CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Virtualisation of Computer Nodes in Radar-Systems

An Investigation into Performance and Power Overheads of Virtual Layers

Master's thesis in Computer science and engineering

SOFIA WERNER

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Virtualisation of Computer Nodes in Radar-Systems

An Investigation into Performance and Power Overheads of Virtual Layers

SOFIA WERNER

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Virtualisation of Computer Nodes in Radar-Systems
An Investigation into Performance and Power Overheads of Virtual Layers
Sofia Werner

Supervisor: Miquel Pericas, Department of Computer Science and Engineering
Advisor: Fredrik Larsson and Jonas Berg, Saab Surveillance
Examiner: Ioannis Sourdis, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A display of different versions of Saabs surface radar system Giraffe. Taken 2019-05-17 from: https://www.defencetalk.com/saab-receives-uk-orders-for-giraffe-radar-systems-65120/

Typeset in LaTeX
Gothenburg, Sweden 2019

Virtualisation of Computer Nodes in Radar-Systems
An Investigation into Performance and Power Overheads of Virtual Layers
SOFIA WERNER
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis work has investigated the impact and implications a virtual layer could have in a transportable radar system in order to determine the feasibility of implementing such a layer in Saab's radar systems. Today, multiple computing boards are used in the Data Processing unit of these radar systems, but in the future Saab wishes to reduce this number. Before this can be done, it is important to know what implications such a change might have. Investigating the potential performance loss and the behaviour of different virtual layers can provide answers to if and what kind of virtual layer could be used in Saab's systems. This has been done by measuring the performance differences between running an application on a physical machine and running the same application in a virtual layer. Our experiments show that the virtual layer itself does not have a significant impact as long as the virtual layer operates under the same conditions as the physical machine, i.e. it has the same clock frequency the same amount of RAM etc. In addition to measuring performance, we have also made an attempt to investigate the impact the virtual layer has on the energy consumption. Our main findings are that the performance in terms of execution time is barely impacted when running CPU heavy applications in Docker containers while Docker containers have a significant impact on I/O operations. A potential drawback from using Docker containers is that our energy measurements suggest that the container could possibly increase the energy consumption significantly. We also find that a Xen guest could possibly have performance close to the physical machine if both machines are running on CPUs with the same clock frequency. We come to the conclusion that Xen is a feasible tool to use in order to facilitate transition from a structure consisting of multiple physical computing boards to a structure only containing one physical computing board. Our observation is that a virtual layer does not have a significant impact on CPU performance, but the performance of I/O operations is noticeably negatively affected. Energy consumption could only be measured on Docker containers and here we observe an increase in energy consumption on a process level.

Keywords: Virtualisation, radar systems, Docker, Xen, performance.

Acknowledgements

I want to thank my advisors Fredrik Larsson and Jonas Berg at Saab for their help, support and encouragement during this project. Also, I want to thank my supervisor Miquel Pericas and examiner Ioannis Sourdis for their help and guidance. A special thanks goes out to the people on Saab that have in all possible ways helped me during this project. A special thanks goes to my fiance for his support. Finally, I want to show my appreciation to Tobias Isenstierna and Stefan Popovic for their valuable input and insights that have helped me progress.

Sofia Werner, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1

Introduction

Virtualisation was first used about fourty years ago. The purpose at that time was to allow multiple users access to the same mainframe as a way of keeping hardware costs down [28]. Today, virtualisation is widely used for other purposes such as distributing workloads, increased system security, or implementing scaling mechanism in networks among other things. A common application for virtualisation is Cloud Computing. In this application, the virtualisation is used for facilitating scaling depending on the workload. This is achieved by letting the cloud consist of virtual machines (VMs) and provisioning new VMs if the workload demands it [31]. Other common applications are load balancing through moving entire VMs between platforms or run legacy software that is not supported by a new operating system [28]. Saab is currently planing to make major changes to the hardware structure by reducig the number of computing boards. Making this change will require that they also make major changes to the software since the currently used software would otherwise be rendered unusable. Virtualisation could here be used to create multiple virtual computing boards on a single physical board and thus eliminate this problem.

This thesis work has been done in collaboration with Saab AB. Focused on investigating if the software used in their mobile radar-system can be virtualised. Saab's software for data processing has been developed over a long period of time and is customised for hardware that was considered good a decade ago. They now plan to replace the OS and hardware in the data processing unit and thereby also change the hardware structure. The software currently running on this unit has been custom-made for this hardware structure which entails that any changes to this structure would demand making changes to the software. In the best case scenario would the software changes be limited to change some parameters in the installation scripts. In the worst case would be the software needing a complete overhaul. By adding a virtual layer between the hardware and the software, it would be possible to create an environment that looks to the software like the old hardware structure on the new hardware structure. This way, it would be possible to change and test the new hardware structure ahead of making eventual software changes.

Before applying virtualisation in the radar-system, it is necessary to investigate how the performance of software might be affected by this. The radar-system is tracking targets in real time and the software needs therefore to be responsive and not prone to lagging. Another important aspect is the energy consumption of the system. Since the target of this project is a mobile radar-system with a limited power supply, it is important that virtualising the software does not increase the

energy consumption in a way that limits the operation time of the system to a point where it is unacceptable. If the risk of lagging software is deemed too great or the operation time is unacceptably limited due to increased energy consumption, it will not be feasible to use a virtual layer in the proposed manner.

There are many techniques for creating a virtual layer. They are usually divided into two groups, hypervisors and containers. These two groups present different advantages and disadvantages, which entails that most systems would benefit differently depending whether you choose to use a hypervisor or create containers. It therefore becomes necessary to study different technologies in accordance with the target system to find a solution that fits that particular system. For this project we used proxy hardware, the choice for new hardware will be based on the outcome of this project. We have therefore opted for running our experiments on a laptop and focus on how the results change from running on a physical machine to running in a VM or a container. Such a comparison between physical machine and virtual layer will provide an estimation as to how the software will be affected by virtualisation under the assumption that a certain virtual layer has the same effect on software performance regardless of the underlying hardware.

We have researched the possibility of introducing a virtual layer into Saabs surface radar system with the goal of replacing the multiple hardware nodes with only one that can, as a first step, contain the old hardware nodes in virtual form and later on run the software either natively or in a suitable virtual layer. The main argument for using virtualisation is that it will be easier to keep the native OS up to date without the risk of it loosing support for the radar software. Reducing the number of computing cards to one will also save physical space inside the radar shelter.

1.1 Aim

This project aims to investigate the feasibility of introducing a virtual layer into the DP subsystem in Saabs radar systems. A virtual layer is likely to degrade performance but this can be tolerated to a certain extent since Saab is planning to upgrade the currently used hardware. The upgraded hardware is likely to have higher performance than the one that is used in the system today. The loss, however, can not be too great since hardware can only compensate to a certain limit. Knowledge about the impact the virtual layer will have on the software will help to determine the performance requirements of the new hardware.

In addition to investigating performance impact we also study the impact a virtual layer can have on the energy consumption. If the virtual layer proves to have a high impact on the energy consumption, this mean that using a virtual layer in the transportable radar-systems is not feasible.

We also looked at what causes the overhead that is observed when running an application in a virtual layer. Understanding the cause behind the overhead can po-

tentially help finding ways to reduce or eliminate it depending on its characteristics.

1.2 Scope

For this project, we have only looked into using virtualisation to solve the possible incompatibility that might arise when making the proposed structural hardware changes. It is possible that the incompatibility between currently used software and the upgraded hardware can be solved without using virtualisation. However, virtualisation can be beneficial in a longer perspective as well, which is why we want to explore virtualisation as an alternative for facilitating the hardware upgrade.

We have also only focused on how performance and energy consumption could be affected by running applications in a virtual layer instead of on a physical machine. An other potential focus is to chose a certain virtalisation technique and investigate how different configurations to that type of layer impacts the performance and energy consumption. We, however considered it more relevant for us to investigate how different techniques impact the performance and energy consumption before diving into how the virtual layer should be constructed in order to achieve the best possible outcome.

Furthermore, we limit the work to only include measuring the performance when running one VM at a time. When having determined how to best implement the virtual layer, it becomes interesting to investigate how performance is affected when running multiple instances of the same of different VMs.

1.3 Contribution

To the best of our knowledge, this is the first time the cause behind the overhead incurred by virtual layers has been studied. Furthermore, the energy overhead has never been studied. We therefore claim to make the following contributions with this paper:

- We study the performance and energy overheads caused by two different virtualisation technologies: Docker and Xen Project Hypervisor. This is done by using a set of benchmark applications and other tools, such as GNU time.
- When analysing the performance overhead we show that neither Docker nor Xen had any significant impact on the performance of CPU heavy applications. I/O heavy applications on the other hand were significantly impacted by both virtual layers.
- Based on the GNU time output we hypothesise that it is possible that the Docker overhead involves an increase in context switching.
- The analysis we perform on the energy overhead suggests that Docker incurs a significant overhead.

1.4 Structure

The report is organised as follows:

- Chapter 2 introduces topics that are important to our work. Here we introduce the virtualisation techniques we have used as well as how our work relates to work previously done in this area.
- Chapter 3 describes how we used the virtualisation techniques described in Chapter 2 in order to measure the performance and energy consumption. We also describe how we evaluated the techniques.
- Chapter 4 presents the results of our experiments. We also analyse the results in this chapter.
- Chapter 5 provides a discussion of how our results might have been impacted by the chosen methods. We discuss how certain things could have been done differently in order to achieve more reliable results.
- Chapter 6 presents our recommendation on how to introduce a virtual layer into the radar system.
- Chapter 7 concludes the report and presents how our work could be continued.

2
Background

This chapter will provide an introduction to the most important topics that is addressed in this report. We will also use this chapter to highlight why virtualisation is such a hot topic right now. The chapter starts with an overview of where the virtualisation research is today and continues with describing the two largest families of virtualisation technologies from which we have chosen the four virtualisation technologies we investigated during this project. The chapter is concluded with an introduction to the four virtualisation technologies we used.

2.1 Virtualisation technologies

Virtualisation technologies can be divided into two groups: containers and hypervisors. Which technology one should use depends on the software that is intended to be run inside the VM or the container.

2.1.1 Hypervisor

A hypervisor, or virtual machine manager (VMM) is a software layer that mimics the underlying hardware by having the same instruction-set [5]. There are two types of hypervisor, type I and type II. The type I hypervisor, e.g. Microsoft Hyper-V or Project Xen hypervisor [13], is also known as bare metal hypervisor. This type of hypervisor is loaded as part of the system kernel and boots before the OS. It therefore require the kernel to support such kernel modules. Some Linux distributions are shipped with such a kernel, but not all. It is however possible to change kernel in order to gain the required support. A type II hypervisor, e.g. VirtualBox or Java Virtual Machine [2], is similar to a regular application in terms of it can be started after the OS instead of always booting before the OS. Figure 2.1 shows a diagram for a hypervisor II that manages concurrent 3 fully virtualised machines. Full virtualisation means that the VMM runs on the host OS and enables the user to install a complete guest OS inside a virtual machine. The guest OS will not know that it is running on virtual hardware and will be able to interact with physical hardware through the VMM in the same manner as it should if it were running directly on hardware. Though this approach provides the most flexibility since it allows applications to be independent of the host OS, it comes with the disadvantage of a large overhead which could in 2010 result in 30% performance loss [33].

Another approach to virtualisation is paravirtualisation, which is similar to full

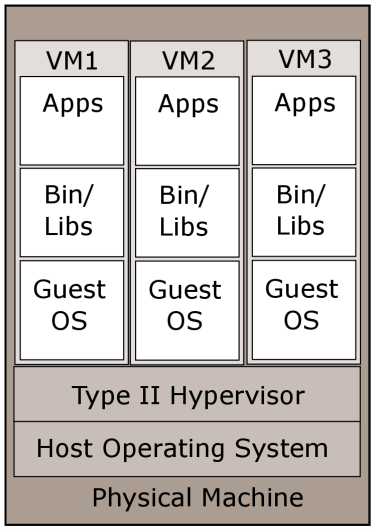


Figure 2.1: This figure shows the Hypervisor type II architecture

virtualisation in terms of both using a hypervisor. The main difference between the two is that a paravirtualised OS is aware of the hypervisor and coordinates operations with it to reduce the number of expensive privileged instructions which in turn increases the performance of the paravirtualised machine above a similar fully virtualised machine [38]. The hypervisor-awareness requires the OS to be modified before running as paravirtualised guest OS. Some Linux distributions, such as Debian, has built-in tools that facilitates creating OS-images that can be used in a paravirtualised machine [41].

2.1.2 Container-based virtualisation

This virtualisation technique is also called OS-layer virtualisation [33]. A container does not contain a full OS image, but rather only an isolated filesystem of a certain OS. Container-based virtualisation are more lightweight than the hypervisor due to this and can thus in most cases provide higher performance than the hypervisor. It is possible to reach performance levels almost on par with the host machine using the container-based virtual-layer [20]. Containerising an application allows for limiting the applications usage of certain resources, such as CPU, disk I/O and memory to mention a few. The container includes all that the application needs to run, i.e. system libraries, settings, code, runtime, system tools. This makes the container able to run on different systems regardless of host OS [11].
Containerising an application can have a number of advantages. For instance, containerising facilitates moving an application from one machine to another. Only requirement would be that the new machine is able to run containers. When containerising an application, the developer often divides it into smaller parts with well-defined functions. These parts will then run together in so called microservices and thus form the application [8].

A typical container life cycle consists of three phases: Image creation, testing and

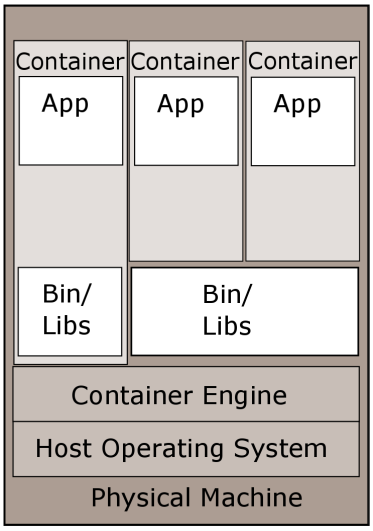


Figure 2.2: This figure shows the container architecture

accreditation; Image storage and retrieval; Container deployment and management. During the first phase are application components created and packaged into images together with something that can prove the credibility of the image, most likely some kind of certificate. The images are then tested in order to ensure correct functionality. In the next phase are the images stored in a registry, either internal or external depending on the purpose of the image. Images can also be pulled from the registry by an orchestrator. In the last phase, the orchestrator deploys copies of images into containers and thus start running instances of the application. The orchestrator can also be used to manage the containers during runtime.

The orchestrator is a tool or multiple tools that a system administrator can use for the above mentioned purposes. This tool adds an abstraction layer that allows an administrator or an other automated tool to simply tell the orchestrator how many containers and of what kind should be started, stopped or what ever is needed to be done.

2.2 Virtualisation technologies used in this project

There exist a multitude of virtualisation techniques and testing all of them would be a too big a task for this projects and we did therefore choose to work with four; two hypervisors and two containers.

The following sections provides a overview of the four virtualisation techniques used in this project. A discussion about why these were chosen instead of any other can be found in Section 3.

2.2.1 Docker

Docker is a container engine that comes in multiple flavours. For this thesis project, the Docker Community Edition (Docker CE) 18.09 used due to it being free and open source as well as been proven to provide relatively good performance while being one of the most prominent container technologies [11, 18].

The container manager makes use of namespaces to isolate processes and create containers [3]. When building an image, the Docker engine instructs the kernel to create a new namespace and install necessary dependencies inside that namespace. What these are is determined by what process will be running in the container. Every process inside the container will be isolated from processes outside of the container and thus be able to e.g. use other software versions than is installed on the host. The containers does not have their own kernel but rather shares the native kernel with each other and the native processes [15]. As default will containers be blocked from directly interacting with anything listed under `/dev/`. If needed can access to a specific resources be given at container startup by using the `-device` flag and specifying which devices the container will visible to the container [1]. Furthermore can the Docker Daemon be used to limit a containers usage of specific resources, such as CPU time. `cgroup`, which is a tool for structuring process in a group hierarchy in order to control access to such resources [7] is used for that purpose.

Docker images are defined using `Dockerfiles`, and how they are built depend largely on what filesystem is being used by the Docker engine [30]. In this project was OverlayFS used since that is the only file system supported by Docker CE. OverlayFS combines filesystems or directory trees into an upper and a lower filesystem, where the lower filesystem can be either another OverlayFS filesystem or an arbitrary other filesystem. If a file is present in both the upper and the lower filesystem will only the version form the upper be visible. If a directory is present in both filesystem will the directories be merged [29]. The Dockerfile describes how the filesystem in the docker container should be built and each line in the file will represent one or more layers to be added to the container image filesystem. The first line of the file either specifies a image or filesystem to base your new image on or tells the engine that you want to build an image from scratch [16]. The rest of the lines defines layers that will be added on top of each other, starting on the base image [30]. Each layer in turn will contain information about which one is its parent, its content and a directory with the combined content of that layer and its parent layer. The building of an image is performed lazily, meaning that the engine reuses layer from cache when ever possible in order to speed up the build process.

2.2.2 Linux Container

Linux Container (LXC) is a Linux native container manager. The container manager allocates a file system representing a given OS for the container and isolates it from the rest of the host. A user can then log into the container as if it was any other machine as well as install applications inside of it. As opposed to the Docker

container is the LXC container built from only one layer and changes made to the container during runtime is automatically saved between sessions. A LXC container can be viewed as a lightweight VM due to it having the same file system as a full OS but is not using any virtualised hardware [23].

The LXC container manager uses namespaces among other things to isolate the container file system for the rest of the OS. LXC supports both privileged and unprivileged containers. A privileged container will have almost the same capabilities as the host while an unprivileged does not. As always is best practice to give out the least possible privileges. An unprivileged container will not be able to access or change anything outside the container. This is possible due to the use of namespaces as well as user and groupID. Processes running inside an unprivileged container might from the containers perspective have 0 as UID and GID, while the host actually have given them IDs above 10000 [22].

The function of a LXC container is like a mix between `chroot` and a VM. `chroot` changes the root directory of a specified process while LXC creates a new root directory that will be the container and is isolated from the rest of the filesystem [9].

2.2.3 Xen Project Hypervisor

Xen Project Hypervisor (Xen) is a Type I Hypervisor, also called bare-metal hypervisor. bare-metal means that the hypervisor is running directly on the hardware instead of inside the OS. This hypervisor is developed by the Xen community as an open source software. The Xen hypervisor provides flexiblity as to hardware configuration by allowing the user to chose size of RAM, vCPUs among other things. The CPU, memory and interrupt managment is handled by he hypervisor in a Xen system.

Xen is installed as a part of the kernel and thus requires the kernel to have support for this, all Linux kernels does not. The hypervisor will boot at start up and simultaniously start the first VM which is the control domain, typically referred to as Dom0. This is the only VM that will have privileges to access hardware. From Dom0 can other guests or domains be started. These will be unprivileged and are typically referred to as DomUs. Each DomU will be running their own full OS and will be independent from each other [39]. Quick Emulator (QEMU) is used for virtualising hardware devices that is used by the DomUs. Each DomU will have their own device drivers and a call to these drivers will incur a call to QEMU on Dom0 which in turn will make a call to the device drivers in the Dom0 kernel, see Figure 2.3.
When setting up the Xen system one needs to allocate and mount a volume that can be used by Dom0 while leaving enough storage space for the other guests. Each VM in the Xen system will have their own Logical Volume where their OS and applications will be installed. The DomUs can be accessed from Dom0 either by secure shell or by using a graphical desktop sharing system such as VNC.

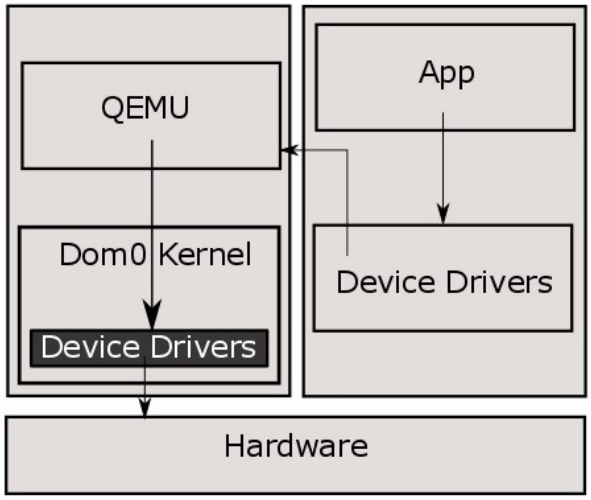


Figure 2.3: This figure shows how a hardware call is handled in a Xen system. The arrows shows the path the call takes before reaching the hardware.

2.2.4 Kernel-based Virtual Machine

Kernel-based Virtual Machine (KVM) is a Linux native virtual machine manager which provides means for creating fully virtualised VMs. It is a type II hypervisor, meaning that the hypervisor runs inside the OS. The kernel component has been shipped with mainline Linux kernels since Linux kernel version 2.6.20 [19]. The hypervisor QEMU to emulate hardware and the userspace component of KVM has been shipped with QEMU since QEMU 1.3.

KVM can be installed and run on the host much like any other Linux application. A guest OS can be installed inside a virtual machine in the same manner as it would have been installed on a host machine.

2.3 Related work

As mentioned in Section 1 has most research on virtualisation been directed towards uses in cloud computing and data centres. Tseng et al. proposes hypervisors and containers can be used together in a fog platform [36]. The fog platform can be viewed as an extension or a complement to a cloud platform [4]. The services in a fog network is most often located at the edge of the network in order to provide low latency. Typical applications in a fog network are streaming and other real-time applications. The network function virtualisation infrastructure (NFVI) described by Tseng et al. consists of a VM that runs Docker Engine. Functions such as load balancing and orchestration are virtualised as virtual network functions (VNFs). Furthermore, they use orchestrator-agents that take over some of the tasks that the orchestrator normally would have, thus reducing the loading the orchestrator. All monitoring of the virtual private clouds in the fog network is done by the agents, while the decision making is left to the orchestrator. This platform is used to test

the authors fuzzy-based real-time autoscaling (FRAS) mechanism that is supposed to help increase the availability of the services in the fog. The FRAS mechanism uses the data collected by the orchestrator-agents during a time period to calculate how much resources will be needed the next time period. The number of deployed VNFs will be increased or decreased according to these calculations.

The performance of containers and VMs has been extensively investigated. Kozhirbayev and Sinnott studied the CPU, memory and I/O device performance when using Docker or Flockport LXC [18]. Flockport is a tool that is used distributing LXC containers and facilitated the utilization of these conainers. Kozhirbayev and Sinnott present comparisons between running different benchmarks natively, in Docker containers and Flockport containers. They conclude that which of these two virtualisation technologies should be used in any one system depends on the software and requirements in that system. Their experiments showed that there was just a small or no overhead in using a container when running CPU or memory intensive applications, but interactions with OS or I/O operations had a more noticeable overhead.

Another performance study was performed by Li et al. [20]. This study focused on comparing computation, memory and storage overhead in containers and VMs. In their experiments they used Docker containers and VMware Workstation to create a standalone virtual machine. They used different benchmark applications such as Bonnie++ and STREAM to measure the different overheads. They concluded that the overhead with regard to memory could be ignored for containers due to the observed overhead was in their margin of observational error, while the overhead for the VM was approximatly 4%. When regarding computational overhead, it depended on what kind of resource, CPU or floating point unit (FPU), was being used to perform the computation. They showed that the overhead was generally lower when the CPU was used and higher when using the FPU. Also in this case had the container in general a smaller overhead than the VM. They calculated the storage overhead based on the number of disk tests that could be finished in 24 hours by the physical machine, the container and the VM. These calculations also showed that the container incurred a smaller overhead than the VM.

The study conducted by Tseng et al. looked into usage of virtualisation while Kozhirbayev and Sinnott and Li et al. studied the performance of different virtualisaiton techniques, but neither of them have have looked into the reason for the overhead incurred by the virtual layer. In our study, we will attempt to provide some insight to the reason for the overhead. Furthermore, we have not found any research into potential increases in energy consumption caused by the use of virtual layers, which is an important aspect for Saabs mobile surface radar systems. We will therefore extend the study to include this in addition to studying performance overheads.

3
Methods

We choose to investigate two hypervisors and two container technologies in order to have the possibility to compare both implementations of the same type and compare implementations that are quite different. This chapter describes how the experiments in this project was conducted and how the evaluation was done.

3.1 Current System Setup

The computer systems inside the radar systems can be divided into subsystems, of which we will focus on the Data Processing subsystem (DP).

The DP subsystem consists of multiple connected so called data processing boards (DP-boards), solid state drives (SSDs), network cards, a switch and cards for handling I/O. One of the DP-cards serves as a master node at start up and each application is allocated to a node at startup and will run on that node until shutdown. The allocation is defined in a startup script. The system also includes some peripheral hardware such as solid state drives (SSDs) for installing applications and operating system, switches and cards handling I/O.

The system in use today consists of a number of computer boards running multiple applications in parallel. The system designers chose this design since there did not exist any cost-efficient hardware that offered enough performance for running all applications on the same card. The performance of CPU and memory has today increased to such an extent that it is possible to run all applications concurrently on the same hardware and thus reducing the hardware cost.

3.2 Experimental setup

We used a Dell Precision M4800 as a baseline system for our experiments. The four chosen virtualisation techniques was implemented one by one on the laptop and in between the laptop was reset in order to create as similar starting conditions as possible for the different layers. Published literature was studied to select the four technologies to use as a starting point for our study of virtualisation. These four was screened by using three suites from Phoronix Test Suite (PTS) that aimed to measure CPU, memory and disk performance. Out of these four was two, one container and one hypervisor, subjected to further testing where we measured energy consumption and tried to determine the cause for overheads. The second selection

was done after running benchmark applications from PTS both with and without virtualisation. For the second run of benchmarking seven different tools were used:

- pTop. A tools for measuring energy consumption on process-level.
- GNU time. A tools for measuring resources used by a process.
- STREAM. Benchmarking application used for measuring memory data throughput.
- IOzone. Benchmarking application used for measuring I/O speed.
- John the Ripper. A decryption tool.
- ping. A tool for measuring e.g. network latency.
- iperf3. A tool for measuring e.g. network bandwidth and jitter.

All of the above listed applications and tools was run both on the physical machine and in virtual layers. The percentage difference between native and virtual were then calculated. We assume here that the virtual layers have the same effect on other platforms than the one used for our experiments in order to be able to draw conclusions about how the Data Processing (DP) software would be affected by these layers.

3.3 Criteria for first technology selection

Four virtualisation technologies were chosen after the following criteria:

- Availability for CentOS. Saab has decided to change OS in their radar-systems and therefore need the virtualisation technology be available for this OS.
- Perceived popularity. Evaluation of this criteria will be subjectively based on how often certain technologies are mentioned in the studied literature. Ease of use will also be to some extent considered but is very likely to be influenced by our own preferences.
- Licence costs. It be preferable to use open source technologies during this project in order to avoid unnecessary license costs.
- Availability of commercial support. It would be preferable if there are commercial support for the investigated technologies since one of the reasons for this project is to investigate possibilities for long term use of virtualisation.

The four chosen virtualisation techniques were then subject to an initial screening using synthetic applications that aims at stressing certain parts of the system. The results of this screening form the basis for a second selection where we chose one container and one hypervisor to priorities when we continued with study performance in more detail.

Details about the benchmark evaluation process will be provided in this chapter along with descriptions of how the different technologies were chosen and how they are deployed.

Other virtualisation technologies then the four presented here, such as VMware vSphere, were considered but discarded since they did not fulfil all four criteria. In the case of vSphere was only a 60 day free trial available before one needed to purchase the software [37].

3.4 Setup of virtual layers

Each virtual layer was installed separately on the lab laptop and the native operating system was reinstalled between virtual layers in order to make the conditions for the layers as equal as possible. CentOS 7 with 3.10.0-957.5.1.el7.x86_64 kernel was used as host OS for all virtual layers, except Xen which required a different kernel since the kernel that ships with CentOS 7 does not provide support for Xen.

3.4.1 Docker

Two different strategies was used when creating the images for the Docker containers that we used. The first strategy was to write Dockerfiles describing the image. The second strategy was to create a simple container that just contained a filesystem using a Dockerfile, then install the necessary applications in the running container and then commiting the changes to a different image using `docker container commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]` where `CONTAINER` is the running container and `TAG` is an existing image you want to change or a new Docker image.

The first strategy was used when creating the images for running the PTS suites. In total was six images used:

1. Base image: An image containing only the filesystem and its accompanying libraries and binaries such as `yum`.
2. Library image: An image with libraries and binaries that we find useful no matter what the container is supposed to be used for, e.g. `vim`. This image uses the base image as a base.
3. PTS image: An image with PTS without and test suites installed. This image has the library image as a base.
4. CPU test suite image: An image based in the PTS image where the CPU suite has been installed.
5. CPU test suite image: An image based in the PTS image where the memory suite has been installed.
6. Disk test suite image: An image based in the PTS image where the disk suite has been installed.

An illustration of this image hierarchy can be found in Figure 3.1.
The first strategy can be a bit cumbersome to use if the container is used for some kind of software development since changes done to content in the container will be discarded when the container stops, unless the container is committed to an image before shutdown. Committing a container to an image is also necessary in order to be able to push the image to a docker image repository. Images can be pushed and pulled to and from such repositories when more then one developer is working together on the same container project. The Docker engine can automatically search repositories if it is missing an image locally when building a new image.

We used the second strategy when running IOzone, John the Ripper and STREAM in Docker containers. For this we used the Library image as a base and installed one of the benchmark applications in it before committing the modified container

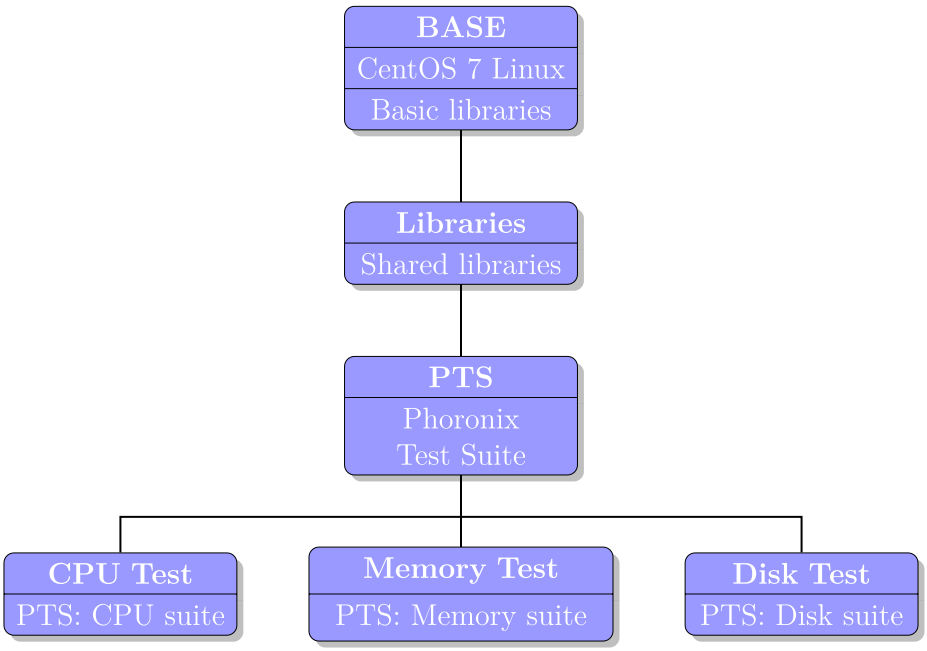


Figure 3.1: Graph showing a visual representation of the container configuration. Each box represents a container and each container inherits the contents from the one above it in the hierarchy.

to a new image and stopping the Library container. This process was repeated for each benchmark application we planned to use.

Container configuration

We used Docker CE 18.09 during our experiments. The container used for running benchmark applications was constructed using a hierarchy of five containers, see figure 3.1. These five containers could have been combined into only one container, but we chose to split it up in order to decrease the complexity of each container even though all five of them was relatively simple. The reason for dividing the test environment into multiple containers was to learn more about structure and configure a Docker container hierarchy. Dividing makes the environment modular and minimises the risk of dependency conflicts.

Our hierarchy consists of one base image that holds a minimal installation of CentOS 7 Linux along with libraries that is installed together with the operating system. Next image contains PTS and its dependencies. This image is based on the previous one and will therefore inherit the OS installed in it. The next three images inherit the content of the PTS-container and will therefore not have to install PTS. These three containers contains the test suites we want to run, one suite in each container.

3.4.2 Xen Project Hypervisor

We had to do the installation process of Xen Project Hypervisor (Xen) a bit differently since we were running CentOS 7 Linux. To get support for running a Dom0 on CentOS was a 4.9.127-32.el7.x86_64 Linux kernel installed and used. This kernel has been configured for this purpose and can be found in the Centos-Extras repository [40]. We used Xen 4.8 during our experiments.

Installation

Can not create all volumes at OS installation. During installation you need to only create 3 partitions `/` `/boot` and `swap`. Use `fdisk` to create a partition that contains the rest of the physical disk space and then use `lvresize` to expand the volume group to include this space.

Domain configuration

The default domain was not reconfigured even though it is possible to do so in order to trim the machine to increase performance. We chose to leave it as it was to, once again, keep the impact of other things than the virtual layer as low as possible. Though this is a choice we have to keep in mind when analysing the result from the benchmarking tests. The guest domain, DomU_1, was configured to be as similar to the Dell laptop as possible in order to reduce performance differences caused by hardware resource differences.

3.4.3 LXC

The latest LXC manager can be installed directly from a CentOS repository using the package manager `yum`. In our case was LXC 3.0.3 the latest version. The package `lxc-templates` is also needed if one does not wish to define their own container template.

Container configuration

We decided to use the CentOS template that was provided by this package since we could not find any guide we could follow to write our own. Getting the LXC container up and running was therefore a quick and easy job. As described earlier could PTS be installed inside the container after connecting a console to it.

3.4.4 KVM

KVM 2.10.0 was installed on the laptop from a CentOS repository using `yum`. Unlike when installing Xen did we not need to install a new kernel in order to gain support for running KVM.

VM configuration

We used virt-manager to configure the guest to be as similar to the Xen guest as possible. The tool virt-manager allows the user to through a GUI define how many CPU cores the VM will have as well as the amount of RAM and disk storage. After this was the procedure for running the benchmark suites identical to the procedure for Xen.

3.5 Measuring performance and energy consumption

We used six different benchmark applications or performance measuring tools to determine what technologies might be suitable to implement in the radar systems. The first one, Phoronix Testing Suite, was used to compare the four chosen technologies against each other in order to determine which two to focus on in the next step. The reports from PTS are not limited to the results of the different benchmarks included in a suite, they also include a description of the syste on which the suite was run. Table 3.1 shows the system report after running a suite on our physical machine.

Table 3.1: This table shows the specifications of the Dell Precision M4800 laptop that was used for scoring the virtualisation techniques. This is our baseline system.

Processor	Intel Core i7-4700MQ @ 3.40GHz (4 Cores/8 Threads)
Motherboard	Dell 0FVDR2 (A16 BIOS)
Chipset	Intel XEON E3-1200 v3/4th
Memory	4 x 4096 MB DDR3-1600MHz HMT451S6AFR8A-PB
Disk	256GB LITEONIT LCS_256
Graphics	Intel Haswell Mobile 2048MB (1150MHz)
Audio	Intel XEON E3-1200 v3/4th
Network	Intel Connection I217-LM + Intel Centrino Ultimate-N 6300
OS	CentOS Linux 7
Kernel	3.10.0-862.14.4.el7.x86_64 (x86_64)
Desktop	GNOME Shell 3.25.4
Display Server	X Server 1.19.5
Display Driver	modesetting 1.19.5
OpenGL	4.5 Mesa 17.2.3
Compiler	GCC 4.8.5 20150623
File System	xfs
Resolution	1920x1080

The other to applications that we used after PTS were, John the Ripper (ripper), STREAM, IOzone, iperf and ping. The first three of the previously mentioned applications (ripper, STREAM and IOzone) were run together with a tool called GNU time.

We were also given access to the data processing applications (DP applications)

in order to perform the same measurements on those as we did on ripper, STREAM and IOzone. This could only be done with the applications running in a testing lab. The reports from these runs were compared to the reports of the other applications to get an estimate of how the DP applications would behave in a virtual layer. The time frame and some security hurdles made it impossible to run the DP applications in a VM or container set up by us.

Ripper, STREAM and IOzone were chosen after talking to people from the teams developing the DP applications about which parts of the system are most critical to the performance of the DP applications. These three applications (ripper, STREAM and IOzone) can be used to put stress on the CPU, the memory and the storage systems which according to the people we talked to are important for the performance. Network latency was another important factor and ping is a good tool for measuring this, while iperf was recommended to use for measuring network performance.

The GNU time application can be used to run another application and afterwards provide a list of the resources used as output [35]. We used this application to measure execution time, RSS, page faults, context switches, file system I/O and page size. The time application reports more statistics than is presented here in this report [25], but we decided to omit some metrics from the output of `time` since they were reported as 0.

When considering energy consumption, data centres (where most previous research on virtualisation has been conducted) and radar systems do differ in some important aspects. Access to power is one such aspect. It is important in both cases to keep the power dissipation as low as possible to keep temperature down and in the radar case will a low energy consumption prolong the operation time. The radar systems are mobile and need therefore to carry fuel that power a generator that provides the radar system with energy. It is important to study the impact a virtual layer has on the power dissipation since a certain type of layer might not be usable if it increases the energy consumption in such a way that it has a significant impact on the operation time.

3.5.1 pTop

The energy consumed by the benchmark processes was measured using pTop, which is marketed as a processes-level power profiling tool [14]. This application runs concurrently with the processes whose energy consumption you want to investigate and the output from it can be used to calculate the power consumed by that process. pTop uses information found in `/proc` to calculate the energy consumed by a process. The results are presented to the user in realtime divided into CPU, disk and memory energy consumption.

A few additional kernel modules was necessary in order to use pTop, such as `cpufreq_stats` and `acpi_cpufreq`. The service `cpuspeed` was used by the kernel to collect the time the cpu cores spent in different frequency states. This service

was not part of any repository know to us and therefore needed to be downloaded from a third-party website [10].

The output from the application is based on calculations done using parameters the user needs to set in the source code beforehand. These parameters include, but are not limited to, L2 cache latency, CPU energy consumption and cache line size. the application calculates the energy consumption of a process by monitoring the time the CPU spends in different frequency states, number of cache misses combined with the time it takes to resolve a miss and the time spent on reading or writing from and to storage [14]. The accuracy of the output is therefore heavily reliant on the user entering the correct values for CPU power etc. We deemed this to not be a major issue since we are most interested in the difference between the energy consumption of the processes with and without a virtual layer. We assume that the results will be reliable enough for our purposes as long as we use the same parameters for each test.

pTop calculates the energy consumption once every three seconds and stores the results in a database. To extract the numbers we are interested in, we query the database for the relevant entries using the process ID (pid) belonging to the application in question. Since pTop operates on a process level, it is necessary to include the energy consumed by the container orchestrator or the hypervisor when making the comparison between native and virtual layer runs.

We made some small changes to the source-code, in addition to those that was mentioned above. These changes consisted of instructing the program to store the calculated energy in the table `process_energy` in the accompanied database. The application developers had prepared functions that did this, but they were not used by the actual program. We have not been able to find an explanation as to why these functions was not used in the application that had been published. A guess would be that an old version had been uploaded, maybe by mistake.

3.6 Benchmark applications and tools

This section presents the different tools and benchmark applications that we used to perform the performance and energy measurements.

3.6.1 Phoronix test suite

The Phoronix Test Suite (PTS) is an open source set of benchmark suites. The suites are stored in a remote repository and can be downloaded to the machine a user wish to run it on. It is easy to use and compatible with multiple Linux distributions including CentOS 7, which is why we chose to use PTS to gather benchmark scores for our configurations. This section will describe in more detail what is needed to run PTS.

Phoronix Test Suite is available from the EPEL repository as well as the developers website [32]. The benchmark applications used by PTS is available from

`www.openbenchmark.org`.

We chose to run three suites from PTS, one that measures CPU performance, one that measures memory performance and one that measures disk performance. These three suites were considered to be a good starting point for us when evaluating the virtual layers. The results we got from the PTS runs formed the basis on which we chose what techniques to study in more detail. PTS presents the system setup after each run in addition to printing the results.

The suites are configured to run most of the tests 3 times and afterwards present an average for these runs as a test result afterwards. We decided to only run each suite once since the individual tests where automatically run multiple times when using the suite. A couple of tests from the CPU suite did not run properly and was therefore disregarded.

All three test suites was run both in a virtual environment and on our baseline system which consisted of the Dell laptop, after which we calculated the percentage difference between the baseline test results and the respective test results from each virtual environment, which resulted in 4 percentage differences for each individual test in the three suites. We then used these percentages to comprise an overall impact average, positive impact average and negative impact average for each test suite and virtualisation technology. We define positive impact as when a test run in a virtual layer scores higher on a ”higher is better”-test (HiB) or lower on a ”lower is better”-test (LiB) than when the same test is run on the physical baseline machine. Negative impact is defined as the opposite of positive impact. The positive impact average is the average calculated by using only the results of the tests where the virtual layer had a positive impact and the negative impact average is the same but using the results where the virtual layer had a negative impact. For the overall impact average is all the test results included without considering if they are negative or positive.

$$\sum_{suite} \frac{(\frac{VL_{score}}{Native_{score}})}{N_{suite}} \tag{3.1}$$

$$\sum_{suite} \frac{(\frac{(PosetiveVL_{score}}{Native_{score}})}{N_{suite}} \tag{3.2}$$

$$\sum_{suite} \frac{(\frac{NegativeVL_{score}}{Native_{score}})}{N_{suite}} \tag{3.3}$$

We calculated these three averages for each virtualisation technique and used them to compare the impact the four different virtualisation technologies had on the overall system performance.

3.6.2 STREAM

The STREAM benchmarking application is used to measure the memory bandwidth in a computer system by measuring the performance of four long vector calculations, including copying of the vector, scaling the vector and adding a value to each element in the vector [27]. This application is also included in two of the three PTS suites we used. We chose to use this benchmark again since it despite being relatively old (it was published in 1995) is still used today in its original form or in an augmented form to measure memory bandwidth [26, 12].

We downloaded the source code for STREAM from the STREAM website and compiled it in 10 versions [34]. The different versions execute the four vector operations 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 times before reporting the best memory bandwidth, which is calculated using the shortest execution time, excluding the first execution. We ran all different 10 version once consecutively for the energy measurement, while we only used the 50 version and ran it 10 times for the performance measurement.

The size of the vectors used in the operations are set at compile time. We configured them to be 100 million elements long, where each element is 8 bytes in order to conform to instructions for STREAM:

- The array size should be roughly 4 times the system cache size.
- The execution time should be at least 20 system clock ticks.

Our chosen array size satisfy these requirements while having some margin.

3.6.3 IOzone benchmark

The radars are continuously generating a huge amount of data each second and even though the data size will be, to some extent, reduced during the signal processing, the amount of data will still be large when the radar signals reaches the data processing stage. The data processing is not strictly done in real time, which means that the data can be pushed to and fetched from disk when needed instead of being stored in memory.

IOzone benchmark applications allows the user to configure test cases for disk reading and writing files to disk. The user can either specify the size of the file to be written exactly or define an interval. The application will divide the file into records and write the file to disk record by record. The record size can also be defined in the same ways as the file size. IOzone also allow the user to specify what kinds of tests he or she wants to run to measure read and write speed. These tests include, but are not limited to:

- Measuring read speed by
  - reading a file from disk
  - reading random locations on disk
- Measuring write speed
  - writing a file to disk
  - writing to random locations on disk

We constructed two different test cases in IOzone, one with big files to get longer read and writes when measuring energy consumption and one with a fixed files size when measuring page faults, context switching, write speed and read speed. These test cases is summarised in Table 3.2.

Table 3.2: This table shows a summary of the two test cases that was used to measure disk energy consumption and disk performance. Test case 1 was used for the former and Test case 2 was used for the latter. Test case 1 was run 1 time on the physical machine and in a Docker container while Test case 2 was run 10 times on the physical machine, in a Docker container and on a Xen guest.

	Test case 1	Test case 2
Tests	read, re-read, write, re-write, random read, stride read, random write	read, re-read, write, re-write, random read, stride read, random write
File size(s)	1, 2, 4, 8 GB	25 MB
Record sizes in	1024, 2048, 4096, 8192, 16384 kB	1024, 2048, 4096, 8192, 16384 kB

All tests listed in both test cases were run with all possible combinations of file and record size. IOzone loops through all tests on one combination of file and record size before changing the record size. After it has looped through all record sizes with one file size is the file size increased. We used a flag telling IOzone to flush all contents to disk when we ran our tests.

The data output from IOzone quickly grows into being almost overwhelming and we chose due to the limited time frame of the project to somewhat limit the data we analysed. When analysing the data collected by running Test case 2 we chose to only look at the tests configurations that on average reported the highest speeds when run on the physical machine. This way we would look at the configuration that had the optimal record size for reading or writing a 25 MB file on our file system.

3.6.4 John the Ripper

John the Ripper is a decryption application and will therefore use the CPU heavily. This test is included in the PTS CPU suite, but could not run properly in the suite. We were able to obtain information about memory and CPU usage by running John the Ripper from the GNU application time, that is, not the built-in function called time.

As input to to the decryption program, we used an unshadowed copy of the password file present on the laptop. This file contained 2 password, one root password and one user password. Both passwords were the same, six characters long and can be found using a wordlist, meaning they are real human words. We used the default wordlist and settings for John the Ripper.

The application was run both natively and in a virtual layer 10 times each from

which we calculated an average for each metric provided by time, the same unshadowed password file was used each time. The two sets of 10 John the Ripper executions inside a virtual layer and natively was both run in sequence and concurrently/in parallel to investigate if having an active container that puts pressure on the CPU will affect a native process. We were also interested in seeing if there was any differences in number of page faults, CPU time and system time among other things.

3.6.5 Network performance

Network latency, packet loss and jitter are three factors that are important in the radar-system. We measured the latency by using ping and we used iper3 to measure jitter and packet loss. These three characteristics were measured on five different network routes:

- Physical machine to physical machine
- Physical machine to Docker container
- Docker container to Docker container
- Physical machine to Xen guest
- Xen guest to Xen guest

Both applications were run for 30 seconds. Iperf3 was configured to send UDP traffic and measure jitter and packet loss once every second. The goal bandwidth was set the same data rate that is used in the surface radar systems.

3.6.6 Data Processing applications

These are some of the applications that are used in some of Saabs surface radar systems. We can not give any details about the behaviour of the data processing applications due to confidentiality. We will use GNU time to gather the same data as we do on ripper, STREAM and IOzone. This will allow us to estimate how the DP applications will behave in a virtual environment.

We can only run the DP applications in a testing lab. This lab is not connected to the internet and installation of third party software is restricted which results in it being impossible for us to install Docker engine in the lab. Instead, we will use our observations from the measurement's on the three other applications to estimate the behaviour of the DP applications in Docker containers and Xen guests.

4
Results

This section will describe how all tests that were performed and why those tests were chosen. Phoronix Test Suite (PTS) was used to create a benchmarking score for each virtualisation technique. These scores were then used to decided which two of the four would be our focus when running the rest of the benchmarks. We will refer to this as the initial screening.

We have focused on the differences between running an application directly in the OS and running the same application in a virtual layer since our experiment setup was not the system that is used in the radar products. This chapter contains the results from the benchmark tests (Sections 4.1, 4.2 and 4.3), an analysis of the energy consumption (Section 4.4) and lastly a summary of the most important results and observations 4.5.

4.1 Initial screening

The initial screening had two purposes; we used this to learn how use the different virtualisation techniques as well as doing a performance screening by running three different benchmark suites from PTS in each one of them. Table 3.1 shows the specifications of the laptop that was used. The subsections relating to the different virtualisation technologies includes a table showing the system setup PTS reported after running a test suite. We have included all information about the system reported by PTS even if it might not be relevant. A clean reinstall of the OS was performed when finishing testing one virtualisation technique.

The following four sections will show the system setup reported by PTS for each virtual layer as well as give a brief overview of the results from running the PTS suites. The analysis of the results will be provided in section 4.2.

4.1.1 Docker

This technology proved to be flexible and easy to use for modifying containers. One drawback was the fact that persistent data need to be placed in a location that is configured to hold persistent data. This can be inconvenient in cases where data need to be saved between runs. On the other hand, not saving data between runs will save disk space since Docker engine will not allocate any storing space for the VMs if not explicitly told to do so. The system information reported by PTS from

running the program inside docker containers can be seen in Table 4.1.

Table 4.1: This table shows the specifications of the virtual Docker machine as it was reported by PTS.

Processor	Intel Core i7-4700MQ @ 3.40GHz (4 Cores/ 8 Threads)
Motherboard	Dell 0FVDR2 (A16 BIOS)
Memory	16384 MB
Disk	256GB LITEONIT LCS-256
Graphics	inteldrmfb (1150MHz)
OS	CentOS Linux 7
Kernel	3.10.0-862.el7.x86_64 (x86_64)
Compiler	GCC 4.8.5 20150623
File System	overlayfs
Screen Resolution	1920x1080
System Layer	Docker

The Docker container performed quite close to the native machine when running the CPU and memory suites, which can be seen in Figure 4.2a. From what we read in the literature was this to be expected. The impact on the disk performance on the other hand was much higher when considering both negative and positive impact.This was also in accordance with the studied literature since previous studies have found that Docker containers can in some cases have a large impact on the disk performance relative to CPU and memory performance [20].

4.1.2 LXC

In some ways was LXC easier to use than Docker since templates from containers based on several common Linux distributions are available through the package `lxc-templates`. It is possible to create your own customised LXC images by creating a root file-system and writing a metadata-file [24]. We found this less intuitive than writing a Dockerfile and more error prone when done by an inexperienced user, which is why we chose to use the CentOS template from `lxc-templates` package instead of writing our own LXC template. The system information reported by PTS from running the program inside docker containers can be seen in Table 4.2.
Also LXC performed quite close to the native machine on the CPU and memory tests. This was to be expected since LXC like Docker is a container technology. When considering CPU and memory did Docker perform slightly better than LXC, maybe due to us being able to make the Docker container customised for running PTS while not doing the same for LXC. Larger differences observed in the disk suite run when comparing LXC to the native machine. When comparing Docker and LXC did the figures differ quite a lot.

4.1.3 Xen

We used a CentOS dom0 with one CentOS guest domain in our benchmarking configuration. The guest was fully virtualised in order to eliminate result differences

Table 4.2: This table shows the specifications of the virtual LXC machine.

Processor	Intel Core i7-4700MQ @ 3.40GHz (4 Cores/ 8 Threads)
Motherboard	Dell 0FVDR2 (A16 BIOS)
Memory	16384 MB
Disk	256GB LITEONIT LCS-256
Graphics	inteldrmfb (1150MHz)
OS	CentOS Linux 7
Kernel	3.10.0-957.1.3.el7.x86_64 (x86_64)
Compiler	GCC 4.8.5 20150623
File System	xfs
Screen Resolution	1920x1080
System Layer	lxc

stemming from differences in operating systems. Running the guest fully virtualised was necessary due to CentOS 7 does not support running as a paravirtualised guest [17].

Installing the Xen Project Hypervisor required a complete reinstall of the OS as well as installing a new kernel. The user was also required to prepare disk partitions in such a way the it would be possible to have multiple logical volumes on the same physical disk. Each logical volume holds one guest domain (domU) or the dom0. A domU is configured in a configuration file where the user can specify number of cores, RAM, videomemory etc. The system information reported by PTS from running the program inside docker containers can be seen in Table 4.3.

Table 4.3: This table shows the specifications of the virtual Xen machine as it was reported by PTS.

Processor	Intel Core i7-4700MQ @ 2.39GHz (8 Cores)
Motherboard	Xen HVM domU (4.8.4.43.ge52ec4b7 BIOS)
Chipset	Intel 440FX-82441FX PMC
Memory	1 x 6144 MB
Disk	17GB
System Layer	Xen HVM domU 4.8.4.43.ge52ec4b7
OS	CentOS Linux 7
Kernel	3.10.0-862.14.4.el7.x86_64 (x86_64)
Compiler	GCC 4.8.5 20150623
File System	xfs

The Xen VM had overall worse performance than the baseline system. This was to be expected since a hypervisor adds some overhead during runtime and the guests had less memory than the baseline system.

4.1.4 KVM

We used CentOS 7 on the KVM machine as well. Once the hypervisor is installed and a guest is created, you can install any OS of your choice on that guest. We could use the KVM guest in the exact same way as we could the Xen guest when it was up and running.

Table 4.4: This table shows the specifications of the virtual KVM machine as it was reported by PTS.

Processor	8 x Westmere E56xx/L56xx/X56xx (Nehalem-C) (8 Cores)
Motherboard	Red Hat KVM (0.5.1 BIOS)
Chipset	Intel 440FX 82441FX PMC
Memory	1 x 6144 MB
Disk	17GB
Graphics	Cirrus Logic GD 5446
Network	Red Hat Virtio device
OS	CentOS Linux 7
Kernel	3.10.0-957.1.3.el7.x86_64 (x86_64)
Compiler	GCC 4.8.5 20150623
File System	xfs
Screen Resolution	1024x768
System Layer	KVM

The performance of the KVM machine was in some cases higher than the native machine but in most cases was it lower than the native machine. The impact on the performance varied most when running the suites in this virtual layer.

4.2 Phoronix Test Suite

The PTS was only used to make an initial assessment of the four virtualisation techniques. We ran in total 44 unique tests, distributed over 3 test suites. These suites were meant to measure the performance of the CPU and the memory- and disk systems. The numbers by themselves are not really interesting, but the difference between scores form virtual layers and the native environment is. We used the equations 3.1-3.3 to compare the results from the different suites on the different system configurations. The results of these calculations can be seen in Figure 4.1.

Some of the results from the PTS test runs might have been skewed due to having been run by different versions of PTS. The latest release of PTS was PTS 8.2 when we started the benchmark tests, but before we had finished PTS 8.4 was released. Test installation started to fail after the update and we were therefore forced to update to the new version in order to continue. One example of where the two versions differ from each other is in `smallpt`-test which is included in the CPU suite. The developers added the `-O3` flag when compiling the source code for smallpt test in the newer version of the test suite program. This might have influenced the results as this flag determines the level of optimisation to use at compile time.

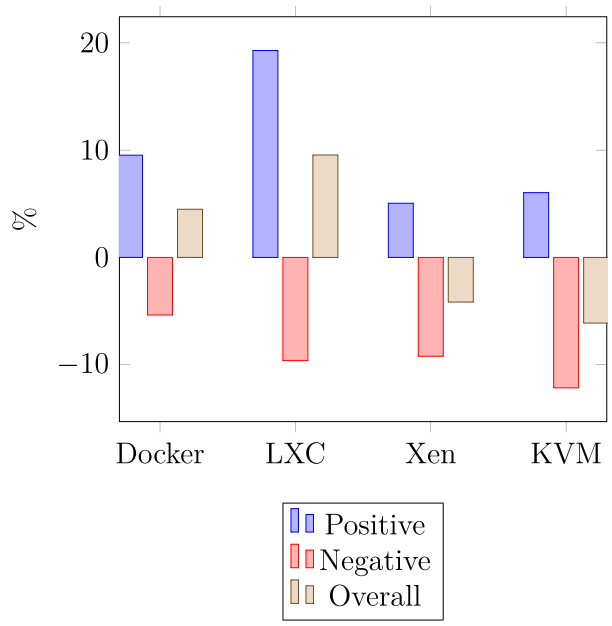


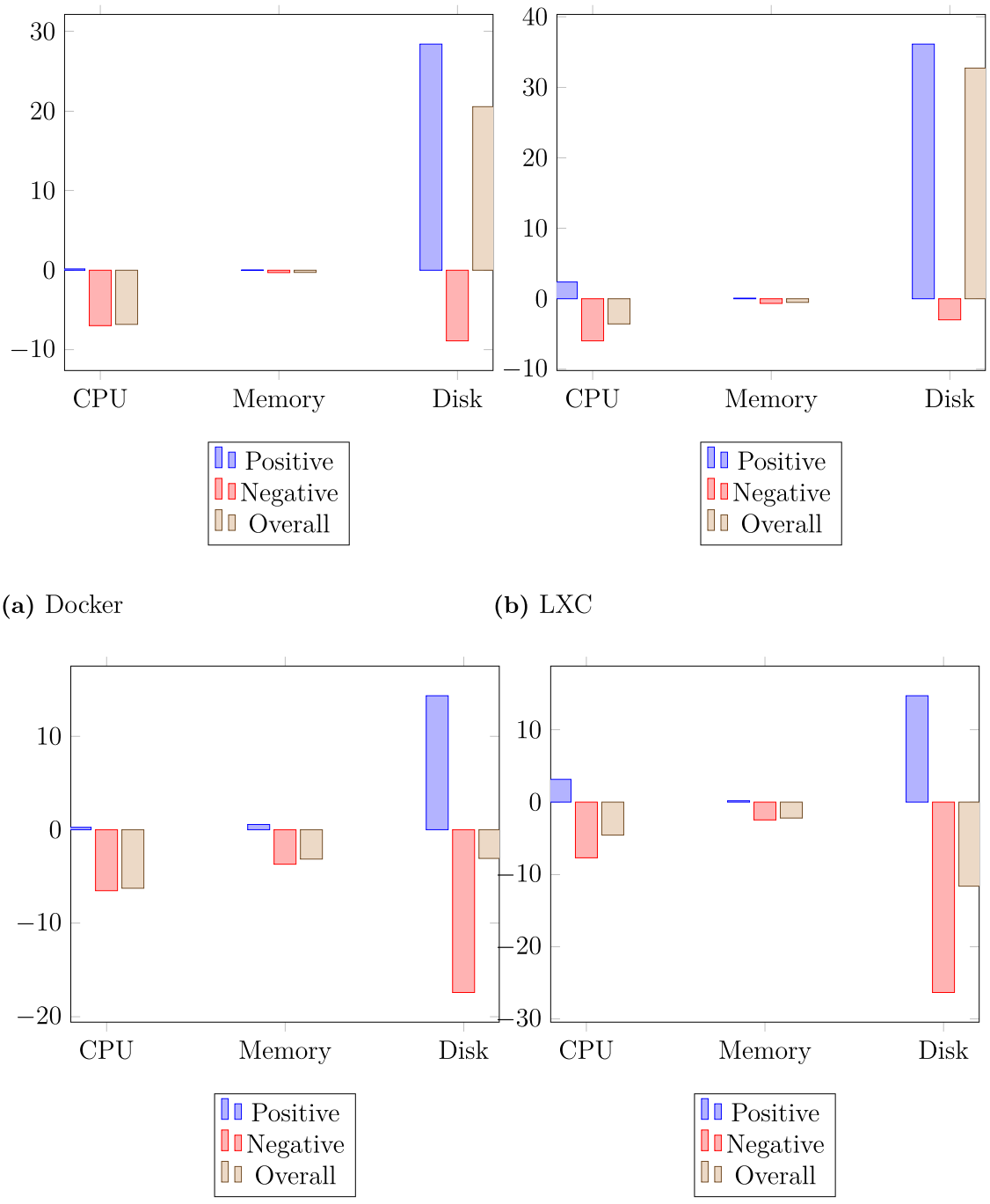
Figure 4.1: The graph shows the combined average impact the virtual layers had on CPU, memory and disk performance. The blue bars show the average of all positive impacts across all three suites. The red bars show the same thing but for negative impacts. The orange bars show the overall impact the different virtual layers had on the PTS tests.

As Figure 4.1 shows, LXC has the highest overall impact on the performance while it also have a larger negative impact than Docker, which is the layer with the second highest overall impact.

The averages calculated for each virtualisation technique and suite using equations 3.1-3.3 are shown in Figures 4.2.

Figure 4.2 shows that out of the measured system parts, the disk system is most affected by the virtual layer. We believe that the reason behind the disk impact being more pronounce lies in how calls to hardware are handled. Neither one of the four virtual layers can access the storage system directly. The VMs uses emulated hardware and therefore are those calls routed through the hypervisor. The containers' access to the storage system is controlled by `cgroup` and a theory is that this masks the disk latency and ths giving an illusion that the disk operations are much faster than they really is. We chose therefore to exclude the disk figures to avoid having the results skewed by the disk figures being much different than the other. The graphs in Figure 4.3 shows the average impact each of the layers had on the CPU and memory performance.

We can see in Figure 4.3 that KVM is the layer that, when only considering CPU and memory performance, provides highest positive impact at the same time as it has one of the highest negative impact on the performance. Since performance improvement is not a priority in this project was a high positive impact not what we were looking for. A stable and relatively predictable is more important and we chose therefore to



(a) Docker

(b) LXC

(c) Xen

(d) KVM

Figure 4.2: This Figure shows the positive, negative and overall impact the four different layers had on the PTS test suites.

continue with Docker and Xen since they had the least difference between the overall impact and the corresponding negative/positive impact. Xen was chosen over LXC

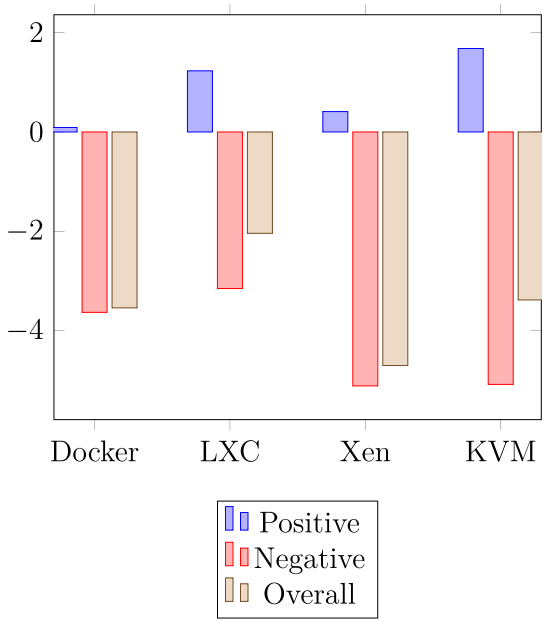


Figure 4.3: This graph shows the combined average impact that was observed when considering the results from the CPU and memory suites from PTS.

even though LXC had less difference between negative and positive impact, but we had decided to continue with one container and one hypervisor which is why we chose Xen.

4.3 Performance benchmarking

We used the same Dell laptop to run STREAM, IOzone, ripper and network tests as we did during the initial screening. The GNU application `time` was used on STREAM, IOzone and ripper to measure execution time, number of page faults and context switches. GNU time defines a involuntary context switch as a context switch caused by time slice expiration and voluntary contaxt switch as a switch caused by the process waiting for something, e.g. an I/O operation. Performance benchmarking was only done on Docker and Xen since they showed the most promise during the initial screening.

We chose to regard the high increase in File system inputs on STREAM and ripper as anomalies, since in both cases was this metric reported as 0 in 9 out of 10 runs.

4.3.1 STREAM

STREAM was used to measure the data throughput in memory. GNU `time` was used to collect data on page faults, context switches etc. The average data throughput for the baseline system and the virtual layers are presented in Tables 4.5 and 4.6 along with the percentage difference between them. Tables 4.7 and 4.8 are showing the same thing, but with the output from GNU time.

Table 4.5: This table shows the averages of the output from STREAM after it was run 10 times in a Docker container, where the vector operations are executed 50 times for each run.

Kernel	Native average	Docker	Difference
Copy	11397.45	11418.89	0.19%
Scale	11322.03	11359.39	0.33%
Add	12660.25	12744.95	0.67%
Triad	12554.83	12635.25	0.64%

Table 4.5 shows that there is a small increase in data throughput in the Docker container compared to the baseline.

Table 4.6: This table shows the averages of the output from STREAM after it was run 10 times on a Xen guest, where the vector operations are executed 50 times for each run.

Kernel	Native average	Xen	Difference
Copy	11397.45	11193.48	-1.79%
Scale	11322.03	11228.59	-0.83%
Add	12660.25	12535.29	-0.99%
Triad	12554.83	12377.91	-1.41%

The Xen guest proved to have a small decrease in data throughput compared to the baseline. This decrease is so small that it is likely that it can be compensated by replacing the current RAM if the same thing would be observed on other hardware.

Table 4.7: This table shows the averages of the output from GNU time after running STREAM 10 times in a Docker container, where the vector operations are executed 50 times for each run.

Metric	Native average	Docker	Difference
User time (s)	34.08	33.84	-0.71%
System time (s)	0.17	0.18	5.29%
Percent of CPU this job got	1.00	0.99	-1.00%
Wall clock time (h:mm:ss)	00:34.25	00:34.04	-0.63%
Maximum resident set size (kB)	2344286.80	2344273.60	0.00%
Major page faults (require I/O)	0.00	0.00	0.00%
Minor page faults (reclaiming a frame)	2041.40	2031.50	-0.48%
Voluntary context switches	1.00	1.00	0.00%
Involuntary context switches	47.20	1741.50	3589.62%
File system inputs	0.00	0.00	0.00%
File system outputs	8.80	10.40	18.18%
Page size	4096	4096	0.00%

As can be seen in Table 4.7 did the measured resource use not differ much between the physical system and the Docker container, except for involuntary context

switching.

Table 4.8: This table shows the averages of the output from GNU time after running STREAM 10 times on a Xen guest, where the vector operations executed 50 times for each run.

Metric	Native average	Xen	Difference
User time (s)	34.08	35.00	2.70%
System time (s)	0.17	0.19	14.12%
Percent of CPU this job got	1.00	0.99	-1.00%
Wall clock time (h:mm:ss)	00:34.25	00:35.21	2.78%
Maximum resident set size (kB)	2344286.80	2344287.20	0.00%
Major page faults (require I/O)	0.00	0.00	0.00%
Minor page faults (reclaiming a frame)	2041.40	2036.50	-0.24%
Voluntary context switches	1.00	1.10	10.00%
Involuntary context switches	47.20	67.10	42.16%
File system inputs	0.00	3.20	0.00%
File system outputs	8.80	10.40	18.18%
Page size	4096	4096	0.00%

The performance of the Xen guest was lower than the physical machine and the Docker container with regard to execution time. The Xen guest had less involuntary context switches than the Docker container but 42% more than the physical machine, which can account for the longer execution time since a context switch is likely to be a little slower on a Xen machine than on a physical machine since the Xen guest does not have direct access to the CPU. We can also see in Tables 3.1 and 4.3 that the Xen machine had a lower clock frequency than the physical machine. We have not been able to determine what caused the decrease in clock frequency.

4.3.2 IOzone benchmark test

We used IOzone to get more detailed information about the disk performance. Test case 2 which is shown in Table 3.2 was used for this purpose. Also here was the GNU application `time` used to collect data on page faults, context switches etc. Tables 4.9 and 4.10 shows the average output from IOzone when running our Test case 2 ten times. Tables 4.11 and 4.12 shows calculated average number of page faults and context switches after 10 runs as well as comparison between the respective virtual layer and the physical machine.
Contrary to what PTS/disk showed did the tests with IOzone show that the read and write speeds are negatively affected by virtualisation using Docker containers. Maybe the tests included in PTS/disk suited the Docker container better than this one did. We have only included one file size and one record length here, maybe would the I/O speed be higher with another file or record size. We saw when we chose record size for the virtual layers that the speeds of read and write was influenced by the record size.
The results observed when running the IOzone test on the Xen guest were more in line with the results from PTS/disk than the Docker results. Interesting to see is

Table 4.9: This table shows the average read and write speed IOzone reported when running the 7 tests we had chosen inside a Docker container.

Kernel	Native average	Docker	Difference
write	874306.90	828226.20	-5.27%
re-write	2824374.10	1198563.90	-57.56%
read	5690218.10	2970650.60	-47.79%
re-read	5684179.00	2978402.00	-47.60%
random read	5605807.60	3879288.60	-30.80%
random write	3220989.70	2941447.50	-8.68%
stride read	8121745.00	3229612.80	-60.23%

Table 4.10: This table shows the average read and write speed IOzone reported when running the 7 tests we had chosen on a Xen guest.

Test	Native average	Xen	Difference
write	874306.90	158048.20	-81.92%
re-write	2824374.10	176358.10	-93.76%
read	5690218.10	2793418.20	-50.91%
re-read	5684179.00	3767506.60	-33.72%
random read	5605807.60	5314107.70	-5.20%
random write	3220989.70	178023.20	-94.47%
stride read	8121745.00	3501814.00	-56.88%

that the write speeds reported are closer together for the Xen guest than for the physical machine.

Table 4.11: This table shows the average output from GNU time after 10 runs of IOzone in a Docker container.

Metric	Native average	Docker	Difference
User time (s)	0.04	0.05	20.93%
System time (s)	0.09	0.14	46.81%
Percent of CPU this job got	0.44	0.49	12.59%
Wall clock time (h:mm:ss)	00:00.33	00:00.40	20.97%
Maximum resident set size (kB)	20960.00	20679.20	-1.34%
Major page faults (require I/O)	0.00	0.00	0.00%
Minor page faults (reclaiming a frame)	697.10	4994.90	616.53%
Voluntary context switches	289.30	247.60	-14.41%
Involuntary context switches	12.30	1.60	-86.99%
File system inputs	0.00	0.00	0.00%
File system outputs	147467.20	147483.20	0.01%
Page size	4096.00	4096.00	0.00%

From Tables 4.9 and 4.10 we can draw the conclusion that virtualisation has a relatively high impact on the I/O performance. However, the two different virtualisation

Table 4.12: This table shows the average output from GNU time after 10 runs of IOzone on a Xen guest.

Metric	Native average	Xen	Difference
User time (s)	0.04	0.05	16.28%
System time (s)	0.09	0.17	78.72%
Percent of CPU this job got	0.44	0.40	-8.70%
Wall clock time (h:mm:ss)	00:00.33	00:00.56	70.82%
Maximum resident set size (kB)	20960.00	20522.20	-2.09%
Major page faults (require I/O)	0.00	0.00	0.00%
Minor page faults (reclaiming a frame)	697.10	585.40	-16.02%
Voluntary context switches	289.30	951.60	228.93%
Involuntary context switches	12.30	1.60	-86.99%
File system inputs	0.00	0.00	0.00%
File system outputs	147467.20	147467.20	0.00%
Page size	4096.00	4096.00	0.00%

techniques seem to impact the I/O differently. The Docker container showed an increase in page faults while the Xen guest showed an increase in voluntary context switches. Different approaches is needed in order to compensate for this in hardware. A large number of voluntary context switches suggests that the system often need to wait for an interrupt. It is likely in this case that this interrupt is caused by a I/O operation which can be sped up by replacing the memory and/or storage system. The impact from increased number of page faults can be compensated by increasing cache size.

4.3.3 John the Ripper

John the Ripper is a password cracking application and will therefore only output the result of the cracked passwords. We used therefore `time` again to collect statistics on page faults and context switches. Table 4.13 and Table 4.14 presents the average number of page faults, context switches etc. for 10 runs of the application in a Docker container and on a Xen guest. These numbers are compared with the corresponding values collected when doing the same run on the physical machine. Worth noting is that many of the metrics collected using time did not differ much between physical machine and Docker container, while three of them differs quite a lot.
Table 4.13 show that three things was affected more than the others when moving the application from the native OS to a Docker container. The system time for the application was increased by 20% when using Docker while the execution time did only increase by 1%. The higher system time can most likely be explained by the increase in context switches, but the execution time is not increasing as much since the number of page faults has decreased.
The most notable difference between the Xen guest ant the physical machine is the system time which is almost 95%. This increase can most likely be explained by the increase in involuntary switching which will increase the system time while the decrease in page faults will do the opposite. The native average presented in Tables

Table 4.13: This table presents the average output of GNU time after running John the Ripper on a password file 10 times in a docker container. The last column presents the percentage difference between these averages.

Metric	Native average	Docker average	Difference
User time (s)	2917.31	2951.75	1.18%
System time (s)	17.98	21.64	20.37%
Percent of CPU this job got	99%	99%	0.00%
Wall clock time (h:mm:ss)	00:49:15	00:49:35	0.65%
Maximum resident set size (kB)	88083.20	87198.22	-1.00%
Major page faults (require I/O)	0.00	0.33	0.00%
Minor page faults (reclaiming a frame)	8621723.40	3832504.44	-55.55%
Voluntary context switches	56.70	67.89	19.73%
Involuntary context switches	2874.70	176584.11	6042.70%
File system inputs	0.80	1322.44	165205.56%
File system outputs	368.00	368.00	0.00%
Page size	4096.00	4096.00	0.00%

Table 4.14: This table presents the average output of GNU time after running John the Ripper on a password file 10 times on a Xen guest. The last column presents the percentage difference between these averages.

Metric	Native average	Xen	Difference
User time (s)	2917.31	4167.325	42.85%
System time (s)	17.98	35.037	94.87%
Percent of CPU this job got	99%	99%	0.00%
Wall clock time (h:mm:ss)	00:49:15	01:10:02	42.18%
Maximum resident set size (kB)	88083.20	88082.4	0.00%
Major page faults (require I/O)	0.00	0	0.00%
Minor page faults (reclaiming a frame)	8621723.40	4790189	-44.44%
Voluntary context switches	56.70	96.3	69.84%
Involuntary context switches	2874.70	3277.2	14.00%
File system inputs	0.80	1131.2	141300.00%
File system outputs	368.00	416	13.04%
Page size	4096.00	4096	0.00%

4.13 and 4.14 refers to when running the application alone on our baseline system.

4.3.4 Networking

The network performance of a Xen guest and a Docker container were evaluated with regard to increased round trip time (RTT), jitter and packet loss. The result from ping is shown in Table 4.15. The network performance baseline was established by pinging 127.0.0.1 as well as connecting an iperf3 client to an iperf3 server on the physical machine. In Tables 4.15 and 4.16 is the physical machine called PM , the Docker container is called DC and the Xen guest XG.

Table 4.15: Round trip times for the five network routes reported by ping after 30 seconds. Numbers are in milliseconds.

Route	min	avg	max	mdev	avg inc.
PM to PM	0.038	0.058	0.092	0.018	0
PM to DC	0.105	0.13	0.184	0.012	124.14%
DC to DC	0.092	0.122	0.141	0.012	110.34%
PM to XG	0.179	0.52	0.602	0.074	796.55%
XG to XG	0.265	0.66	2.876	0.455	1037.93%

As can be seen in Table 4.15, the Docker container added the least time to the RTT and ping between two Docker containers proved to be more efficient than pinging a Docker container from the physical machine. This is likely because the Docker containers are connected to a internal Docker bridge that can be accessed by the physical machine via a Docker gateway. The Xen guests on the other hand is connected to a generic virtual network bridge that is not specifically designed for handling this kind of virtual guests as the Docker bridge is could be an explanation to why the Xen guest increase the RTT by almost 800% and the Docker container increases RTT by 125%. We also see that the communication between two Xen guests is much slower than the communication between two Docker containers.

Table 4.16: This table shows the jitter and packet loss observed when having five different constellations of iperf3 clients and servers.

Client to Server	Jitter	Loss	packets lost
PM to PM	0.018	0.00%	0/3416
PM to DC	0.001	0.00%	0/51631
DC to DC	0.001	0.00%	0/51638
PM to XG	0.026	3.40%	1751/51635
XG to XG	0.029	5.10%	2609/51633

We observed the same phenomenon with iperf3 as with ping, namely that connections between a Xen guest and another Xen guest or the physical machine is less stable then between a Docker container and another Docker container or the physical machine. From Table 4.16, we can draw the conclusion that there is less traffic and therefore less disturbances in the network dedicated to the Docker containers.

4.3.5 DP applications

We can not, because of confidentiality, disclose any details about the performance of the DP applications. We calculated the number of context switches per second in order to be able to do some comparisons between these applications and the benchmark applications. The radar applications had a lot more context switches per second than any of the benchmark applications, but as long as the virtual layer has the same clock frequency as the physical machine should this not have any considerable impact on the execution time.

Another observation is that in all cases, the number of voluntary context switches were higher than the number of involuntary. For our three benchmark applications was this the case only once, IOzone was the only benchmark application that had more voluntary context switches than involuntary. Furthermore was IOzone also the only application that did not consume more energy in the virtual layer than on the physical machine. This suggest that virtualising the DP applications will not increase the energy consumption, or at least not as much as it was increased in the case of STREAM. But this need to be confirmed with further experiments.

4.4 Energy benchmarking

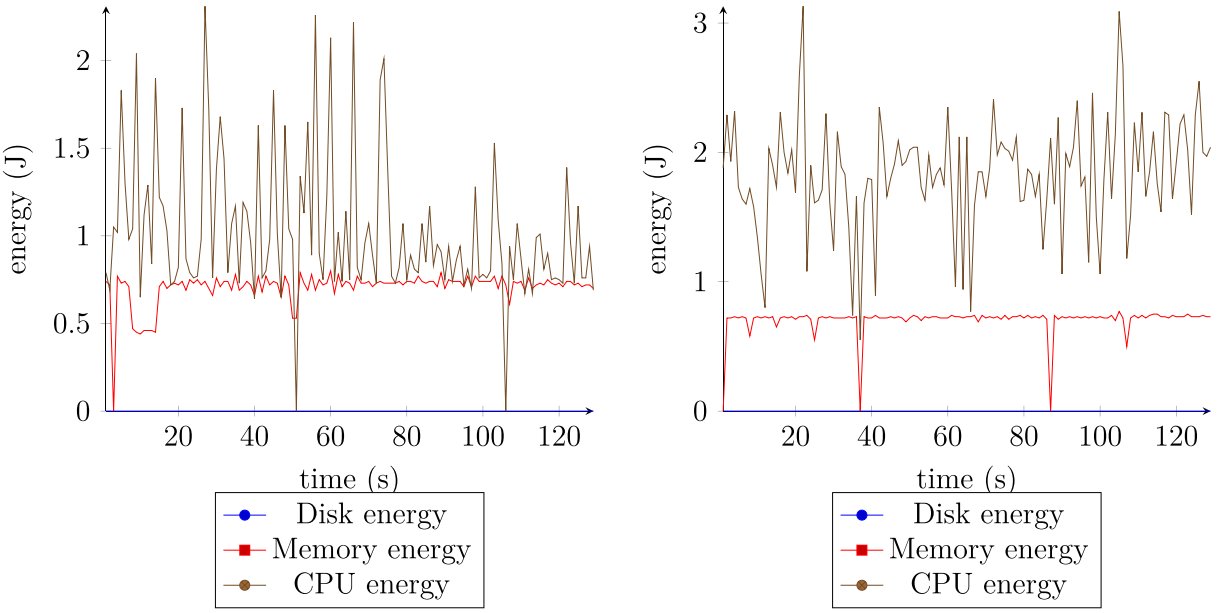
We measured the energy on a process level in order to eliminate the risk of background processes unknown to us influencing the results on one run but not the others. We used a tool called pTop to do the energy measurements. This tool depends on a CPU module called cpufreq_stats being loaded into the kernel. This kernel module is not supported by Linux kernel 4.9 which is the kernel that Xen depends on. We could therefore not do any energy measurements on the Xen layer. It was also not possible to load this module or change kernel in the environment where we ran the Data Processing applications. We could therefore not perform energy analysis on this applications nor test them in a Xen environment. It would have been possible to use an external meter, but that would not have provided process-level measurements and we did not have access to such a device.

energy measurements were performed on three applications, John the Ripper, IOzone and STREAM to investigate if virtualising a process caused it to consume more CPU, memory or disk energy. pTop was run on the host when performing the measurments on the virtual layers in order to capture the energy consumed by the Docker Daemon and other processes necessary for running a container. The first two applications did not suggest that the virtual layer caused any significant increase in energy consumption. Figure 4.6 shows how the energy consumption of the IOzone process varied over time and how much energy the three subsystems CPU, memory and disk consumes each.

4.4.1 STREAM

The memory benchmarking application STREAM was used to investigate if moving a memory heavy process to a virtual layer has any impact on the energy consumed by the process. Figure 4.4 shows how the energy consumption change during execution time. We used 10 stream processes of different length when measuring the energy consumption of this application in order to see if the execution time of the application had any impact in the energy consumption. All did the same calculations with the same array length, but different number of times. They where run consecutively starting with the shortest and ending with the longest.

We expect to see CPU usage since we know that STREAM perform calculations and we expect to see memory usage since the application is used for measuring data



(a) STREAM physical machine.

(b) STREAM Docker.

Figure 4.4: The two graphs show the changes in energy consumption for 10 consecutive STREAM processes split up into disk, memory and CPU energy consumption. Figure 4.6a shows the energy consumption when the processes are run in a docker container and Figure 4.6b shows the same but for a run on the physical machine.

throughput in memory. The load on the memory seem to be relatively consistent compared to the load on the CPU. The memory energy consumption looks to be relatively equal between the physical machine and the Docker container compared to the CPU energy consumption.

4.4.2 IOzone

IOzone was used to investigate the impact Docker had on the energy consumption of the storage system. Test case 1 presented in Table 3.2 was used for this.

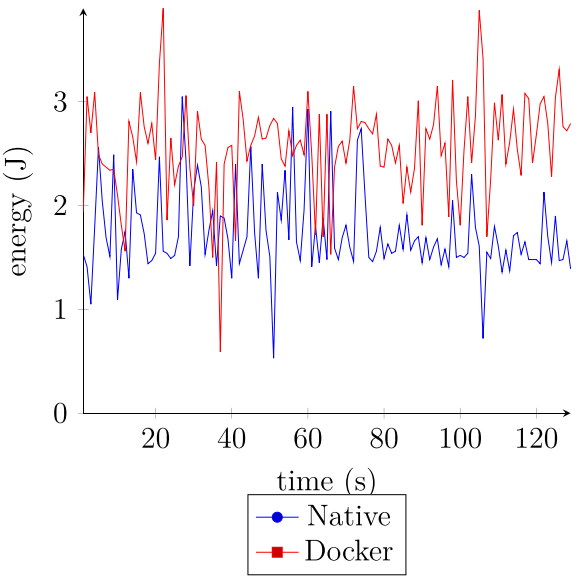
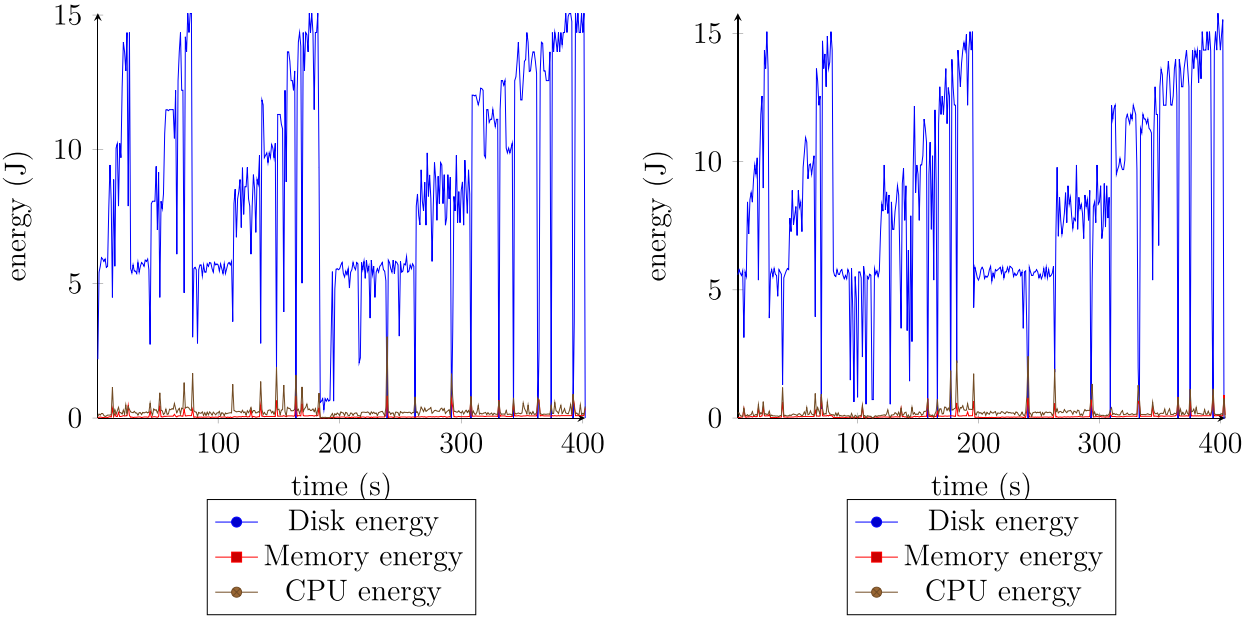


Figure 4.5: This graph shows a comparison between the energy consumption of an STREAM processes run both in the native OS and inside a Docker container. The energy consumed by the Docker Daemon and the Container Daemon is included in the Docker energy.

One can see when comparing Figures 4.6b and 4.6a that the increase in energy consumption when moving the IOzone process from the native environment to a Docker container is at worst minimal based on measuring the energy consumption for during only one run of the application. We got the same result when including the energy consumed by the extra processes that is needed when running Docker containers, such as Docker Daemon and Container Daemon. The result of this is shown in Figure 4.7.
We can conclude from Figure 4.7 that the virtual layer has no impact on the energy consumed by the IOzone application. The fact that the two graphs is a bit out of synch between 100 and 200 seconds is most likely because the Dockerized version was a bit slower than the native version and therefore had the Docker version not reached the same stage in the process at 100 seconds as the native version had.

4.4.3 John the Ripper

The John the Ripper application was used to estimate the increase in CPU energy consumption when moving from native to Docker environment. Figure 4.8 shows the energy consumption of the application when it is run on the baseline system and in a Docker container.
Figure 4.9 suggests that the virtual layer did not add to CPU energy consumption, except for at 3 stages in time where the Docker consumes significantly more energy than the physical machine. What causes this is difficult to determine without going into detail of the application source code which is out of the scope of this project. This is though an important point to consider when in the end choosing a technology to implement. A rerun of this experiment might provide insight into this.



(a) IOzone native.

(b) IOzone Docker.

Figure 4.6: The two graphs show the changes in energy consumption for an IOzone process split up into disk, memory and CPU energy consumption. Figure 4.6a shows the energy consumption when the process is run i a docker container and Figure 4.6b shows the same but for a run on the physical machine.

We calculated the average Watt consumed by these applications to allow for an easier comparison between the baseline and the Dockerized version. The resulting figures is shown in Table 4.17. These numbers suggest that an application that is heavy on the CPU could consume 15% more energy when run in a Docker container instead of in the native OS and an application the is heavy in the memory could consume 46% more energy. It is possible that this phenomenon is cause by the applications being packaged into an image representation of the container, we do not know if this causes any differences between the binaries used by the physical machine and the binaries being extracted from the container image. We know from Figure 4.2 that these differences does not incur performance variations.

Table 4.17: This table shows the calculated average energy consumed by the three measured applications.

Application	Native	Docker	Difference
John the Ripper	1.12	1.29	15.01%
IOzone	8.89	8.84	-0.60%
STREAM	1.73	2.54	48.29%

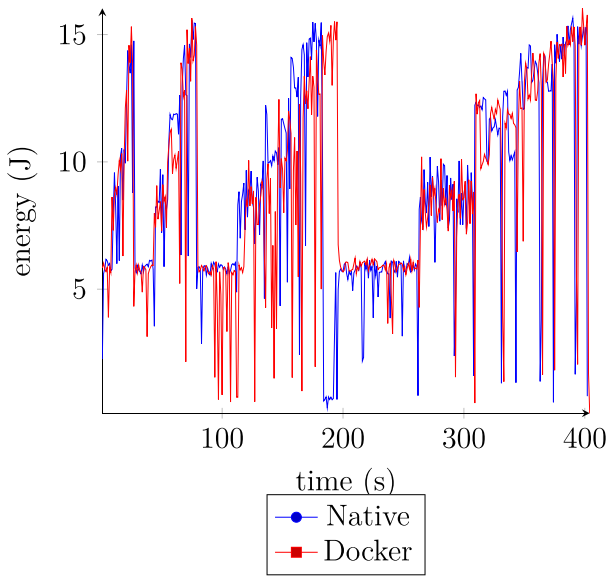


Figure 4.7: This graph shows a comparison between the energy consumption of an IOzone process run both on the physical machine and inside a Docker container. The energy consumed by the Docker Daemon and the Container Daemon is included in the Docker energy.

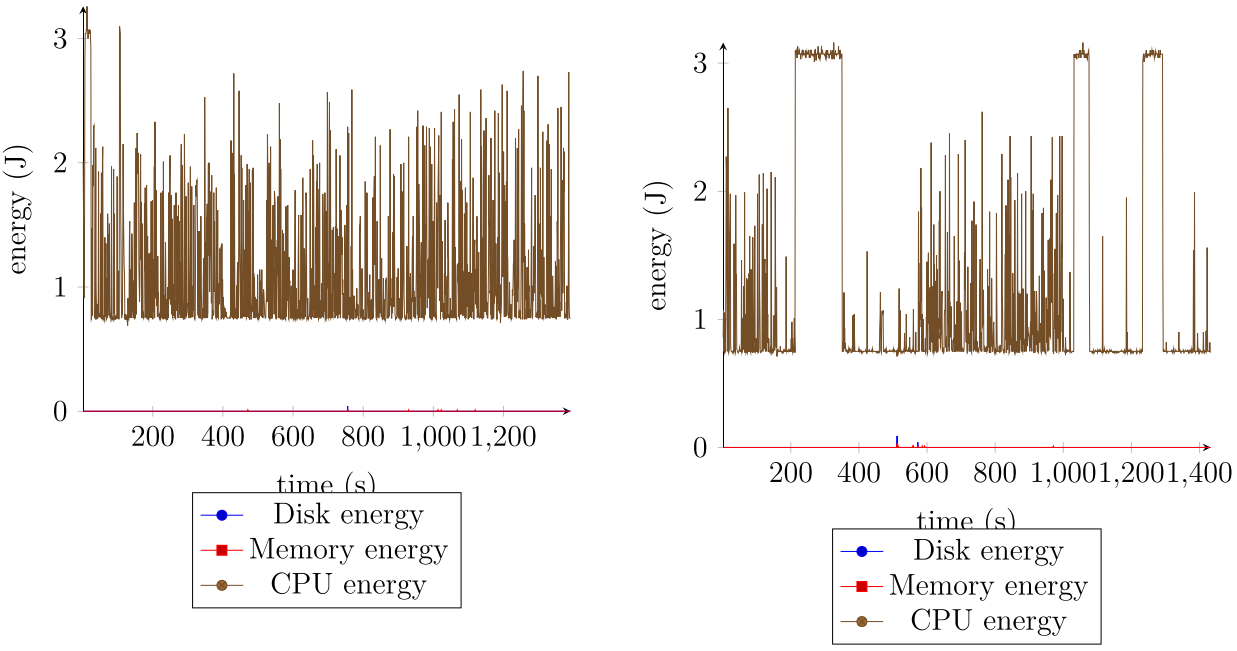
4.5 Summary

The benchmarking experiments consisted of three parts:

- Using PTS, decide which container and hypervisor to study in more detail
- Using other tools and bencharmks, measure performance of Docker and Xen.
- Using pTop, measure energy consumption on Docker.

PTS The first part of the benchmarking where we used PTS was used to determine which of the four virtualisation techniques showed the most promise for implementation in the radar system. As stated in Section 4.2, Docker and Xen had the least difference between when running tests in the respective virtual layers and running the tests on the physical machine. We therefore regarded them to have a more predictable impact on the software performance than KVM or LXC.

For the remainder we decided to do more extensive tests with Xen and Docker since we decided to prioritise a predictable impact. The continued testing confirmed what PTS suggested, namely that Docker does not have any particular impact on the overall performance of an application. When using Docker, the layer adds a few percent to the execution time while context switches and page faults are increased and decreased respectively. This is not the case when using Xen. Using Xen adds about 42% to the total execution time compared to running Ripper on our baseline system. The increase in execution time can be explained by the Xen guests having virtual CPUs with a lower clock frequency than the physical machine.



(a) John the Ripper physical machine.

(b) John the Ripper Docker.

Figure 4.8: The two graphs show the changes in energy consumption for a John the Ripper process split up into disk, memory and CPU energy consumption. Figure 4.8b shows the energy consumption when the process is run i a docker container and Figure 4.8a shows the same but for a run on the physical machine.

Measuring performance The experiments described in 4.3 have shown that virtual layers are best suited for applications that does not need to do a large amount of system calls. IOzone is the application that most likely made the most system calls since it makes alot of I/O operations, and this was the application were we could observe the largest increase in execution time for both Docker and Xen. The application areas for the two technologies differ a bit. Docker containers are suitable for applications where generated data does not need to be saved between sessions or where the data can be logged by a process that is not run in the container. Xen is suitable if you need multiple VMs on your physical machine, e.g. in a server environment.

The overhead incurred by Docker and Xen seem to come from different places depending on what kind of operation is being performed. Context switching seem to have less impact than page faults on the performance of software in a Docker container in terms of execution time. It is harder to draw any conclusion about the overhead of the Xen guest since the CPU frequency differed between the physical machine and the Xen guest. This is likely to have masked increase or decrease in execution time of the software on the Xen guest. However, we did see a decease in page faults on the Xen guest which could suggest that the performance might be higher on a Xen guest then on the physical machine if the CPU frequency is the

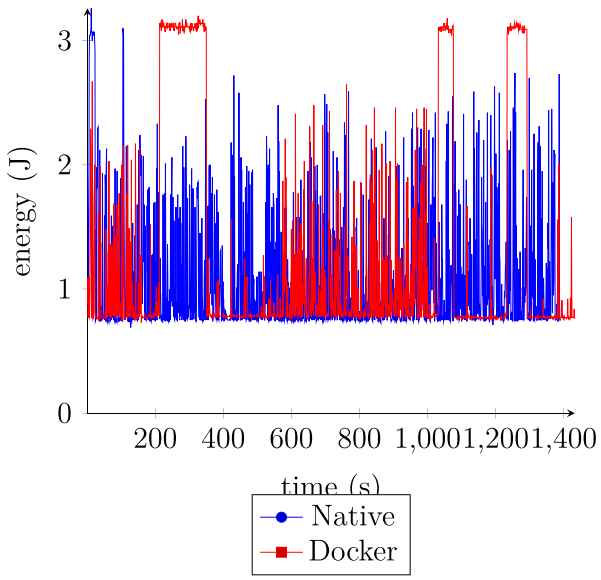


Figure 4.9: This graph shows a comparison between the energy consumption of an John the Ripper process run both in the native OS and inside a Docker container. The energy consumed by the Docker Daemon and the Container Daemon is included in the Docker energy.

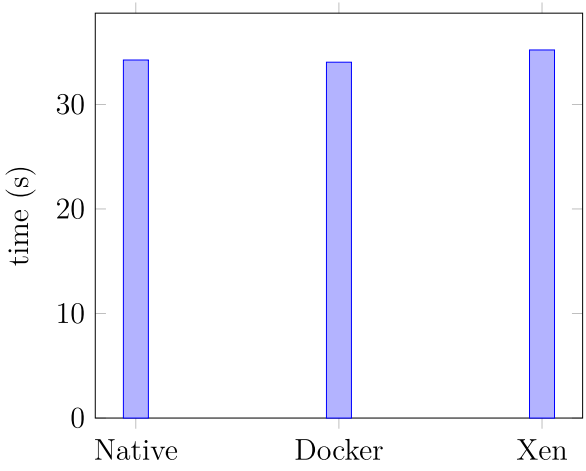
same. A possible explanation for the lower frequency can be found in the difference in workload between the vCPU and the physical CPU on which the vCPU is scheduled [21]. In our case is only the Xen guest executing any significant work-load and therefore will the workload of the vCPU be higher then the work-load of the physical CPU. Throughout the experiments, the CPU governor was set to *ondemand* which meant that a low workload would result in a lower frequency. When running Xen is the vCPU time slice set to 30 ms with 10 ms as the minimum sampling interval for frequency adjustment. This can result in the CPU frequency never being increased to a level suitable for the actual work-load since the governor can not increase the frequency more than one step at each sample interval. There are two ways to address this problem:

- Change governor. If the CPU power governor is set to will the frequency of the physical CPU always be set to highest available frequency, which would mean that the vCPU also would receive this frequency.
- Increase the vCPU time slice. This would provide the opportunity for the Xen power manager to increase the frequency of the vCPU to a more suitable level before the end of the time slice.

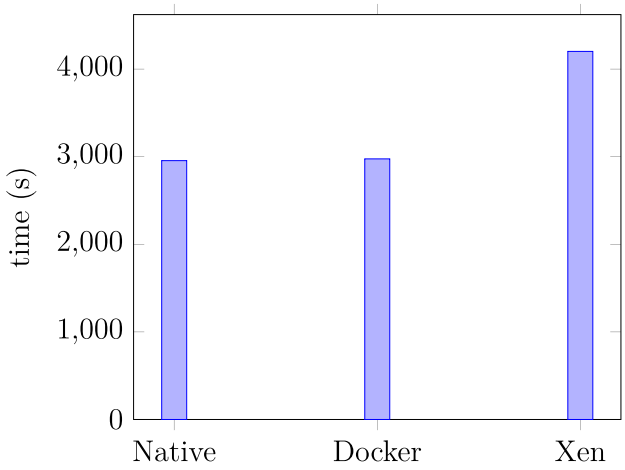
The network experiments showed that the Xen guest added quite a lot to the network latency and even caused packet loss when we ran iperf3. Docker containers showed better performance than the Xen guests even though the RTT more than doubled when pinging between a Docker container and another Docker container or the physical machine.

Figure 4.10 summarises the average execution times and energy consumptions for

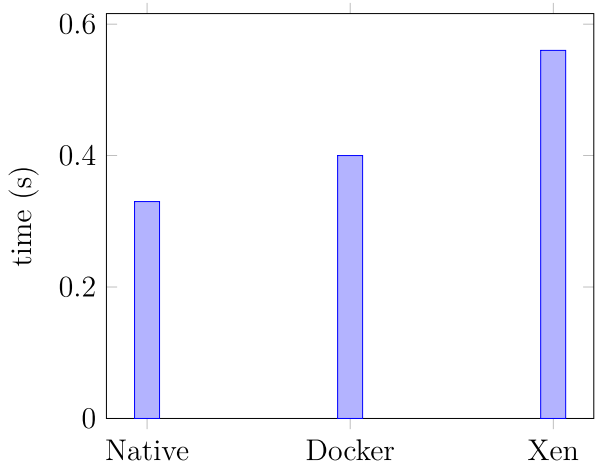
STREAM, IOzone and John the Ripper when they are executed on the physical machine or in a virtual layer. The differences between the heights of the bars show that Xen has the largest overhead in terms of execution time, but as previously mentioned could this be due to the lower clock frequency. The measurements of energy consumption suggests that applications that are heavy on the CPU will get increased energy consumption when run in a virtual layer.



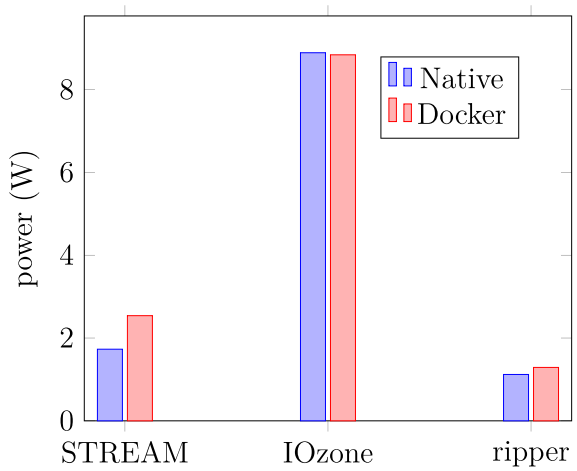
(a) Average execution time for STREAM



(b) Average execution time for John the Ripper



(c) Average execution time for IOzone



(d) Average energy consumption

Figure 4.10: This Figure shows the average execution time for STREAM, IOzone and John the Ripper when they are run on the physical machine, in a Docker container and on a Xen guest.

Measuring energy We could only measure the energy consumption of dockerized processes, not processes run on a Xen guest due to compatibility issues. Running a process inside a Docker container instead of directly on the physical machine seems

to cause an increase in energy consumption in certain parts of the system. By observing Figure 4.4b and Tables 4.7 and 4.17, we can assume that the increase in energy consumption is caused by the increase in context switches. We calculated the number of context switches per second in an attempt to see connection between context switches and energy consumption using the following formula

$$\frac{Voluntary + Involuntary\_context\_switches}{System + User\_time} \tag{4.1}$$

Table 4.18: This table shows the average number of context switches when running our three applications on the physical machine and in a Docker container.

Application	Native cs/s	Docker cs/s	Difference
STREAM	1.41	51.22	3539.80%
IOzone	2201.46	1311.58	-40.42%
ripper	1.00	59.41	5848.97%

By comparing Tables 4.18 and 4.17 we can see that the two applications that had a higher energy consumption also had a higher number of context switches in the virtual layer. As established in Table 4.17 is STREAM the application that had the largest energy consumption increase when changing from running on the physical machine to running in a Docker container and since the most notable difference between the two runs is the number of involuntary context switches is it likely that those context switches are what causes the energy consumption increase. Since a context switch will cause the content of the caches to change and cache latency as well as cache line size are two of the parameters used by pTop to calculate energy is the increase in context switches a likely explanation for the increase in energy consumption.

5
Discussion

In this chapter we discuss the methods we used for measuring the energy consumption and performance. We highlight what parts we could have done differently and how our choices might have impacted the outcome of the project. We conclude the chapter with a discussion on how we chose which technologies to focus on when performing the performance and energy measurements.

5.1 Energy measuring

The tool that was used for measuring energy showed that there is a slight increase in energy consumption when running applications in virtual layers. Since the user is instructed to adjust some parameters before compiling the code and the accuracy of the program relies on these parameters, it is likely that the raw data collected from pTop does not conform precisely to reality. We made an attempt to tune it to our system but are not sure of who successful the attempt was. This however, we do not view as an issue since we are more interested in the relation between virtual layer and baseline system energy consumption. As long as that stays the same no matter the difference between reported energy consumption and real energy consumption will this tool serve our purposes. We believe that the tool has the same accuracy for both the process that is run in a Docker container as it has for the process running directly on the physical machine. This belief is based on the fact that pTop could identify the relevant processes that was run in the container even though pTop itself was run on the physical machine. pTop uses process IDs to monitor how much a certain process is using a certain resource. This means that as long as the process ID for the process in question is visible to the host machine, the accuracy of the measurements should not differ between virtual processes and processes running on the physical machine. The developers of pTop claims that the tool has an error median that is less than 2 Watts [14]. They do not clearly state in their paper if this is per process or per system, but it should be per system since they used a Watts Up power meter to measure the accuracy of pTop.

If our assumptions about the accuracy of pTop are wrong could this mean that the power impact is much greater than indicated here. Higher energy consumption in the same system generally entails more heat to dissipate. Depending on how efficient the cooling of the system is and how much heat the system generated before the increase in energy consumption could this have severe implications. Considering that we are investigating the possibility of using a virtual layer in a mobile radar

system with a limited power supply, it is even more important that we can be sure about how the virtual layer impacts the energy consumption. A too great energy consumption would limit the operation time and too much heat could be a severe issue because the radar-systems are used both in cold and very hot environments. Heat also ages components which means that too much heat might possibly shorten the lifetime of the radar system depending on the part that is currently constraining it.

Our calculations regarding Watts consumed by the applications suggest that the energy consumption could increase rather drastically if the applications are run inside containers rather then on the physical machine. Containerisation of the radar applications might be unfeasible due to this. As we see from comparing Figures 4.4a and 4.4b originates this increase in energy consumption in the processor. An increase of nearly 50% in CPU energy consumption could have a large impact on the operation time depending on the number of such operations and how large portion of the energy consumption the CPU operations are responsible for in the radar applications.

pTop was not used together with virtual machines in the original paper and in hindsight we believe that a power meter would have been a more suitable tool for this kind of measurements. Our reason for choosing a software tool was that we wanted to see how much e.g. Docker Daemon added to the energy consumption alone. According to our measurements using pTop does the Docker Daemon not cause any significant increase in energy consumption which decrease the necessity of measuring energy consumption at a process-level.

5.2 Benchmarking results

Overall did the virtual layers have a negative impact on the performance. Figure 4.1 suggests that the virtual layers had a positive impact on some parts of the system, but by comparing Figure 4.1 with the individual test results one can see that there are a few results that alone increases the average. We have not included the individual test results from the PTS suites since we did not find those numbers being relevant because we are only interested in the differences between native and virtual environment at this stage.

We chose to only use the results from the PTS suites as a screening tool since we thought that we did not have enough insight into exactly what each test in the suites measured and extracting the raw data was too time consuming. Despite this, we thought it was suitable as a screening tool since it provided an efficient way of running multiple benchmark applications that tested different aspects of the system.

As mentioned earlier, the platform that was used does not reassemble the currently used system and might not reassemble the new system either. What we did though was to try to make the virtual environment as similar to our baseline system (the Dell laptop) as possible in order to limit the differences between them to the virtu-

alisation. This way we hoped to be able to measure the impact the virtualisation it self had on the software performance. This way it is possible to estimate by calculations how the DP applications might be affected by virtualisation and thus use that information to choose the replacing hardware.

We chose to only use the most basic configurations when creating the virtual machines, such as number of virtual CPU, RAM and video RAM. This resulted in the Xen guest having a lower clock frequency than the physical machine. This has likely influenced the results reported by GNU time. The Xen guest had a 30% lower clock frequency than the physical machine and the other virtual layers and we can see that when running John the Ripper which was the application that focused on using the CPU is the execution time about 42% longer. The number of context switches could not by themselves account for this increase in execution time, however the difference in CPU frequency explains the increase in execution time since it would make the execution time longer even if the number of context switches and page faults was the same.

To further limit possible interference, we stopped processes such as browsers and daemons that was not necessary for the functions of the process we were currently performing measurements on. If unnecessary processes are would that result in the execution time of the processes we were interested in being longer which would give the impression that the performance was worse than it really is. There is a possibility that we missed some unnecessary processes. Assuming that these processes interfered with the measured processes equally should the interfering processes not impact our results significantly since we are interested in the physical vs. virtual environment differences. If this assumption is wrong and only the processes run on the physical machine were affected by interference, it is possible that the physical machine should exhibit a higher performance and the difference between physical and virtual machine is greater than shown in this report.

5.3 Motivation of prioritisation choice

We decided at the start of the project that it is more important that the virtual layer does not have a too large negative impact on the performance rather then it can provide a positive performance impact. Due to this decision was the negative impact numbers given a larger importance then continuing the work after the benchmarking process.

We chose to continue with more in depth testing of Docker and Xen because they had the smallest difference between overall impact and negative impact when only taking the results from CPU and memory suites into account. The results from the disk suite was not included when making the decision since the impact on the disk performance was so much more pronounced that they were likely to mask the CPU and memory impacts. Instead, we chose to use IOzone to measure disk operation performance in virtual layers.

6

Transitioning to a virtual environment

Since this project should partially determine the feasibility of introducing a virtual layer in the radar systems have we also studied how a transition from running traditional applications natively on the machines to running them in a virtual layer. Using containers or a hypervisor will require different things when making the transition from running natively to running in a virtual layer.

6.1 Feasibility

From a performance perspective, we believe that even though a virtual layer have some disadvantages would the advantages it presents outweigh them. A virtual layer will almost certainly decrease the performance compared to when running an application natively. The current hardware is, as stated in Section 1.1, going to be replaced with newer that provides more performance. We believe that overall it is likely that the new hardware can mitigate the performance degradation caused by introducing a virtual layer.

Using a virtual environment presents multiple advantages. As an example, applications requiring conflicting libraries can be placed in separate VMs as a workaround to this problem. Virtual environments also enables using any OS that support virtual environments even though it might not support the application that need to run in the system. This is possible since VMs on the same host do not need to run the same OS as each other or the host.

The disadvantages of using virtual environments is mostly noticeable during system setup. The setup will be different depending on what virtualisation technique is being used. Hypervisors type II might be smoother than containers to use, but they are also likely to have the highest performance degradation. A type I hypervisor might be a bit cumbersome install since it is not as straight forward as the type II, but we found it to be quite smooth to use once installed. Xen also seemed to have slightly more stable performance differences compared to Docker, but this need to be further investigated in order to arrive at any definitive conclusion.

From an energy perspective is it possible that the virtual layer could have a negative impact on the operation time of the surface radar system. Our process-level mea-

surements in the physical system and the Docker container suggest that the could increase drastically depending on which system parts are most heavily used.

Our results from the network experiments suggest that using a Xen guest in the radar-systems might not be feasible. The increase in latency could cause the target tracking to lag and the packet loss reduces the reliability of the system. In a live scenario could this be devastating for the people relying on the radar for detecting incoming threats. The Docker containers also increased the latency compared to our baseline, but not as much as the Xen guest did. It might be possible to compensate for the latency increase caused by the Docker container due to the fact that the jitter and the standard deviation of the RTT was lower when using Docker containers then when sending data internally on the physical machine.

6.2 How to do it

A hypervisor would allow the currently used applications to be moved from the native environment into the virtual machine. This would be useful in order to smoothly move the old radar applications from the current computer configuration which uses multiple computer cards to a new one with only one computer card. In the new system could a hypervisor be used to create multiple VMs that could have the same function as the physical cards have today. From the applications' point of view would it seam like they still are spread out over multiple physical cards even though there are only one physical card. The drawback from this configuration is that this would require a computer card that has at least X times the performance as one of the cards that is used today, where X is the number of cards that is currently being used.

Containers have proven to be the technology that have the least impact on the performance compared to running applications on the physical machine. To be able to use containers at their full potential, the applications should be divided into smaller components and each component should be placed in a container. The containers will then work together to form the app. The container configuration would much like the hypervisor require a hardware performance increase at least equal to the number of computer cards. Maybe not as high as the hypervisor since containers have less overhead. It would require more work to move the radar applications into a container environment since they will need to be containerised. How the containerisation should be done is out of the scope for this project and should be left to the software engineers. A guideline for containerisation is to divide the applications into well-defined functions.

6.3 Recommendations

Our recommendation for the transitioning from multiple physical computer cards to only one is to as a first step use a hypervisor to create the required amount of VMs that can substitute for the physical cards. This configuration will allow any legacy software to run as if the hardware has not been changed while the software

engineers work with containerising the applications. However, it is important to study how increased network latency can impact the performance of the system as a whole before replacing physical boards with VMs.

We recommend to use containers in the long run. Containers add a small overhead compared to a hypervisor while still isolating processes from each other. Containerisation will also allow developers to push new features and fixes to the applications much faster compared to when using traditional development. Security might also be better due to organisations will be able to configure their orchistrators to always pull the newest version of the app and thus be up-to-date with every security patch [8].

7
Conclusion

In this thesis we have made an attempt to address the problem of introducing a virtual layer into transportable radar systems. This was done by investigating the impact the layer would have on the software performance and the energy consumption. In addition to this, we also looked at the cause behind the overhead of two virtualisation techniques. The results from our experiments suggest that CPU operations are not significantly affected when moved from a physical environment to a virtual one. Operations that require system calls however are significantly affected. We therefore conclude that computation heavy applications are better suited for virtualisation than I/O heavy applications when only considering performance. Concerning energy consumption, our experiments indicate that CPU heavy applications will have a higher energy consumption in a virtual environment. This observation indicates that virtualisaiton is not suitable for implementing in a system with a limited power supply. However, this need to be confirmed with further research.

7.1 Future work

This work hope to act as a baseline for continued investigation of how virtual layers can be used in surface radar-systems. The energy consumption of the individual processes indicates that a virtual layer could limit the operation time of a system with a limited power supply. However, we do not know how big the increase is on a system level since we only measured the energy consumption on specific processes at process-level.

Other possible areas for further research is to investigate how multiple running containers or VMs might impact each other. We did some research into this, but due to time limitations were we not able to arrive at any conclusions on this part. This is highly interesting since we propose to run multiple VMs on the same computing board after updating the hardware but before the software is fully prepared for a virtual environment.

Since the hardware configuration differed between the physical machine and the Xen guest would it be relevant to repeat the experiments done in this project using a Xen guest with the same hardware configuration. One could conclude from this project that the increased execution time mainly stems from the lower CPU frequency which suggests that if the two machines had the same frequency would their performance in terms of execution time be closer than observed here.

We have focused on how virtualisation can be used to solve the potential incompatibility problem between currently used software and updated hardware. Future projects might include investigating how the virtual layer should be implemented in order to ensure the best possible outcome.

Bibliography

[1] add host device. Add host device to container (–device). https://docs.docker.com/engine/reference/commandline/run/#add-host-device-to-container---device. Accessed: 2019-06-06.

[2] Samuel A Ajila and Omhenimhen Iyamu. Efficient live wide area vm migration with ip address change using type ii hypervisor. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 372–379. IEEE, 2013.

[3] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3): 102–c3, 2015.

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, February 1996. ISSN 0734-2071. doi: 10.1145/225535.225538. URL http://doi.acm.org/10.1145/225535.225538.

[6] CentOS Docker build scripts. Centos docker build scripts. https://github.com/CentOS/sig-cloud-instance-build/tree/master/docker. Accessed: 2018-11-20.

[7] cgroups. cgroups - linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 2019-06-06.

[8] Ramaswamy Chandramouli, Murugiah P Souppaya, and Karen Scarfone. Nist guidance on application container security. Technical report, 2017.

[9] chroot - change root directory. chroot - change root directory. http://man7.org/linux/man-pages/man2/chroot.2.html. Accessed: 2019-06-06.

[10] cpuspeed. cpuspeed download. https://carlthompson.net/downloads/cpuspeed/cpuspeed-1.5.tar.bz2. Accessed: 2019-03-20.

[11] Vitor Goncalves da Silva, Marite Kirikova, and Gundars Alksnis. Containers for Virtualization: An Overview. *Applied Computer Systems*, 23(1):21–27, 2018. ISSN 2255-8691. doi: 10.2478/acss-2018-0003. URL `http://content.sciendo.com/view/journals/acss/23/1/article-p21.xml`.

[12] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Evaluating attainable memory bandwidth of parallel programming models via babelstream. *International Journal of Computational Science and Engineering*, 17(3):247–262, 2018.

[13] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, 2(3):222–225, 2013.

[14] Thanh Do, Suhib Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool, 2009.

[15] Docker get started. Get started, part 1: Orientation and setup. `https://docs.docker.com/get-started/`. Accessed: 2019-06-06.

[16] Docker Overview. Docker overview. `https://docs.docker.com/engine/docker-overview/`. Accessed: 2018-12-13.

[17] DomU Support for Xen. Domu support for xen. `https://wiki.xenproject.org/wiki/DomU_Support_for_Xen`. Accessed: 2018-11-30.

[18] Zhanibek Kozhirbayev and Richard O. Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182, 2017. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2016.08.025. URL `http://www.sciencedirect.com/science/article/pii/S0167739X16303041`.

[19] KVM. Kernel virtual machine. `https://www.linux-kvm.org/page/Main_Page`. Accessed: 2018-12-14.

[20] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *Advanced Information Networking and Applications (AINA), 2017 IEEE 31st International Conference on*, pages 955–962. IEEE, 2017.

[21] Ming Liu, Chao Li, and Tao Li. Understanding the impact of vcpu scheduling on dvfs-based power management in virtualized cloud environment. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 295–304. IEEE, 2014.

[22] LXC manpage5. lxc. `https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html`. Accessed: 2019-01-15.

[23] LXC vs Docker. Understanding the key differences between lxc and docker. https://archives.flockport.com/lxc-vs-docker/. Accessed: 2019-06-02.

[24] LXD 2.0: Image management. Lxd 2.0: Image management [5/12]. https://stgraber.org/2016/03/30/lxd-2-0-image-management-512/. Accessed: 2019-01-07.

[25] man time. time(1) - linux man page. https://linux.die.net/man/1/time. Accessed: 2019-03-08.

[26] I Masliah, A Abdelfattah, A Haidar, S Tomov, M Baboulin, J Falcou, and J Dongarra. Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices. *Parallel Computing*, 81: 1–21, 2019.

[27] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 1995:19–25, 1995.

[28] Daniel A Menascé. Virtualization: Concepts, applications, and performance modeling. In *Int. CMG Conference*, pages 407–414, 2005.

[29] overlayfs. Overlay filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Accessed: 2019-06-06.

[30] OverlayFS. Use the overlayfs storage driver. https://docs.docker.com/storage/storagedriver/overlayfs-driver/. Accessed: 2018-12-13.

[31] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018.

[32] Phoronix Test Suite Package. phoronix-test-suite-8.2.0-1.el7.noarch.rpm. https://centos.pkgs.org/7/epel-x86_64/phoronix-test-suite-8.2.0-1.el7.noarch.rpm.html. Accessed: 2018-11-20.

[33] J. Sahoo, S. Mohapatra, and R. Lath. virtualization: A survey on concepts, taxonomy and associated security issues. In *2010 Second International Conference on Computer and Network Technology*.

[34] stream website. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/. Accessed: 2019-03-22.

[35] time. Gnu time. https://www.gnu.org/software/time/. Accessed: 2019-05-14.

[36] Fan-Hsun Tseng, Ming-Shiun Tsai, Chia-Wei Tseng, Yao-Tsung Yang, Chien-Chang Liu, and Li-Der Chou. A lightweight autoscaling mechanism for fog computing in industrial applications. *IEEE Transactions on Industrial Informatics*, 14(10):4529–4537, 2018.

[37] vSphere. Product evaluation center for vmware vsphere 6.7. https://my.vmware.com/en/web/vmware/evalcenter?p=vsphere-eval. Accessed: 2019-06-06.

[38] J. P. Walters, V. Chaudhary, M. Cha, S. G. Jr., and S. Gallo. A comparison of virtualization technologies for hpc. In *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, pages 861–868, March 2008. doi: 10.1109/AINA.2008.45.

[39] Xen overview. Xen project software overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Accessed: 2019-06-06.

[40] Xen4QuickStart. Xen4 centos quickstart. https://wiki.centos.org/HowTos/Xen/Xen4QuickStart. Accessed: 2018-11-28.

[41] Xenwiki. Xen project beginners guide. https://wiki.xenproject.org/wiki/Xen_Project_Beginners_Guide. Accessed: 2018-11-12.

A

Appendix 1

This appendix holds all code and configurations files used throughout the project.

A.1 Dockerfiles

Below is listings showing the Dockerfiles used to create the containers that was used to run PTS during the benchmarking process. The code in Listing A.1 was generated using a script and a kickstart file from the CentOS Project GitHub repository [6].

Listing A.1: Base image for CentOS 7 Linux

```
#mycentos
FROM scratch
ADD centos-7-docker.tar.xz /

LABEL org.label-schema.schema-version="1.0" \
    org.label-schema.name="CentOS Base Image" \
    org.label-schema.vendor="CentOS" \
    org.label-schema.license="GPLv2" \
    org.label-schema.build-date="20181006"

CMD /bin/sh
```

Listing A.2: Creating an image with the shared libraries needed by PTS

```
#mylibs
FROM mycentos:latest AS build
WORKDIR /home
RUN yum update
RUN yum install -y wget
RUN wget http://dl.fedoraproject.org/pub/epel/
    epel-release-latest-7.noarch.rpm
RUN rpm -ivh epel-release-latest-7.noarch.rpm
RUN yum install php-cli php-pdo php-xml -y
CMD /bin/sh
```

Listing A.3: Creating an image with PTS

```
#mypts
```

```
FROM mylibs:latest
RUN yum install phoronix-test-suite -y
CMD /bin/sh
```

Listing A.4: Creating an image for a container that will run the CPU tests from PTS

```
#mypts-cpu
FROM mypts:latest
VOLUME /var/lib/phoronix-test-suite
RUN phoronix-test-suite install pts/cpu
CMD /bin/sh
```

Listing A.5: Creating an image for a container that will run the memory tests from PTS

```
#mypts-mem
FROM mypts:latest
VOLUME /var/lib/phoronix-test-suite
RUN phoronix-test-suite install pts/memory
CMD /bin/sh
```

A.2 Xen configuration file

Listing A.6: Configuration file for the Xen VM used during the benchmarking process

```
#centos-test.cfg
builder = "hvm"
name = "centos-test"
memory = "8192"
vcpus = 8
serial = 'pty'
vif = ['bridge=virbr0']
disk = ['phy:/dev/centos/centos-test,xvda,rw',
    'file:/opt/isos/CentOS-7-x86_64-Minimal-1804.iso,xvdb:cdrom,r']
boot = "c"
sdl = 0
vnc = 1
vncconsole = 1
vnclisten ="127.0.0.1"
vncpassword = ""
stdvga = 1
videoram = 2048
keymap = "sv"
```