Bachelor Thesis Project

Container Hosts as Virtual Machines

- A performance study

Author: Andreas Aspernäs
Author: Mattias Nensén
Supervisor: Jacob Lindehoff
Semester: VT 2016
Subject: Computer Science

Abstract

Virtualization is a technique used to abstract the operating system from the hardware. The primary gains of virtualization is increased server consolidation, leading to greater hardware utilization and infrastructure manageability. Another technology that can be used to achieve similar goals is *containerization*. Containerization is an operating-system level virtualization technique which allows applications to run in partial isolation on the same hardware. Containerized applications share the same Linux kernel but run in packaged containers which includes just enough binaries and libraries for the application to function. In recent years it has become more common to see hardware virtualization beneath the container host operating systems. An upcoming technology to further this development is *VMware*'s *vSphere Integrated Containers* which aims to integrate management of *Linux Containers* with the *vSphere* (a hardware virtualization platform by *VMware*) management interface. With these technologies as background we set out to measure the impact of hardware virtualization on Linux Container performance by running a suite of macro-benchmarks on a *LAMP-application* stack. We perform the macro-benchmarks on three different operating systems (*CentOS*, *CoreOS* and *Photon OS*) in order to see if the choice of container host affects the performance. Our results show a decrease in performance when comparing a hardware virtualized container host to a container hosts running directly on the hardware. However, the impact on containerized application performance can vary depending on the actual application, the choice of operating system and even the type of operation performed. It is therefore important to consider these three items before implementing container hosts as virtual machines.

Keywords: Virtualization, Containerization, Cloud Computing, Container Host, Linux Container, Performance, Virtual Machine, Wordpress, Linux, Apache, MySQL, PHP, CoreOS, Photon OS, VMware, LAMP, Software Container, Docker, Hardware Virtualization, Full Virtualization

Preface

We would like to thank our supervisor *Jacob Lindehoff* for taking the time to review and assist our work. Our thanks goes to *Linnaeus University* for allowing us to use the laboratory in which our work was conducted. The laboratory has been an invaluable asset for developing our skills when it comes to virtualization and server infrastructure. We would also like to give a shout out to the software companies across the world which allows universities to use their software for educational purposes.

Contents

1 Introduction

In this chapter we describe the background of this thesis, the problem we intend to examine, as well as previous research done in the field. It will further explain the motivation behind our study, why we believe it holds a scientific value and also introduce the questions we seek to answer. The chapter ends with a description of the scope of the project, its intended target group and a brief outline of the report.

1.1 Background

Hardware virtualization is used for application isolation with one application for each server while still increasing application density on a physical host by means of server consolidation. Virtual server consolidation allows multiple servers to run on any single host which gives greater utilization of hardware capacity, reduce floor space, power consumption and computer cooling, which all help to reduce costs [1]. Similarly, *containerization* is a technique that in recent years has increased in popularity with the development of *Docker*[2], a container management software. Containerization creates lightweight operating system-level isolation on a single physical host or virtual machine, that allows for multiple applications to run on different operating systems while sharing the same kernel. Containerization removes the need to create a new virtual machine running its own full operating system for every new application which would produce greater storage and memory overhead than the lightweight shared-OS approach of containerization[3, Linux Kernel Containment]. However, because the applications share the same kernel, it does not achieve the hardware isolation for applications provided by hardware virtualization. In hardware virtualization each virtual machine has its own set of virtualized hardware and the isolation between virtual machines makes them agnostic to the hardware utilization of other virtual machines and thus increasing operational security[4]. A solution to this problem is to combine hardware virtualization and containerization to run container hosts as virtual machines. The result is hardware isolated virtual machines hosting lightweight containers and is one of the common implementation for *Platform as a Service (PaaS)* vendors[5]. Hardware virtualized containerization is also supported by companies like *VMware* that specialises in virtualization and are developing their own container host operating system called *Photon OS* for the purpose of running virtualized containers[6].

1.2 Previous research

In 2014 Mathijs Jeroen Scheepers did a study comparing hardware virtualization to container-based virtualization by macro-benchmarking the performance of a *LAMP-application* stack. The setup was two virtual machines running on *XenServer* and two *Linux Containers* (LXC)[7] running on *CoreOS* with *Docker* as the container engine. *Ubuntu Server* was used as operating system for both virtual machines and containers. One virtual machine and one container ran as an *Apache* web server with *WordPress* and the other virtual machine and container ran as a *MySQL* database server. On each of the two setups two things were benchmarked: The first benchmark focuses on the application performance when the web server is utilized by an increasing number of clients over a time of 800 seconds. The benchmark showed that *LXC* setup could process about four times the number

of requests totally than that of the *XenServer* setup. Jeroen Scheepers states that the difference could be explained by how CPU isolation is handled. The *Xen Project* hypervisor used by *XenServer* isolates per CPU core compared to the *cgroup priority* based isolation of *LXC* containers.

The second benchmark was comprised of two *SQL* tests, the purpose of which was to measure the performance of inter-virtual machine communication compared to inter-container communication by having the web server perform *SQL* queries to the database server. The first test was to measure the time it takes to complete one *SQL-SELECT* query. Results show that *XenServer* was slightly slower than *CoreOS* which means *LXC* containers introduce less overhead than the virtual machines running on the *Xen* hypervisor. The second test is a *PHP* script that inserts randomly generated data into the database with 10 000 *SQL-INSERT* queries. Results show that the applications running on *XenServer* completes this in 16 seconds compared to 355 seconds on the *CoreOS/LXC* setup. Mathijs Jeroen Scheepers explains that the massive performance drop on *CoreOS/LXC* could be explained by the lack of hardware isolation between the *LXC* containers[4].

Another research paper is the IBM Research Report *An Updated Performance Comparison of Virtual Machines and Linux Containers* by Wes Felter, Alexandre Ferreira, Ram Rajamony and Juan Rubio. In the paper they attempt to contrast the performance between virtual machine deployments and *Linux Containers*. The method used was to use a suite of different workloads to put strain on the hardware resources and compare how containerized performance stand in comparison to hardware virtualized performance. The results show that *Linux Containers* perform better than virtual machines based on a number of metrics, such as IOPS, CPU and memory utilization because of the extra abstraction layer added when virtualizing the hardware. In their *Conclusions and Future Work* section they question the deployment of containers inside virtual machines since it imposes the extra overhead while not, in their view, adding any benefits[8].

1.3 Problem formulation

There are advantages to using both virtualization and containerization; hardware virtualization has the advantage of hardware isolation which creates greater security and isolates applications from each other on a hardware level[9]. Containers are on the other hand very lightweight and can be created and removed very fast[10]. With coming technologies like *vSphere Integrated Containers* we believe nested use of container hosts like *CoreOS* and *Photon OS* running as virtual machines could become more common.

However, questions remain how this can affect the performance of applications running inside of the containers. No research which makes a macro-benchmark performance analysis comparing non-virtualized container hosts to virtualized container hosts, while also comparing possible container hosts operating systems, was found when searching on *Google Scholar* with the keywords *hardware virtualization* and *container host*. We want to study the implications of running container hosts as virtual machines, specifically the impact it could have on the performance of applications inside of the containers.

1.4 Motivation

We believe combining both hardware virtualization and containerization could potentially allow better utilization of the hardware while still be able to provide hardware isolation between applications or groups of applications. There is also value in being able to consolidate all servers unto one infrastructure and today's containerization clusters only support Linux. In a hardware virtualization infrastructure there's an option to have Windows servers running beside container hosts. However, there remains a question on how the performance of applications could be affected when combing these two technologies, and using them on different operating systems. These are questions we deem important for any IT-administrators looking to implement these kinds of solutions as a decrease in performance could hurt the benefits of using containers inside of virtual machines. Even if virtual container hosts is a common implementation among PaaS vendors[5] and and technologies like *vSphere Integrated Containers* are being developed, IBM claims in a report that there would be no benefit in running containers inside virtual machines[8] and the article *Why Containers Instead of Hypervisors?* by Steven J Vaughan-Nichols[11] argues that containers are better then virtual machines because of greater application density when using containers, particular in application development. As we found no research into the subject and find insufficient consensus to rule out the implementation entirely, we believe the implications of using virtual container hosts needs to be further investigated.

1.5 Research Question

Based on work by Mathijs Jeroen Scheepers[4] mentioned in *Previous Research*, server setups using hardware virtualization is expected to bring additional overhead compared to letting software containers run on a non-virtualized implementation. The reason being that the hypervisor adds an additional abstraction layer. Making a macro-benchmark performance test comparing non-virtualized container hosts to virtualized container hosts by measuring the performance of the applications inside of the containers, the research questions we will attempt to answer are as follows:

RQ1	How will containerized application performance be affected by: a) virtualized and non-virtualized container hosts? b) different container host operating system?

1.6 Scope/Limitation

The scope of this thesis project is limited to answering the specific research questions mentioned in the previous *Research Question* chapter. Macro-benchmark performance tests will be conducted with the primary goal of comparing how well an *Apache* and *Wordpress*-application stack runs in containers on *CentOS*, *CoreOS* and *Photon OS*, in both in a bare metal and virtualized environment. *Photon OS* will however not run non-virtualized because it is designed for virtualization. When we ask the question *How will containerized application performance be affected by: b) different container host operating system?* we are only interested in seeing the differences. A deeper discussion about the reasons behind the differences is beyond the scope of this thesis project.

In our motivation we point out that using virtual container hosts is a common implementation among PaaS vendors as one argument why this is an interesting research, however we are not looking to answer any question to why they are doing to, but rather would could happen when one does.

1.7 Target group

The target group is administrators of virtualization infrastructures and IT-technicians which are interested in virtualization concepts and/or may be in the process of or are considering to move from a non-virtualized containerization approach to a hardware virtualized approach of running container hosts. The information we wish to provide is the difference in performance of the two and the performance difference between individual container hosts.

1.8 Outline

This chapter outlines the different sections of the project report. It will start with the section *Technical Background* which aims to give the necessary background information about virtualization and technologies used to answer the research questions outlined in the section *Research Questions*. Next is the *Method* section which explains the method used to answer the research questions. Following the *Method* section is the *Results* section which contains the results of our experiments as described in the *Method* section, and the *Analysis* section which evaluates the results. The report ends with *Discussion* which discusses our findings, *Conclusion* where we present our conclusions and *Future research* present possible areas outside the limit of this thesis project that could be worthy of future research.

2 Technical background

This chapter intends to give a description of the technologies used in this thesis project. It also aims to give a theoretical background for the subjects which are addressed in this report. The purpose is to give the reader a basic understanding of the technologies and concepts to create a baseline of discussion.

2.1 Cloud Computing

With evolving network technology and the expansion of the Internet it is now possible to centralize computing resources, a concept called *cloud computing*. The *National Institute of Standards and Technology* (*NIST*) has developed a document defining cloud computing. They define cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources" and describe its essential characteristics. The primary characteristics that NIST describes are the ability for a consumer to provision computer resources with the service provider, the ability to reach resources through network access, the ability to pool resources using a multi-tenant model, scalability and monitoring of resource usage. Cloud computing can be available through several different service models. The *Software-as-a-Service* service model is to deliver application access to users without giving them insight to the underlying platform on which the application is running. Another service model is *Platform-as-a-Service* which lets consumers deploy their own services on the centralized platform but with no insight into the underlying infrastructure on which the services are running. The last service model is *Infrastructure-as-a-Service* which lets consumers set up their own virtual data center with control over operating systems and service deployment. The consumer has no control over the underlying cloud infrastructure. [12] Besides the need for the networking technology enabling access to these resources there has to be something enabling the isolation and management of the different resources running in the cloud. One such technology which is widely used is *virtualization*.

2.2 Virtualization

The virtualization concept is believed to originate in the 1960s and early 1970s. The pioneers were International Business Machines Corporation, IBM, which spent considerable effort trying to develop efficient time-sharing technology for their mainframes. Time-sharing enables the computing power of a mainframe to be divided into shares which could be distributed to different groups of users[13]. Skipping ahead to the late 1990s, the technology company *VMware* developed the first virtualization products which could virtualize the x86 architecture[14, The Challenges of x86 Hardware Virtualization]. With x86 virtualization the hardware and operating system was split in two by an abstraction layer. The abstraction layer allows the operating systems and applications running on the physical machine to become hardware agnostic and with that comes increased agility and business continuity. Services no longer needs to be taken down for hardware maintenance or backups because the applications can easily be migrated to other physical hosts or copied. In a white paper from year 2007 *VMware* stated it had customers with production servers which had been running for over three years without downtime[14, Overview of x86 Virtualization].

2.2.1 Hardware Virtualization

Hardware virtualization is a technology which creates an abstraction layer between the hardware and the operating system. It is done by software called *hypervisors*. Hypervisors come in different forms but there are two primary types; Type-1 hypervisors and Type-2 hypervisors. Type-1 hypervisors are also referred to as bare-metal hypervisors because they run directly on the hardware. Examples of Type-1 hypervisors are *Microsoft Hyper-V*, *VMware ESXi* and *Citrix XenServer*. Type-2 hypervisors run on a host operating system and is therefore come with increased overhead for the virtual machines. Examples of Type-2 hypervisors are *VMware Workstation* and *Oracle VM VirtualBox*[15].

The primary functionality of hardware virtualization is to make multiple operating systems and applications runnable in parallel on the same hardware by creating virtual instances of the hardware for each virtual machine and thereby make cost savings by increasing hardware utilization[4]. The earliest server-side product which managed to do this was *VMware ESX*. It used a custom-built kernel called VMkernel which was designed to run and manage virtual machine workloads. The VMkernel itself runs a Virtual Machine Monitor, VMM, for each virtual machine in the system. The VMM is responsible for implementing the virtual hardware and to execute the virtual machine.

In order to have multiple virtual machines running on the same host, they have to be isolated from each other and the actual actual hardware. To exemplify why this is important one can imagine a virtual machine sending a privileged instruction to turn itself off. In order for this instruction to only turn the virtual machine off and not the entire system, the call must be interpreted in the correct way. For this purpose a technique called *binary translation* is used. The technique traps privileged instructions coming from virtual machines and translates them into what the instruction means in the context of the source being a virtual machine. Figure 2.1 describes how *full virtualization with binary translation* works in relation to the instruction execution rings of the x86 architecture[14, Technique 1 – Full Virtualization using Binary Translation].

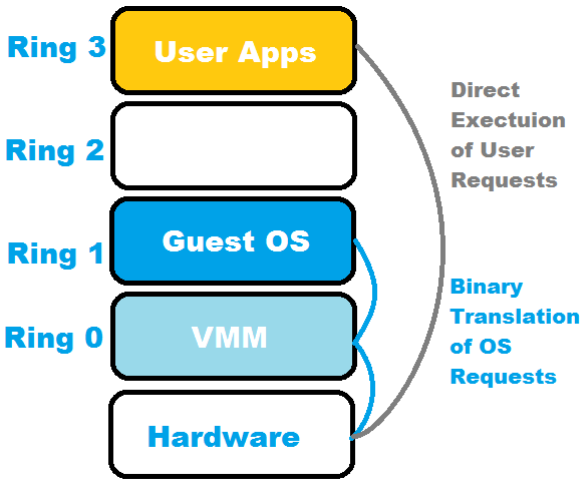

Figure 2.1: Full Virtualization with Binary Translation

Another early approach to hardware virtualization is *paravirtualization*. Using paravirtualization requires the guest operating system to be modified with virtualization management applications and device drivers. The advantage of this is to remove the need for binary translation by making the guest operating system aware that it is being virtualized. With this extra self awareness the guest operating system replaces the non virtualizable privileged instructions with *hypercalls*. Hypercalls are special instructions which are sent to the hypervisor. The hypervisor then forwards the privileged instructions to the hardware. Paravirtualization is supposed to decrease the virtualization overhead by removing the need for binary translation but *VMware* claims performance advantage varies greatly depending on the server workload and presses the fact that paravirtualization causes greater complexity when it comes to management and support because of the need to modify the guest operating system kernels. Figure 2.2 describes how paravirtualization works in relation to the instruction execution rings of the x86 architecture[14, Technique 2 - OS Assisted Virtualization or Paravirtualization].

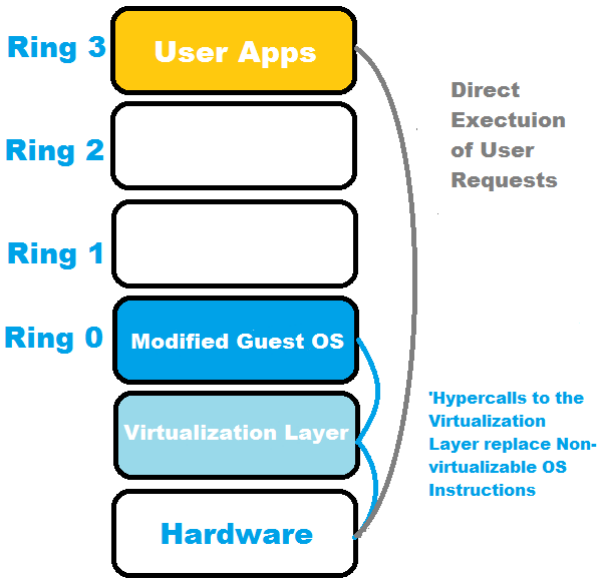

Figure 2.2: Paravirtualization with hypercalls

In the recent decade a third approach to hardware virtualization has been used which is hardware-assisted virtualization. It first appeared in 2006 when Intel Corporation and Advanced Micro Devices released their respective support features for hardware virtualization, Intel-VT and AMD-V. The new additions added a new CPU execution mode feature which meant the VMM could be run below ring 0, in a new mode called *root* (sometimes referred to as *ring -1*). Privileged instruction calls are automatically trapped to the hypervisor and thus removing the need for binary translation and it also removes the need to run modified operating systems using the paravirtualization approach. Figure 2.3 describes how hardware-assisted virtualization works in relation to the updated instruction execution rings of the x86 architecture[14, Technique 3 - Hardware Assisted Virtualization].

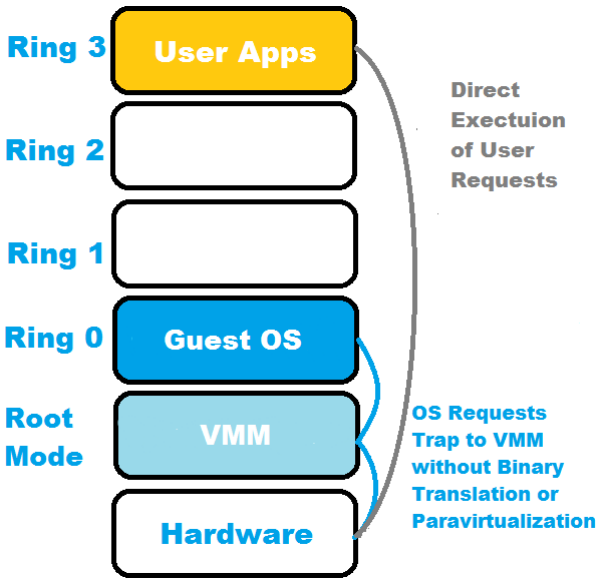

Figure 2.3: Hardware-assisted Virtualization

2.2.2 Containerization

Containerization is a operating system-level virtualization used to provide isolation and management of resources, primarily in Linux environments. The name containerization is derived from the way shipping containers is standardized and in the context of applications it refers to an agile way of packaging applications in an isolated execution environment. The isolation is created by three main components; *chroot*, *cgroups* and *kernel namespaces*. *chroot* is a command in Linux which lets a process change root directory to create container specific file systems. *cgroups* is the kernel subsystem by which the processes can be assigned resource quotas. The *kernel namespaces* enables every container to receive its own network configuration and inter-process communication, *IPC*, namespaces[5, IV. Container Overview]. Figure 2.4 shows an overview of the containerization architechture.

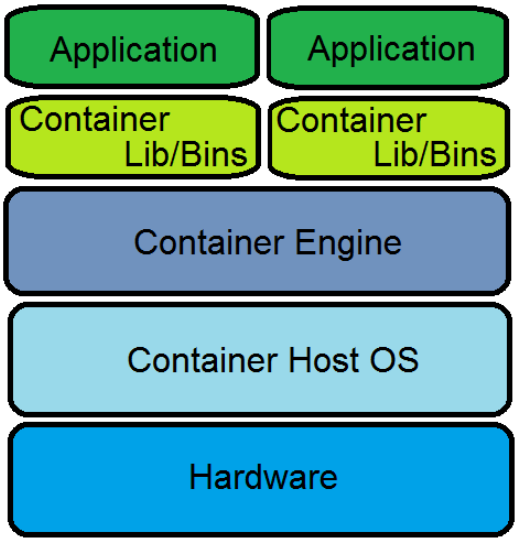

Figure 2.4: Containerization

Examples of *Container Hosts OS* as seen in figure 2.4 are *CoreOS*, *Photon OS* that are specificly designed for this purpose but most major Linux distributions have support to run a *Container Engine*. A *Container Engine*, for example *Docker* or *rkt*, is the software which runs the containers.

Containerization allows creation of multiple user space instances which are isolated from each other, and it is these segmented instances which are referred to as containers. In Linux where multiple Linux distributions uses the same kernel this allows each container to have a separate distribution from the container host OS. Applications running inside them can have its own libraries and containers can be tailored to fit that particular application. The parts of the container host operating system that are shared by the containers are *read-only* for the containers while they have their own share in which they can write. The sharing of kernel resources makes the containers much more lightweight than its virtual machine counterpart. Inside the containers is just enough to run each application, keeping its size to a minimum, drivers are kept inside the shared OS as a part of the kernel. Containers therefore allow multiple applications to run isolated from each other a single shared OS. To run multiple isolated applications on the same host in traditional server virtualization, one virtual machine (VM) would have to be created for each application[16].

2.3 VMware vSphere

VMware vSphere is a proprietary server virtualization platform from *VMware*. The products core is the *vSphere ESXi hypervisor*. *vSphere ESXi* is a Type-1 hypervisor which means it is installed directly onto the hardware and works like a miniature OS. It uses a VMkernel at its core with a installation footprint of 150 MB which means it has minimal effects on the storage resources available and its small size also decrease likelihood of security breaches because of the minimal attack surface[17].

The other components of vSphere is *vCenter Server*, *vSphere Client* and *vSphere Web Client*; these are used to manage one or more hosts. *ESXi* itself has only a slimmed down terminal interface for only basic configuration such as setting hostname and doing network configuration. When managing a single host a administrator logs in remotely using the *vSphere Client* which runs on the local computer. When hosts are joined together in a cluster, this cluster is managed by the *vCenter Server* which controls all the individual hosts. The *vCenter Server* is can be run outside or as part of the cluster. When administrating a cluster the administrators connect to *vCenter Server* using either the *vSphere Client* or the *vSphere Web Client*, the later which offers additional features and management tools.

VMware vSphere has a *High Availability* feature which provides failover protection to the virtual machines. The feature can monitor both virtualization hosts and virtual machines and if there is a hardware failure or guest operating system failure the virtual machine is automatically started on another functional host. The failover can be done fairly fast depending on the operating system which needs to be booted[18]. *vSphere* also offers a feature called *Fault Tolerance* which creates a live *shadow instance* of a virtual machine which runs on a different virtualization host. The shadow instance is at any time ready to step forward and take over if the other virtual machine crashes or if the virtualization host goes down. The result is that the downtime of a *Fault Tolerance* enabled virtual machine is limited to the time it takes for the system to notice the failure and activate the shadow instance[19].

2.4 Photon OS

Photon OS is an open source minimal Linux container host in development by *VMware*. Being a container host operating system means it is built for the specific purpose of providing the common kernel for containers running on the host. To run containers *Photon OS* has support for container deployment with *Docker*, *rkt* and *Pivotal Garden* and while it is very minimalistic it comes with a yum-compatible package manager for life cycle management[6].

Photon OS is a part of another *VMware* project called *Project Bonneville* and its goal is to make the relationship between virtual machines and application containers a complementary one rather than competitive. The project is an extension of *VMware vSphere* and will enable *Docker* containers to be run as virtual machines and thus gaining the isolation advantages of hardware virtualization. It works by forking the kernel from a running *Photon OS* instance to create a new virtual container host for every new container that is deployed.[20]. By forking only the kernel and a few supporting resources necessary to run containers allows much faster deployment then that of regular virtual machines.

2.5 CoreOS

CoreOS is a minimal operating system which is designed to run in clusters to host Linux containers. In the core of *CoreOS* is *Docker*, the container engine used to run the containers. *CoreOS* does not come with a package manager for lifecycle management, instead, administrators of a *CoreOS* cluster must run the tools within containers[21]. *CoreOS* utilizes *fleet*, a software that makes an administrator able to treat a large group of *CoreOS* machines as a single system with a shared *init*-system. Using *fleet*, the cluster also is able to maintain high availability because if a cluster host fails, the containers running on that host is automatically started on a different host in the cluster[22].

2.6 CentOS

CentOS is a Linux distribution which is a community-driven derivative to the *Red Hat Enterprise Linux* (*RHEL*) operating system. In contrast to *RHEL*, *CentOS* is completely free and its goal is to provide a platform for the open source communities. The operating system has been in development since 2004 and the goal is to be functionally compatible with RHEL[23].

2.7 Docker

Docker is an open-source application that enables deployment of applications inside of software containers. All of the applications dependencies are included in the container, anything from code and run-time to system tools and system libraries, everything that is needed to run the application. *Docker* makes use of images to launch new containers and they are templates from which many containers can be created, each container is then an instance of a particular image. Images are created from layered file systems which enables sharing of common files, this in turn can help reduce disk storage and speed up uploading and downloading of images. One of the possible use-case scenarios for *Docker* is optimization of infrastructure. *Docker* creates lightweight containers that all share the same kernel so there is no need for additional guest operating systems to isolate applications. Containers create less overhead compared to virtual machines, reduces the time to spin up applications and take up less storage[24].

2.8 Apache

Apache HTTP Server is a web server which as of November 2015 hosts 37% of all web sites[25], making it the most popular web server in the world. *Apache* had its initial release in 1995 and is a free and open-source software; a part of the *Apache HTTP Server Project* which is a developer community to further develop and support *Apache*, all supervised by the *Apache* Software Foundation.[26]. *Apache* can be used on *Linux*, *OS X* and *Windows* among others.

2.9 Apache JMeter

Apache JMeter is a *Java* application which is used to benchmark web application performance. The application was first developed by the *Apache Software Foundation* and the project was headed by Stefano Mazzacchi. The *Apache Software Foundation* has since then developed extra functionalities and a graphical user interface which helps visualising both the construction of tests and results. *JMeter*'s only works at the protocol level and does not render images like a regular browser[27].

2.10 MySQL

MySQL is an open source database developed the company *MySQL AB*, headquartered in Cupertino, California, United States and Uppsala, Sweden. In 2008 the company was aquired by *Sun Microsystems, Inc* and is since then a joint venture. In 2008 the database software had been downloaded 100 million times with 50,000 downloads daily[28]. *MySQL* is a transactional database which means the data is exchanged within *transactions*. If a transaction is not completed in its entirety, the data exchange is rolled back, thus maintaining the coherency of the transaction in case of a potential system failure[29].

3 Method

This chapter contains a description of the method that will be used to answer the research questions put forward in this thesis project. Under *Method Description* a topology will be presented describing the different hardware and software components used in the experiments. A more detailed system documentation can be seen in appendix A.

3.1 Scientific approach

The scientific approach of this report is an inductive approach using an empirical method to collect quantitative data. In order to measure the performance of applications inside of containers a number of experiments will be conducted. The data gathered from these experiments will be time to complete service requests and number of completed service requests.

3.2 Method description

To answer our research questions three experiments was conducted: One *HTTP-GET* test, one *SQL-SELECT* test and one *SQL-INSERT* test. The *HTTP-GET* test was intended to measure the overhead of *CPU* virtualization. The web server put the page that was fetched in its cache and worked at delivering the page to as many parallel users as possible. In the *Apache* web server, a new process was created for each new user. The page was *10 000* bytes big and the Gigabit Ethernet network could support a total throughput of 1 000 000 000/8 = 125 MB per second, meaning that the network could support about 12 500 transactions per second not taking into consideration overhead introduced by Ethernet, TCP and HTTP. The overhead is usually around 2-9% according to forum posts on *StackOverflow*[30] which still leaves the network supporting more than 10 000 transactions. If the results show less transactions per second, the network was not overloaded. At maximum a total number of 100 users was working towards the website, this meant that if the web server response time was for example 100 milliseconds, no more than 1000 transactions could traverse the network at any given second. If a bottleneck occurred then it would likely be because the CPU did not have the compute capacity to handle that many requests.

The *SQL-SELECT* experiments measured the latency of which a small *SQL-SELECT* query could be performed between two application containers running on the same container host. When the database server received the query, it also had to read from the disk. Differences between virtualized and non-virtualized setups was intended to show the overhead of read operations and if there was a difference in inter-container communication latency. The third test, the *SQL-INSERT* test, measured the overhead of write operations.

The purpose of these experiments was to measure the performance of the applications running inside of containers deployed by *Docker*, and to compare non-virtualized container hosts to virtualized container hosts. Differences in containerized application performance between the container host operating systems themselves was also examined. The following description section is divided into: Topology for the experiment setups, a prestudy

benchmarking the physical hardware verifying their equal *I/O* performance, ending with a detailed description of the individual experiments.

3.2.1 Topology

The lab environment consisted of an IBM BladeCenter chassis containing four blade servers and two Cisco Ethernet switches. All four blades had the same physical configuration which can be seen in futher detail in appendix A.1. The experiments was executed in five different setups. Each of the blade servers had one or three setups, though no more then one active at a time.

The four setups were (as seen in figure 3.5):

1. CentOS running on the ESXi hypervisor
2. CoreOS running on the ESXi hypervisor
3. Photon OS running on the ESXi hypervisor
4. CentOS running directly on the hardware
5. CoreOS running directly on the hardware

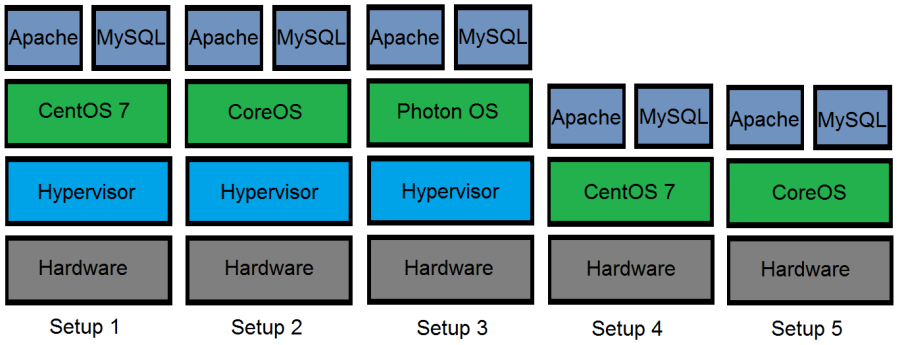

Figure 3.5: The figure shows the five different setups tested in this thesis project

Of the four blade servers, one hosted setup 1, 2 and 3, two others hosted setup 4 and 5, and the last one acted as the testing client that tested all the other setups using *Apache JMeter 2.13*. For details concerning each setup, see appendix A. Kernel versions, *Docker* versions and OS releases can be seen in appendix A.3. Each of the blade servers was connected to two Cisco Ethernet switches through two network interfaces, one for each switch. To separate management traffic from test traffic on the network, one interface was a dedicated administration interface used to handle management of hosts, virtual machines and software containers. The other interface was used to handle the traffic generated by *Apache JMeter*. Internet was also be available through the management interface to pull down *Docker* containers from the public repositories. For a full view of the topology, see appendix A.2. Each setup consisted of two applications running inside of its own software container; an *Apache* web server and a *MySQL* database. For a full description of how to deploy the applications using *Docker*, see appendix A.5.

3.2.2 Prestudy: Hardware I/O Benchmarks

The three test servers all shared the same hardware specifications but to be able to draw any valid conclusions in the comparison between setups on different physical servers they had to be equal in performance relevant to our benchmarks. In this case the read and write performance of the disks were important because the hardware used is 10 years old and mechanical disk performance is known to deteriorate over time. During the benchmarks, *CentOS* was installed on all three servers to ensure there are no unknown operating-system-specific variables affecting the performance. The Linux benchmarking tool *FIO* was used to perform the tests and the method was based on what is considered good practice for simulating web server and database workloads on I/O according to the guide *How to benchmark disk I/O* at the website *Binary Lane*[31]. According to the guide, it is generally best to measure the disk subsystem performance in the unit *IOPS* (Input/Output operations per second) in three different tests; *random reads/writes*, *random reads* and *random writes*, all with a block size of 4 kilobytes. *FIO* also allowed for running multiple threads which was done to simulate multiple visits to a website where multiple users might want to read and write to the disk at once. The results from the I/O benchmarks showed that all three servers offered similar performance as seen in figure 3.6 where Blade01-03 represents each of the three IBM Blade servers. The benchmark also tested the latency to the disks and the results are presented in figure 3.7. Although Blade01 has less then half the latency of Blade02 and Blade03, the longest latency that of Blade03 is ca 0.50 milliseconds and we believe it is not enough to make a noticeable effect on our performance experiments. The full details of the implementation of the benchmarks see appendix A.4.

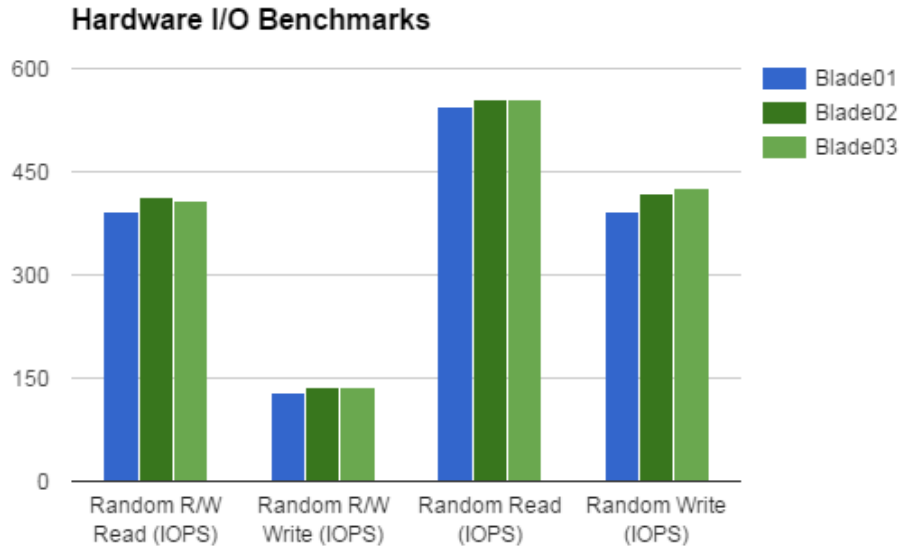

Figure 3.6: Read and Write IOPS (More is better)

3.2.3 HTTP Requests

The forth server was not hosting any setup and instead acted as an external load generator that sent traffic to each of the five test setups. The tests was done by letting *Apache JMeter* simulate an increasing number of active users, starting with an expected parallel

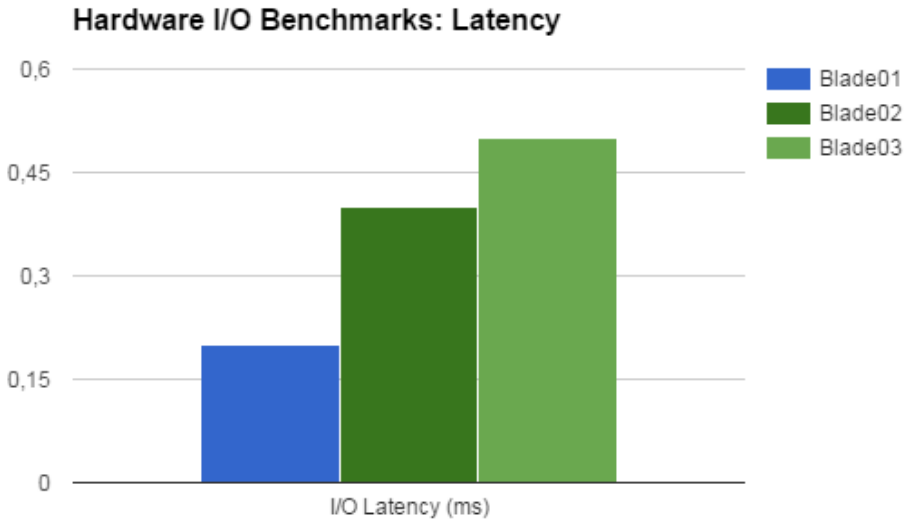

Figure 3.7: I/O Latency (Less is better)

user count of 1 which then increased linearly, as seen in figure 3.8, to 100 over a duration of 720 seconds. The tests continued for 80 seconds with 100 simultaneous users working against the website.

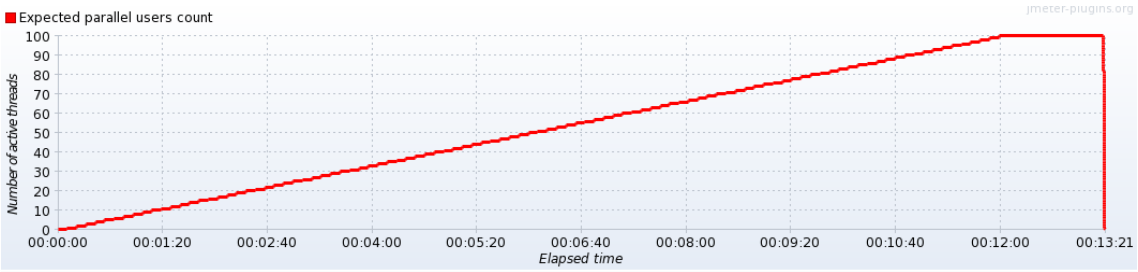

Figure 3.8: The figure shows the ramp up of simultaneous users during the tests

Apache JMeter, for each user, requested the starting page of the *Wordpress* installation on each of the four setups and the time it took for the application to service the requests was registered in a CSV file. The procedure was possible because *Apache JMeter* registered the time the simulated user sends the *HTTP GET Request* and registered the time when the user received the response. Every setup was tested five times each. The independent variable between experiments were different setups seen in figure 3.5. The dependent variable was the *LAMP-application* stack performance measured in *response time*, *number of completed transactions* and *transactions per second*.

The CSV file contained each successful *HTTP Request* on a separate row and from the file we extracted information using two different *Listeners* in *Apache JMeter*; *response Time over Time* and *transactions per second*. These listeners were data interpreters which presented the data in different ways and the *response time over time* showed how fast the client could receive a response from the server at the different load levels while the *transactions per second* listener showed how many successful transactions that were completed by the server. From the CSV files it was also possible to extract the total number of successful transactions completed during the tests by counting the total number of rows in the CSV files, subtracting 1 for each file to not count the headers. The difference in these three types of data allowed us to draw conclusions on how well the different setups

performed.

The data was then be analyzed by comparing the non-virtualized setups to the virtualized setups while also comparing the virtualized operating systems, *CentOS 7*, *CoreOS* and the technical preview of *Photon OS* to non-virtualized operating systems *CentOS 7*, *CoreOS*. The intent was to find and quantify the overhead caused by hardware virtualization but also to see how well *Photon OS* performed in relation to other operating systems running as container hosts.

3.2.4 SQL Queries

To measure how well applications in two different containers on the same host communicate with each other and read/write data, two experiments were conducted. The first test measured the time it took to complete a single *SQL-SELECT* query from the web server to the database server. The second test measured the time it took to complete 10 000 *SQL-INSERT* queries from the web server to the database server. To do this we created two *PHP* scripts to execute on each of the web servers. The *SQL*-query tests were performed on one setup at a time, with the independent variables being the setup used as seen in the figure 3.5. The dependent variables were the time it took to complete an *SQL-SELECT* query and the time it took to execute 10 000 *SQL-INSERT* queries.

To run the experiments two *PHP* scripts were used, one for the *SQL-SELECT* queries and one for the *SQL-INSERT* queries. Both scripts worked by first creating a *MySQL* connection to the database container by specifying the name of the container and the *username* and *password* for the *MySQL* server. A query was then made to set the active database to the *Wordpress* database. In the *SQL-SELECT* experiments a query was made to get two entries from the table*wp_users* through a *SQL-SELECT* query. The script recorded the current time and executed the query a single time after which it recorded the time again. The time it took to complete the query was then calculated and printed to the screen. The *SQL INSERT* experiments script worked in much the same way but instead of reading a table containing two posts a single time, a *SQL-INSERT* query was done 10 000 times, writing a simple word ”test” to a pre-created empty table. The script ended by calculating the time to complete the queries and the result was printed to the screen. For full detail of the scripts, see appendix A.6.

3.3 Method Discussion

The method was based on the previous research[4] by Mathijs Jeroen Scheepers in 2014. The method we used differed slightly from his because he didn't leave a detailed description for us to follow. We created our own *PHP*-scipts because he didn't present which scripts he had used. He wanted to measure the difference in performance when running applications inside of virtual machines compared to inside of containers using macro-benchmarks. Our goal was to measure the performance difference when running containers on virtual container hosts compared to non-virtualized container hosts. Even though our research questions differed slightly, we decided to use his method because the goal of his research was the same; to measure virtualized performance compared to non-virtualized performance.

When looking to answer our research questions we also considered at another method in

the report *An Updated Performance Comparison of Virtual Machines and Linux Containers*[8] by IBM. The objective was the same as in the the report by Mathijs Jeroen Scheepers, to strain the resources in the systems to compare and to quantify the performance to see differences. A difference between the two methods were that the IBM research was a series of benchmarks of more specific parts of a system while the research conducted by *Jeroen Scheepers* was a macro-benchmark. Being a macro-benchmark method means it measured the performance of a system by running an application which itself used the different parts of the system. A reason we did not chose the method by IBM was that it was was hard to understand and would therefore be hard to replicate. We chose to do a macro-benchmark because we wanted to measure the performance of the application stack as a whole and not just individual components.

The *VMware ESXi* hypervisor was chosen as it was closely linked to *Photon OS*, which is designed to run within the context of *vSphere Integrated Containers* and is tuned for *vSphere*. *CentOS* was chosen as a representative heavyweight enterprise operating system to use as a container host. *CoreOS* was chosen as a representative lightweight container host operating system because it is specifically designed to act as a container host operating system and is commonly used and referred to in containerization articles.

There are ways our method could have been done differently though. In a real-world scenario the system would most likely not have ran at 100% capacity for any extensive period of time. To simulate this, the tests could have been set to stop at an average number of user connections for a web site, possibly utilizing 70% resources of the web server. However, this was not chosen because the overhead was thought of to be more visible at high resource utilization.

3.4 Reliability and Validity

To improve reliability of the results each test was conducted five times. The difference between containerized application performance on non-virtualized and virtualized container hosts should have been similar on all systems using the same hypervisor and testing method regardless of the hardware used. The prestudy with measurements of I/O performance was conducted to increase the reliability of the results since I/O could have been a bottleneck.

We weren't able to run the same version of *Docker* on all the container hosts. The latest version on *Photon OS* was *1.8.1*, as seen in appendix A.3, and we attempted to run versions on other hosts close to the version on *Photon OS*. The reliability threat imposed by the different versions could however only affect the comparison between different operating systems because both the virtualized and the non-virtualized setup of the same container host used the same Docker version.

One thing that might have affected the results was that the hypervisor itself consumed about 0-0,5% of the CPU in the virtualized setups because it had services running listening to administrator input. Since our tests very quickly utilized all of the computing resources there may have been an internal validity threat because the CPU utilization of the hypervisor could alter the performance. The hypervisor also consumed some memory but our tests did not strain the memory resources of the systems.

4 Results and Analysis

In this section the results of our experiments is presented and analysed. Information will be given for each graph and table. To increase readability of the results and the analysis they are combined into one section. The graphs illustrate the results while the surrounding text contains analysis of the results.

4.1 HTTP Requests

The results seen in figure 4.9 showed a decrease in performance in virtualized environments compared to the non-virtualized environments. The non-virtualized *CentOS* machine performed 20% better on average than the virtualized *CentOS* and the non-virtualized *CoreOS* machine performed 27% better on average than the virtualized *CoreOS* when comparing the number of successful *HTTP Requests* completed during the tests. It should be noted however, that the first test run on the *CoreOS* virtual machine performed significantly worse than its own average, a performance reduction of 33%. The cause was unknown but we can speculate that an event like a scheduled job was performed on the hypervisor during the test. Discounting the first test run on the virtualized *CoreOS* machine, the performance of the virtualized *CoreOS* machine compared to the non-virtualized *CoreOS* machine was 18% lower . By those numbers, the average decrease in performance when hardware virtualizing a container host was 19%. The results in figure 4.9 showed that the *Photon OS* container host performed better than the other virtualized operating systems. Although still a technical preview, *VMware* explains that *Photon OS* is ”validated on and tuned for *VMware* product and provider platforms”[6, What is Photon OS] which could explain the slight performance increase compared to the other operating systems.

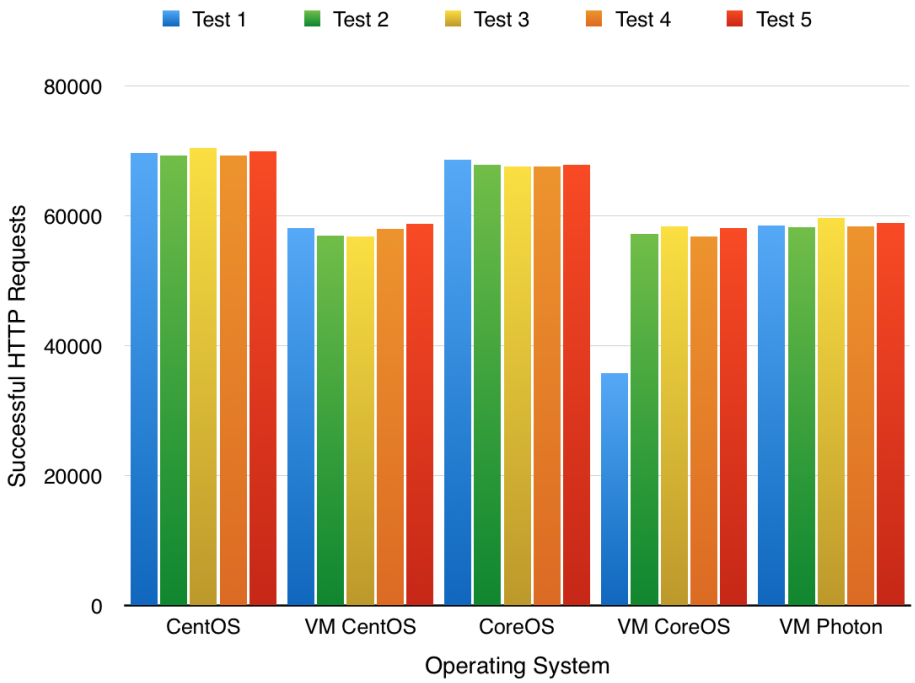

Figure 4.9: Successful HTTP Requests (More is better)

As seen in figure 4.10 the transactions per second increased more quickly in the non-

virtualized systems but since the number of transactions was initially low the difference was barely noticeable. The difference became more noticeable at higher number transactions per second. A possible explanation was that the overhead of every transaction, although small in comparison to the total transaction time, still stacked up. The overhead itself was in most part likely attributed to the hardware virtualization which introduced an extra abstraction layer between the application and the resources it used. A reason for the decreased performance could also have been the internal virtual switching in the hypervisor. Virtual machines running on the *VMware vSphere* hypervisor was connected to internal virtual switches which were connected to the physical network interface cards on the hypervisor. Traffic going to the virtual machines needed to go through one extra layer of switches before it got to the applications compared to the non-virtualized container hosts.

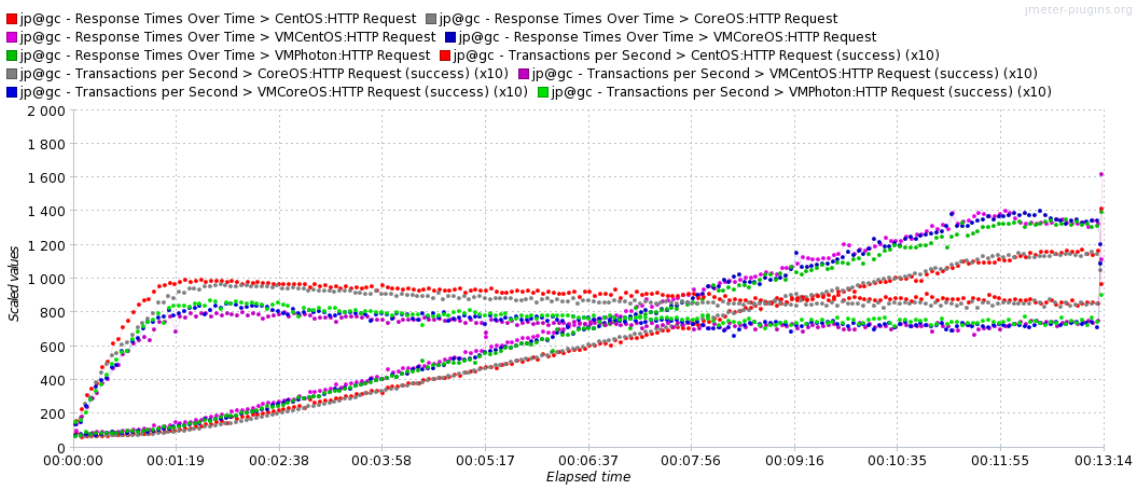

Figure 4.10: Respons Time over Time vs. Transactions Per Second

The full results of the *HTTP Request* tests can be seen in appendix B. The graph in figure 4.10 is only presented as representative results in relation to the actual full test results in appendix B. This is done because of the difficulties of combining all the *JMeter* results into a graph which can more easily be interpreted. In figure 4.10 the amount of transactions per second is scaled up by a factor of ten to make a combination of the graphs possible.

Examining all of the test results in appendix B, the ”spread” of the response times over time differed greatly between the different tests although the time granularity was the same, i.e. 100 milliseconds. When increasing the time granularity to produce averages with a time granularity of for example 4 seconds, the graphs were close to the same, something that could mean that there was a slight measurement problem. One hypothesis was that because the servers were fully utilized, some of the simultaneous requests were put on hold while waiting for CPU time to become available. Since the response time was measured on the *Apache JMeter* client, the CPU time needed to complete the request may have been the same but the order in which the responses were sent from the server might not have been in the order of which the simultaneous requests were sent from the *Apache JMeter* client.

4.2 SQL Queries

In the second part of the experiments *SQL* queries were done from the web server container to the database container to measure the difference in completion time. The average results for *SQL-SELECT* queries is presented in figure 4.11. The values are the average execution times to complete a single *SQL-SELECT* query where less is better. For detailed results see table 3.3 in appendix C.

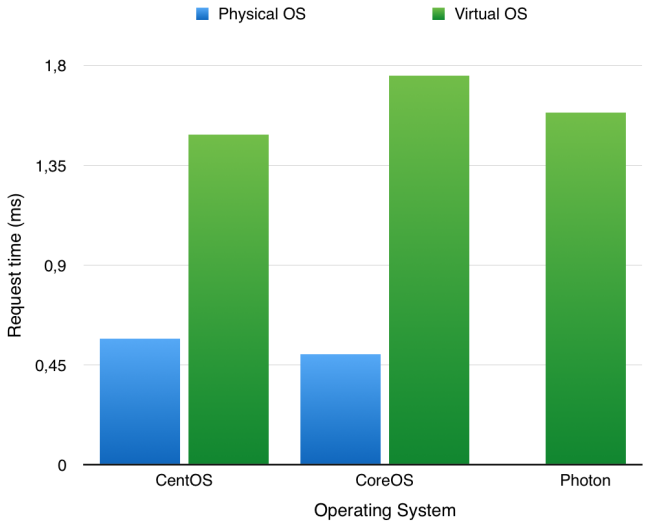

Figure 4.11: Execution times to complete 1 *SQL-SELECT* query shown as a graph. (Less is better)

The results from *SQL-SELECT* query experiments (figure 4.11) showed an increase in execution time when running the operating systems virtualized compared to non-virtualized. On average *CentOS* showed an increase of 162% and *CoreOS* showed an increase of 253%. *Photon OS* could not be compared due to it not being non-virtualized but performs on par with the other virtualized operating systems with a difference of 6.6% compared to *CentOS* and 9.6% difference compared to *CoreOS*. The difference between the operating systems were lower compared to the difference between virtualized or non-virtualized, with the greatest difference being between *CentOS* and *CoreOS* that of 15% virtualized.

An observation is the varying time it took to complete a single *SQL-SELECT* between the different test runs. One explanation could be that because we used physical hard drives the *disk read-and-write head* is not likely positioned at the location of the data being sought for by the *SQL-SELECT* query. Every query would then act as a random read with different completion time as a result.

The results from the *SQL-INSERT* queries where the time to complete 10 000 *SQL-INSERT* queries from the web server to the database server are presented in figure 4.12. The values presented are values in number of seconds and less is better. For detailed results see table 3.4 in appendix C.

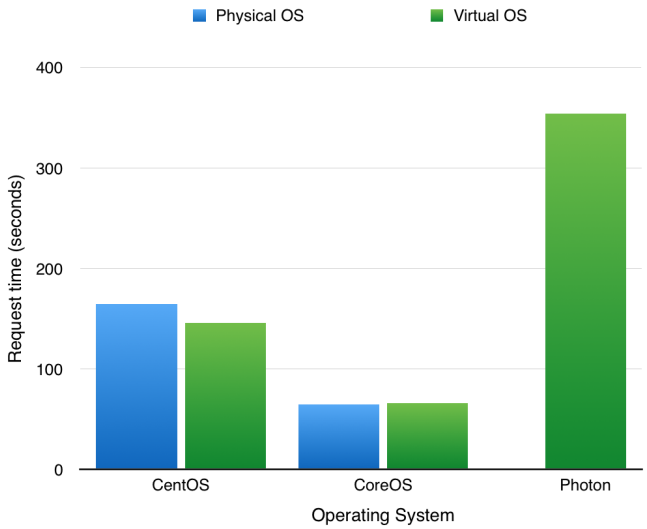

Figure 4.12: Execution times for 10 000 *SQL-INSERT* queries shown as a graph. (Less is better)

The results from the *SQL-INSERT* query experiments showed a relatively low difference between non-virtualized operating systems and virtualized operating systems, the greatest difference being for *CentOS* with a difference of 13%. The difference between operating systems were significantly larger by comparison; *CoreOS* (virtual) had the lowest execution time of 65,7 seconds on average, *CentOS* was 122% slower than *CoreOS* virtualized and *Photon OS* was 143% slower compared to *CentOS* virtualized.

When comparing the *SQL-SELECT* and *SQL-INSERT* results to each other we made two observations: The first was that in the *SQL-INSERT* tests the average execution time for the virtualized container hosts were lower or comparable to that of when running it non-virtualized. One hypothesis was that the total time it took for the data to be written to disk was substantially larger than the time it took for the operating system to, through the hypervisor, call for the data to be written. This could mean that for time-consuming operations like writing data the performance cost of hardware virtualization had less of an impact. The other observation is related to the difference between non-virtualized and virtualized *SQL-SELECT* query results. The results showed that there was a difference between non-virtualized an virtualized performance. The nature of the workload was here the opposite of the *SQL-INSERT* query experiment. Instead of a large quantity of write-operations a double read-operation was done. The results suggested that the smaller the physical work (for example I/O-operations) the more was the impact of hardware virtualization.

5 Discussion

The purpose of this study is to answer the following research question: How will containerized application performance be affected by 1) virtualized and non-virtualized container hosts, and 2) choice of container host operating systems. To answer the questions we adopt a method where application performance is measured with the independent variables first being the choice of whether or not to use hardware virtualization and second variable being the choice of container host operating system. Given the setup of an *LAMP*-application stack with a web server and a database server in two separate containers; how well does containerized applications perform when simulating *HTTP Requests* and *SQL* queries.

The results show that there is a performance decrease in the number of answerable *HTTP Requests* when running the container host as a virtual machine. There is approximately an 18-20% decrease of completed *HTTP Requests*. A decrease is expected because of the extra abstraction layer added when virtualizing an operating system, which has to communicate to hardware through the hypervisor. The network traffic must also be switched through the hypervisor from the physical port to the virtual switch which the virtualized container host is connected to. *Photon OS* performs slightly better than the other virtualized container host operating systems when measuring *HTTP Requests*, possibly because it is said be tuned to run efficiently in *vSphere*. It might also be tuned in ways which don't affect *LAMP-stack* performance. It will be interesting to see how *VMware* can integrate *Photon OS* more and add container management support to *vSphere* in the future when *vSphere Integrated Containers* is launched and *Photon OS* leaves the technical preview phase.

In the analysis we explain a difference in the results between the two types of *SQL* experiments. There is a notable difference in time to complete a single *SQL-SELECT* query between containers running on virtualized compared to non-virtualized operating systems, while only showing a minor difference between the different operating systems themselves. In the *SQL-INSERT* experiments the choice of operating system had a much greater impact on performance, whilst showing almost no difference between non-virtualized and virtualized operating systems. We believe the reason for this is that when the number of queries increases it becomes more important how different operating systems handle the workload. Also the nature of the workload seem to be a major reason for differences in performance, virtualized compared to non-virtualized, as described in the analysis of the *SQL* tests.

IBM claims as described in *Previous Research* that the extra abstraction layer of hardware virtualization only adds overhead without any additional benefits[8] when deploying containers inside of virtual machines. It is true that it comes with extra overhead, even when running container host operating systems like *Photon OS* which is specifically designed to run in a *vSphere* environment. However, hardware virtualization comes with several benefits such as hardware isolation and features such as *Fault Tolerance* in *vSphere* which helps reduce downtime for critical systems. Software containers are fast to boot though so the difference in downtime in case of a failure in a *CoreOS*-fleet cluster compared to *vSphere* cluster with *Fault Tolerance* may be negligible.

6 Conclusion

In this study we examined the implications for application performance when running containers inside of virtual machines. This was done by comparing non-virtual container hosts to virtual container hosts and measure the performance of two of their applications.

Our conclusion is that running virtual container hosts could affect the performance of their applications negatively, something to be expected considering the abstraction layer provided by the hypervisor. However, the impact of virtualizing a container hosts differs between the type of operation performed inside the applications running on it. We found that on larger amounts of consecutive *SQL-INSERT* queries the performance on virtual container hosts were even on par with physical ones. The opposite was found in the *SQL-SELECT* experiments. From the *HTTP Request* test we saw a 20% decrease in completed *HTTP-Requests* on average when running on a virtual machine instead of a physical installation. Non-virtualized *CoreOS* and *CentOS* had both faster response time over time as well as more transactions per second than virtualized *CoreOS* and *CentOS*. These experiments show that CPU intensive applications will see greater a performance decrease in a virtualized environment.

The choice of container host operating system is a complicated issue because they all offer different advantages. The results of the *HTTP-Request* experiments show that *Photon OS* performs slightly better in the virtualized setups than *CoreOS* and *CentOS* when it comes to the total number of completed *HTTP Requests*. *CoreOS* and *Photon OS* is also very small distribution which compared to *CentOS* requires less storage and memory resources to be installed and to be run. However, in the *SQL-INSERT* test *Photon OS* performed worse then all other tested operating systems, be it virtualized or non-virtualized. This shows that it is also important to consider the application before choosing an operating system for this kind of implementation.

The results is likely not fully generalizable. This is because our research found that the impact of hardware virtualization depends on the operation performed. When doing heavier operations like writing to disk the impact of hardware virtualization was negligible but when doing smaller operations like a small read operation the impact was large. Our conclusion based on those results is that the smaller the operation, the bigger the impact. To make the result more generalizable we would want to make more measurements with different amounts of read and writes. The overhead caused by hardware virtualization also becomes more visible at higher utilization levels in our experiments meaning that our results in the HTTP tests may only be general at those utilization levels.

Our method was based on previous research by Mathijs Jeroen Scheepers[4], but when analysing our data we found that our conclusions could be made stronger by improving or supplementing our experiments. In the *SQL* experiments there could be a comparison between doing a single query and multiple queries of the same type because time to execute can be significantly different between reading and writing data. Each of our experiments only included a single *SQL-SELECT* query and 10 000 *SQL-INSERT* queries. In retrospect we could have analyzed the methods of earlier research more critically. In the *HTTP-Request* experiments it might also have been better to test the systems at lower utilization levels to get a picture of performance at workload closer to what would be in a production environment. A test could be done to see at which number of simultaneous users simulated by *Apache JMeter* would push the system to about 70-90% CPU load

and then do a linear increase from one user to that specified number. In our current tests, the maximum amount of transactions per second are at *t=100s* (12% into the test), the number then decreases as the system becomes overloaded. Our reasoning for choosing method is that we wanted to push the utilization high to make the overhead caused by the hardware virtualization more visible. The overhead is on a operation-basis which means the overhead would not be as visible at low utilization. Now we get a picture of how much overhead exists from hardware virtualization in a fully utilized system.

Another thing that could improve our method is to also test using less hardware (CPU and memory) for the non-virtualized operating systems and then reduced the hardware for the virtual machines to the same amount of CPU and memory. In our tests the virtual machine running on the hypervisor was assigned all of the hardware on the virtualization host because the hypervisor itself only utilizes about 0-0,5% of the hardware resources according to our management view through the *vSphere Client*. It is however unknown if the virtual container host was actually competing for resources with the hypervisor during the high utilization phase of the tests in a way that affected the results. To analyze this more resources could be assigned to the hypervisor or less hardware resources installed on the physical blades hosting the non-virtualized installations of *CentOS* and *CoreOS*.

One possible scenario for this kind of implementation is to use virtual machines as application platforms consisting of multiple components where each component is a container. The application platform could be a web shop and the necessary components could be the web server to host site for the customers interact with. There might also be a database server to hold information about the products and an inventory system used by the employees. Instead of running each component in it's own virtual machine each component is run as a container inside of a single virtual machine. The components would be lightweight with inter-container communication being more effective compared to inter-virtual machine communication, while also receiving hardware isolation between different application platforms running on the same physical hardware. Having all components necessary to run the application platform inside the same virtual machine also makes migration between physical machines easier because the administrators does not have to move each component separately.

We believe these results to be relevant to the IT industry because even though companies like *VMware* and *Microsoft* are putting both time and money into developing technologies to better support virtual container hosts, we found no academic research on the concept of running containers in virtual machines. Virtual container hosts bring new and interesting possibilities to the data center administrators but questions about the implications of doing so, such as performance, needs to be addressed. As we believe virtual container hosts is going to be a major part of future data centers it is important to fill this knowledge gap.

6.1 Future research

It would be interesting if someone could do our tests with a further developed method. If future research takes into account the proposed improvements discussed in the *Conclusion* chapter the results may be different. In a future where *Photon OS* has left technical preview we would also like to see our tests done again to see if there are any improvements to performance.

Future research could also be to evaluate different container engines. *CoreOS* has its own container engine called *rkt* which could be compared to *Docker*. Why is there a need for two different container engines, is one better than the other and in which ways?

Another future research that could be conducted is a well structured operational comparison between different virtualization platforms. One could for example compare *CoreOS with fleet* to *VMware vSphere* on *provisioning*-, *high availability*-, *fault tolerance*- and *migration*-features to see which one suits which environment and for what reasons. It is especially interesting if the operational comparison is made when container management has been integrated in *vSphere*.

References

[1] D. Kumar. (2015) Server consolidation benefits – with real world examples. (Accessed on 07/27/2016). [Online]. Available: http://www.sysprobs.com/server-consolidation-benefits-with-real-world-examples

[2] Docker. (2016) What is docker? [Online]. Available: https://www.docker.com/what-docker

[3] S. Hogg. (2014) Software containers: Used more frequently than most realize | network world. (Accessed on 05/09/2016). [Online]. Available: http://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html

[4] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," *21st Twente Student Conference on IT*, pp. 1–7, 2014.

[5] R. Dua, A. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, pp. 610–614.

[6] VMware, Inc. (2015) Photon OS by VMware. [Online]. Available: https://vmware.github.io/photon/

[7] Canonical Ltd. What's LXC? [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[9] D. Kumar. (2014) Docker vs vms. (Accessed on 07/27/2016). [Online]. Available: http://devops.com/2014/11/24/docker-vs-vms/

[10] S. Seshachala. (2014, 11) Docker vs vms - devops.com. (Accessed on 05/20/2016). [Online]. Available: http://devops.com/2014/11/24/docker-vs-vms/

[11] S. J. Vaughan-Nichols. (2014, 08) Why Containers Instead of Hypervisors? (Accessed on 05/24/2016). [Online]. Available: http://blog.smartbear.com/web-monitoring/why-containers-instead-of-hypervisors/

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

[13] Oracle. 1.1.1. Brief History of Virtualization. [Online]. Available: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html

[14] VMware, Inc. (2007, 09) Understanding Full Virtualization, Paravirtualization and Hardware Assist. [Online]. Available: https://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[15] R. Vanover. (2009, 06) Type 1 and Type 2 Hypervisors Explained. (Accessed on 05/24/2016). [Online]. Available: https://virtualizationreview.com/blogs/everyday-virtualization/2009/06/type-1-and-type-2-hypervisors-explained.aspx

[16] P. Rubens. (2015) What are containers and why do you need them? | cio. (Accessed on 05/12/2016). [Online]. Available: http://www.cio.com/article/2924995/enterprise-software/what-are-containers-and-why-do-you-need-them.html

[17] VMware, Inc., "vSphere ESXi Hypervisor," 2016. [Online]. Available: http://www.vmware.com/se/products/vsphere/features/esxi-hypervisor.html

[18] (2016) Vmware vsphere with operations management: High availability | vmware sverige. (Accessed on 05/06/2016). [Online]. Available: http://www.vmware.com/se/products/vsphere/features/high-availability

[19] (2016) vsphere fault tolerance: Vmware | vmware sverige. (Accessed on 05/06/2016). [Online]. Available: http://www.vmware.com/se/products/vsphere/features/fault-tolerance.html

[20] B. Corrie. (2015, 06) Introducing Project Bonneville - Cloud-Native AppsCloud-Native Apps - VMware Blogs. (Accessed on 05/24/2016). [Online]. Available: http://blogs.vmware.com/cloudnative/introducing-project-bonneville/

[21] CoreOS. Using CoreOS. (Accessed on 05/24/2016). [Online]. Available: https://coreos.com/using-coreos/

[22] ——, "Using fleet with coreos," https://coreos.com/using-coreos/clustering/, (Accessed on 02/25/2016).

[23] (2016) About centos. (Accessed on 05/06/2016). [Online]. Available: https://www.centos.org/about/

[24] . Docker. (2016) What is docker? (Accessed on 05/20/2016). [Online]. Available: https://www.docker.com/what-docker

[25] Netcraft Ltd. (2016) November 2015 Web Server Survey. [Online]. Available: http://news.netcraft.com/archives/2015/11/16/november-2015-web-server-survey.html

[26] Apache Software Foundation. (2016) Foundation Project. [Online]. Available: http://www.apache.org/foundation/

[27] ——. (2016) Apache JMeter - User's Manual: Introduction. [Online]. Available: http://jmeter.apache.org/usermanual/intro.html

[28] M. AB, "Mysql ab :: Sun to acquire mysql," https://web.archive.org/web/20080117192218/http://www.mysql.com:80/news-and-events/sun-to-acquire-mysql.html, 01 2008, (Accessed on 05/03/2016).

[29] T. T. FAQ, "Transactional database," http://www.tech-faq.com/transactional-database.html, 10 2012, (Accessed on 05/03/2016).

[30] F. Penov. (2010) networking - what % of traffic is network overhead on top of http/s requests - stack overflow. (Accessed on 07/27/2016). [Online]. Available: http://stackoverflow.com/questions/3613989/what-of-traffic-is-network-overhead-on-top-of-http-s-requests

[31] How to benchmark disk I/O. [Online]. Available: https://www.binarylane.com.au/support/solutions/articles/1000055889-how-to-benchmark-disk-i-o

A Appendix A: System Documentation

Appendix A contains all the documentation describing the topology used in this thesis project.

A.1 Physical Configuration

The physical hardware is contained in an IBM BladeCenter. The physical machine is shown in figure 1.13.

Figure 1.13: The IBM BladeCenter used in this thesis project

The BladeCenter contains 14 blade servers. The first blade server seen in 1.13 is used to run *CoreOS* directly on the physical hardware. The second blade server runs the ESXi 6.0 hypervisor which virtualizes the virtual implementations of *CoreOS*, *CentOS* and *Photon OS*. The third blade server is used to run *CentOS* directly on the hardware. The forth blade server is the client which is used to run the JMeter tests towards the different implementations. The last ten blade servers are not used in this thesis project.

Each of the blades have the following hardware specifications:

- CPU 1: Intel Xeon 5320 1,86 GHz 4 cores
- CPU 2: Intel Xeon 5320 1,86 GHz 4 cores
- Memory: 2x 4 GB ECC DDR2 + 2x 2 GB ECC DDR2
- Storage: 2x 10k rpm eServer xSeries SAS disk at 73.4 GB

- 2x Gigabit Ethernet Network Interfaces

A.2 Network Topology

This section describes the network topology used in this thesis project. The core of the network is the *Dell PowerConnect 6248* Gigabit Ethernet switch, as seen in figure 1.14. It connects the BladeCenters two switch modules to the two administration clients. The administration clients are running *VMware Workstation* to host an edge router and a PXE-server. The edge router is used for Internet-connection while the PXE-server is used to PXE-boot the different operating systems onto the blade servers. The networks within the BladeCenter are the **production network** used to do the HTTP GET testing from the JMeter machine(blade04) to the container hosts and the **administration network** used to start containers do general administration tasks. The reason behind dividing the networks up is to ensure the HTTP GET testing is done within its own broadcast domain with as little disturbances as possible.

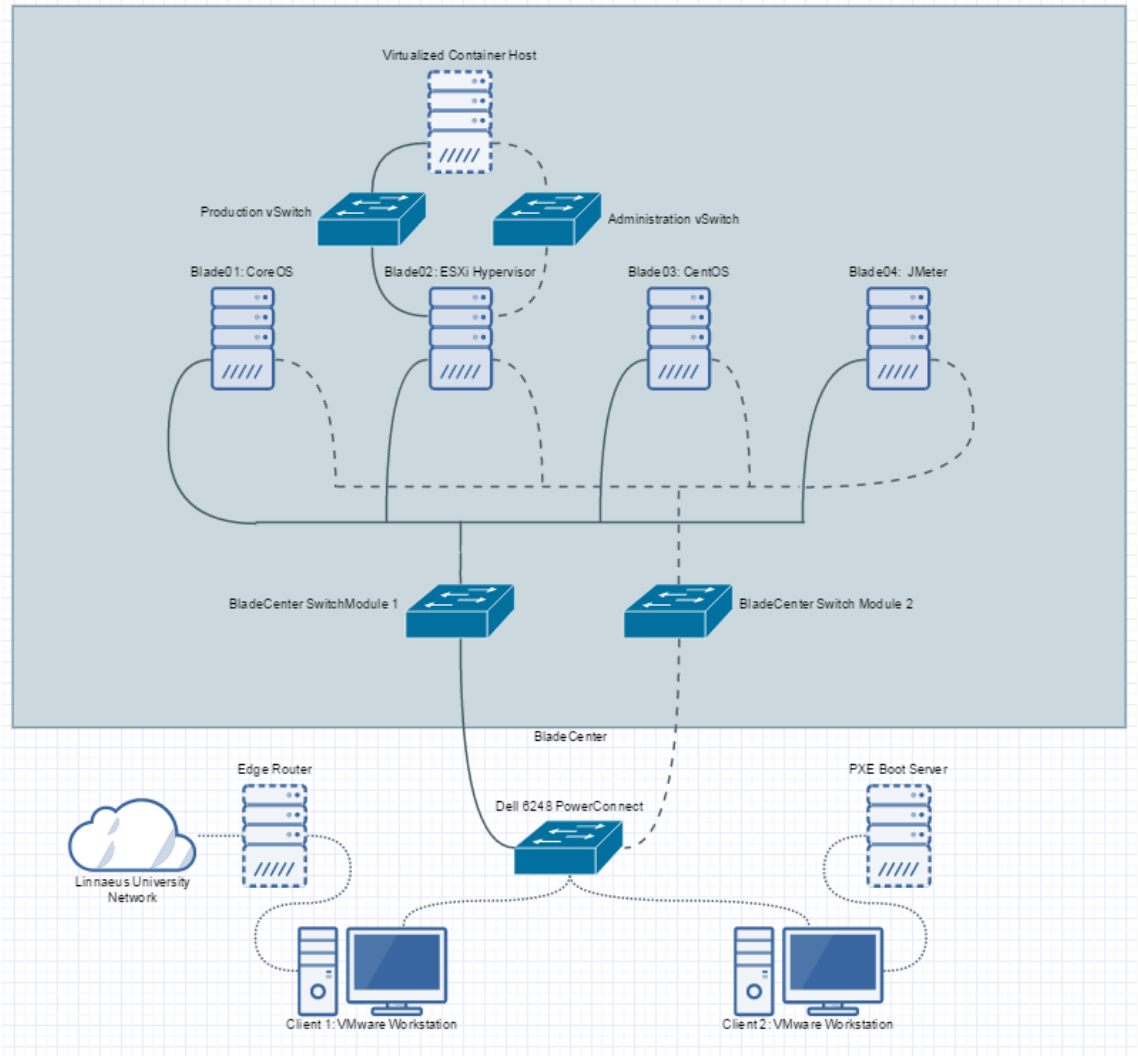

Figure 1.14: Network topology used in this thesis project

A.3 Versions

This chapter contains the versions and releases of the software and operating systems used in this thesis project. Table 1.1 show the kernel-, *Docker*- and operating system-versions used. For *Apache JMeter*, the version number is *2.13* and Jmeter Extras 1.31 is also installed for the extra functionality to merge the data from different tests.

Table 1.1: Kernel-, Docker- and OS-versions of the operating systems used in this thesis project

Type	CentOS	CoreOS	Photon OS
Kernel	3.10.0-327.10.1.el7.x86_64	4.3.7-coreos	4.0.9
Docker	1.8.2-el17.centos, build a01dc02/1.8.2	1.9.1, build 9894698	1.8.1, build d12ea79
OS Release	7.2.1511	899.15.0	1.0 Tech Preview 2

A.4 Prestudy: Hardware Benchmarks

In the beginning of the thesis project, hardware benchmarks were conducted to ensure that the disks of the blade servers would not have different levels of physical deterioration. The software used was *FIO*. In order to install the different components needed to run *Random Read*-, *Random Write*- and *Random R/W*-tests the following commands were used:

```
#Install
cd /root
yum install -y make gcc libaio-devel || ( apt-get update
&& apt-get install -y make gcc libaio-dev  </dev/null )

wget https://github.com/Crowd9/Benchmark/raw/master\
/fio-2.0.9.tar.gz ; tar xf fio*
cd fio*
make
```

To run the three tests the following commands were used:

```
#Random read/write
./fio --randrepeat=1 --ioengine=libaio --direct=1
--gtod_reduce=1 --name=test --filename=test --bs=4k
--iodepth=64 --size=4G --readwrite=randrw --rwmixread=75
#Random Read
./fio --randrepeat=1 --ioengine=libaio --direct=1
--gtod_reduce=1 --name=test --filename=test --bs=4k
--iodepth=64 --size=4G --readwrite=randread
#Random Write
./fio --randrepeat=1 --ioengine=libaio --direct=1
--gtod_reduce=1 --name=test --filename=test --bs=4k
--iodepth=64 --size=4G --readwrite=randwrite
```

Disk latencty was also measured. The software used was IOPing. The command to install IOPing was the following:

```
cd /root
yum install -y make gcc libaio-devel || ( apt-get update &&
apt-get install -y make gcc libaio-dev  </dev/null )
wget https://ioping.googlecode.com/files/ioping-0.6.tar.gz
; tar xf ioping*
cd ioping*
make
```

To run the test the following command was used:

```
#Test
./ioping -c 10 .
```

A.5 Docker Commands

The following script is used to deploy the *LAMP* stack which includes a *MySQL* container and an *Apache* container:

```
#!/bin/bash

# Enviroment variables
export ROOT_PASSWORD=password
export WORDPRESS_PASSWORD=password

# Run MySQL container
docker run --detach \
        --name mysql1 \
        --env MYSQL_ROOT_PASSWORD=$ROOT_PASSWORD \
        --env MYSQL_USER=wordpress \
        --env MYSQL_PASSWORD=$WORDPRESS_PASSWORD \
        --env MYSQL_DATABASE=wordpress \
        mysql

sleep 10
# Run Wordpress container
docker run --detach \
        --publish 80:80 \
        --name wordpress1 \
        --env WORDPRESS_DB_HOST=mysql:3306 \
        --link mysql1:mysql \
        --env WORDPRESS_DB_USER=wordpress \
        --env WORDPRESS_DB_PASSWORD=$WORDPRESS_PASSWORD \
wordpress
```

A.6 PHP Scripts

The two following *PHP* scripts describes how to establish and connect to a *MySQL* database server and measure the time to complete SELECT or INSERT queries.

A.6.1 SQL SELECT

The following *PHP* script is used to measure the time it takes to complete an *SQL-SELECT* query:

```
<?php
$servername = "mysql1";
$username = "wordpress";
$password = "password";
$dbname = "wordpress";

// Create connection
$conn = new mysqli($servername, $username, $password);

// Check connection
if($conn->connect_error) {
die("Connection failed: " . $conn->connect_error);
}
echo "Connected successfully\n";

$sql = "USE wordpress";
$conn->query($sql);

$sql = "SELECT * FROM wp_users";
$time = microtime(TRUE);
$conn->query($sql);
$time = microtime(TRUE) - $time;
echo "Time to complete 1 SQL SELECT: ";
echo $time;
echo "\n";
?>
```

A.6.2 SQL INSERT

The following *PHP* script is used to test the time it takes for the web server to complete 10 000 *SQL-INSERT* queries where the value *"test"* is inserted to a test table within the *Wordpress* database:

```
<?php
$servername = "mysql1";
$username = "wordpress";
$password = "password";
$dbname = "wordpress";

// Create connection
$conn = new mysqli($servername, $username, $password);

// Check connection
if($conn->connect_error) {
die("Connection failed: " . $conn->connect_error);
}
echo "Connected successfully\n";

$sql = "USE wordpress";
$conn->query($sql);

$sql = "INSERT INTO test_table(test) VALUES('test')";
$time = microtime(TRUE);
for($i = 0; $i < 10000; $i++) {
$conn->query($sql);
}
$time = microtime(TRUE) - $time;
echo "Time to complete 10 000 SQL INSERT: ";
echo $time;
echo "\n";
?>
```

B Appendix B: JMeter Results

This section contains all of the results from the tests conducted with JMeter. Table 2.2 shows the full results in table-format.

Run	CentOS	CoreOS	VM CentOS	VM CoreOS	VM Photon
1	69705	68664	58122	35783	58532
2	69341	67828	56898	57263	58263
3	70402	67620	56850	58398	59727
4	69335	67609	57967	56835	58339
5	69898	67817	58729	58116	58882
Average	69736	67908	57713	57653*	58749

Table 2.2: Number of successful HTTP Requests completed during the tests. More is better.

Table 2.2 shows the full results of the *HTTP Request*-tests. The first test on the virtualized *CoreOS* setup is excluded when calculating the average to show a more just comparison.

B.1 CentOS

This section contains all of the results from the JMeter tests towards the non-virtualized *CentOS 7* machine.

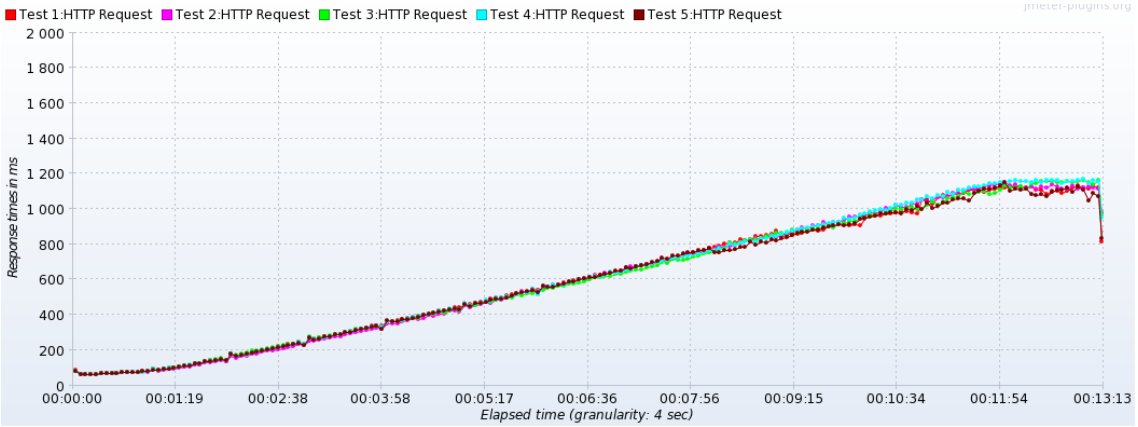

Figure 2.15: CentOS - Test 1 - 5 - Response Time over Time (Less is better)

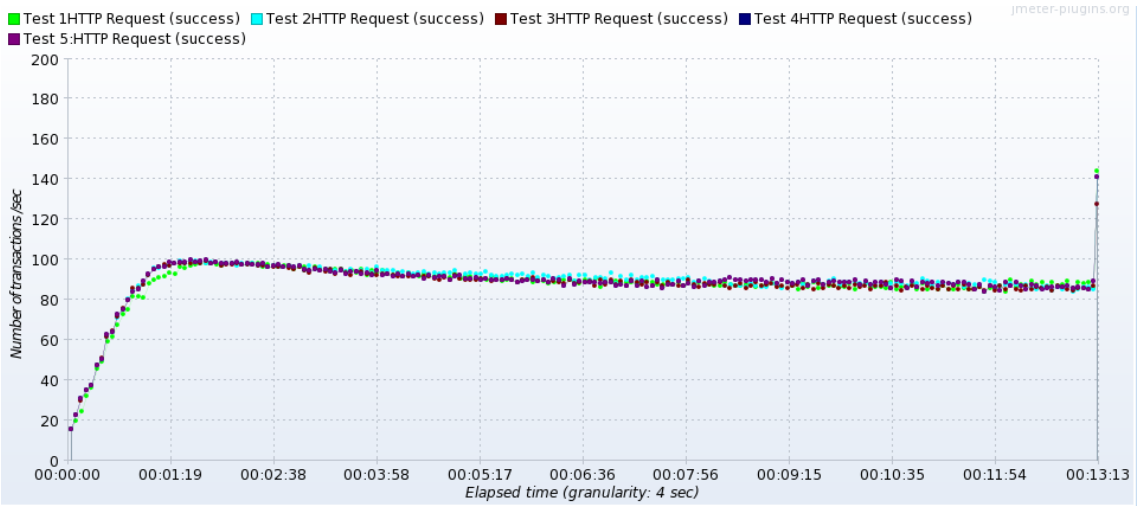

Figure 2.16: CentOS - Test 1 - 5 - Successfull Transactions Per Second (More is better)

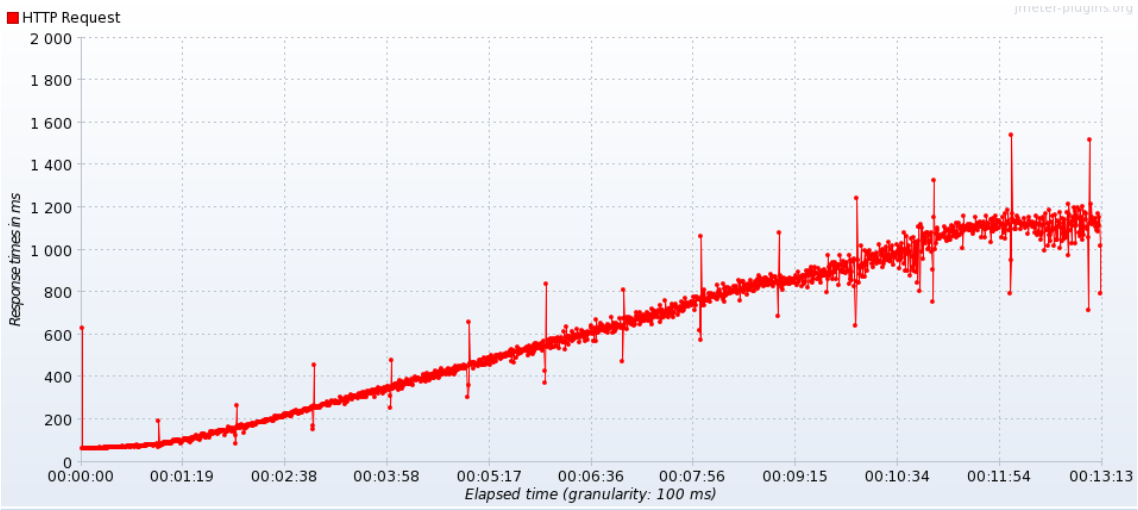

Figure 2.17: CentOS - Test 1 - Response Time over Time (Less is better)

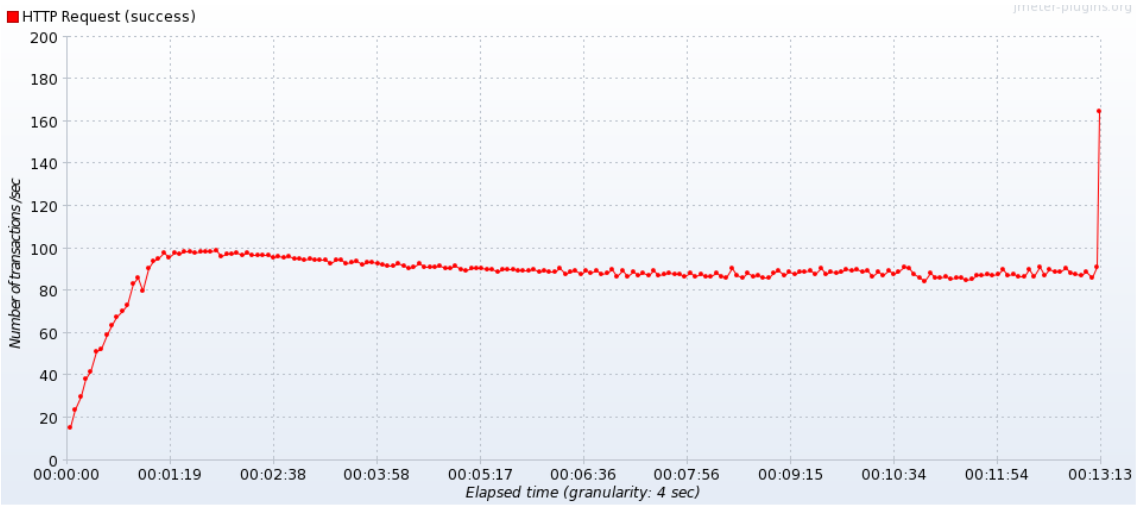

Figure 2.18: CentOS - Test 1 - Successfull Transactions Per Second (More is better)

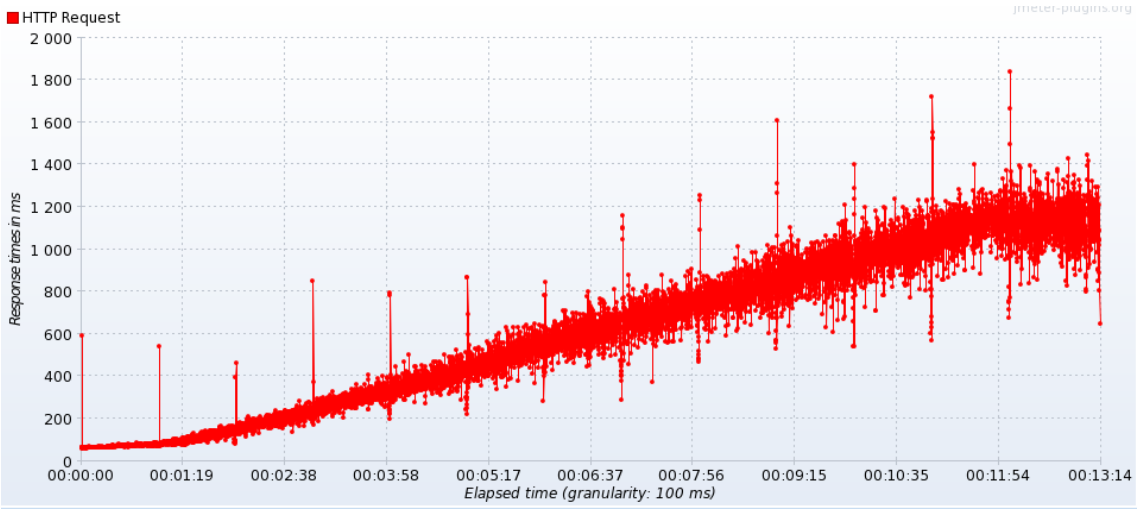

Figure 2.19: CentOS - Test 2 - Response Time over Time (Less is better)

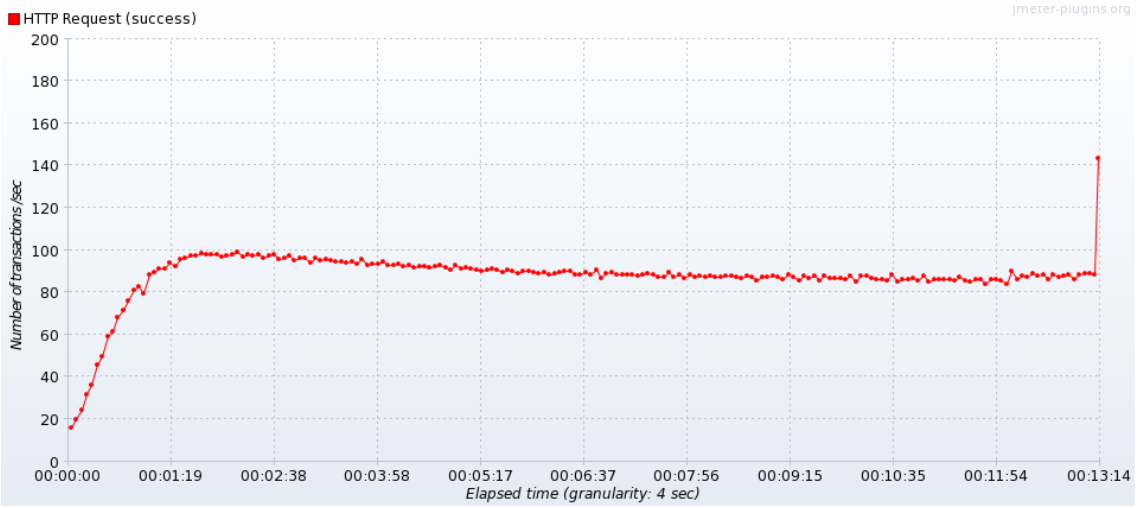

Figure 2.20: CentOS - Test 2 - Successfull Transactions Per Second (More is better)

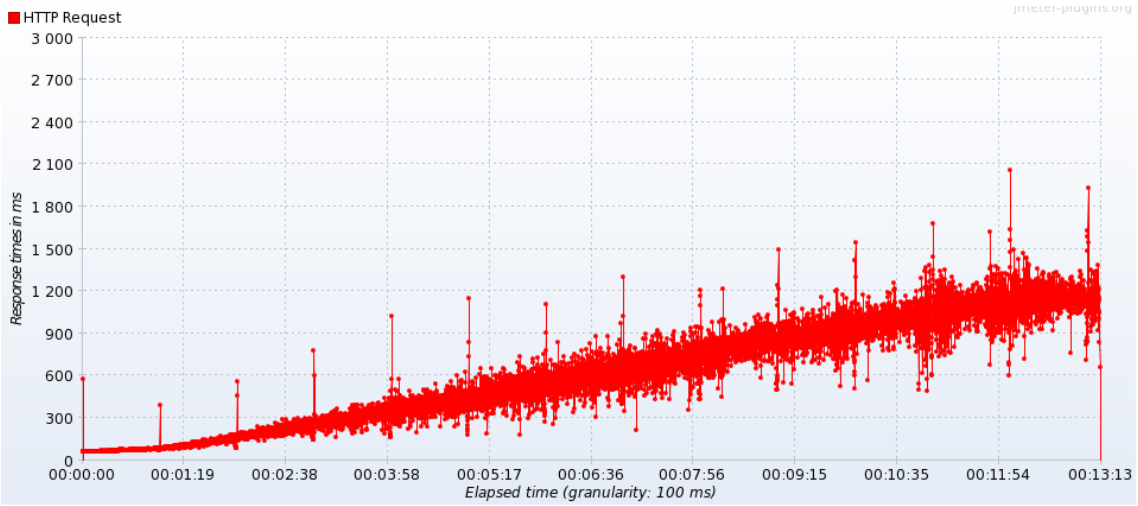

Figure 2.21: CentOS - Test 3 - Response Time over Time (Less is better)

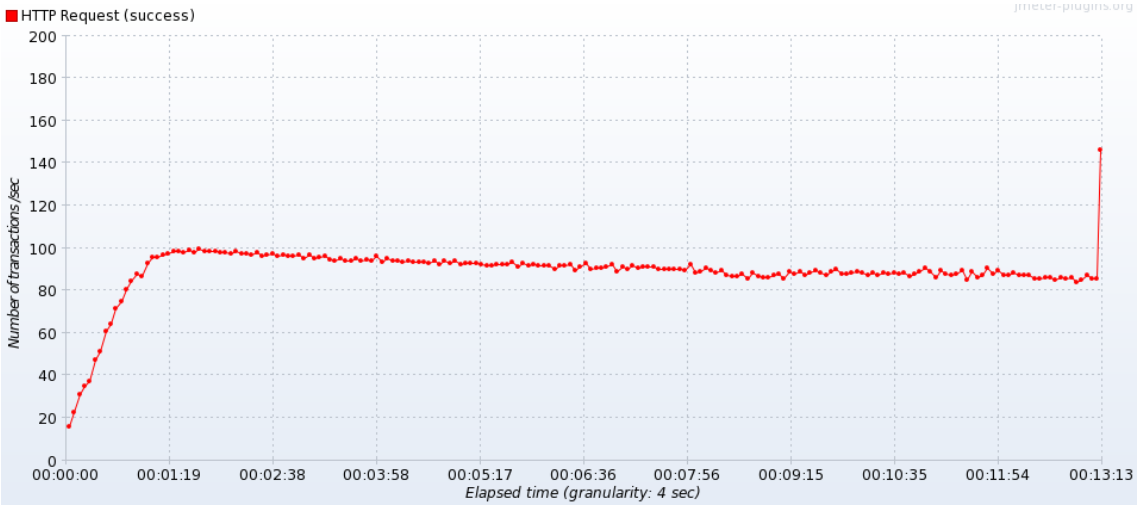

Figure 2.22: CentOS - Test 3 - Successfull Transactions Per Second (More is better)

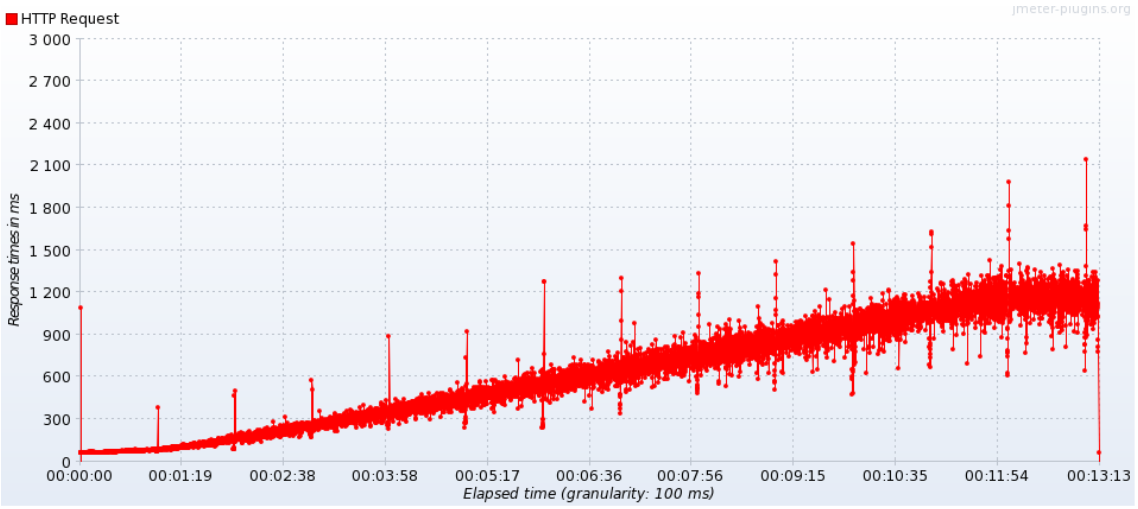

Figure 2.23: CentOS - Test 4 - Response Time over Time (Less is better)

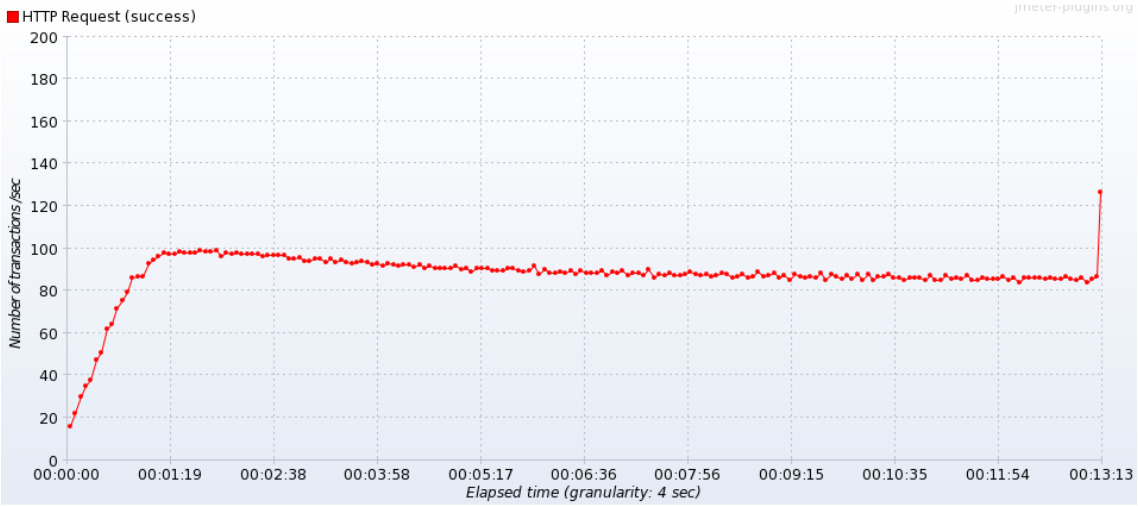

Figure 2.24: CentOS - Test 4 - Successfull Transactions Per Second (More is better)

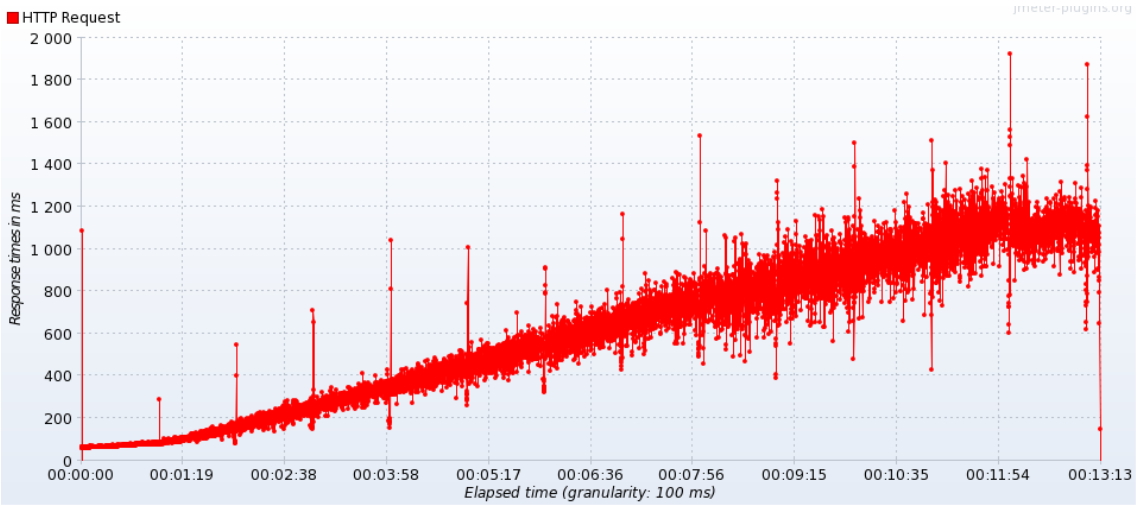

Figure 2.25: CentOS - Test 5 - Response Time over Time (Less is better)

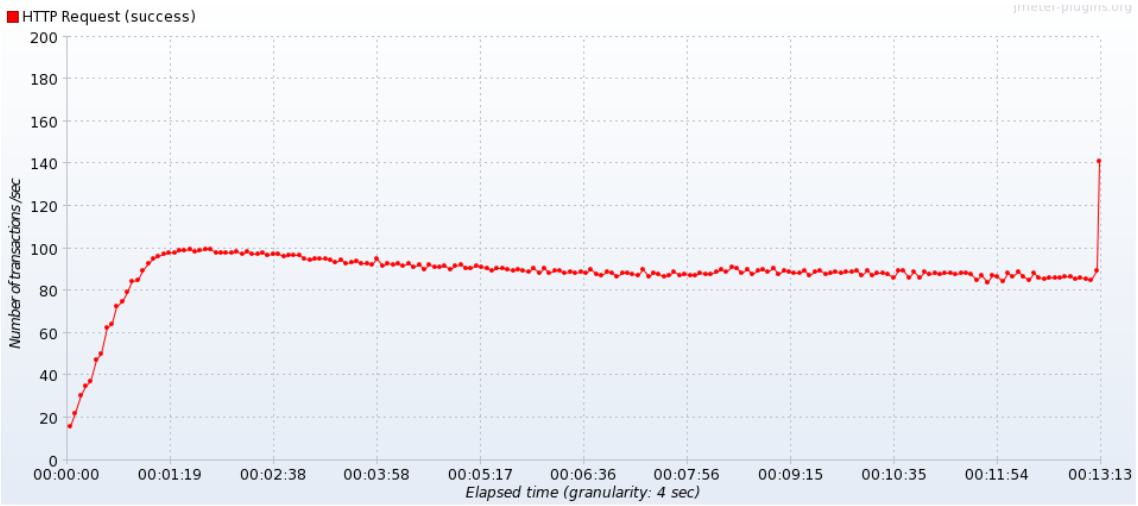

Figure 2.26: CentOS - Test 5 - Successfull Transactions Per Second (More is better)

B.2 CoreOS

This section contains all of the results from the JMeter tests towards the non-virtualized *CoreOS* machine.

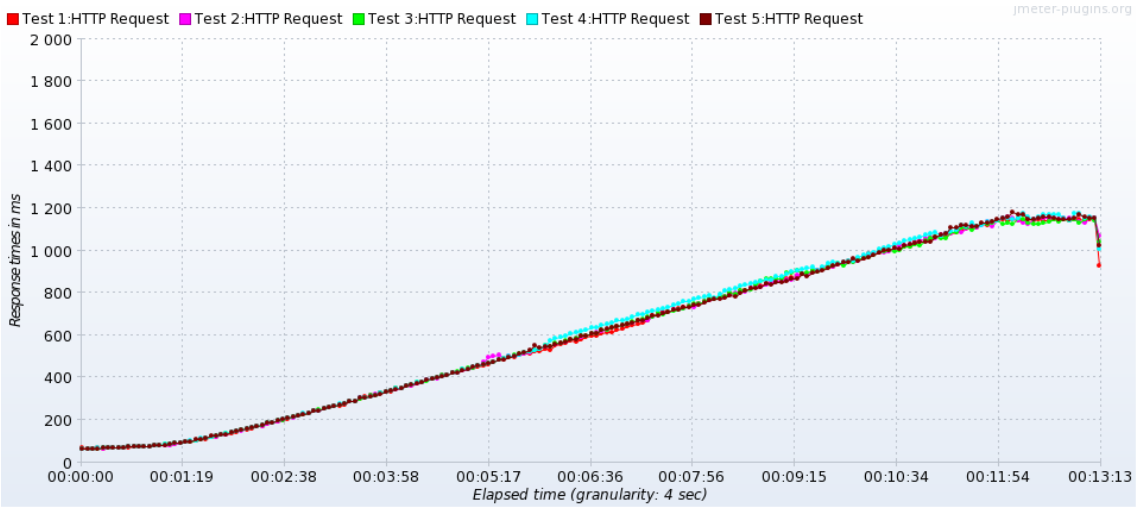

Figure 2.27: CoreOS - Test 1 - 5 - Response Time over Time (Less is better)

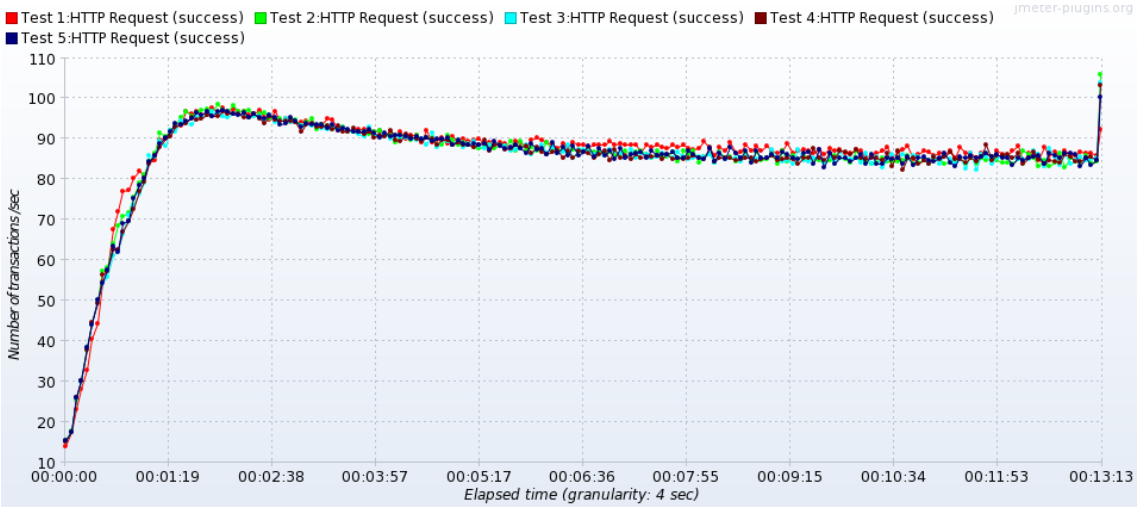

Figure 2.28: CoreOS - Test 1 - 5 - Successfull Transactions Per Second (More is better)

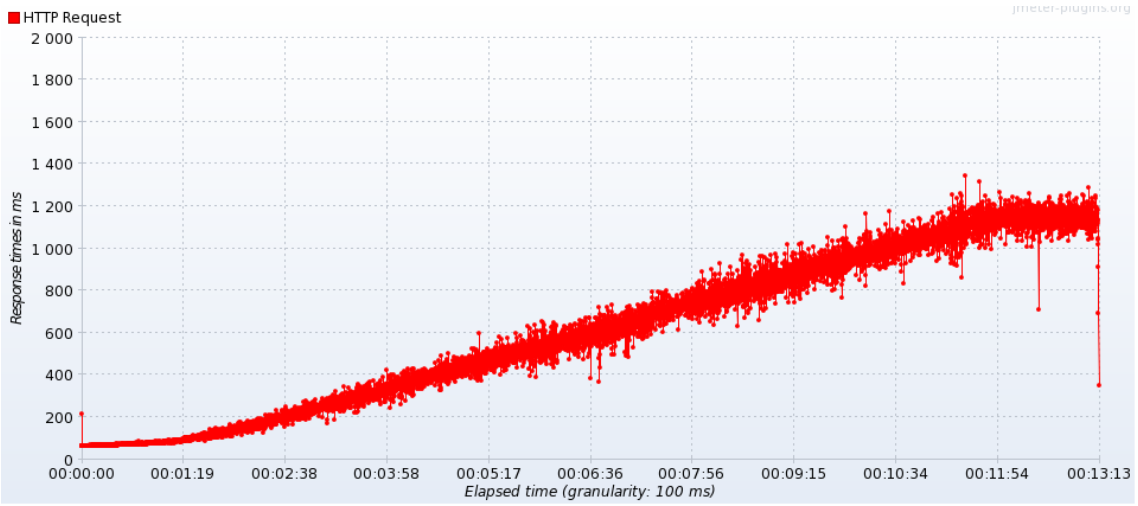

Figure 2.29: CoreOS - Test 1 - Response Time over Time (Less is better)

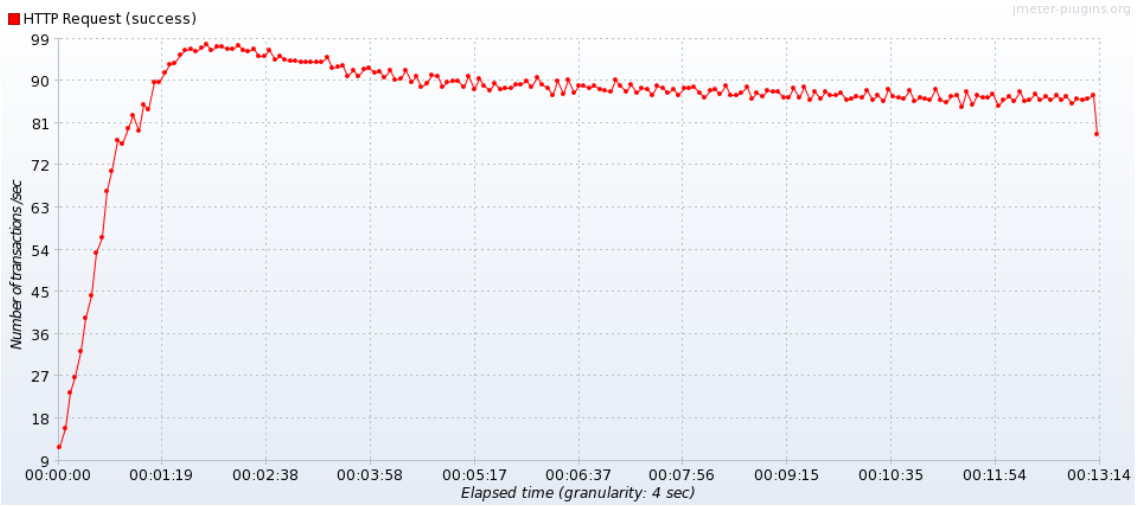

Figure 2.30: CoreOS - Test 1 - Successfull Transactions Per Second (More is better)

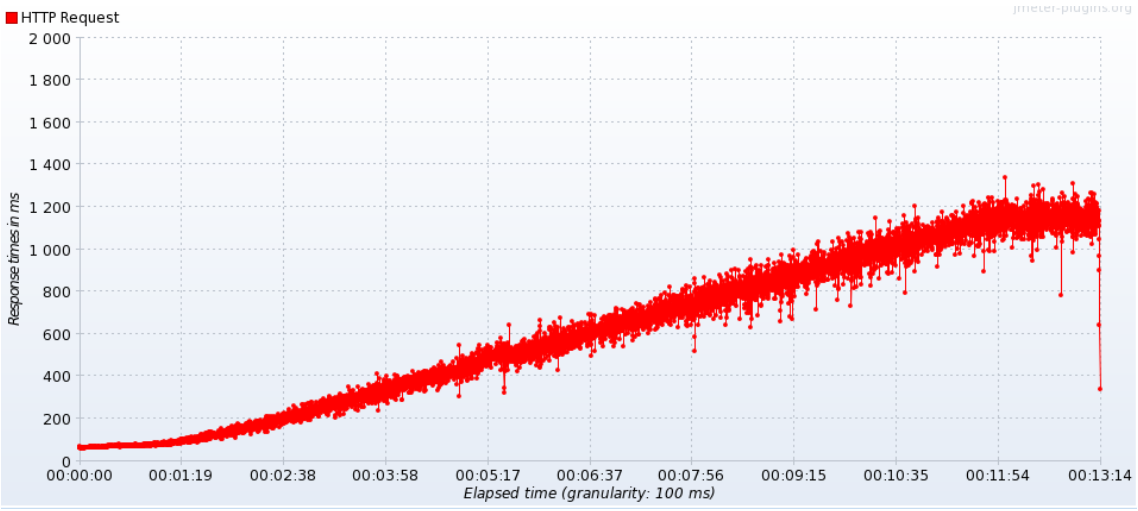

Figure 2.31: CoreOS - Test 2 - Response Time over Time (Less is better)

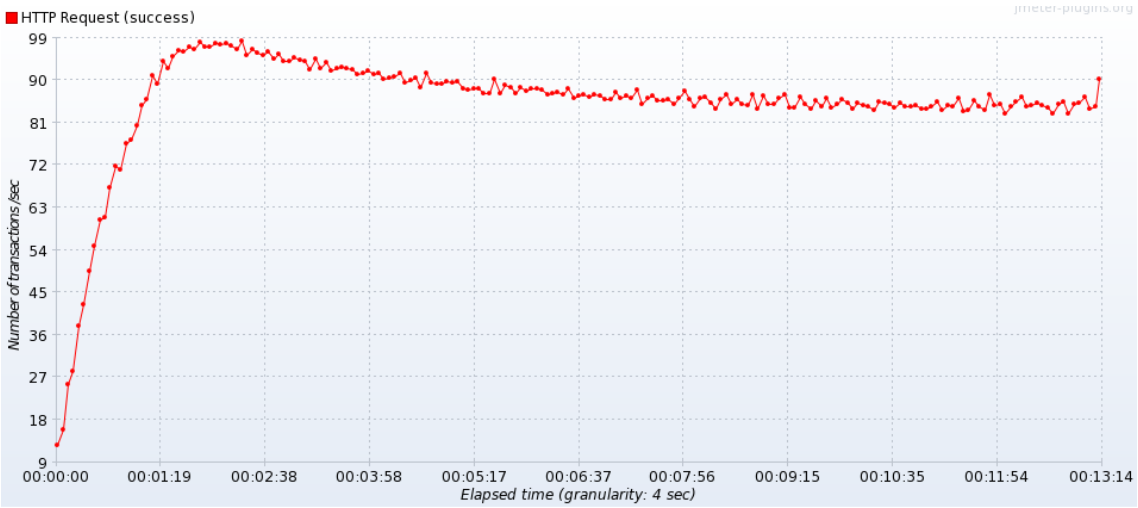

Figure 2.32: CoreOS - Test 2 - Successfull Transactions Per Second (More is better)

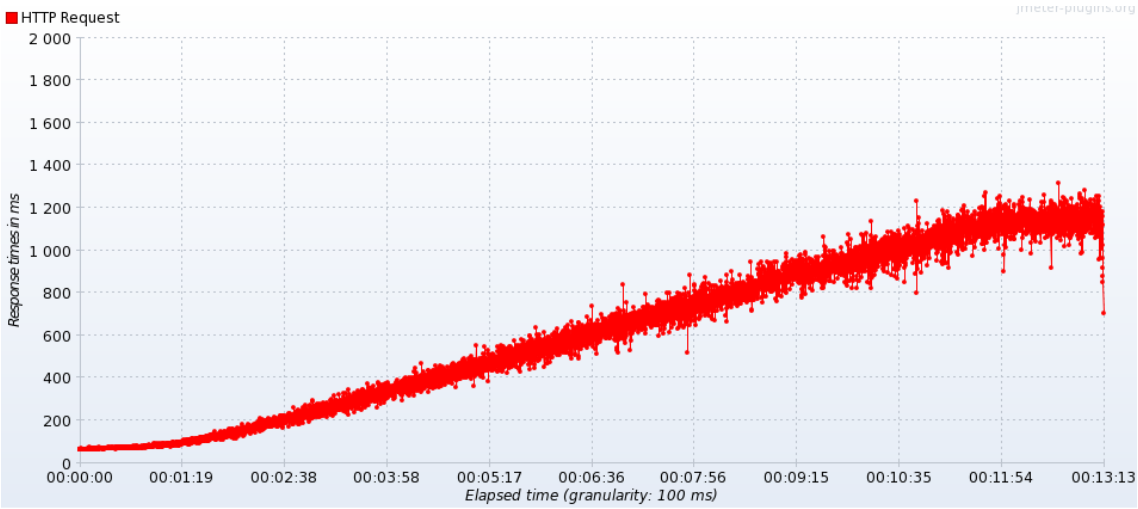

Figure 2.33: CoreOS - Test 3 - Response Time over Time (Less is better)

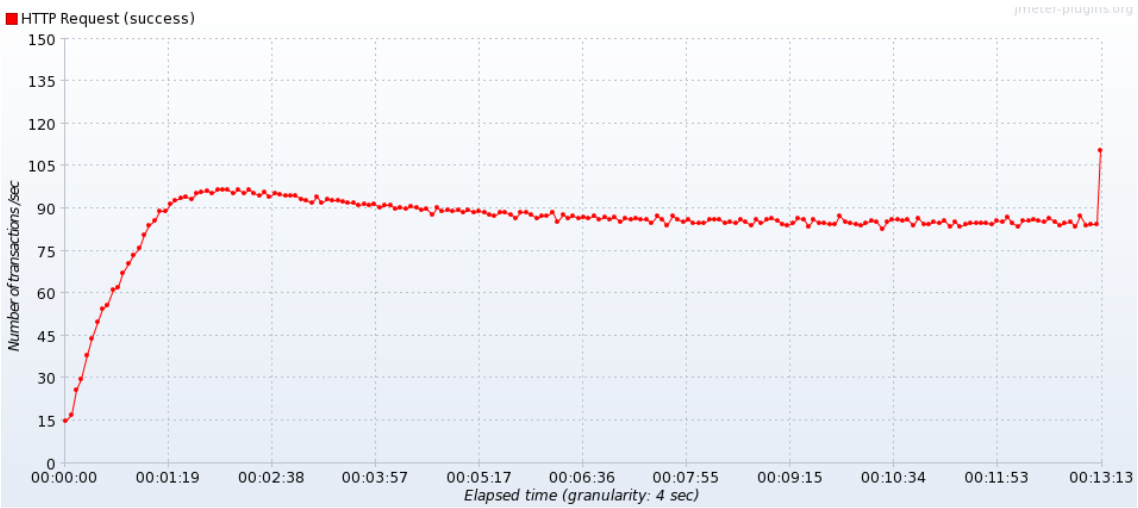

Figure 2.34: CoreOS - Test 3 - Successfull Transactions Per Second (More is better)

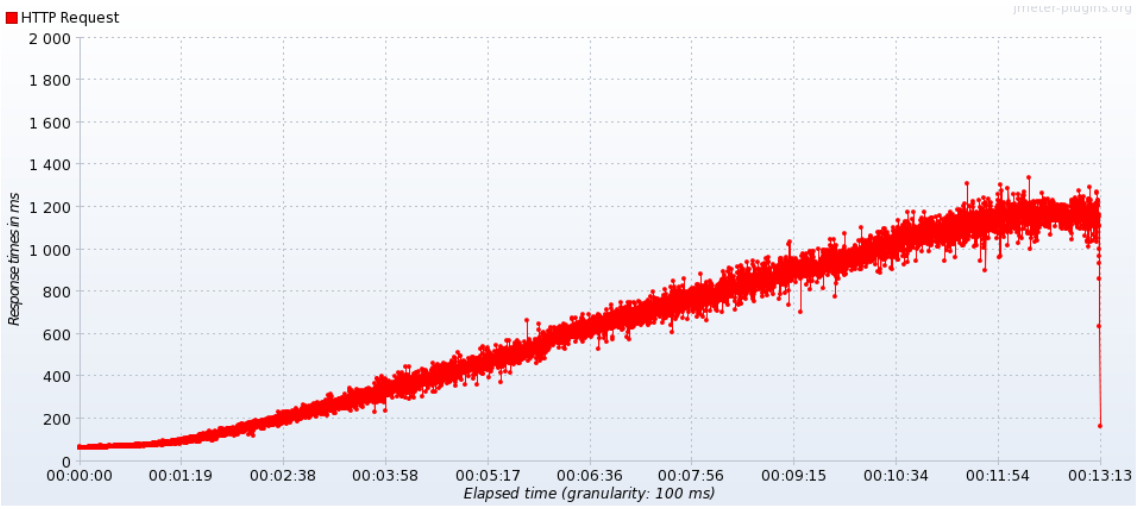

Figure 2.35: CoreOS - Test 4 - Response Time over Time (Less is better)

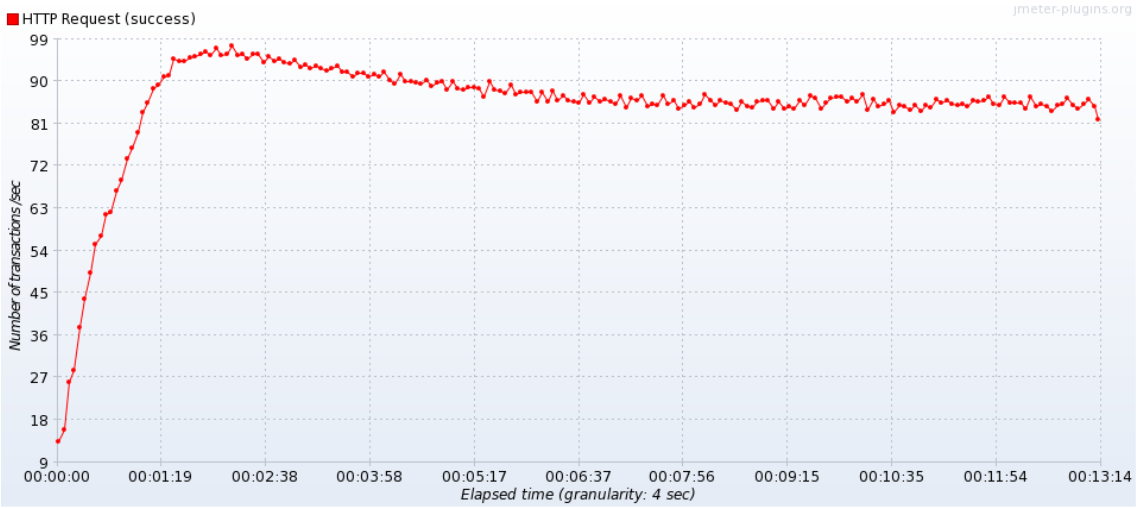

Figure 2.36: CoreOS - Test 4 - Successfull Transactions Per Second (More is better)

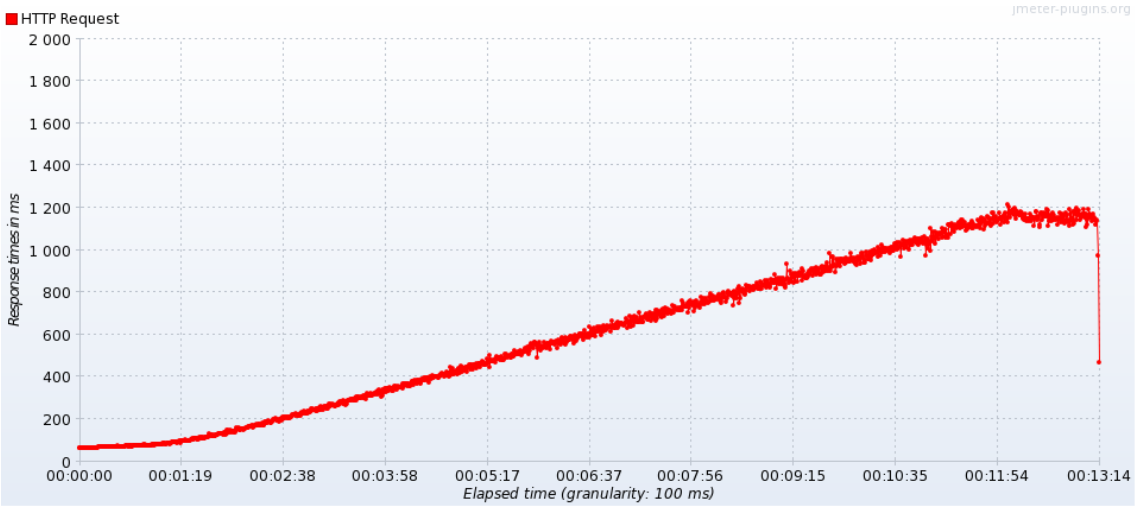

Figure 2.37: CoreOS - Test 5 - Response Time over Time (Less is better)

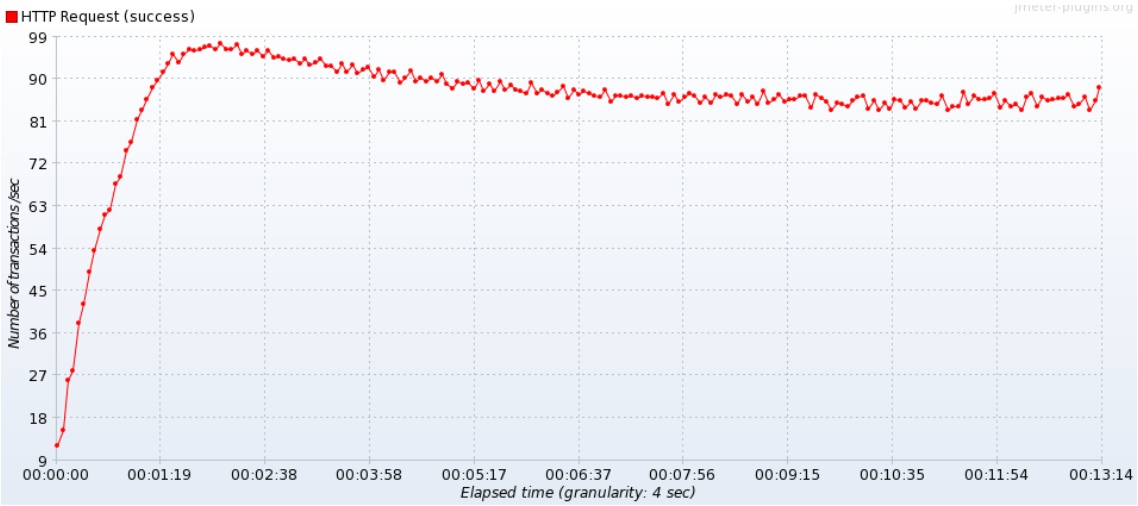

Figure 2.38: CoreOS - Test 5 - Successfull Transactions Per Second (More is better)

B.3 VMCentOS

This section contains all of the results from the JMeter tests towards the virtualized *CentOS 7* machine.

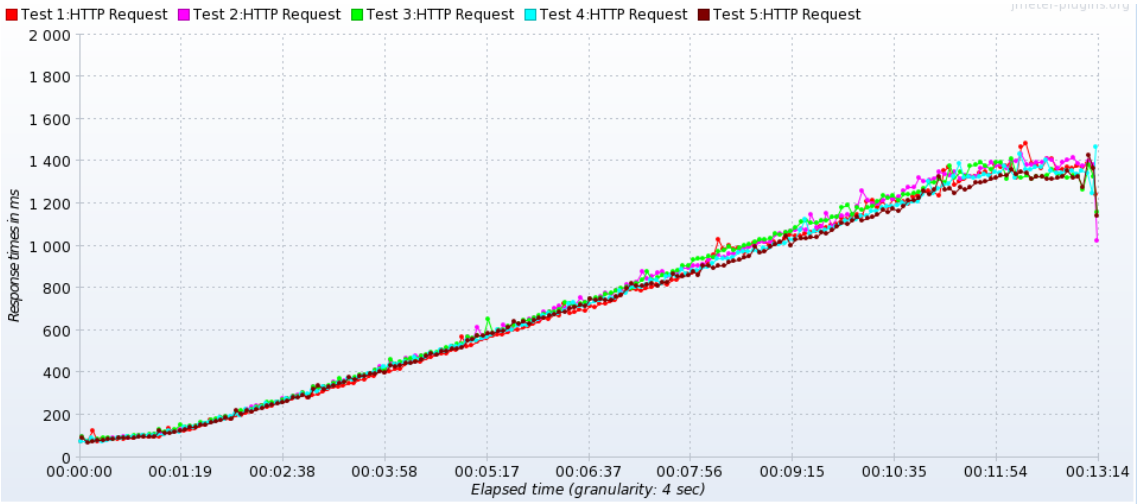

Figure 2.39: VMCentOS - Test 1 - 5 - Response Time over Time (Less is better)

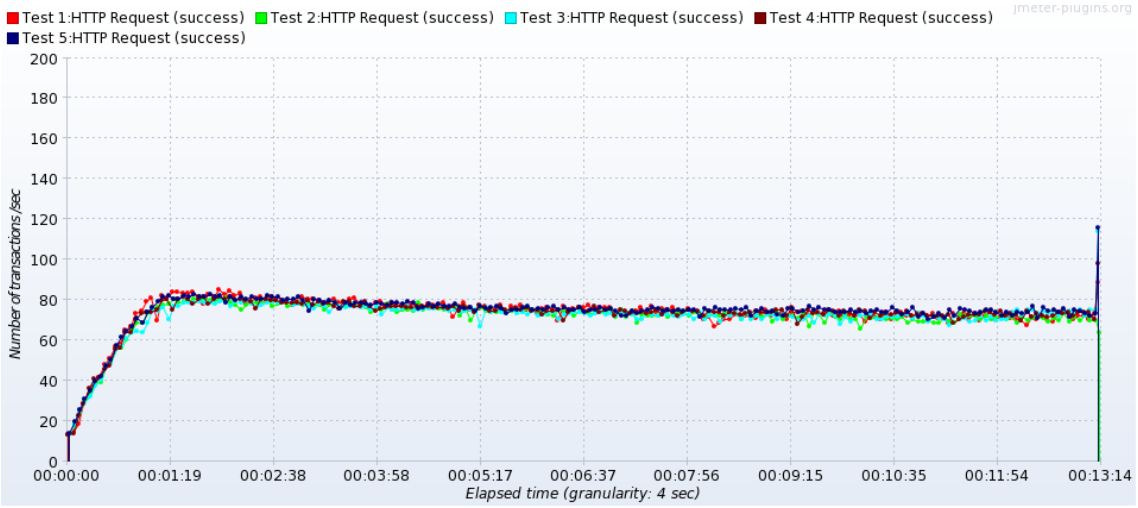

Figure 2.40: VMCentOS - Test 1 - 5 - Successfull Transactions Per Second (More is better)

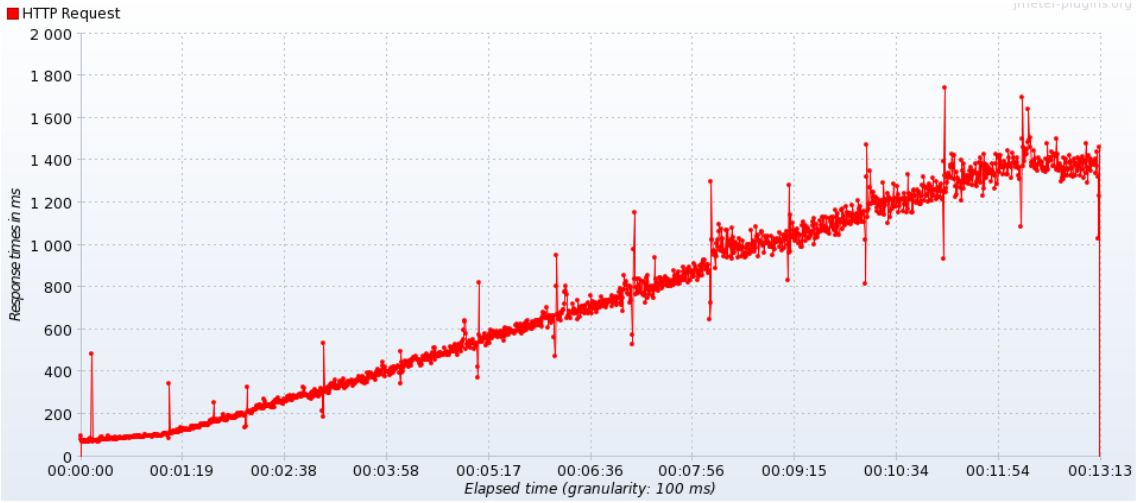

Figure 2.41: VMCentOS - Test 1 - Response Time over Time (Less is better)

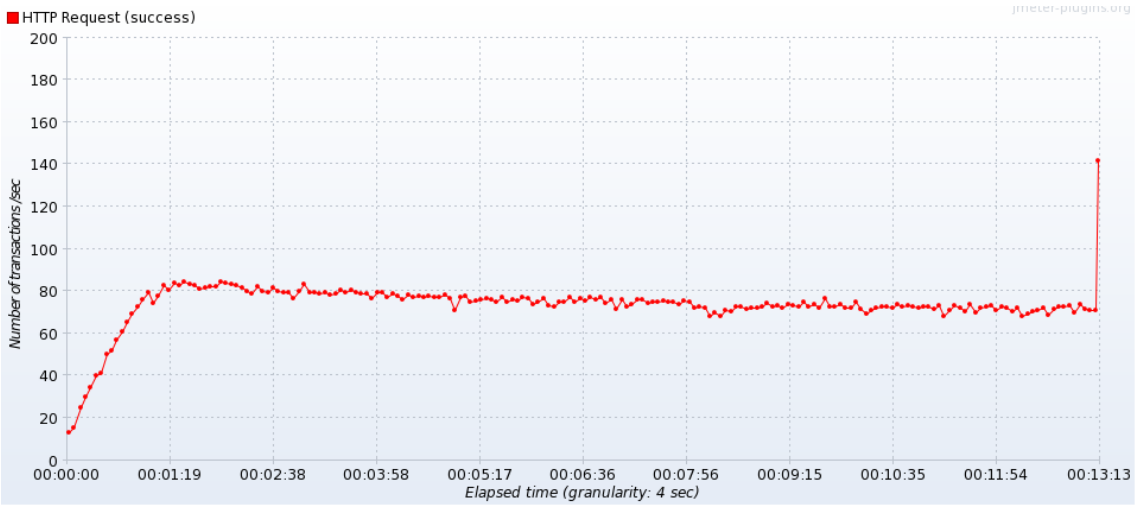

Figure 2.42: VMCentOS - Test 1 - Successfull Transactions Per Second (More is better)

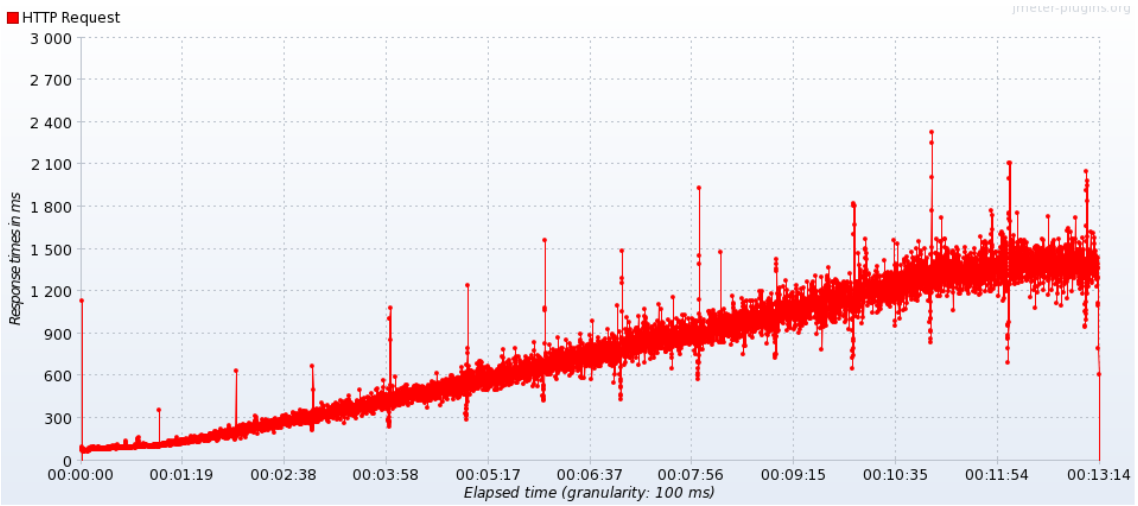

Figure 2.43: VMCentOS - Test 2 - Response Time over Time (Less is better)

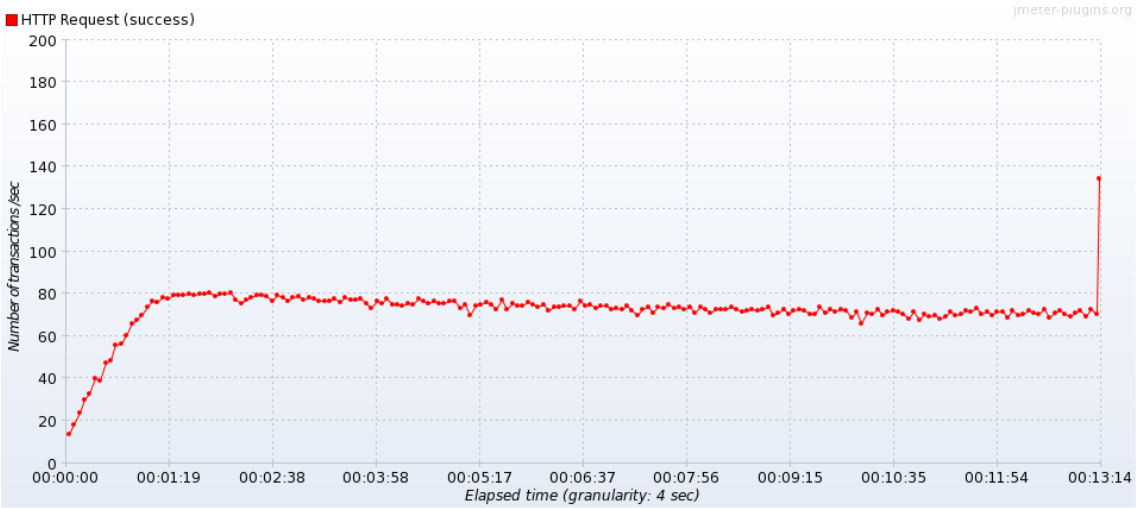

Figure 2.44: VMCentOS - Test 2 - Successfull Transactions Per Second (More is better)

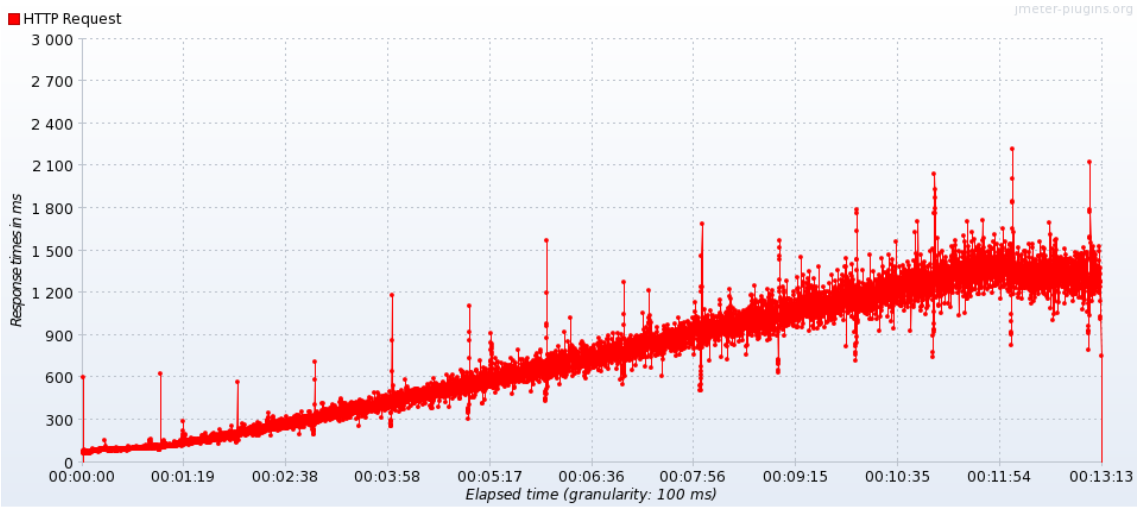

Figure 2.45: VMCentOS - Test 3 - Response Time over Time (Less is better)

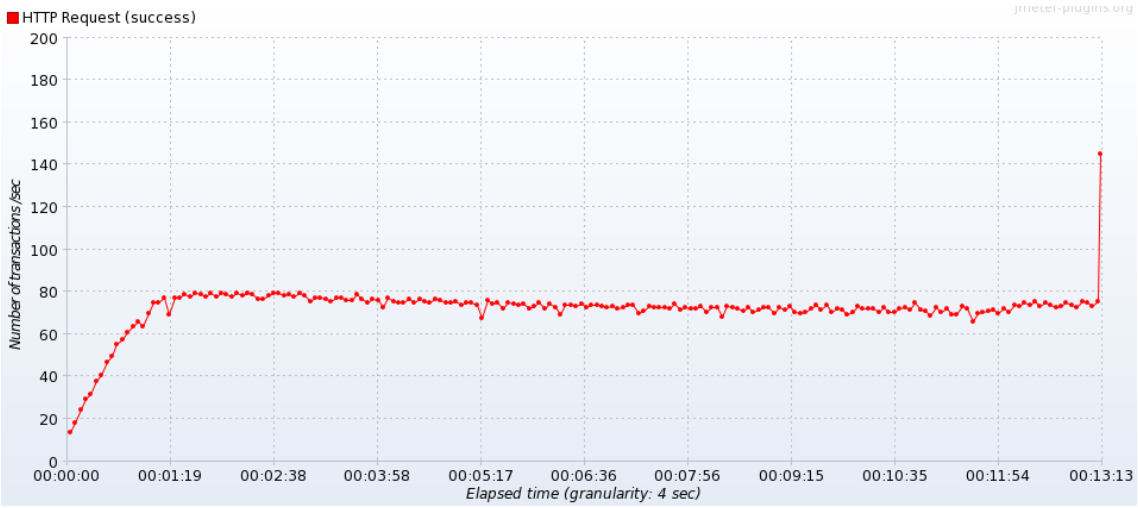

Figure 2.46: VMCentOS - Test 3 - Successfull Transactions Per Second (More is better)

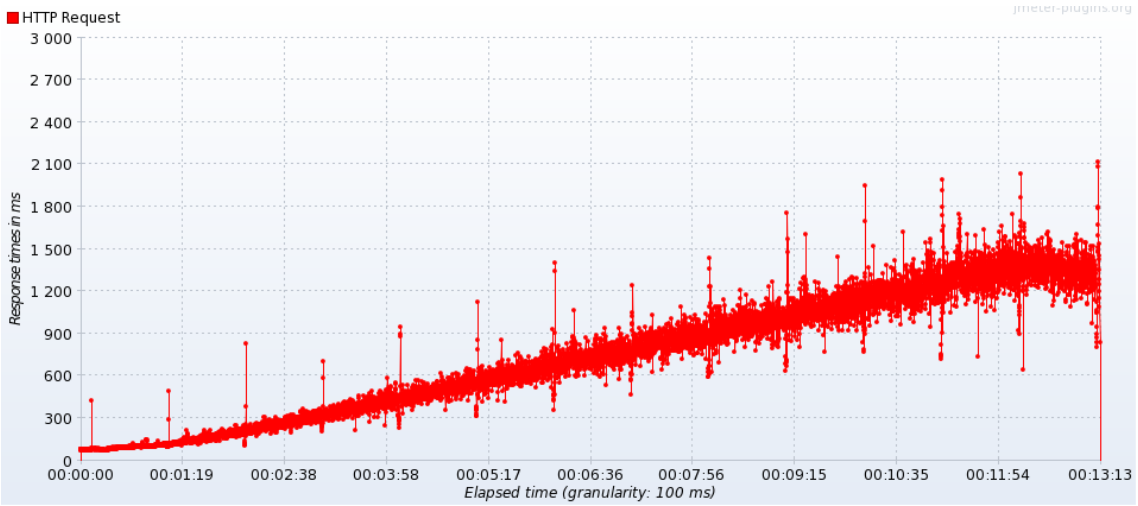

Figure 2.47: VMCentOS - Test 4 - Response Time over Time (Less is better)

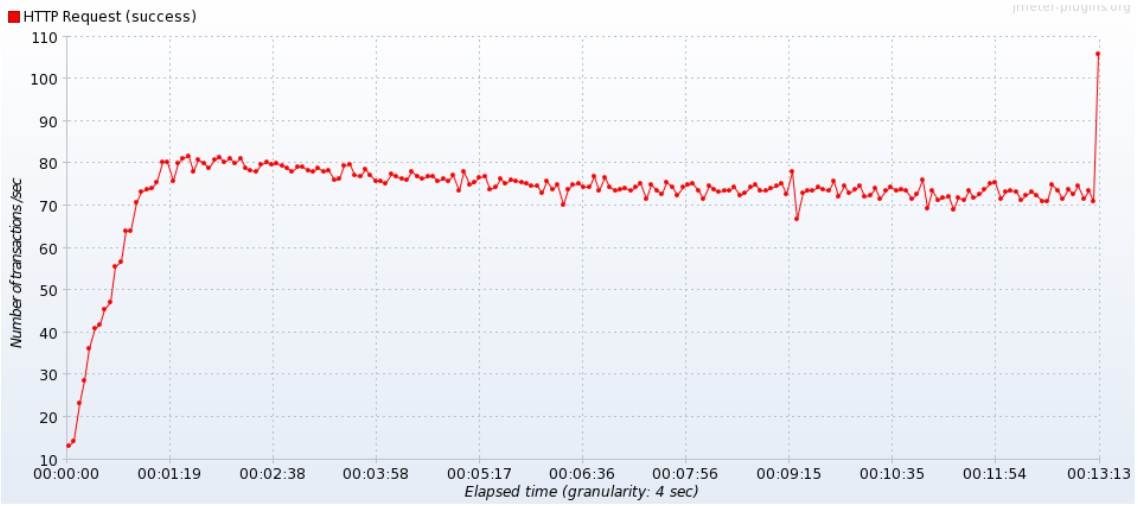

Figure 2.48: VMCentOS - Test 4 - Successfull Transactions Per Second (More is better)

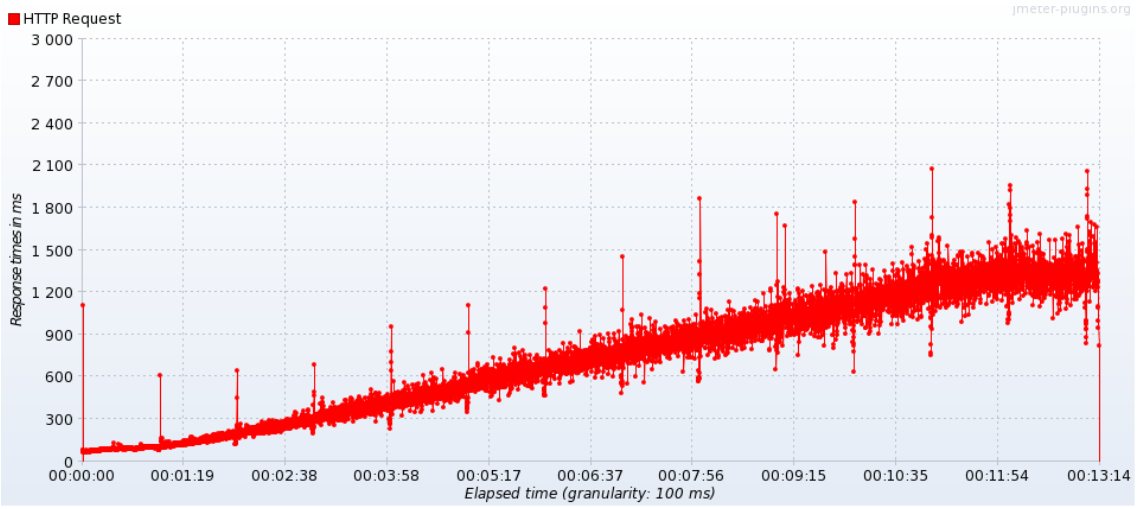

Figure 2.49: VMCentOS - Test 5 - Response Time over Time (Less is better)

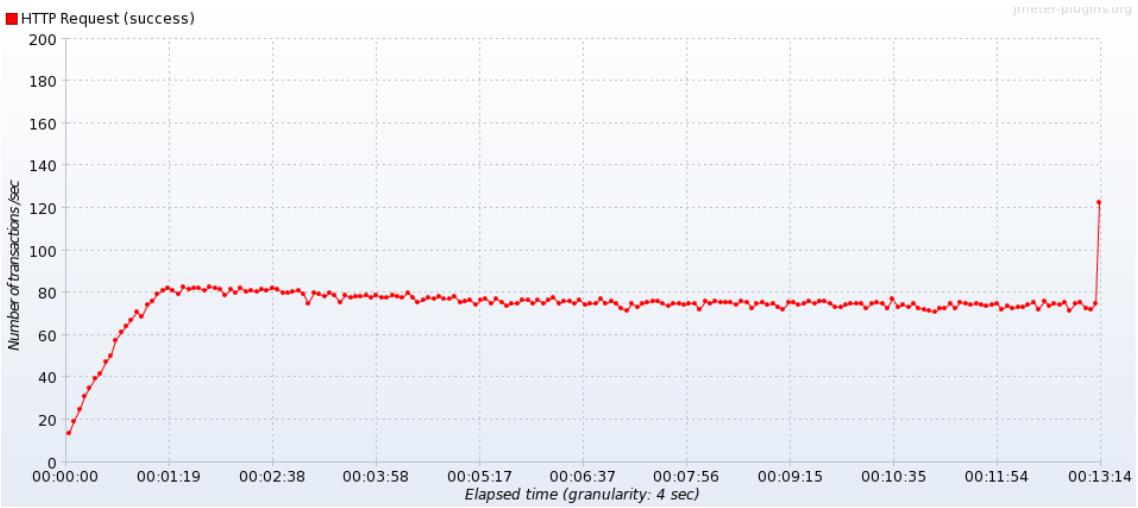

Figure 2.50: VMCentOS - Test 5 - Successfull Transactions Per Second (More is better)

B.4 VMCoreOS

This section contains all of the results from the JMeter tests towards the virtualized CoreOS machine.

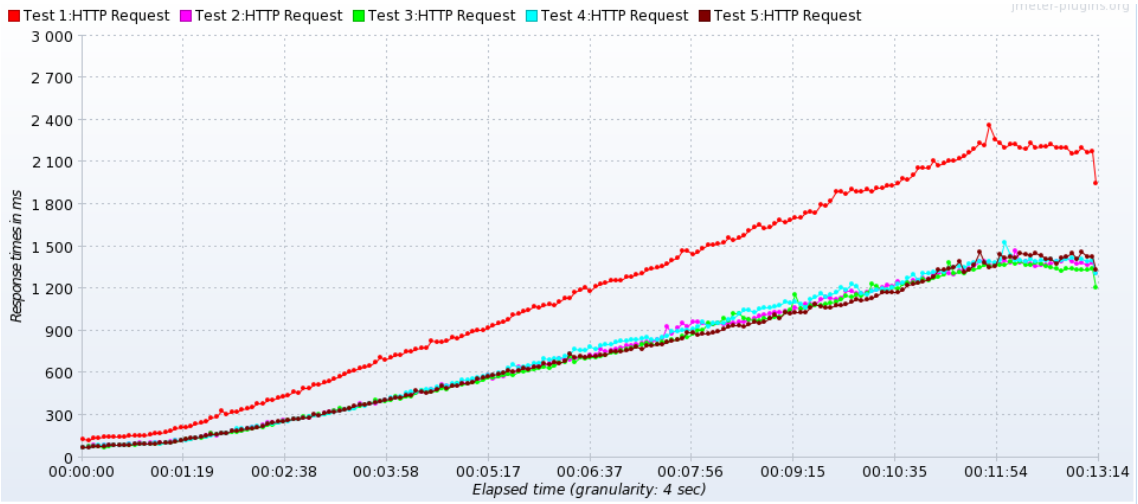

Figure 2.51: VMCoreOS - Test 1 - 5 - Response Time over Time (Less is better)

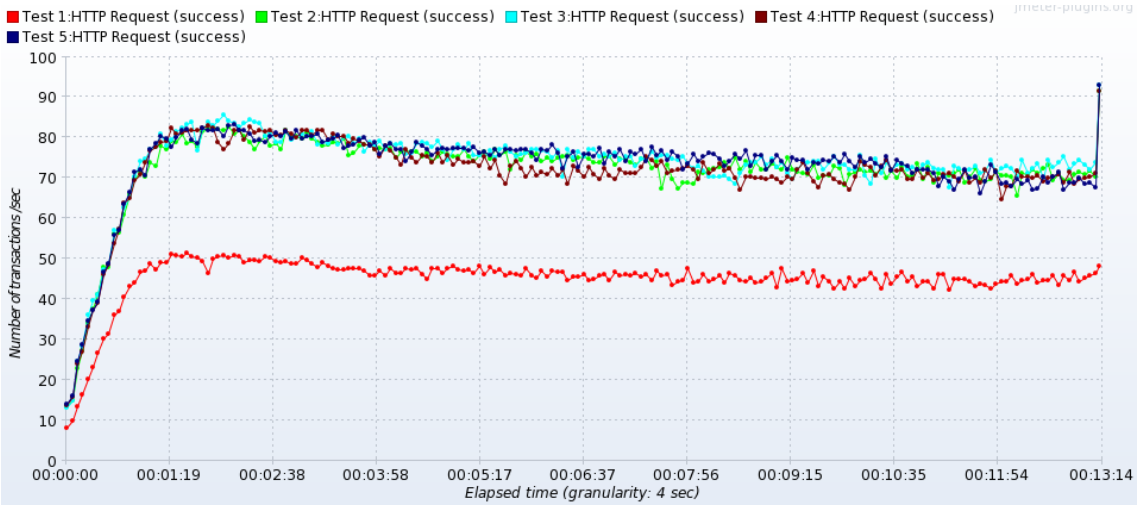

Figure 2.52: VMCoreOS - Test 1 - 5 - Successfull Transactions Per Second (More is better)

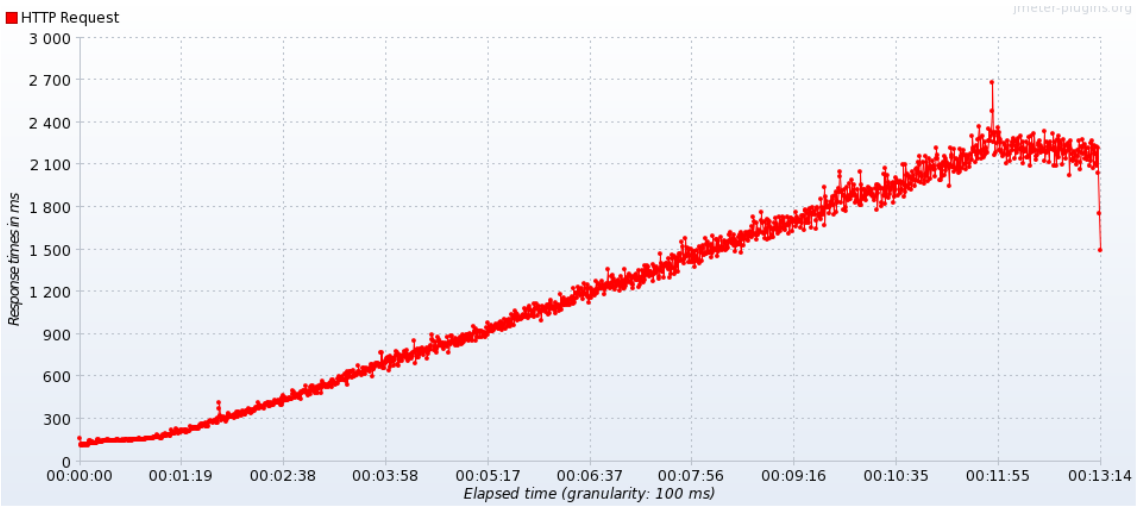

Figure 2.53: VMCoreOS - Test 1 - Response Time over Time (Less is better)

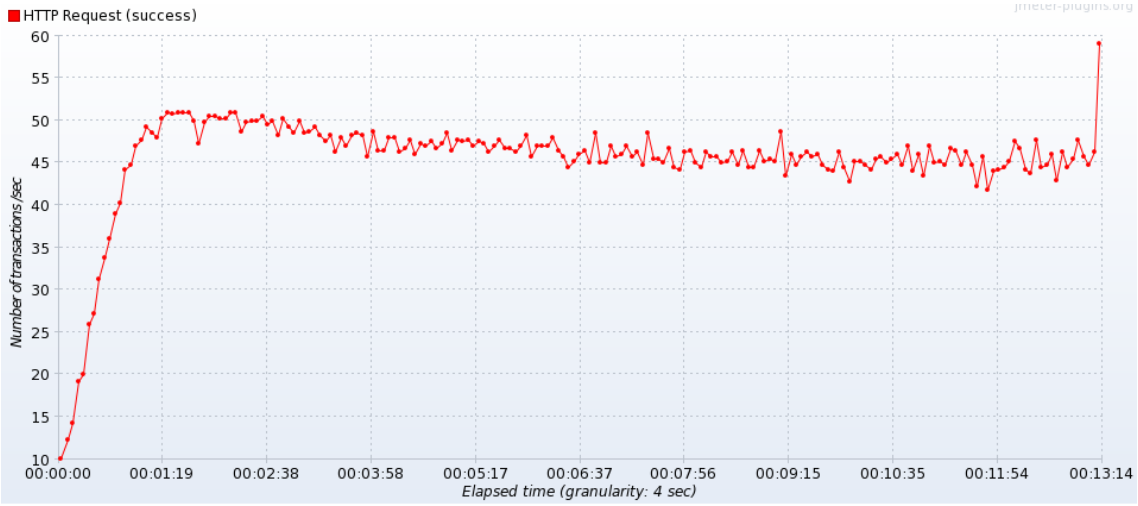

Figure 2.54: VMCoreOS - Test 1 - Successfull Transactions Per Second (More is better)

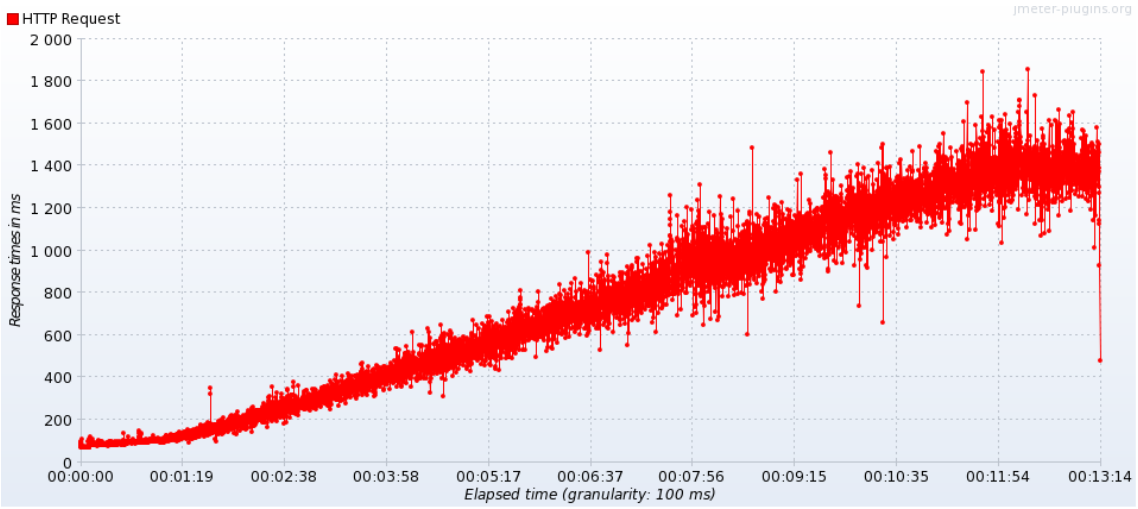

Figure 2.55: VMCoreOS - Test 2 - Response Time over Time (Less is better)

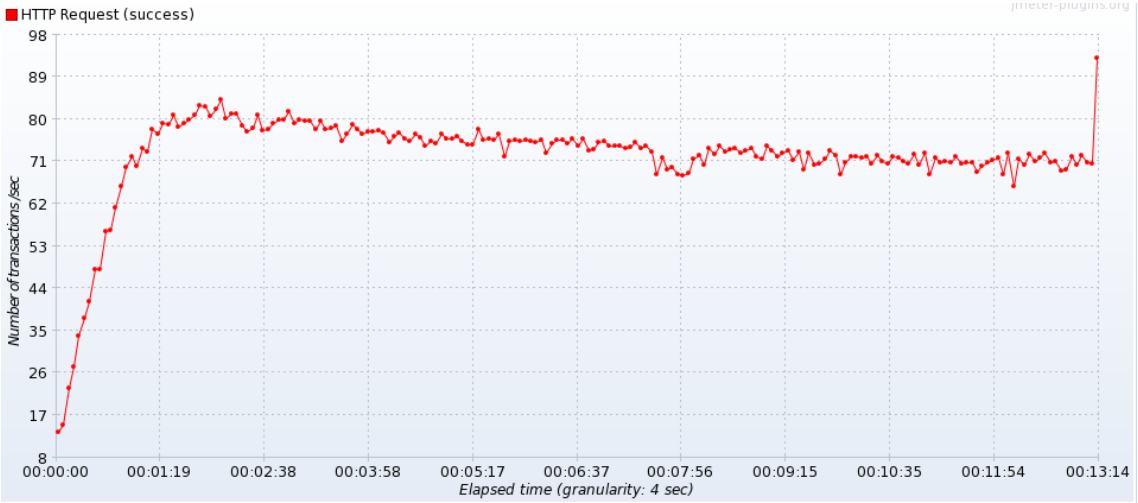

Figure 2.56: VMCoreOS - Test 2 - Successfull Transactions Per Second (More is better)

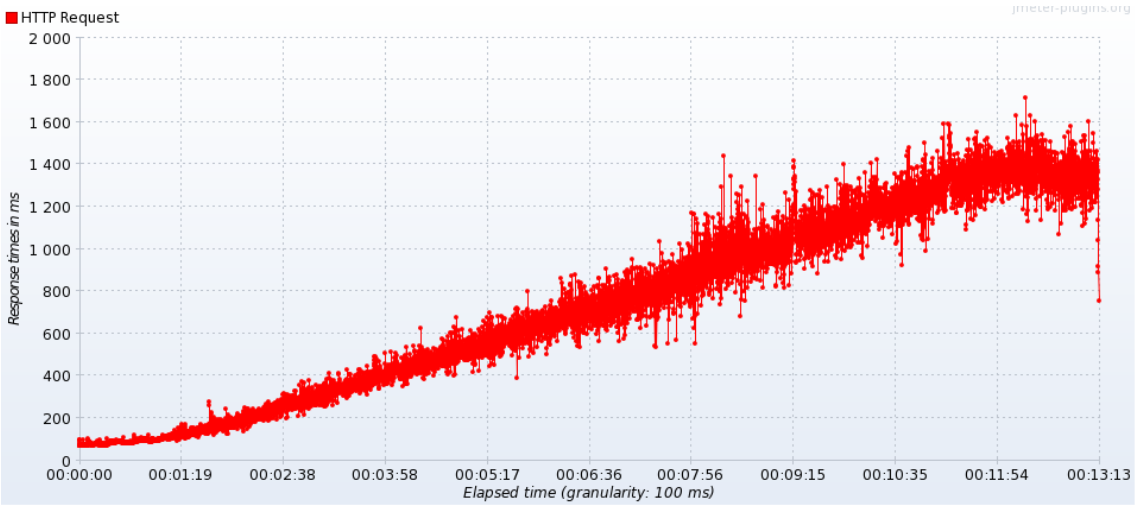

Figure 2.57: VMCoreOS - Test 3 - Response Time over Time (Less is better)

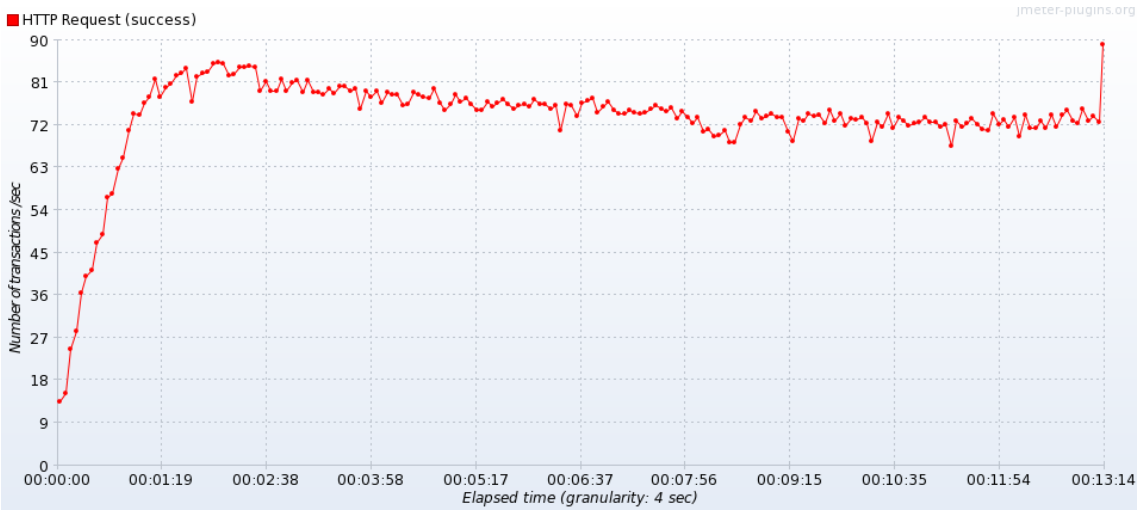

Figure 2.58: VMCoreOS - Test 3 - Successfull Transactions Per Second (More is better)

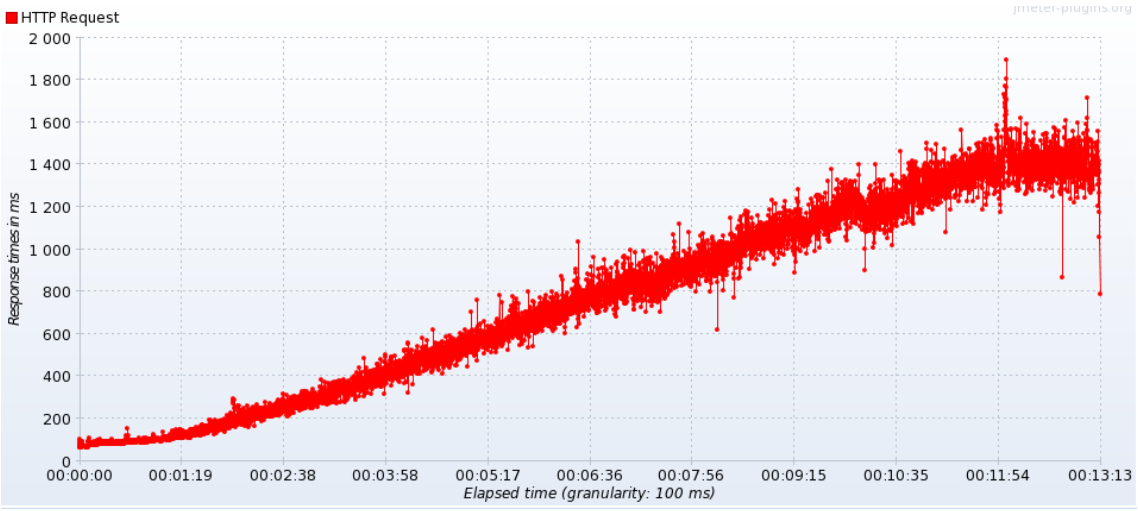

Figure 2.59: VMCoreOS - Test 4 - Response Time over Time (Less is better)

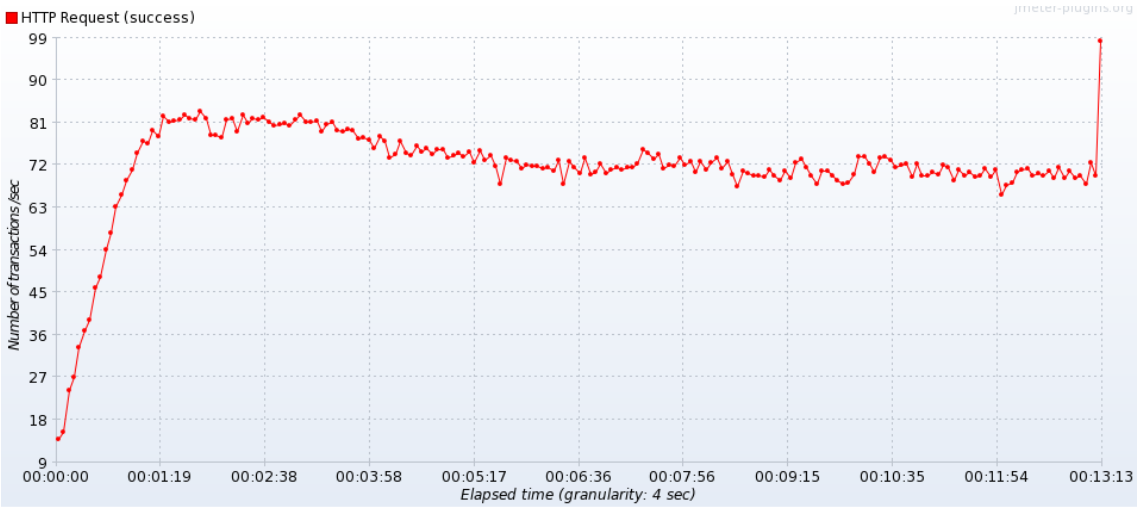

Figure 2.60: VMCoreOS - Test 4 - Successfull Transactions Per Second (More is better)

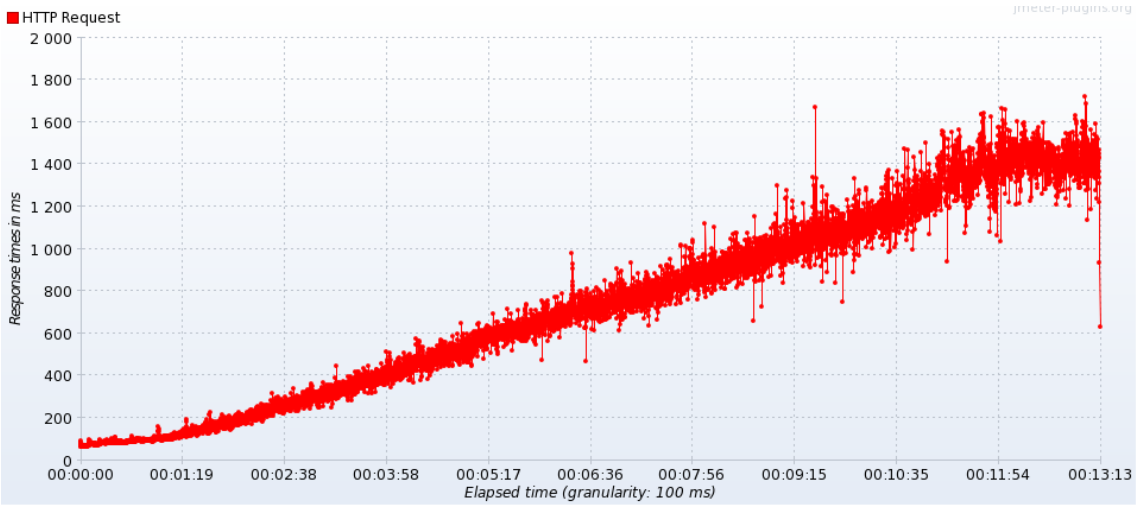

Figure 2.61: VMCoreOS - Test 5 - Response Time over Time (Less is better)

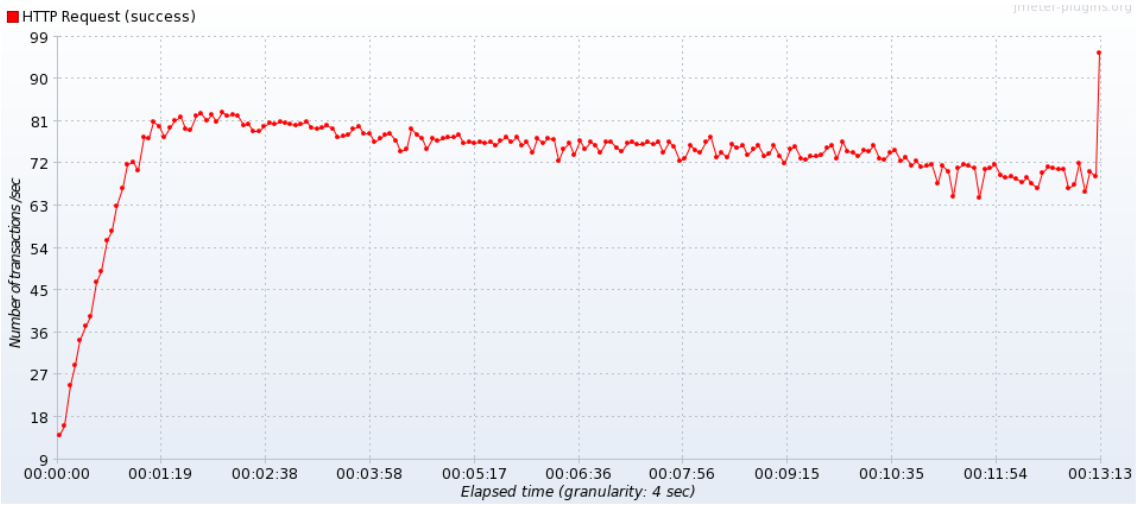

Figure 2.62: VMCoreOS - Test 5 - Successfull Transactions Per Second (More is better)

B.5 VMPhoton

This section contains all of the results from the JMeter tests towards the virtualized *Photon OS* machine.

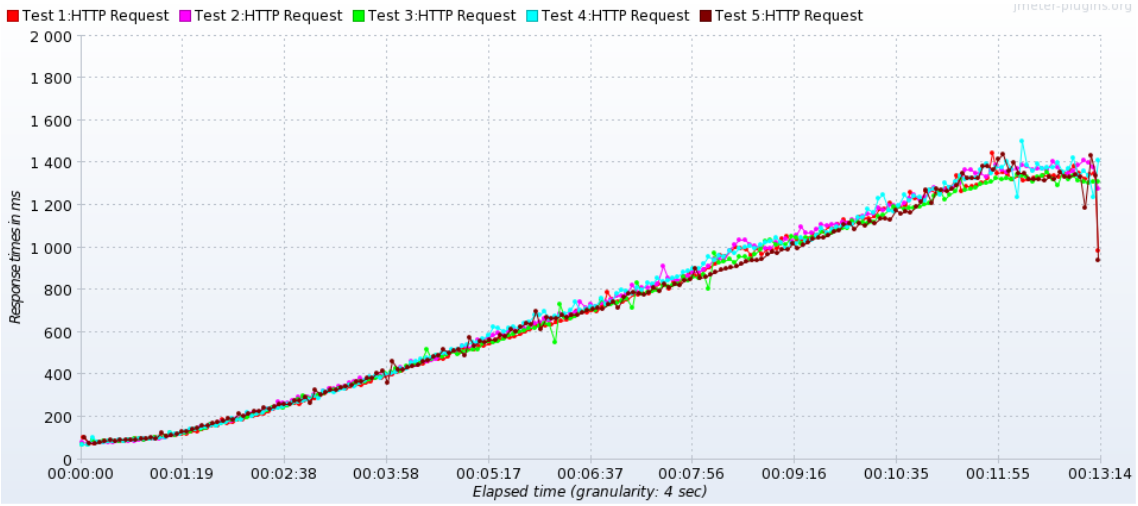

Figure 2.63: VMPhoton - Test 1 - 5 - Response Time over Time (Less is better)

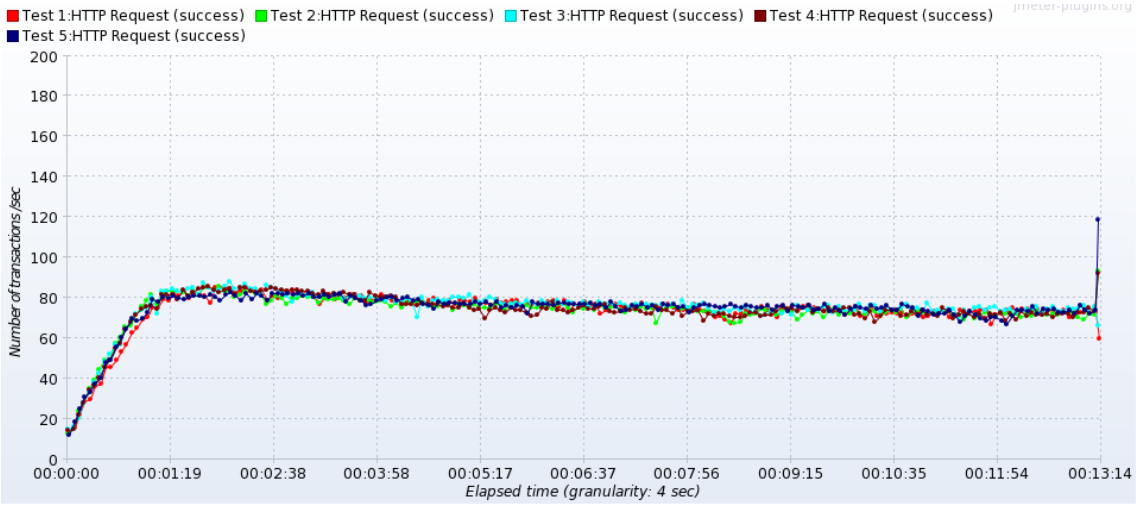

Figure 2.64: VMPhoton - Test 1 - 5 - Successfull Transactions Per Second (More is better)

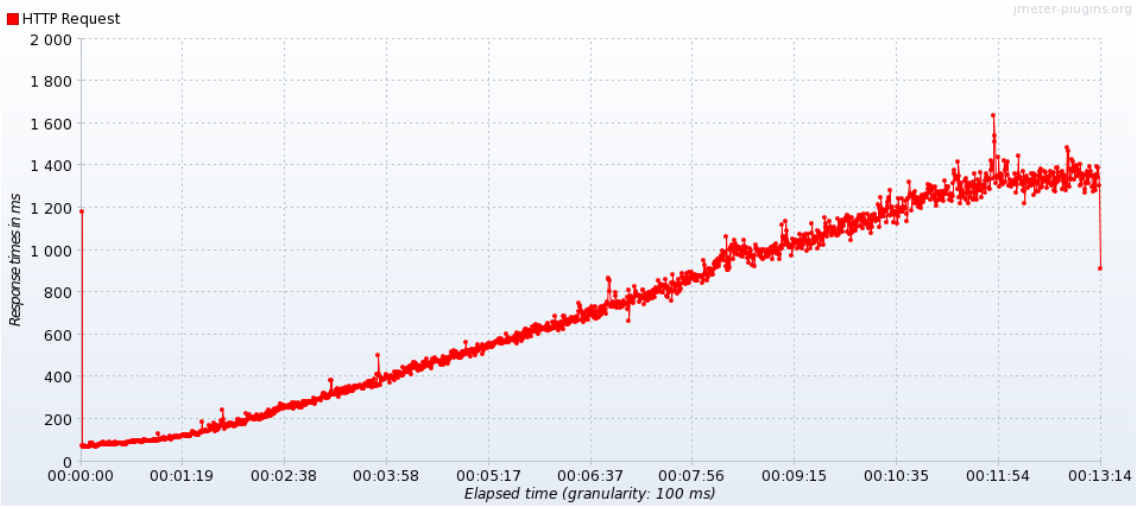

Figure 2.65: VMPhoton - Test 1 - Response Time over Time (Less is better)

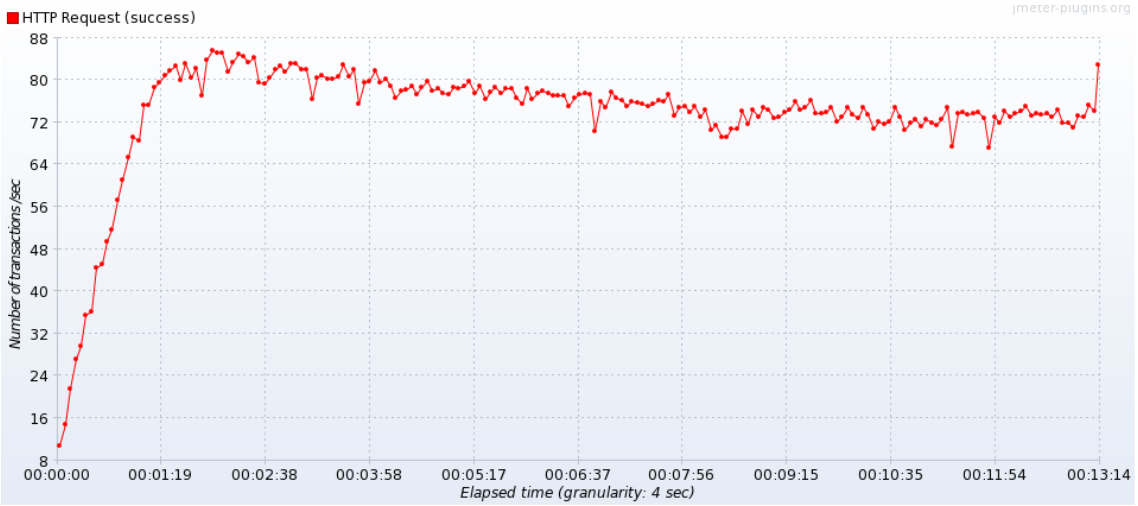

Figure 2.66: VMPhoton - Test 1 - Successfull Transactions Per Second (More is better)

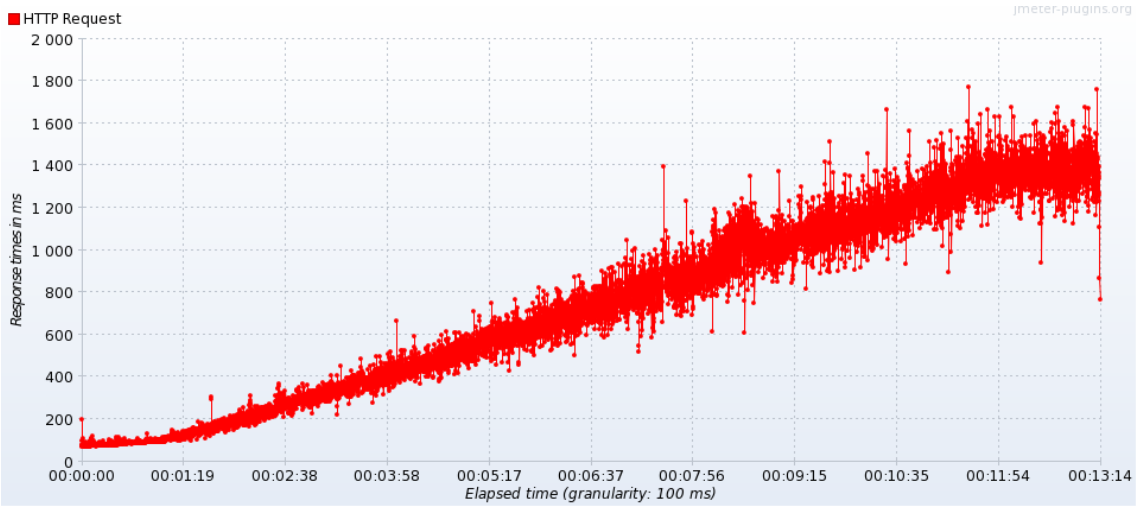

Figure 2.67: VMPhoton - Test 2 - Response Time over Time (Less is better)

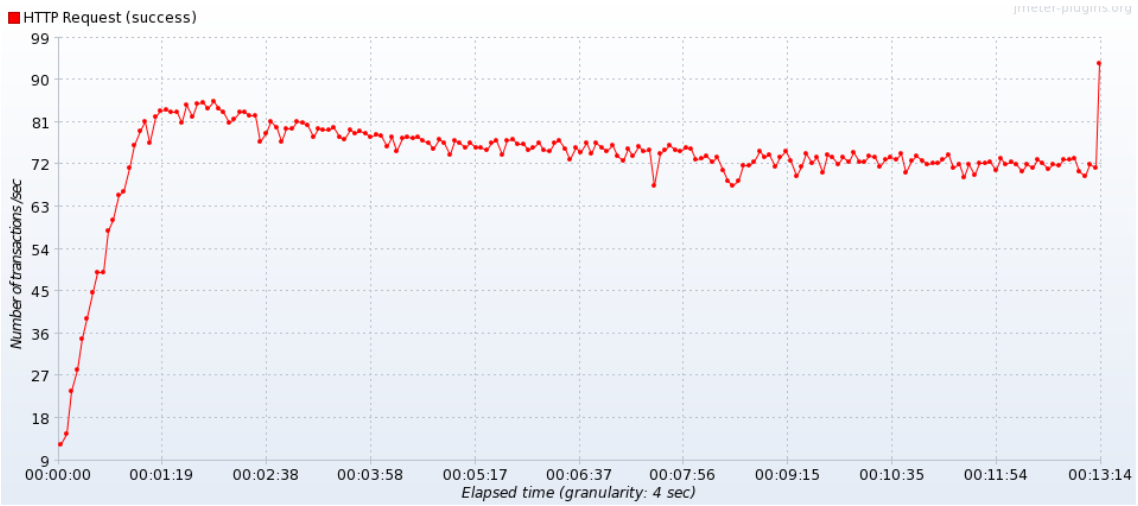

Figure 2.68: VMPhoton - Test 2 - Successfull Transactions Per Second (More is better)

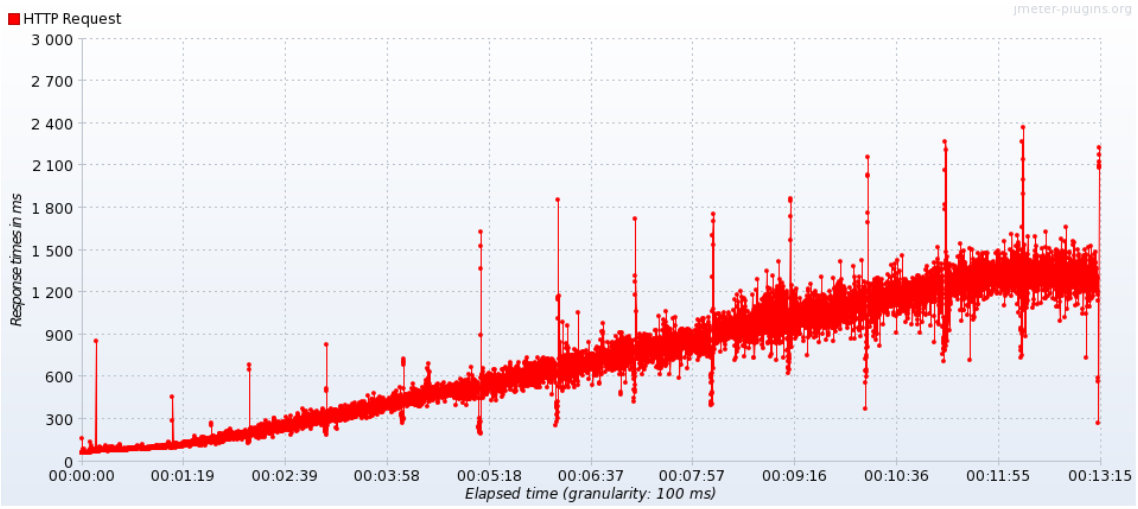

Figure 2.69: VMPhoton - Test 3 - Response Time over Time (Less is better)

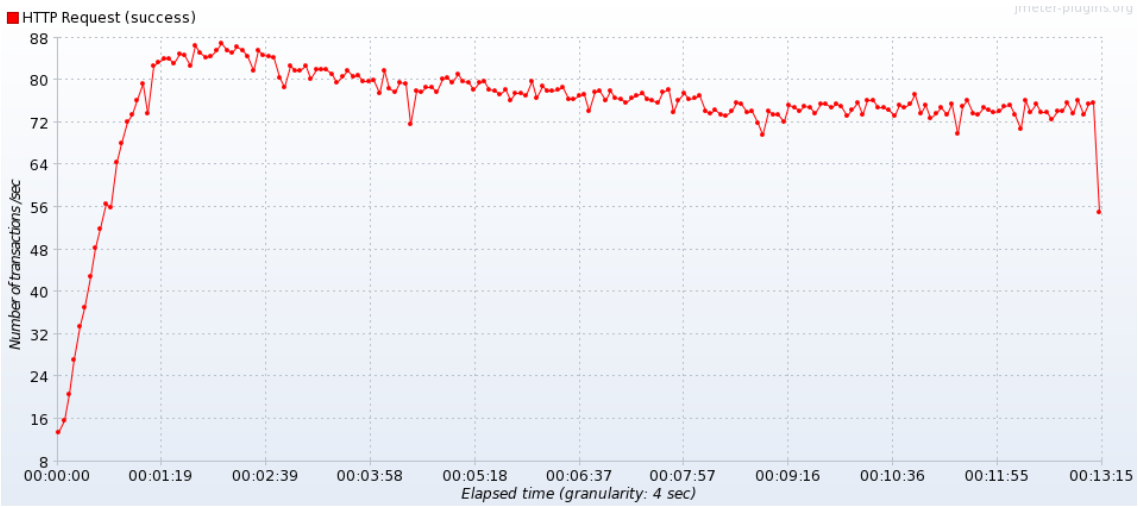

Figure 2.70: VMPhoton - Test 3 - Successfull Transactions Per Second (More is better)

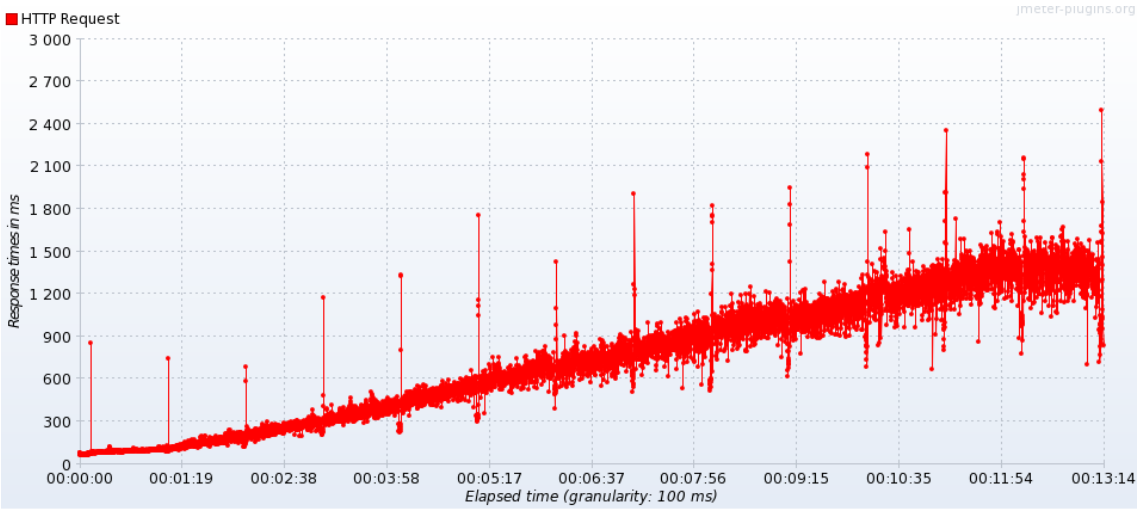

Figure 2.71: VMPhoton - Test 4 - Response Time over Time (Less is better)

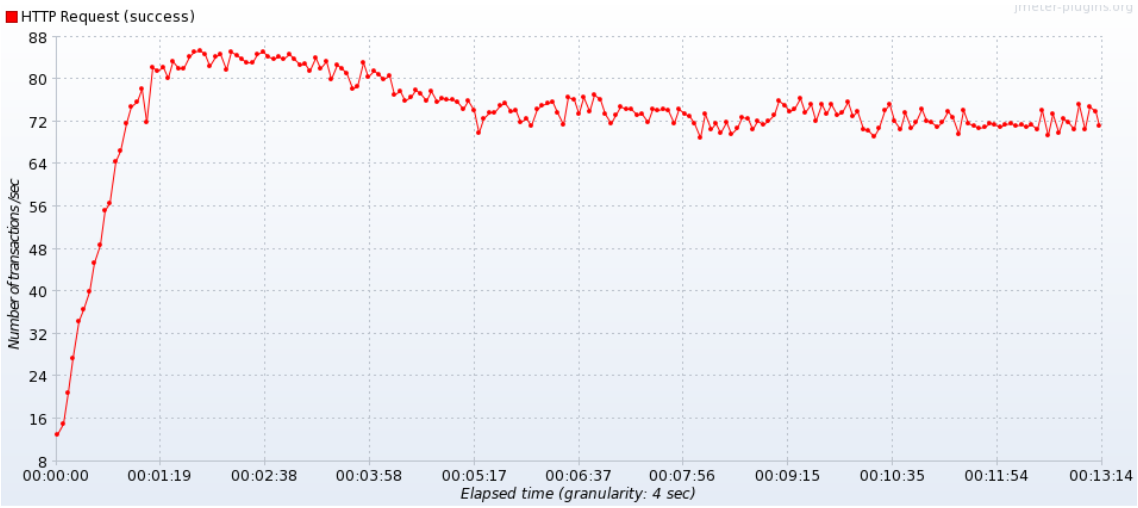

Figure 2.72: VMPhoton - Test 4 - Successfull Transactions Per Second (More is better)

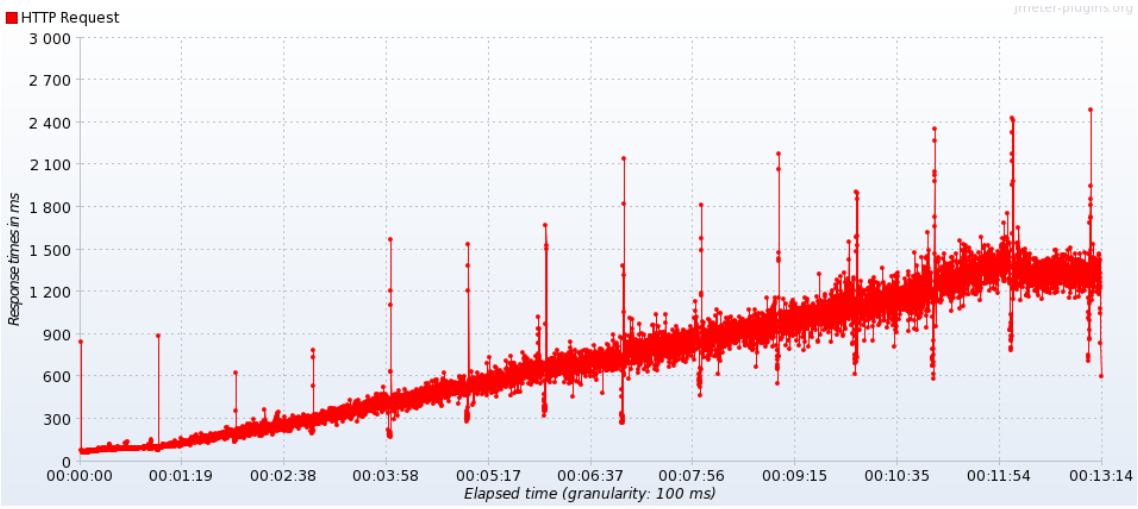

Figure 2.73: VMPhoton - Test 5 - Response Time over Time (Less is better)

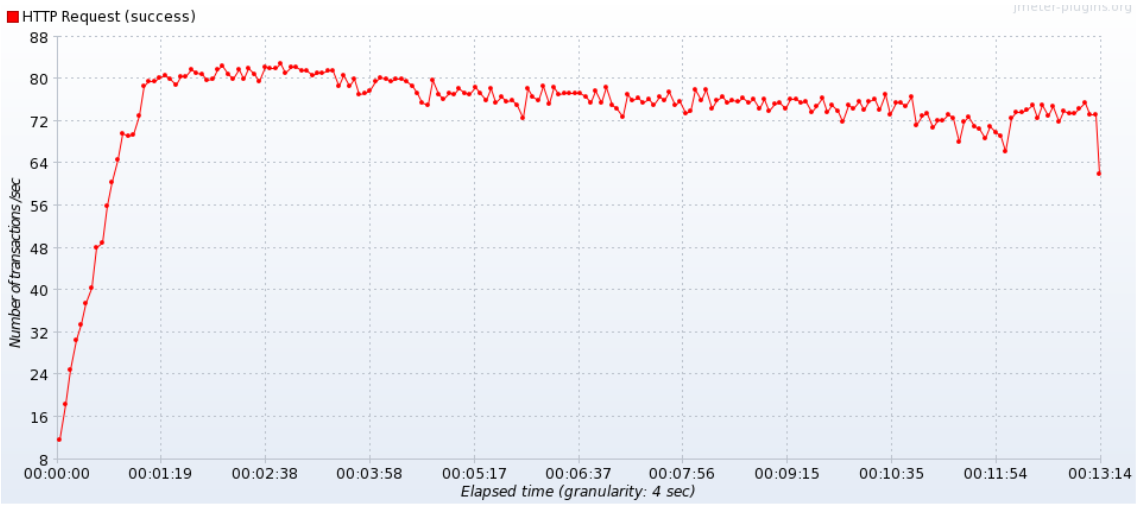

Figure 2.74: VMPhoton - Test 5 - Successfull Transactions Per Second (More is better)

C Appendix C: SQL Query Results

This appendix presents the results from the SQL queries experiments.

SQL-SELECCT, measured in miliseconds.

Run	CentOS	CoreOS	VM CentOS	VM CoreOS	VM Photon
1	0.728	0.453	1.290	1.514	1.885
2	0.521	0.522	2.246	2.042	2.105
3	0.570	0.490	1.242	1.746	1.632
4	0.514	0.505	1.812	1.651	1.249
5	0.505	0.513	0.850	1.815	1.061
Average	0.568	0.497	1.488	1.754	1.586

Table 3.3: Execution times to complete 1 *SQL-SELECT* query shown as a graph. (Less is better

SQL-INSERT, measured in seconds.

Run	CentOS	CoreOS	VM CentOS	VM CoreOS	VM Photon
1	174.197	64.388	145.171	66.479	414.104
2	164.675	64.270	148.839	64.520	355.149
3	161.202	64.835	145.204	65.698	332.823
4	161.954	64.971	141.341	65.789	333.578
5	159.437	65.118	146.912	66.020	333.317
Average	164.293	64.716	145.493	65.701	353.794

Table 3.4: Execution times for 10 000 *SQL-INSERT* queries shown as a graph. (Less is better)