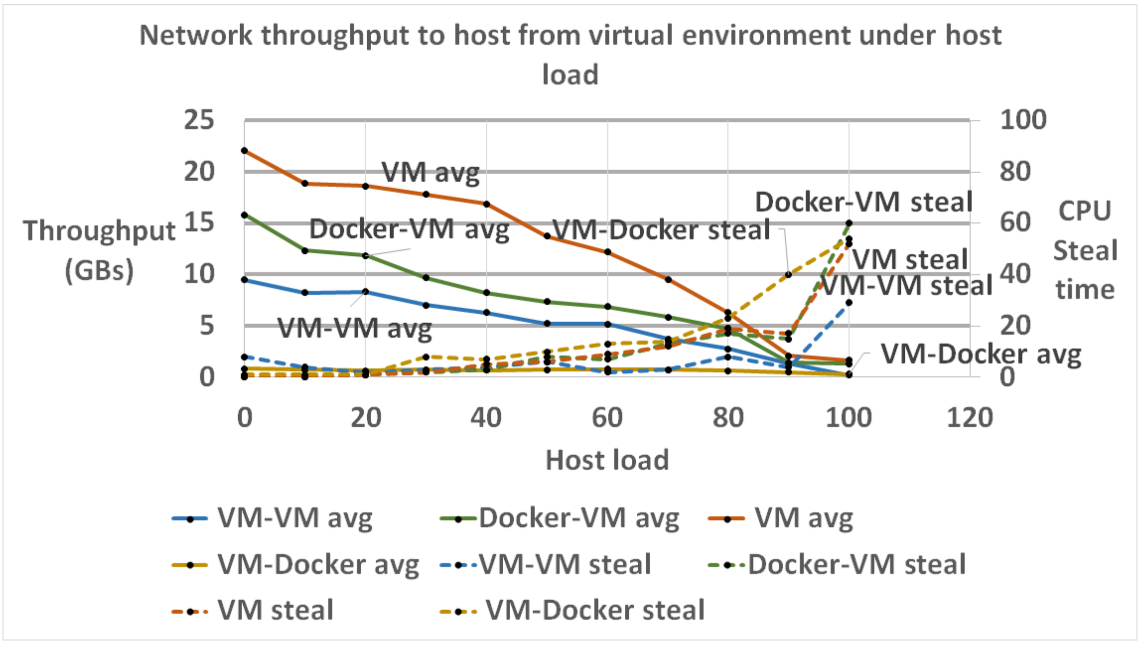

Figure 15: Graph demonstrating how measured steal time correlates with decreased throughput.

Table 9: Table describing normalized network throughput benchmark average of each environment, and how it is effected by increased host load.

Load	VM	steal	Docker-VM	steal	VM-Docker	steal	VM-VM	steal
0	100	0	100	0	100	1	100	8
10	86	0	78	0	92	1	87	4
20	84	1	75	1	81	1	88	2
30	81	2	61	2	91	8	74	3
40	76	5	52	3	85	7	66	4
50	62	6	47	8	91	10	55	6
60	55	9	43	7	94	13	55	2
70	43	12	37	13	89	14	39	3
80	29	19	30	17	79	23	29	8
90	9	17	9	15	61	40	14	4
100	7	52	8	60	34	54	2	29

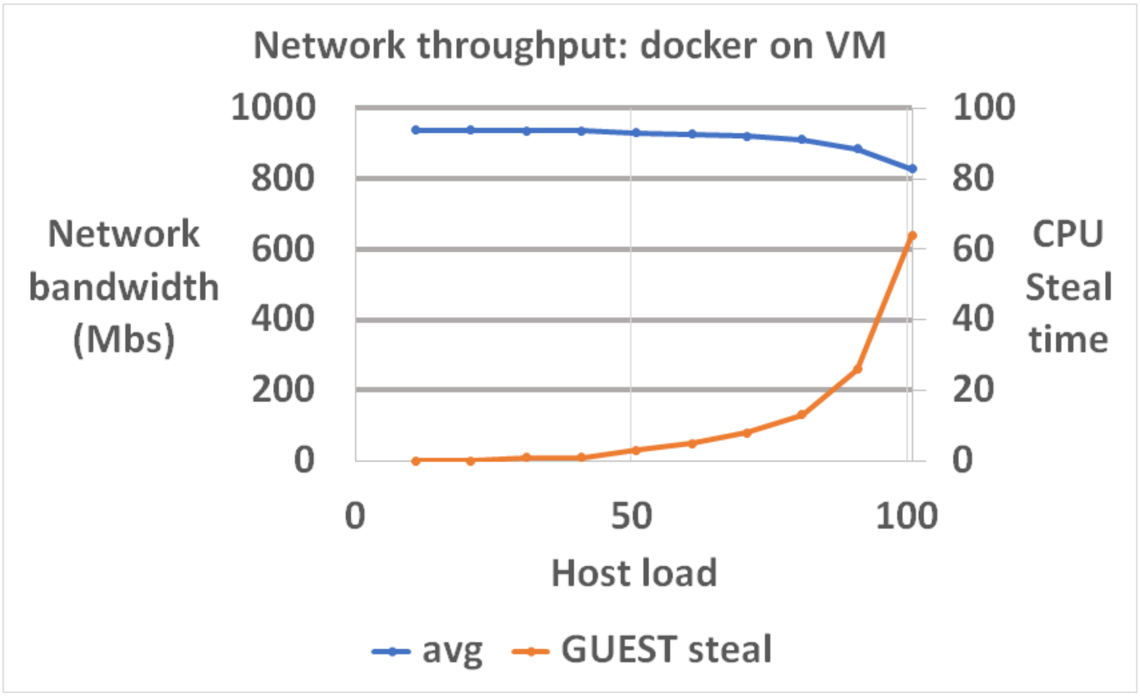

Figure 16: Graph demonstrating how measured steal time correlates with decreased throughput when transferring data to external server.

8 Discussion

This section discusses the benchmark results and aims to draw conclusions on them. Also another kind of experiment is discussed, an attempt to start multiple level nested VMs. Last, some concepts that were noticed during the thesis work are presented, that might be of use when considering future benchmarking.

8.1 Conclusions about benchmark results

From results, it is clear that performance of host and docker is very similar in many areas. The docker storage driver is documented to bring slight overhead to disk accesses, in order to support image layering. This overhead can then be seen in the kernel compilation benchmark. The only difference in compilation benchmark between host and docker is expected to emerge from disk I/O performance. This difference grows when docker is virtualized, by running it inside a VM. The docker I/O overhead was barely visible in our disk bandwidth benchmarks but shows up again in the disk operation latency benchmark. This overhead was again hidden in compilation benchmark, when we run a VM inside docker. The compilation results are exactly same as with the native VM.

Docker default network bridge with Network Address Translation creates a performance penalty to network throughput and latency. This penalty is smaller than in libvirtd default network bridge, when benchmarking connections from VMs. For docker this is not the only way to achieve network connection, and even the host network could be exposed to it. This kind of ordeal would not introduce any performance penalty. Interestingly, docker loses with 50ms to host network latency, and when it is virtualized, again it loses to VM with the same difference of 50ms.

Pure single/multicore CPU or memory performance benchmarks were not done in this thesis. A process inside docker container is able to use the processor the same way as other processes on the system. It is usually also under same scheduler governance. The same applies to main memory accesses. VM with hardware assisted virtualization is expected to have same or slightly worse CPU performance as the host.

8.2 Conclusions about steal time measurements

In compilation benchmark under host load some intresting observations were done. First, the VM-VM -setup result time increases with the increased host load. But this does not show at all in the measured steal time, which only slightly correlates with the results or host load. Seems like the underlying VM is unable to add up the steal time properly. With VM and docker-VM environments the steal time correlates with the increased host load and compilation time. From host load 50 to 60 steal time increases more rapidly than before. Docker-VM keeps up with the new trend but with VM, after the increased trend, the steal time then goes back to previous rate.

Disk bandwidth benchmark shows that small host load does not affect guest's disk access based operations. Only after host load of 40 the disk bandwidth starts to decrease, and it greatly decreases after host load of 80. The steal times for these two trends are respectively: around 5 and around 10. Disk operation latency is not affected by host load.

Our network throughput benchmark shows that the maximum guest throughput suffers even after a slight increase of host load. The steal time is barely measurable at this stage. However, this maximum throughput is not often realizable in real cases, as it would require for example, a 40GBs capable NIC. If we assume the docker-VM -setup is using as 10 Gbps NIC, the throughput is first time affected at host load of 40.

According to our measurements a baseline for steal time can determined. First of all, it is likely that a VM will always get little steal time. For single core, steal time value of 1 over one second is very typical. When the value reaches 5, it is safe to assume that an application running inside virtual environment is somehow affected. After the value is over 10, the application is expected to be heavily effected.

8.3 Multi-level nested virtual machines

An attempt to define limits of nested virtualization with KVM was done. This was done just out of curiosity as this kind of environments do not have many real life use cases.

In multi-level setup three VMs were started, a new one being started on top of the previous one. In following terminology, L0 means the host hypervisor. L1-L3 are VMs nested on top of each other. This is illustrated in Figure 17. In this experiment, installing and booting L3 was observerd to be very unstable. Often when trying to boot the VM, the VM execution ended in an unrecoverable error called Kernel panic. In those cases, which the L3 VM was able to boot, the CPU performance was measured to be around 5% of the host, which hints of reducing from hardware accelerated virtualization into software virtualization.

8.4 What could have been done differently

As benchmarking virtual and nested virtual environments is a complicated process with lots of tunable parameters, this section discusses some of them. Also some other notes for future benchmarking are presented.

Measuring steal time and its effect on application is challenging for many reasons. Simulating *generic host load* is difficult, because in real case this type of load could be consisting of many different combinations of CPU, memory, disk I/O and network I/O -type loads. Also the levels of stress they introduce to those system resources are likely to vary. Our host load only stressed CPU and memory, constantly on the same level during one benchmark.

Limiting the host load to a certain level was done with cpulimit. Now cpulimit itself is a program which also uses some, although little, CPU time. Other choices of limiting the process CPU time also exist, for example control group cpuset.

Doing compilation benchmark for 4 environments, with 10 different host loads and 10 repetitions would last too long for some environments. Thus in disk bandwidth

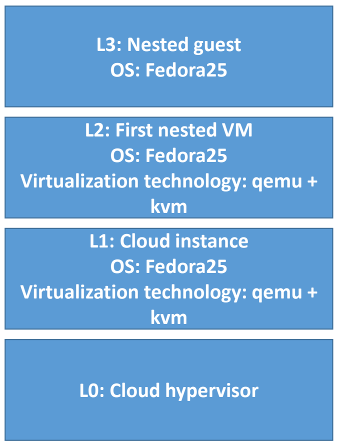

Figure 17: Image shows virtual machine relations to each other

and application benchmark runs were cut down to 2, which lead to high standard deviation in some cases. In some cases the results were even counter intuitive, by benchmark with low host load being slower than benchmark with higher host load.

We could also include host environment in steal time benchmarks. This would demonstrate how native processes compete for CPU time.

Also our CPU time logging script was itself also using CPU time, which is then included in the results. CPU user time and system time of a benchmarking process could be read directly as recoreded by kernel, in proc-filesystem. However, this does not show steal time, as it is accounted for a whole virtual CPU, and not for a single process.

Our testing environments were limited to Linux Fedora distribution, KVM hypervisor and Docker container environment. Other opensource options also remain, and their conception of CPU steal time may differ. The tests could be repeated for example for Xen hypervisor or LXC-container solution.

We only measured network benchmark to external server over 1 Gbps link. Almost all of the environments can easily provide this much throughput, even when the system is under a heavy stress. This benchmark should be repeated with 10 Gbps link.

Docker inside docker was not included in our test environments. Whether it is interesting or not, it should reveal something about nested docker storage drivers. Also network performance would likely be effected.

Kernel module KVM records lots of VM related statistics, and these statistics could be observed and included in future measurements. Those statistics show for example reasons for VM exits.

9 Future work

How Docker and software containers are adopted remains to be seen. Docker allows exposing many container isolation features. For example, filesystem and network of host can be exposed to container, so that the processes inside the container can freely access these resources on the host. Docker also has a option to run container privileged, which means that the processes which have access to host resources, can access these resources without any restrictions i.e., with root priviledges.

Exposing host files to container is necessary for example, in a case where the process running inside container needs to start a VM with KVM. This is only possible if file */dev/kvm* is bound to container filesystem. With VM, the virtual kernel could just create new, virtualized KVM. How these kinds of limitations are solved remains to be seen.

Follow up for this thesis, could be deeper dive into the cloud applications and CPU steal time. We discovered that under Linux host, steal time is actually retrieved from scheduler data structures inside kernel space. How disk access based host load shows up as steal time? Or does it perhaps show as increased IOWait time? A lot of questions remain open about steal time and how cloud applications should measure it, and also, how to interpret it.

References

[1] *CPU Steal Time. Now on Amazon EC2* Leonid Mamchenkov Available: http://mamchenkov.net/wordpress/2015/12/12/cpu-steal-time-now-on-amazon-ec2/ [27 Feb 2017]

[2] *Amazon EC2 FAQs* Available: https://aws.amazon.com/ec2/faqs/ [9 May 2017]

[3] Oracle *HVX: Virtual infrastructure for the cloud* Available: https://www.ravellosystems.com/technology/hvx [21 Apr 2017]

[4] *Cpusets* Simon Derr (2004) Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt [13 Mar 2017]

[5] Simon Dredge (2015) *Accelerating the NFV Data Plane: SR-IOV and DPDK in my own words* Available: http://www.metaswitch.com/the-switch/accelerating-the-nfv-data-plane [16 Mar 2017]

[6] *Docker overview* Available: https://docs.docker.com/engine/understanding-docker [7 Nov 2016]

[7] *Docker and the Device Mapper storage driver* Available: https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/ [31 Jan 2017]

[8] *Understand images, containers, and storage drivers* Available: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/ [30 Jan 2017]

[9] *Docker for Virtualization Admin* Available: https://goto.docker.com/docker-for-the-virtualization-admin.html [16 Dec 2016]

[10] *About Docker Engine* Available: https://docks.docker.com/engine [16 Dec 2016]

[11] *Data Sharing on traffic pattern inside Facebook's datacenter network* Available: https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/ [7 Feb 2017]

[12] *Fedora 25 - System Administrator's Guide* Available: https://docs.fedoraproject.org/en-US/Fedora/25/html/System_Administrators_Guide/ [9 May 2017]

[13] *Flexible I/O Tester* Available: https://github.com/axboe/fio [2 May 2017]

[14] *Best practice: KVM guest caching modes* Available: https://www.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaatbpkvmguestcache.htm [6 Apr 2017]

[15] *Intel Xeon Processor E5-2630* Available: http://ark.intel.com/products/64593/Intel-Xeon-Processor-E5-2630-15M-Cache-2_30-GHz-7_20-GTs-Intel-QPI [10 Apr 2017]

[16] Ryan Chamberlain, Jennifer Schommer (2014) *Using Docker to support reproducible research* Available: http://pfigshare-u-files.s3.amazonaws.com/1590657/WSSSPE2_Docker.pdf [4 May 2017]

[17] *capabilities - overview of Linux capabilities* Available: http://man7.org/linux/man-pages/man7/capabilities.7.html [12 Apr 2017]

[18] *add documentation about kvmclock* Available: https://lkml.org/lkml/2010/4/15/355 [3 Apr 2017]

[19] Ingo Molnar. *[kvm-devel] [announce] KVM/NET, paravirtual network device* Available: http://www.mail-archive.com/kvm-devel@lists.sourceforge.net/msg00824.html [22 Feb 2017]

[20] *Linux kernel, source code* Available: https://www.kernel.org [3 Apr 2017]

[21] *Overview of Linux namespaces* Available: http://man7.org/linux/man-pages/man7/namespaces.7.html [7 Nov 2016]

[22] *Netflix and Stolen Time* Dave Link Available: https://www.sciencelogic.com/blog/netflix-steals-time-in-the-cloud-and-from-users [27 Feb 2017]

[23] *Overview of Linux PID namespaces* Available: http://man7.org/linux/man-pages/man7/pid_namespaces.7.html [7 Nov 2016]

[24] *Overview of Linux user namespaces* Available: http://man7.org/linux/man-pages/man7/user_namespaces.7.html [7 Nov 2016]

[25] *Overview of Linux mount namespaces* Available: http://man7.org/linux/man-pages/man7/mount_namespaces.7.html [8 Nov 2016]

[26] Jones, M. (2010) *Virtio: An I/O virtualization framework for Linux* Available: https://www.ibm.com/developerworks/library/l-virtio/ [8 Nov 2016]

[27] Matthew Gillespie (2009) *Best Practices for Paravirtualization Enhancements from Intel Virtualization Technology: EPT and VT-d* Available: https://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization [12 Jan 2017]

[28] Menage, P. *CGROUPS* Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt [8 Nov 2016]

[29] *What is OpenStack?* Available: https://www.openstack.org/software [10 Nov 2016]

[30] *Steal Time Accounting* Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Virtualization_Deployment_and_Administration_Guide/sect-KVM_guest_timing_management-Steal_time_accounting.html [2 Dec 2016]

[31] *Can you run Intel's Data-plane Development Kit (DPDK) in a Docker container? Yep.* Available: https://developers.redhat.com/blog/2015/06/02/can-you-run-intels-data-plane-development-kit-dpdk-in-a-docker-container-yep [20 Dec 2016]

[32] *Scheduling domains* Jonathan Corbet (2004) Available: https://lwn.net/Articles/80911/ [18 Apr 2017]

[33] *Software-Defined Networking (SDN) Definition* Available: https://www.opennetworking.org/sdn-resources/sdn-definition [3 Feb 2017]

[34] Rusty Russel *virtio: Towards a De-Facto Standard For Virtual I/O Devices* Available: http://www.ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf [21 Feb 2017]

[35] Zhang, Fengzhe and Chen, Jin and Chen, Haibo and Zang, Binyu (2011) 'Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles' *CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization*

[36] Aleksandra Checko, Henrik L. Christiansen, Ying Yan, Lara Scolari, Georgios Kardaras, Michael S. Berger, and Lars Dittmann (2015) 'IEEE COMMUNICATION SURVEYS & TUTORIALS, VOL. 17, NO. 1, FIRST QUARTER 2015 405' *Cloud RAN for Mobile Networks - A Technology Overview*

[37] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, Dmitrii Zagorodnov (2009) 'CCGRID '09 Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid' *The Eucalyptus Open-source Cloud-computing System*

[38] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra and R. Biswas (2011) '18th International Conference on High Performance Computing, Bangalore, 2011, pp. 1-10' *The impact of hyper-threading on processor resource utilization in production applications*

[39] R. Morabito, J. Kjällman and M. Komu (2015) 'Cloud Engineering (IC2E), 2015 IEEE International Conference on' *Hypervisors vs. Lightweight Virtualization: A Performance Comparison*

[40] Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A. (2007) 'kvm: the Linux Virtual Machine Monitor' *2007 Linux Symposium*

[41] Kashyap Chamarthy (2013) 'Notes on taking KVM-on-KVM nested virtualization for a spin' *CloudOpen Europe 2013* Available: https://events.linuxfoundation.org/sites/events/files/slides/nested-virt-kvm-on-kvm-CloudOpen-Eu-2013-Kashyap-Chamarthy_0.pdf [11 Apr 2017]

[42] Bandan Das, Yang Z Zhang, Jan Kiszka (2010) 'Nested Virtualization: State of the art and future directions' *KVM Forum 2010* Available: https://www.linux-kvm.org/images/3/33/02x03-NestedVirtualization.pdf [2 May 2017]

[43] Gerald J. Popek, Robert P. Goldberg (1974) 'Formal Requirements for Virtualizable Third Generation Architectures' *Fourth Association for Computing Machinery Symposioum on Operating Systems Principles*

[44] Robert P. Goldberg (1974) 'Survey of Virtual Machine Research' *Computer - IEEE Computer Society*

[45] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, Anja Feldman (2012) 'Logically Centralized? State Distribution Trade-offs in Software Defined Networks' *HotSDN*

[46] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, Ben-Ami Yassou (2011) 'The turtles project: design and implementation of nested virtualization' *OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*

[47] Paul Emmerich, Daniel Raumer, Floarian Wohlfart, Georg Carle (2015) 'Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems' *ICN 2015: The Fourteenth International Conference on Networks*

[48] Fabrice Bellard (2005) 'QEMU, a Fast and Portable Dynamic Translator' *FREENIX Track: 2005 USENIX Annual Technical Conference*

[49] Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio (2015) 'Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on' *An updated performance comparison of virtual machines and Linux containers*

[50] Michael T. Goodrich, Roberto Tamassia (2015) *Algorithm Design and Applications*, John Wiley & Sons

[51] Marcus K. Weldon (2016) *The Future X Network: A Bell Labs Perspective* CRC Press

[52] Ken Gray and Thomas D. Nadeau (2016) *Network Function Virtualization*, Morgan Kaufmann Publishers

[53] Matthew Portnoy (2016) *Virtualization Essentials*, Sybex

[54] Dan C. Marinescu (2013) *Cloud Computing: Theory And Practice*, Morgan Kaufmann Publishers

[55] Vicki Stanfield, Roderick W. Smith (2002) *Linux System Administration, Second Edition* Sybex

[56] Michael Kerrisk, (2010) *The Linux Programming Interface*, No Starch Press

[57] Sandra K. Johnson, Gerrit Huizenga and Badari Pulavarty (2005) *Performance Tuning for Linux Servers* IBM Press

[58] Naresh Chauhan (2014) *Principles of Operating Systems*, Oxford University Press

[59] Paul Göransson, Chuck Black (2014) *Software Defined Networks: A Comprehensive Approach*, Morgan Kaufmann Publishers

[60] Ken Gray and Thomas D. Nadeau (2013) *SDN: Software Defined Networks*, O'Reilly

[61] Intel (2011) *Intel 64 and IA-32 Architectures Software Developer's Manual, 3B* Exeter: Author

[62] Intel (2016) *Intel 64 and IA-32 Architectures Software Developer's Manual* Exeter: Author

[63] (2016) *Cisco Global Cloud Index: Forecast and Methodology, 2015-2020* Available: http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf [20 Mar 2017]

[64] 2012 *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action* Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf [8 March 2017]

[65] Intel (2013) *APIC Virtualization Performance Testing and Iozone*, Nguyen Khang T

[66] Intel (2008) *Design Patterns for Packet Processing Applications on Multi-core Intel Architecture Processors*, Christian F. Dumitrescu

[67] Intel (2015) *DPDK Programmers Guide*, Exeter: Author

[68] Intel (2011) *PCI-SIG SR-IOV Primer*, Exeter: Author

[69] *Constellation ES - Seagate* Available: http://www.seagate.com/www-content/product-content/constellation-fam/constellation-es/constellation-es-3/en-us/docs/constellation-es-3-data-sheet-ds1769-1-1210us.pdf [21 Apr 2017]

[70] Intel (2013) *4th Generation Intel Core vpro Processors With Shadow VMCS* Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf [27 Mar 2017]

[71] *What is ONF?* ONF (2016) Available: https://www.opennetworking.org/images/stories/downloads/about/onf-what-why-2016.pdf [3 Feb 2017]

[72] VMWare (2008) *Performance Evaluation of Intel EPT Hardware Assist*

[73] VMWare (2006) *A comparison of Software and Hardware Techniques for x86 Virtualization* Keith Adams, Ole Agesen

[74] (2013) *Network Function Virtualization: Use Cases* 'ETSI Industry Specification Group'

[75] IETF (2015) *RFC 7665* 'Service Function Chaining (SFC) Architecture'

A Preliminary measurements

This appendix describes two measurements that were done before the actual benchmarks. Three different types of storage were tested, just to see that our disk benchmark technique yields acceptable results. Then, the kernel compilation benchmark was also done on all of these three storages, to understand the effect the storage type has to the benchmark.

A.1 Storage disk speeds

Three different types of storages were tested: *ephemeral* which is a physical harddisk connected to the machine, *Elastic Block Storage* which is a network storage system, *ramfs* where all of the data is actually stored on RAM. Different storage system speeds are illustrated in A1. Two different type of storage benchmarks were done: Sequential read/write and random read/write.

The results are as expected. Ramfs is much faster than the two permanent storages, and ephemeral is faster than EBS.

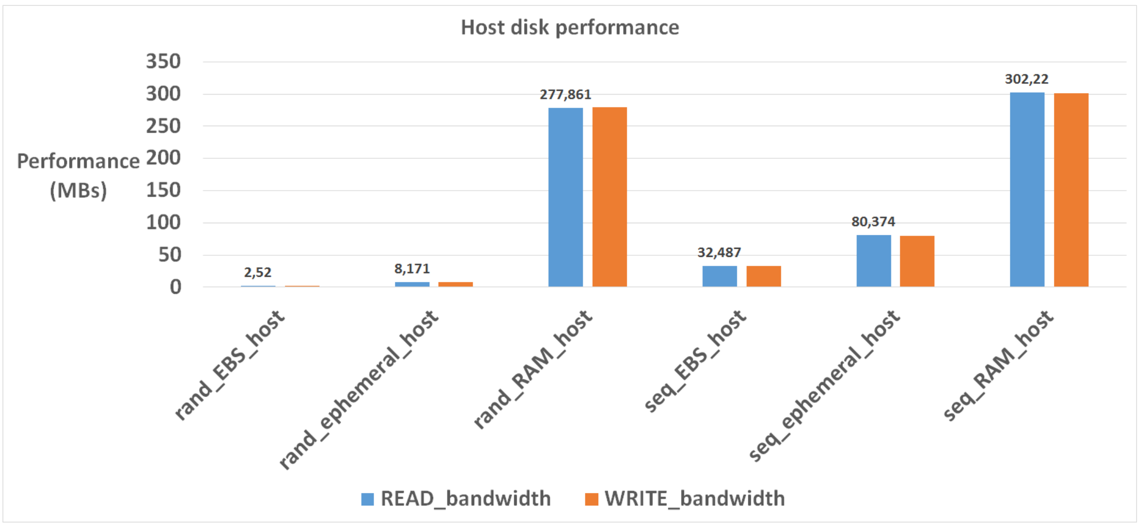

Figure A1: Figure illustrating performance of different file storage systems. As a general observation, EBS is slower than ephemeral, and ephemeral is slower than RAM-disk.

A.2 Storage performance effect on compilation benchmark

Figure A2 shows how usage of disk cache effects on compilation benchmark. Commanding kernel to drop caches before benchmark, increases the compilation time. Slower the disk, more the compilation time is increased. On RAM, the effect is rougly 2%, and on permanent storages it is around 5-7%.

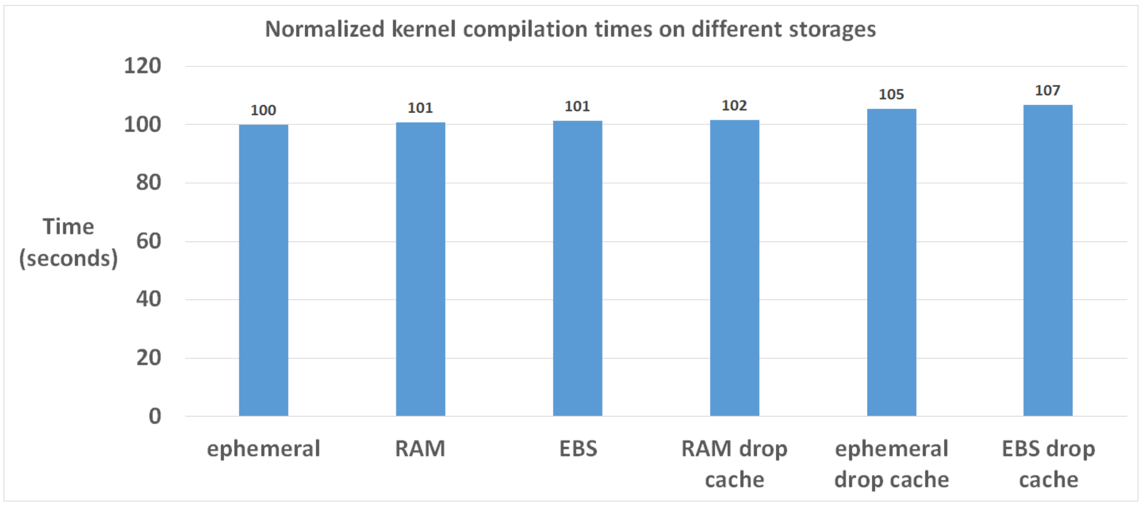

Figure A2: Graph demonstrating how much storage speed effects compilation benchmark

B DPDK with VM from VM

DPDK based packet forwarding was going to be used as one of the benchmarks of this thesis but considering how long time it took to setup one of the cases (VM-VM), it was replaced with iperf. Unlike DPDK, iperf uses linux IP stack normally, but it is much simpler to setup for benchmarking network throughput.

B.1 DPDK inside nested environments

DPDK source code was downloaded and the project was built using standard tools.

To measure DPDK performance, two VMs connected via Linux bridge were used. In the tests the host was also a VM. DPDK pktgen -example application was used to generate OSI-layer 2 packets on generator VM, which were send to the bridge. Then forwarder VM read the packets from the first bridge and forwarded them to a second bridge. The second bridge then lead back to the Generator VM. This setup is described in B1.

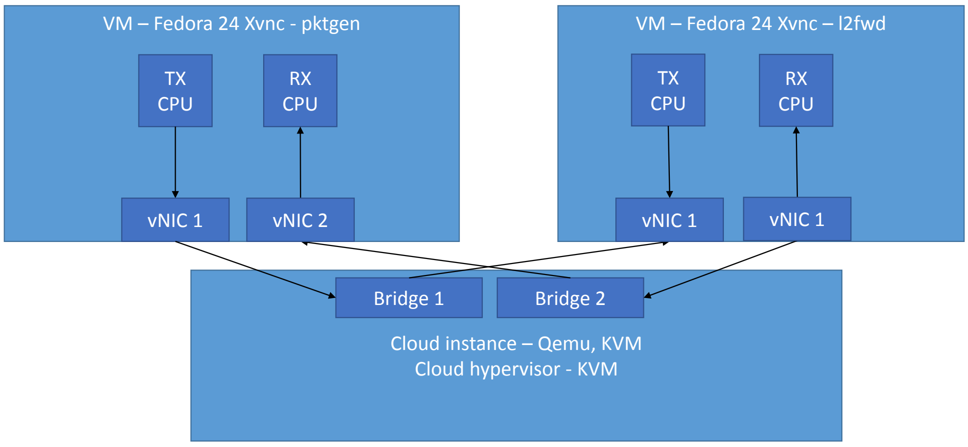

Figure B1: DPDK Forwarding test setup. One VM is generating packets as the other one is forwarding them from one linux bridge to another.

B.2 DPDK CPU loads

As first test, DPDK pktgen application CPU loads were measured. These were measured for two cases. First case, shown in Figure B2, where DPDK was using kernel IP stack, just like a normal program that wishes to send IP packets inside Linux. The second case can be seen in Figure B3. In this case, NICs were bound directly to DPDK, which is how the DPDK is designed to be used. CPU times show clearly how the DPDK bypasses the Kernel IP stack. Packet throughput is described in Figure B4. It was measured from host side, as the guest kernel is unaware of any network activity on DPDK bound NIC.

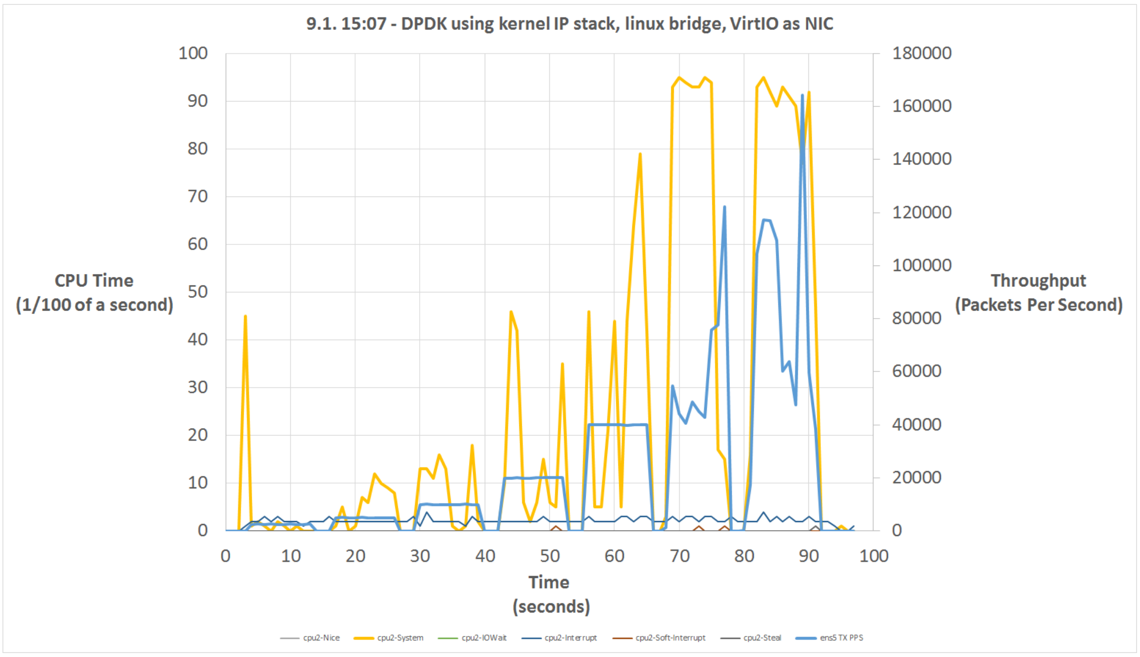

Figure B2: Graph showing system CPU time of a CPU core running DPDK pktgen application and packets per second, measured from a specific network interface. pktgen is configured to generate packets for 10 seconds at a time, then pause for 5 seconds and generate packets again with doubling the desired throughput after every pause. Graph shows how the system time (yellow) increases, as the throughput gets higher (light blue).

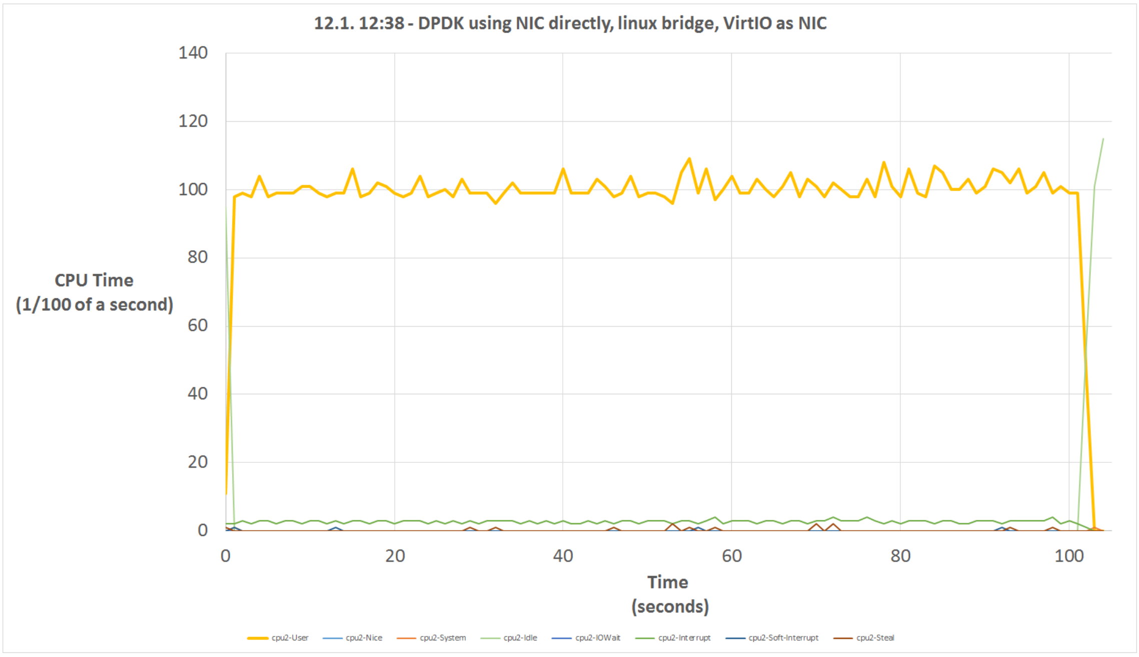

Figure B3: Graph showing user CPU time of a CPU core running DPDK pktgen application. As mainly user time is used for packet generation and sending, the graph varies very little.

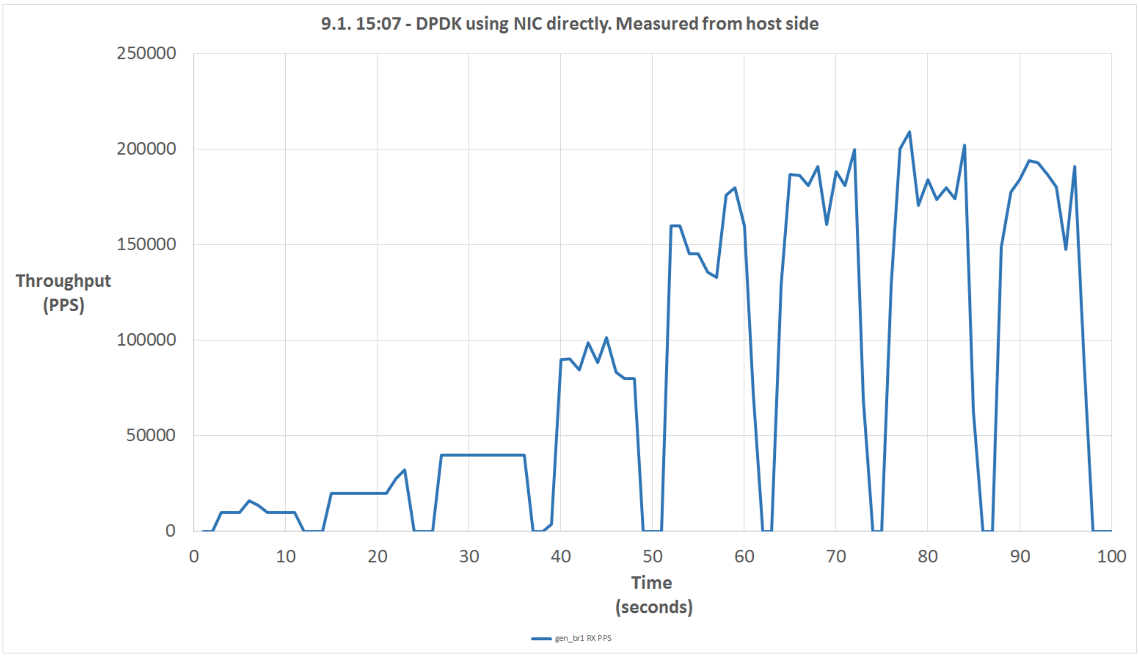

Figure B4: Graph showing packet throughput from host side. Note how the rate has increased when comparing to the DPDK Kernel IP-stack -case.