Bachelor Degree Project

UNIVERSITY OF SKÖVDE

CONTAINER PERFORMANCE BENCHMARK BETWEEN DOCKER, LXD, PODMAN & BUILDAH

Bachelor Degree Project in Computer Science
G2E, 22.5 ECTS
Spring term 2020
2017-05-25

Rasmus Emilsson (a17rasem@student.his.se)

Supervisor: Thomas Fischer
Examiner: Jianguo Ding

Contents

Abstract

Virtualization is a much-used technology by small and big companies alike as running several applications on the same server is a flexible and resource-saving measure. Containers which is another way of virtualizing has become a popular choice for companies in the past years seeing even more flexibility and use cases in continuous integration and continuous development.

This study aims to explore how the different leading container solutions perform in relation to one another in a test scenario that replicates a continuous integration use case which is compiling a big project from source, in this case, Firefox.

The tested containers are Docker, LXD, Podman, and Buildah which will have their CPU and RAM usage tested while also looking at the time to complete the compilation. The containers perform almost on par with bare-metal except Podman/Buildah that perform worse during compilation, falling a few minutes behind.

Keywords: Containers, virtualization, compilation, Docker, LXD, Podman.

1 Introduction

Virtualization has been available since the 1960s with systems such as IBM CP-40 (Comeau, L. W., 1982), and the IBM System/360-67 (IBM 1972). But, the technologies "…wasn't widely adopted until the early 2000s" (Redhat, 2020).

With no indication of losing traction as it per Redhat (2020) is the foundation of cloud computing, virtualization has shifted towards the use of containers instead of fully-fledged OS virtualization for certain tasks. For example, "As of February 2017, 80% of backend services in production run as containers. As a result, Docker has gone from an experiment to being a critical piece of Spotify's backend infrastructure." (Xia, 2017)

This report is structured in 9 parts including this part. Chapter 2 will give some general background information and basic knowledge needed to understand the experiment. Chapter 3 introduces the problem and the research question in detail.

Chapter 4 covers the methodology that will be used to conduct the experiment in the lab, design of the experiment, the tools used to gather data is discussed in chapter 5. Chapter 6 gathers and describes how to interpret the data while chapter 7 analyses the data in question using diagrams and text.

Chapter 8 is the concluding chapter where the analysis of the data will be made into conclusions. Chapter 9 and 10 are the discussion and future work chapters.

2 Background

This chapter will contain the needed information on the concepts the reader needs to know to understand the problem in question.

2.1 Virtualization / Containerization

The background of Virtualization was introduced in chapter 1; this chapter will detail the advantages of virtualization and how both traditional virtualization and containerization works.

Virtualization offers advantages over a bare-metal approach. Flexibility and scalability are one main advantage as new VMs can be created quickly, sent to another server if needed for workload balancing. On top of that, the efficiency of putting ten programs on one server instead of dedicating ten servers for those tasks is maximized.

The difference between a virtual machine and a container is that a container contains only what it needs to complete a given task such as running a web server or a database. There is no need for a complete guest operating system that expends resources to run and store which means that a container may have better performance and is quicker to start and migrate. Explaining this connect is easier done with figure 1:

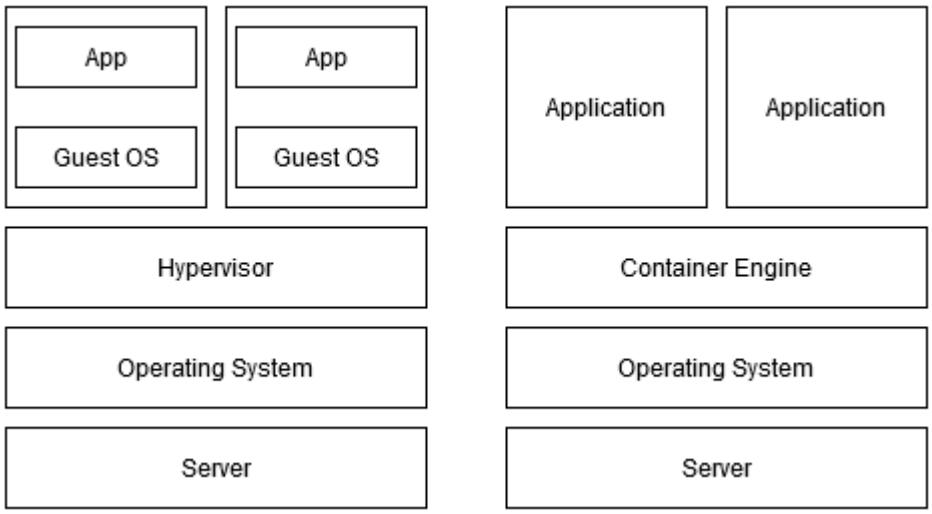

Figure 1 Hypervisor vs Container Engine (Authors own)

2.2 Continuous Integration, Continuous Development

According to CodeShip (2020), Continuous Integration "…is a development practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration can then be verified by an automated build and automated tests." Continuous Integration will henceforth be called "CI"

The practice of CI and the benefits it brings like revision control, automated testing, and build automation has made it a best practice for software development. The difference between Continuous Integration and Continuous Development is that in a Continuous Development environment, is that the deployment of the code to production is automated too. The flow of CI and CD is shown in figure 2.

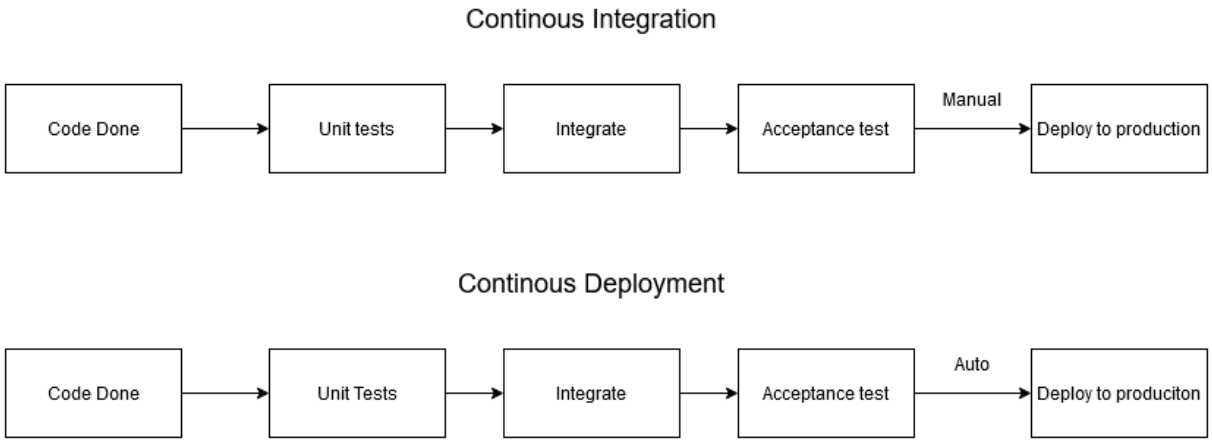

Figure 2 CI/CD flowchart (Authors own)

This is of particular interest when it comes to containers as a container can quickly be started and test the code, the image can also be run anywhere on any computer that has the prerequisite tools like Docker installed. This means that no matter what environment the image ends up in, whether it is Red Hat, Ubuntu or CentOS the container can be run and the content tested.

2.3 Docker and template

Docker first started in 2013 and quickly become prominent in the market as the product was collaborated upon by Fedora, Red Hat, and OpenShift (Techcrunch, 2013).

Docker is particular in the market because of the ease of use as the images used are created by using a Dockerfile. This is a text file that details what the daemon should do to create the custom image. To illustrate this process, a Dockerfile can look like the following example:

FROM ubuntu:18.04

LABEL maintainer="Rasmus Emilsson"

ENV SHELL /bin/bash

*RUN apt-get update && *

apt-get install -y vim

RUN echo "Hello world!"

This Dockerfile would create an image with an Ubuntu 18.04 operating system as a base, tagging Rasmus Emilsson as the author of the image. It would then run apt-get update and apt-get install -y vim to install vim and then say Hello World! Using the bash echo command.

This ease of use is attractive as small images can quickly be created for a given task using the regular Linux commands like apt-get, echo, etc.

Everything for this experiment could not be made to work in the Dockerfile that was used to create the final image. In particular, the bootstrap file required human interaction to be able to compile Firefox. The final Dockerfile for this experiment can be found in Appendix A.

Most of the commands are to install dependencies that the Firefox compiler needs to work. The bootstrap file that is supplied by Mozilla seen in the final Dockerfile contains most of the dependencies needed but not all. This was a problem while setting up as Firefox as the build instruction page specifies that only the bootstrap file is needed. (Mozilla, 2020)

This did not work so the bootstrap in the source code also needed to be invoked which was fully dependent on human interaction. This is why the source code is downloaded directly to the container which is true for every subsequent solution too so that every implementation is the same. After this bootstrap has been run using the "./mach bootstrap" command, it is possible to use "./mach build" to start the compiling. This is done outside of the terminal of the container as to not affect the performance as one can invoke commands upon a running container.

The source code can be mounted to the container as a volume or mount point but this made no difference in performance in preliminary tests on the machine. It also gave rise to

permission problems on the other container solutions. This also allows for the image to be used anywhere without having to download anything but the image.

The Dockerfile found in Appendix A served as the template to every other solution as the same commands and the last bootstrap needs to be run.

2.4 LXC (LXD)

LXC was the second choice of container engine because of its big market share and was first released in 2008, 5 years before Docker. It bears little resemblance to how images are created to docker as it had to be more manually created. LXD was used to create the image which is "…basically an alternative to LXC's tools and distribution template system with the added features that come from being controllable over the network." (Linuxcontainers, 2020). LXD in essence then is the same as LXC but with a better user experience creating and maintaining the images and containers created.

In essence, the steps to set up the LXC image is the same as Docker but each command has to be run manually. Using the Dockerfile as the template, the container is created first and then accessed using a terminal, the source code is downloaded and added to the image directly in this case too to eliminate any inconsistencies.

2.5 Podman

Podman was chosen as previously mentioned because of the growth potential of the project. The project was created and funded by Red Hat which makes it an alluring prospect for Linux containerization in an enterprise environment.

Two approaches were explored for this project. The first was to use the existing docker image and creating a container from that which translates the Docker instructions that exist in the file into code that Podman can understand. This means that any Docker image created for any task can be used by Podman making a potential migration to the platform from Docker easier.

A consideration with this approach however was that interpreting another service's way of creating an image might give rise to overhead performance differences. The second approach to this project then was to use the underlying container building tool "Buildah" directly to break the dependence of Docker and potential performance differences.

2.5.1 Buildah

With Buildah, the same Dockerfile that was used to create the Docker image can be used to create the buildah image. This once again makes migration to the platform easier as no new config files need to be created which gives credence to the growth potential argument.

Using this approach instead should eliminate the potential image performance differences as the image is native to buildah / podman.

2.6 Sysbench

Sysbench is used to test the performance in a synthetic setting as it can push the containers to their maximum performance. This workload tries to be representative of the performance of a given piece of software or hardware and in the case of this experiment, the data gathered from Sysbench is used to test the performance of the CPU and RAM in operations per 10 seconds.

This is interesting for the experiment as the compilation process of Firefox already pushes the CPU and RAM, this can then be used to determine where a potential bottleneck of a container solution lies.

2.7 Related work and limitations

Many studies have been made comparing traditional bare-metal, virtual machines, and containers like Spoiala, Calinciuc, Turcu & Filote (2016) who studied the difference between Docker and KVM. Not much or any work has been done to compare the difference in performance between the different container platforms while searching through IEEE Xplore, ACM Digital Library, and Google scholar.

The related works cited in this paper have mostly been focused on the performance of the RAM and CPU using various benchmarking tools. This study continues on that path focusing on those parameters too. This is because for the experiment that is run, compiling Firefox, it is unclear what to focus on when it comes to I/O or network performance and the related works do not have a good way of measuring this performance either. It is also out of the scope of the experiment considering the extensive amount of testing that will be done which takes up much time. More on this can be found in chapter 5.2.

3 Problem description

This chapter details the problem the project focuses on in more detail, as well as explaining why this research is important, and what it can add to the scientific community.

3.1 Motivation

As different CI solutions are used in larger capacities, the code that is compiled and tested needs to be tested quickly. For this reason, containers are usually used to compile the code as the image can be applied quickly for this task.

What has not been done before in an academic fashion is to test the performance of the different container solutions in this regard. One can find comparisons to bare-metal for many of the container solutions but not against each other. There may be performance comparisons done by the companies who create the container solutions but these have not been found.

The choice of container for a given task in a CI capability could be influenced by the performance of the container, for example how quickly it handles a process like compiling a big program. To have data which container solution performs better or worse would be an advantage for someone who needs to choose between them, for example, a system administrator that is setting up a corporate environment for Continuous Integration.

3.2 Research question

The concrete research question this experiment aims to answer is:

> *"How does performance differ between Docker, LXD, Podman, and Buildah running a heavy workload"*

The aim then of this experiment is to test the performance of a number of popular container solutions to see which one works best in a given scenario. In this experiment, the workload aims to be representative of a real scenario that a container may be used for which is to compile code.

The experiment is conducted on a laptop that has been factory reset so as to not alter the performance running other programs than needed as the computer had been used for leisure before, more on this can be found in the experiment design section. Specifications for this computer can be found in Appendix C.

The hypothesis is that there will be little variance in performance based on experiments (Felter, Ferreiera, Rajamony & Rubio, 2015) where Docker performed 2% slower than bare-metal and Dan Julius (2016) who found that LXC performed roughly 5% slower than bare-metal in experiments.

3.3 Objectives

To produce a satisfactory result, the workflow is divided into four parts as per the process outlined by Wohlin (2012). The steps will be discussed in greater detail in the following chapter. The steps in order are as follows:

1. **Gather data on containers –** This step is to establish what container solutions exist and which ones to focus on This is done in the background, starting in chapter 2.3. This includes feasibility in the context of the lab and the market share that each solution has. A deprecated container solution like RKT for example is not eligible and neither is cloud provided instances like Amazon EC2 as they are costly, introduces network variables, and a loss of control.

2. **Design of experiment –** When the appropriate container solutions have been picked from the criteria set in step 1, designing the process of the experiment is next. This includes how to make a container compile Firefox and how to test the performance.

3. **Perform the experiment –** From the parameters and criteria set, perform several tests, and gather the data that is extracted in this step. The tests are to compile Firefox in the container and extract time to complete, CPU usage and RAM usage.
4. **Analyze the data –** Analyse the data taken from step 3 while applying appropriate statistical methods.

4 Methodology

This chapter will explain the steps that were outlined in chapter 3.3. This study is an experiment and was chosen because it was the most fitting method to answer the given research question. This is mostly due to not finding that this research question has been explored before where the raw computational performance of containers has been tested quite this rigorously against each other.

Using Wohlin (2012) per recommendation, the steps of creating an experiment is conducted in five distinct steps but may be done in a different order than what is presented here.

3 Scoping

4 Planning

5 Operation

6 Analysis and Interpretation

7 Presentation and Package

This experiment and study follow this framework but may omit parts that do not apply to an experiment in a computational environment. What these steps entail is explained further in the next sub-chapters.

4.1 Scoping

Scoping determines foundation for an experiment and according to Wohlin (2012) where "The purpose of the scoping phase is to define the goals of an experiment according to a defined framework"

The object of the study is arbitrary and can be "products, processes, resources, models, metrics or theories". Purpose determines the intent of the experiment while quality focus deals with the primary effect that is being studied. Perspective outlines the viewpoint from where the results of a given experiment are interpreted and context who and/or what is involved in the experiment.

Applying this model to the experiment at hand, the template becomes the following:

Analyse **Selected Container Solutions**

for the purpose of **comparing performance**

with respect to their **implementation tactics**

from the point of view **of developers/administrators of CI solutions.**

in the context of **a home computer**

With this template filled out, the scope and aim of the experiment is clearly laid out so as to both know what to do and what not to do to keep the experiment grounded and appropriate for the given time frame.

4.2 Planning

The next step after scoping is planning, where scoping determines why the experiment is to be conducted, planning determines how the experiment should be conducted. This is also the longest and most comprehensive part of Wohlin's five steps.

The planning stage is divided into seven steps which will be explained in short form in the following subchapters. All seven steps have not been used as they don't fit the experiment done; the omitted steps are context selection which mostly deals with risk in a company context. Selection of subjects deals with people to choose for an experiment like interviews, surveys, etc. which are not used in this experiment.

4.2.1 Hypothesis formulation

This is where the hypotheses for the experiment is created. According to Wohlin, two hypotheses need to be formulated based on the experiment definition.

The Null hypothesis "states that there are no real underlying trends or patterns in the experiment setting; the only reasons for differences in our observations are coincidental" and is something that one wants to avoid when doing an experiment. That is, the data that is extracted while measuring CPU, RAM, and Wall Time while compiling and during Sysbench tests shows nothing.

The Alternative hypothesis instead "is the hypothesis in favor of which the null hypothesis is rejected." This means that a pattern has emerged and conclusions can be drawn from the results which is the preferred state.

4.2.2 Variables Selection

The variables spoken about here are the selection of independent and dependent variables. The independent variables, in this case, are variables that can be controlled and may change in the experiment. The dependent variable is what one can measure the effect of changes in the independent variables from.

For this experiment, the independent variables consist of the container solutions that have been picked based on criteria that have been introduced in the background chapter. The containers that have been selected are Docker, LXD, Podman, and Buildah.

The dependent variables then become the benchmark method which in this case is the compilation of Firefox itself and the subsequent statistics that are collected by the compilation process and output in human-readable form. These statistics are expired wall time, RAM usage, and CPU usage. It is also the benchmark information gathered through Sysbench like CPU and RAM operations.

Measurement scales are also determined at this stage. The most important part is time to complete the compilation as what is determined by this experiment is which solution will be able to compile Firefox the quickest to facilitate CD/CI. To determine what can be improved upon, CPU and memory utilization will be monitored but also IO operations. These statistics are all compiled by the Firefox build mechanism.

4.2.3 Experiment Design

"To draw meaningful conclusions from an experiment, we apply statistical analysis methods on the collected data to interpret the results" which means that the design of the experiment needs to be correct for the results to also be correct and valid. The general principles for designing an experiment are Randomization, Blocking, and Balancing in combination with each other." These concepts are explained in further detail below.

Randomization: All statistical methods as per Wohlin et al. requires that observations are made from random variables and are used to average out an effect that might otherwise present itself as a larger effect. In this experiment, this could be random fluctuations in CPU usage that are then averaged out through an extensive number of tests. In a study of people, this would be to choose people that can represent a larger number of people.

Blocking: When an experiment contains a factor that is known to have an effect on the result but is not interesting, blocking can be used to eliminate this factor. That is to place the factor in question in one block and the rest that is interesting in another block.

In this experiment, this is done to I/O performance as what is looked at is Wall time, CPU usage, and RAM usage.

Balancing: A balanced design of an experiment is to have an equal number of tests done. This is done to simplify the analysis of the data and to strengthen the significance of it. In this case, balancing is done in this experiment by running the tests an equal amount of times.

4.2.4 Instrumentation

There are three types of instrumentation in an experiment, Objects, guidelines, and measurement tools.

The objects in this instance are the specification of machines and config files that may be used. Measurement tools are the tool that the compiler of Firefox uses when it runs which gives an output of Wall time and CPU usage. The “top” command is used to measure RAM usage.

The guidelines detail how the experiment should be run, for example, how many times per container solution or what operating system to use.

4.2.5 Validity Evaluation

Wohlin et al. describe four threats to the validity of an experiment where two of the main threats can be applied to this experiment. The different threats are conclusion validity, internal validity, external validity, and construct validity. The internal threats are moot as they are only applicable when people are involved in the data like in a survey or interview.

What will then be detailed is how the approach to conclusion validity, external validity and construct validity is done to mitigate the threats.

Conclusion validity is described by Wohlin as “Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment”

The biggest threat in this category is *Low Statistical Power*. That is if the data is not powerful enough to give a true pattern in the data, the risk of the conclusion being wrong is high. This is mitigated in this experiment mainly by running the tests 100 times for each container solution which should smoothen give enough data to be valid. All tests will also be processed with a T-test which can be found in appendix E.

Fishing and the error rate deal with the bias of the researcher that “fishes” after a specific result which would make the data invalid because it loses independence. This is something that the individual researcher needs to be aware of and try to negate by discarding bias and/or do a thorough analysis of the data without seeing a pattern where there Is none.

Reliability of measures. The tests for a given implementation should output a similar result. If one benchmark run takes 100 seconds and the next one takes 1000 seconds, something has gone wrong and the outlier in the test should be discarded so as not to taint the conclusion data.

Construct validity “…concerns generalizing the result of the experiment to the concept

or theory behind the experiment."

Inadequate preoperational explication of constructs is the threat that the constructs of a given experiment are not properly defined. This can be mitigated by

Mono-method bias states that if only one measurement is taken the risk of faulty data is high. This is mitigated by measuring not only time to complete compilation but also CPU and RAM utilization which gives the experiment several performance values.

"Threats to **external validity** are conditions that limit our ability to generalize the results of our experiment to industrial practice." In this category, *Interaction of setting and treatment* is the relevant concept. This is a threat if the experiment is not representative of for example the industrial practice. For example, if one uses old tools to conduct an experiment while newer tools are usually used in the industry. This is mitigated in this case by running only relevant container solutions that have a sizeable market share or are expected to see growth and not using deprecated solutions.

4.3 Operation

This is the phase where data is collected through the treatments set in the previous steps. In this step everything should be prepared so the experiment can be run in a smooth manner and so that it can follow the plans.

4.4 Analysis & Interpretation

When the experiment has been run and data collected the result needs to be analyzed using a statistical method. This is usually done using descriptive statistics first to display the data in human-readable form and to perform hypothesis testing.

4.5 Presentation & Package

The last step is to present the data through some appropriate medium. This report is this experiment's presentation and package which has been iterated throughout the course of the experiment.

5 Designing the experiment

This chapter will explain the setup that is used to run the benchmarks and collect the data. It will explain the environment of which became quite particular and then go into detail about the different container solutions.

The choice of containers is not an arbitrary one but reflects the market as of the current date. The data from which the choices come from is a company called Datanyze (2020) that has compiled the top 1 million websites from the Alexa service provided by Amazon. From this list, the market share of the container solutions can be seen. The chosen technologies

are Docker with around 30% market share and LXC which also holds around a 30% market share.

This was deemed as too few solutions so Podman / Buildah was chosen because of the growth potential as it is backed by IBM/Redhat.

5.1 Host server and operating system

The environment that was first planned to conduct the experiment in was the NSA lab on the HiS campus. But due to the situation with the coronavirus at the moment where the experiment was to be started, this had to be changed. The best choice that was available was a moderately powerful laptop that was deemed sufficient for the purpose of compiling Firefox and giving a fair result. The specifications for the laptop will be provided in Appendix A.

This laptop was formatted and a new install of Ubuntu 19.10 was installed. 19.10 was chosen over version 20.04 because that is what the containers used as an operating system. This is because the Firefox compilation is fully dependent on Python 2.7 and Python 3.X to work which was is not supported in Ubuntu version 20.04 and could not be made to work in a reasonable time frame. While it should make no difference in regards to performance to use different operating systems, Ubuntu 19.10 is still a supported operating system at the time of writing.

This laptop also serves as the baseline of the experiment as the bare-metal approach to building Firefox and the Sysbench tests.

The solutions all have their own settings to use and optimize with combined with global cgroups that could limit or delimit the resources given to the containers. None of these settings nor cgroups have been touched as by default in Ubuntu, the container solutions get full access to the resources when running as root as is the case in this experiment.

The compilation process is Firefox's own called "Mach". Invoking the "Mach Build" command on the source code folder will start the compilation process and the capturing of performance data.

5.2 Environment consistency

To minimize the effects of the surrounding environment on the tests, some measures have been taken.

Heat and ambient temperature: The tests were all done during the same month, April, which kept the ambient temperature similar between each test and each container solution. The laptop was not moved and was allowed to rest for ten minutes between each benchmark run.

Operating system and existing programs: The computer had been used for leisure before the experiment so the computer was factory reset and formatted before installing a new image of Ubuntu 19.10 on it. This was repeated between each container solution to keep the environment clean and consistent between each solution so that for example Docker was not running in the background by mistake while testing LXC.

From the clean 19.10 image found on the official Ubuntu website, only what was needed to run a specific solution was installed to ensure no other programs were interfering with the benchmarks.

Network: Network was not measured in this experiment because this parameter was hard to keep consistent. There were many devices on the network which were hard to control, laptops, tablets, phones, other computers and so on that could potentially interfere with the results. Testing network performance was therefore deemed not applicable in this scenario because the validity threat was too great.

I/O: Testing the memory of the computer, both the storage media (SSD) and the RAM is interesting but there were no clear indications on how this is to be done nor what parameters to look for because these are not things that been taken into account in the related works.

6 Data Gathering

This chapter will detail how the data used for analysis was captured to allow for analysis which is chapter 7.

6.1 Mach statistics

Capturing the data from a container is a somewhat different endeavor compared to capturing data from the host system. This is because as per Docker (2020) “Docker makes this difficult because it relies on lxc-start, which carefully cleans up after itself. It is usually easier to collect metrics at regular intervals…”.

It was because of the difficulty of capturing the data as the container exited that the built-in metric collection tool of mach was used. This data looks like this for example:

Overall system resources - Wall time: 2979s; CPU: 94%; Read bytes: 11947162624; Write bytes: 9137963008; Read time: 174444; Write time: 76997

This data was then manually entered into a text file for later processing. This method is understood to be error-prone so good care has been taken to ensure each test’s results were copied down correctly. All tests have also gone through T-testing to prove the validity of the numbers. After each run, the computer was rebooted to eliminate any cached information, and the test runs again repeated 100 times for each container solution.

6.2 RAM usage through Top

As one can see, RAM is not measured in this tool. The chosen method to measure RAM usage is to use the top command in conjunction with the command to start the compilation. This top command is then written to a file for later analysis.

The top command is not run at once as that would skew the results when the compilation has not started completely. Therefore, the top command is "slept" until compilation has started which was determined to be fully done by 15 minutes after preliminary tests. After the first top command is done, RAM is measured in the same way every 5 minutes 4 times more to determine an average. The command then to start a given container, start the compilation, start top and then writing the output to a file became:

"X start container ; sleep 10 ; exec /usr/local/src/firefox/mach build ; sleep 900 ; top -b -n 1 -p1 >> Xcontainer.txt ; sleep 300 ; top -b -n 1 >> Xcontainer.txt ; sleep 300 ; top -b -n 1 >> Xcontainer.txt ; sleep 300 ; top -b -n 1 >> Xcontainer.txt ; sleep 300 ; top -b -n 1 >> Xcontainer.txt "

The relevant data can then be extracted from this file and entered into the data to be used. This is admittedly not the best way to extract the RAM usage data as the top command measures the overall usage of the entire system. This problem is mitigated by having a new install of Ubuntu for each container solution with a reboot between each test which should still give a fair representation of the memory usage.

6.3 Sysbench statistics

The benchmarks done through Sysbench were written to a log file as it ran using a small script in conjunction with a command line. The script looks as following for the CPU tests:

for each in 1 2 4; do sysbench --test=cpu --cleanup --cpu-max-prime=20000 --num-threads=$each run; done

This was invoked in the command line using:

for each in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20; do /script; done | cat >> container.txt

The data was then parsed through in an online text extractor called "molbiotools.com" with a regular expression which to get the number of events was: "(total number of events:)\s*\d*". This could then be downloaded as a CSV file and imported into excel for processing. An example of how this data looks can be found in Appendix D.

6.4 Parameters

The parameters that are of particular interest for this experiment is wall time serving as the basis and then CPU and RAM to determine why a given solution has better or lower performance than another solution. Wall time is sensitive to outside influences but is mitigated by running the tests a large number of times and restarting the computer between each test to reset any cache or stop any program that has started.

During the sysbench tests, the time to complete a task was not important but the raw performance of the CPU and RAM was measured.

7 Data analysis

This chapter will detail how the collected data will be processed to later be analyzed in the conclusion in Chapter 8.

7.1 CPU, RAM and Wall Time

As mentioned in chapter 6.1, the data that was used for CPU and Wall Time was provided by the Mach build command. This information was then entered into a text file for later processing. The “top” data was as previously mentioned written to file and then entered into excel for processing. Bare metal numbers collected the same way are used as the baseline for all the tests.

Each measured variable had their median value calculated using Excel based on the data which is shown in figure 3.

7.2 Wall Time

In figure 3 the Wall Time of the different compilations' median can be seen over 100 benchmarks each.

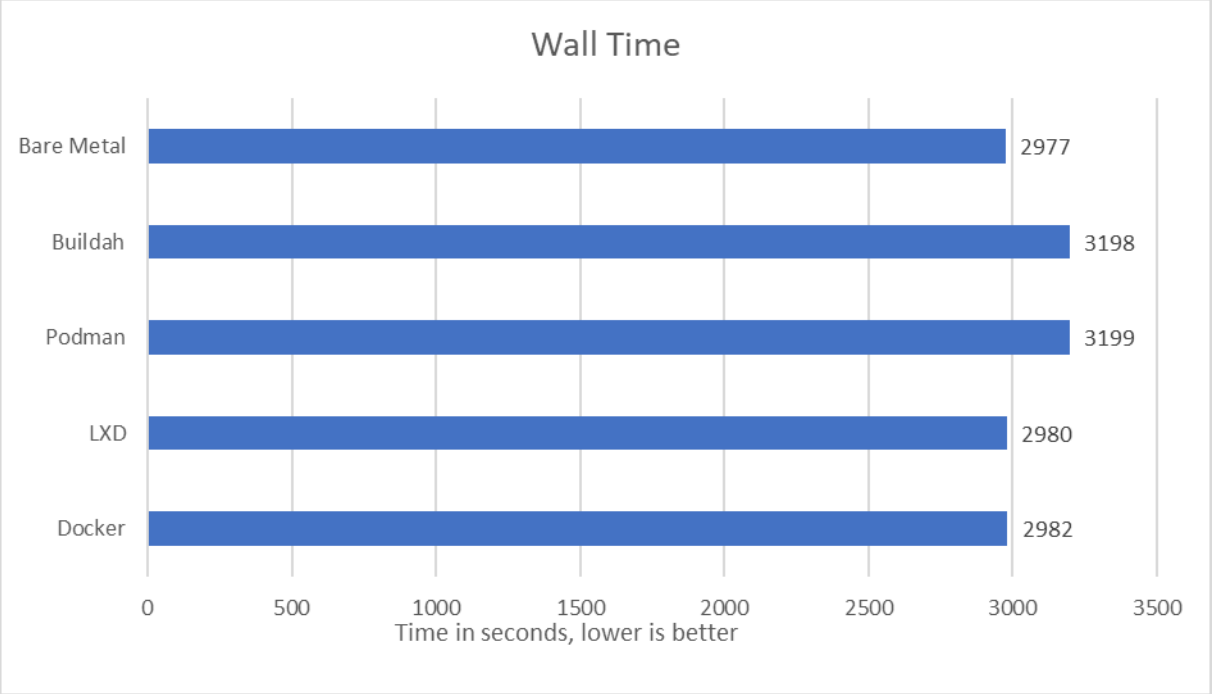

Figure 3 Wall Time (Authors own)

Wall time for this experiment is especially interesting because of the implications to Continous development where time to build the code is imperative which means that lower is better. As can be seen, LXD and Docker are very close to tied with only a few seconds' difference. Each compilation runs for approximately 2980 seconds equating to around 49 and a half minutes.

What is interesting is that Podman and Buildah fall behind the other containers with a few hundred seconds. This is especially interesting considering that RAM and CPU usage as can be seen in figure 5 and 6 is similar throughout all the tests. Potential reasons for this behavior will be explored in Chapter 8.

Docker	LXC	Podman	Buildah	Bare
4,570284	4,853236	15,9088	15,79897	4,312436

Table 1 Standard Deviation of wall time (Authors own)

Table 1 shows the standard deviation calculated in Excel using the wall time numbers. Docker, LXC, and the bare-metal solution show a fairly low standard deviation at around 4.5 seconds each while Podman and Buildah shows a higher deviation overall at roughly 15.5 seconds each. This may be indicative of something that Podman does that is different than

the other implementations that make it more unstable. The difference is quite small, around ten seconds but it is a pattern that is discernible.

7.3 CPU

Figure 5 shows the CPU usage for each solution. The raw data from the mach build command is averaged as per the developers on the #build channel on Mozilla's Matrix server. The data used in the chart is the median calculated using the averaged numbers.

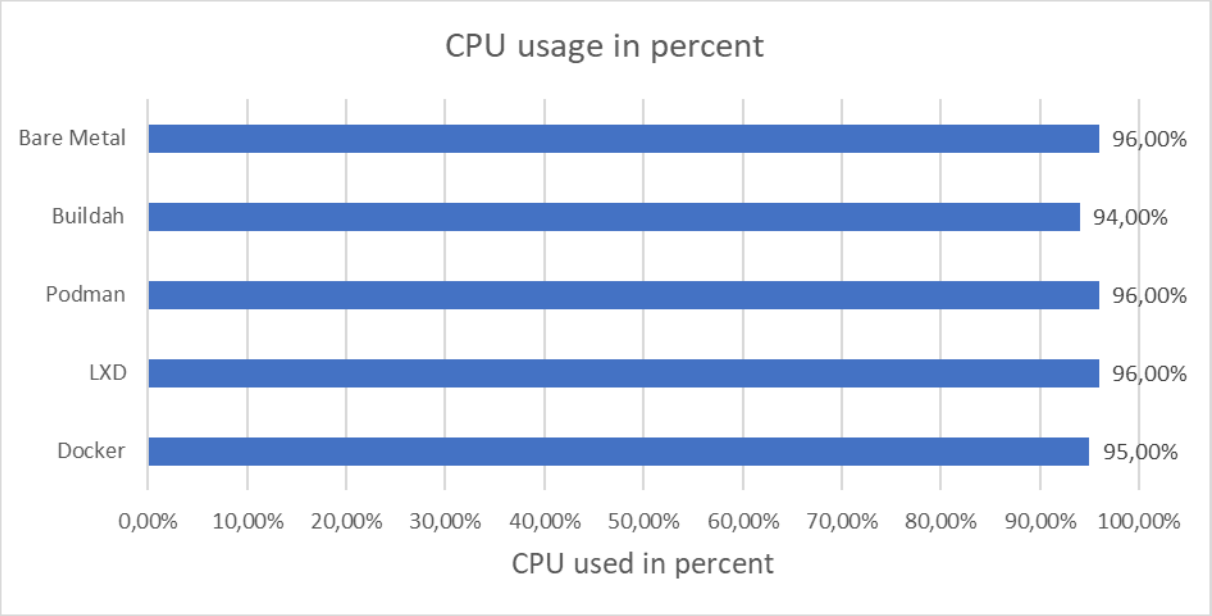

Figure 4 CPU (Authors own)

As one can see in the chart. The amount of CPU percent used during the compilation remains roughly the same which is to be expected as no limits have been set on the containers or any other settings have been set limiting the performance of the CPU.

The average CPU usage is slightly higher for Podman and LXD at 96% while the average for Docker is 95% and 94% for Buildah. This is a high percentage compared to RAM as seen in figure 4 so that might point to a bottleneck in the system.

Unlike wall time, no big fluctuations between each solution could be seen and all of the tested solutions seem to expend the same amount of CPU resources. This shows that despite the differences in wall time, the compilation process uses similar amounts of total CPU in percent. The cause for the discrepancy in wall time is therefore apparently not the CPU.

7.4 CPU operations per 10 seconds / Sysbench

In table 1, the median CPU operations per 10 seconds can be seen captured through Sysbench.

Bare Metal Tot. CPU OPS	Docker Tot. CPU OPS	LXD Tot. CPU OPS	Podman Tot. CPU OPS
8946,5	8935	8943,5	8941

Table 2 Median CPU ops, higher is better. (Authors own)

This median is calculated using 1, 2, and 4 threads ran 20 times per thread. The higher the number in the table the better as more operations have been done. the median does not differ much between the different solutions, the bare metal solutions have a slightly higher median than the rest, and LXD has the highest median of the container solutions. This is in contrast to the Firefox compilation where Podman was slower than the rest.

7.5 RAM

Figure 3 shows the RAM usage collected through top and calculated into a median over 5 snapshots per compilation.

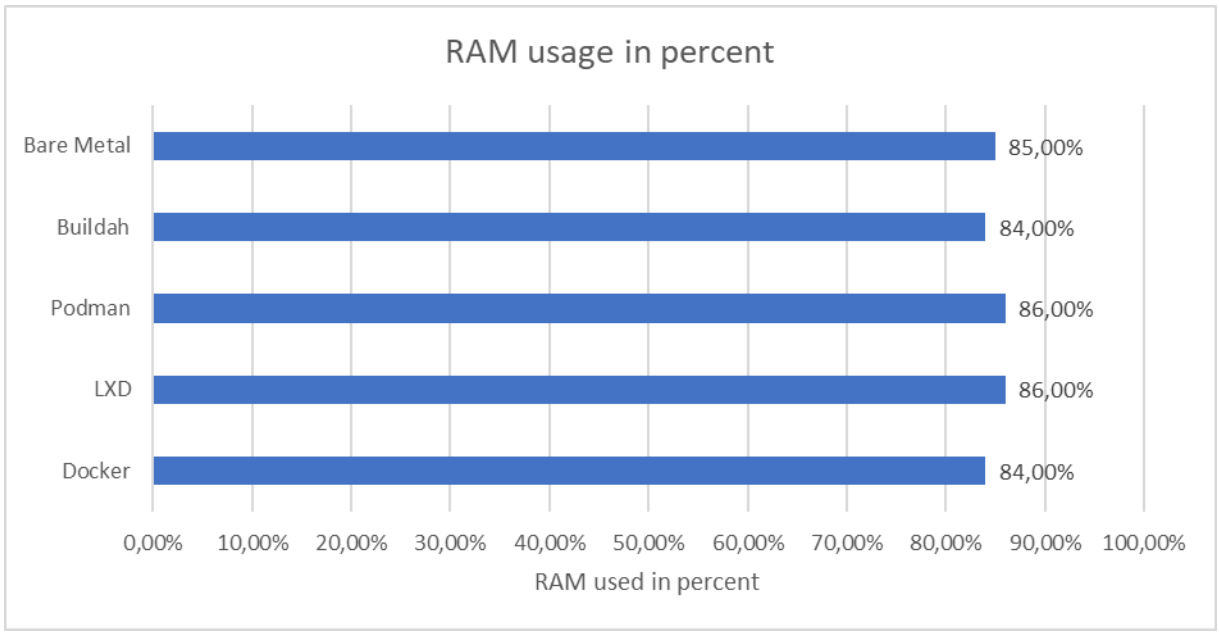

Figure 5 RAM usage (Authors own)

As with CPU in chapter 7.3, the average RAM used is consistent compared to each other. Podman and LXD utilize the most RAM at 86% each while Docker and Buildah use 84% each.

This is an interesting figure as the CPU is working at almost max while RAM works at a lower rate. This is not due to any of the snapshots showing a lower used percent as shown in table 3 which shows the standard deviation of the median for each run. As with the CPU, the

amount of used RAM is apparently not the cause for the fluctuation in wall time seen in Podman.

Docker	LXD	Podman	Buildah	Bare Metal
1,423903	0,764918	0,847585	1,468469	1,085311

Table 3 Standard Deviation of RAM usage (Authors own)

Reasons for this behavior are explored in Chapter 8.1 but the standard deviation shows that the snapshots of ram usage are not the cause for the discrepancy. The highest standard deviation that is seen is Docker and Buildah at 1.4% each, bare metal sits at almost exactly 1% while LXD and Podman are under 1%.

7.6 RAM operations per 10 seconds / Sysbench

In table 2, the median RAM operations per 10 seconds can be seen captured through Sysbench.

Docker Tot. RAM OPS	LXD Tot. RAM OPS	Podman Tot. RAM OPS	Bare Metal Tot. RAM OPS
86039333	87679615	86339320	87806082

Table 4 Median RAM operations, higher is better. (Authors own)

Running the test 20 times and calculating the median, the numbers in the table above are revealed where the higher number is better. As with CPU operations in chapter 7.4, the median does not differ much between the different solutions, the bare metal solutions have a slightly higher median than the rest, and LXD has the highest median of the container solutions. This is in contrast to the Firefox compilation where Podman was slower than the rest.

8 Conclusions

This chapter will from the data gathered during the experiments draw conclusions as to how the different containers perform during heavy load and discuss the research question and if it has been answered.

8.1 Analysis and Conclusion

Based on the gathered data in chapter 7, the differences between each solution was small to insignificant. The one interesting difference that sticks out is the disparity that can be seen with Podman looking at Wall Time.

Here, Podman's compilation takes roughly 7% longer averaging out to around 220 seconds difference per benchmark run. This is especially strange considering that the amount of CPU and RAM used is consistent or even higher than for example Docker. This is not completely

unprecedented as pointed out in Chapter 3.2, LXC in one test performed 5% worse than bare-metal. A 7% difference may therefore also be in the normal span of performance for containers overall.

Analysis of both the posted mach statistics and the top commands shows no immediate difference to Docker and LXD in how it handles the compilation process.

As per figure 3, there seems to be a bottleneck in the system as the CPU is running at almost 100% while the RAM runs at around 85% capacity. This is further evidenced by Table 3 that shows that this is how the RAM actually performs while compiling Firefox. This can affect the performance of the different solutions but even so, all the solutions have the same bottleneck. However, the Sysbench tests show that the different solutions can perform at similar levels.

The problem might, therefore, lie in how the build mechanism that Mozilla uses interacts with the way Podman works. Mozilla has been reached out to, in specific their build team on the Matrix server but no answer has been given.

8.1.2 Podman / Buildah explanation

Considering this disparity that was unexplainable by a cursory look at the statistics. The developers of Podman were contacted to get a better explanation. This was done through email first where the first contact became a developer that wants to remain anonymous. The conversation moved to the podman IRC channel on irc.freenode.net. Several more developers joined the conversation to try to remedy the problem or diagnose the cause of the disparity.

Unfortunately, no clear reason could be found for the disparity as of writing between the platforms although hypotheses have been formed and proposed by the developers. These range from how the CPU and RAM are used to network overhead but the culprit speaking to the developers seems to indicate that the problem is CPU, a quote from one of the developers: “it might be nice to use a simple reproducer that is CPU only”.

Considering the benchmarks done with Sysbench, Podman showed no signs of being much slower to carry out CPU operations than the other solutions. The problem might then lie in how Firefox is built instead of pointing to a problem with the container at large.

Trying to remedy the problem and dig that deep down into the workings of Podman was out of the scope for this experiment but is something that future work could look at.

8.2 Answer to research question

The research question posed in chapter 3.1 was “How does performance differ between Docker, LXD, Podman, and Buildah while running a heavy workload”

The hypothesis was that no major difference in performance would be seen between LXC and Docker as previous benchmarks have shown there are few differences in other applications (Tesfatsion et al. 2018). This proved to be true when testing for Docker and LXC considering that their performance was on par with each other.

Podman and Buildah were the outliers when testing and it is unclear why this is even after speaking to the developers. Considering the extensive measures to mitigate any disturbances in the testing, the set up of the lab is unlikely to be the reason for the difference in performance considering the good T-test score.

9 Discussion

In producing the data for this experiment, it became apparent that at least for this set up of hardware, the differences regarding performance are small where the only outlier is Podman. But considering that Podman is still worked on and has the backing of Red Hat behind it, the performance may see a boost to be on par with Docker and LXC.

What platform a system administrator would use for their continuous integration purposes then becomes a question of which features one wants for their platform of choice. If security is of importance for example, then LXC might be a good candidate based on the many permission problems found while implementing in this study where LXC seems to place security at the forefront by default.

The reason for not finding any real performance tests between the different containers is probably because there are very few in real use cases. The vendors have most likely done their own performance tests but no one is marketing their solution as faster than any other and none is posting their results.

The plan was also to test Docker for Windows and how it would perform in comparison. However, due to time restraints and also how to implement it to make it a fair comparison, this was omitted. Some testing was done on another computer for this and no implementation problems were found but these tests could not be used.

9.1 Contribution

The data produced in this experiment gives insight into the performance of the two most popular and used containers combined with one solution that has growth potential. While the performance differences were minimal except for Podman, the data is valuable for someone that is setting up an environment using containers. Many parameters need to be taken into account while setting up a corporate environment, security, performance, sustainability, economics, and so on.

This experiment could be used as-is to not have to care about the performance side or expand upon it by testing for I/O and/or network performance.

10 Future work

Potential future work can be done to study if the same pattern of discrepancies in performance can be seen with optimization and/or with better hardware. The study could also incorporate more platforms and move to the cloud. An interesting prospect would be to test the network performance of the platforms to see if the network makes a difference.

This experiment tested one use case for CI/CD containers which is to compile a big project. Other use cases for containers could be tested like a WebRTC server like Kurento Media Server as (Spoiala, Calinciuc, Turcu & Filote, 2016) did but test more than Docker and bare metal.

Podman is also working on version 2.0 at the moment which will be released shortly as of the time of writing (Podman, 2020). The question of the inconsistent results could be revisited in that case to see if the problem has been remedied.

REFERENCES:

Baude, B., (2020). *Update On Podman V2*. [online] podman.io. Available at: <https://podman.io/blogs/2020/05/13/podman-v2-update.html> [Accessed 14 May 2020].

Comeu L. W. (1982). *CP/40 – The Original of VM/370*. [online] Available at: <https://www.garlic.com/~lynn/cp40seas1982.txt> [Accessed 25 March 2020]

Datanyze. (2020). *Containerization Market Share Report | Competitor Analysis | LXC, Docker, Kubernetes*. [online] Available at: <https://www.datanyze.com/market-share/containerization--321/Alexa%20top%201M> [Accessed 15 March 2020].

IBM. (1972). *IBM System/360 Model 67 Functional Characteristics*. [online] Available at: <http://www.bitsavers.org/pdf/ibm/360/funcChar/GA27-2719-2_360-67_funcChar.pdf> [Accessed 25 March 2020]

Linuxcontainers.org. (2020). *Linux Containers - LXD - Introduction*. [online] Available at: <https://linuxcontainers.org/lxd/introduction/> [Accessed 17 March 2020].

MDN Web Docs. (2020). *Linux Build Preparation*. [online] Available at: <https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Simple_Firefox_build/Linux_and_MacOS_build_preparation> [Accessed 20 March 2020].

Redhat.com. (2020). *What Is Virtualization?*. [online] Available at: <https://www.redhat.com/en/topics/virtualization/what-is-virtualization> [Accessed 5 May 2020].

Spoiala, C., Calinciuc, A., Turcu, C. and Filote, C. (2016). *Performance comparison of a WebRTC server on Docker versus virtual machine*. [online] IEEEXplore. Available at: https://ieeexplore-ieee-org.libraryproxy.his.se/document/7492590 [Accessed 24 Oct. 2019].

Techcrunch.com. (2020). *Techcrunch Is Now A Part Of Verizon Media*. [online] Available at: <https://techcrunch.com/2013/09/19/dotcloud-pivots-and-wins-big-with-docker-the-cloud-service-now-part-of-red-hat-openshift/?guccounter=1> [Accessed 18 April 2020].

Tesfatsion, S., Klein C., & Tordsson, J. (2018). *Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds*. [online] ACM Digital Library. Available at: <https://dl.acm.org/doi/10.1145/3184407.3184414>

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-29043-5

Xia, D., (2017). *Improving Critical Infrastructure Rollouts*. [online] Labs. Available at: <https://labs.spotify.com/2017/06/22/improving-critical-infrastructure-rollouts/> [Accessed 11 May 2020]

APPENDIX A:

Final Dockerfile, these are also the exact commands that are run on LXD.

FROM ubuntu:19.10
LABEL maintainer="Rasmus Emilsson"

ENV SHELL /bin/bash

ENV PATH="/root/.cargo/bin:${PATH}"

RUN apt-get update

RUN apt-get install -y wget python clang llvm mercurial

RUN apt-get install -y cargo

RUN wget -q https://hg.mozilla.org/mozilla-central/raw-file/default/python/mozboot/bin/bootstrap.py -O /tmp/bootstrap.py

RUN chmod +x /tmp/bootstrap.py

RUN python /tmp/bootstrap.py --application-choice=browser --no-interactive; exit 0

RUN mkdir -p /usr/local/src/firefox-src

RUN hg clone https://hg.mozilla.org/mozilla-central/ /usr/local/src/firefox-src
WORKDIR /usr/local/src/firefox-src

Appendix B:

Wall Time in seconds, Std.dev and Median

Table 5 shows the statistics of wall time that was collected during the compilation process. Standard deviation is then shown on the second to last row and the median is shown last compiled from the previous numbers in the column.

Docker	LXC	Podman	Buildah	Bare
2987	2973	3178	3213	2972
2984	2973	3202	3181	2974
2981	2973	3220	3203	2977
2988	2973	3222	3201	2975
2989	2973	3224	3193	2979
2976	2973	3208	3178	2971
2976	2974	3223	3204	2983
2985	2974	3201	3216	2982
2981	2974	3181	3206	2972
2982	2974	3188	3210	2977
2984	2975	3199	3204	2974
2981	2975	3187	3180	2984
2979	2975	3187	3184	2970
2981	2975	3196	3195	2984
2989	2975	3194	3221	2980
2988	2975	3201	3223	2982
2979	2976	3201	3179	2982
2986	2976	3186	3196	2979
2986	2976	3209	3174	2976
2987	2976	3177	3189	2974
2984	2976	3175	3199	2980
2989	2976	3198	3197	2984
2986	2976	3204	3181	2977
2987	2976	3218	3215	2978
2983	2977	3198	3186	2970
2987	2977	3201	3188	2974
2976	2977	3198	3199	2977
2986	2977	3177	3195	2977
2978	2977	3203	3193	2974
2976	2977	3175	3213	2983
2980	2978	3208	3181	2974

2983	2978	3186	3210	2983
2988	2978	3204	3210	2970
2977	2978	3218	3205	2981
2976	2979	3194	3196	2980
2983	2979	3189	3187	2981
2981	2979	3187	3175	2974
2979	2979	3214	3205	2971
2983	2979	3188	3193	2975
2977	2979	3202	3211	2981
2977	2979	3185	3224	2972
2980	2979	3181	3173	2974
2989	2979	3201	3194	2970
2984	2979	3193	3225	2975
2977	2979	3181	3182	2980
2988	2980	3219	3208	2983
2976	2980	3185	3175	2980
2979	2980	3180	3173	2971
2985	2980	3174	3221	2983
2981	2980	3228	3216	2984
2989	2980	3224	3173	2983
2976	2980	3211	3195	2980
2976	2981	3206	3204	2976
2981	2981	3228	3191	2979
2989	2981	3177	3212	2981
2975	2981	3211	3189	2975
2985	2981	3223	3194	2978
2977	2981	3194	3217	2977
2976	2982	3179	3173	2972
2986	2982	3205	3174	2976
2984	2982	3223	3214	2983
2983	2982	3202	3205	2974
2986	2982	3228	3224	2973
2987	2982	3184	3224	2979
2989	2982	3227	3182	2973
2989	2982	3223	3180	2971
2983	2983	3212	3204	2984
2980	2983	3193	3210	2977
2977	2984	3223	3215	2972
2987	2984	3194	3177	2978

2989	2984	3180	3183	2983
2983	2985	3219	3207	2973
2976	2985	3178	3210	2970
2988	2985	3181	3212	2979
2979	2985	3205	3216	2982
2979	2985	3187	3219	2981
2980	2986	3209	3218	2975
2982	2986	3191	3183	2980
2988	2986	3191	3219	2978
2982	2986	3181	3192	2977
2975	2986	3219	3184	2979
2975	2986	3184	3180	2971
2978	2987	3209	3213	2980
2976	2987	3199	3176	2982
2978	2987	3220	3203	2974
2989	2987	3196	3184	2983
2977	2987	3222	3197	2975
2979	2988	3177	3177	2975
2983	2988	3222	3216	2971
2989	2988	3213	3180	2981
2981	2988	3226	3174	2982
2975	2988	3203	3207	2977
2984	2988	3205	3220	2983
2980	2988	3177	3210	2972
2987	2989	3188	3211	2983
2977	2989	3199	3210	2975
2975	2989	3190	3196	2980
2982	2989	3174	3223	2970
2987	2989	3190	3205	2977
2983	2989	3201	3173	2971
4,570284	4,853236	15,9088	15,79897	4,312436
2982	2980	3199	3199	2977

Table 5 Wall time, Std dev, Median

RAM usage in percent, Std.dev and Median

Table 6 shows the RAM percent used while compiling Firefox. The second to last row shows the standard deviation while the last row shows the median percent used.

Docker	LXD	Podman	Buildah	Bare Metal
84	86	85	85	84
85	86	85	83	86
85	85	86	86	84
84	86	85	87	85
84	85	86	84	86
83	85	87	83	84
83	86	87	84	84
86	87	85	87	87
84	87	85	87	86
84	86	87	86	85
85	86	87	83	85
83	87	85	83	85
83	87	86	84	87
83	85	86	85	85
86	85	87	87	86
83	87	85	83	87
83	86	87	83	87
84	86	87	86	84
84	86	86	87	85
85	87	85	84	84
84	85	86	84	85
85	87	86	83	85
83	87	87	87	84
83	85	86	84	86
84	86	85	83	85
86	86	87	85	86
84	87	85	87	86
83	85	87	84	86
87	85	87	86	84
84	86	86	85	85
84	85	85	85	87
87	87	85	87	84
87	86	85	84	84
83	87	85	84	86

87	85	86	85	87
83	86	87	83	84
87	85	85	86	87
87	85	87	83	85
86	87	87	87	86
84	86	87	84	87
87	86	85	86	84
83	86	87	84	85
84	86	86	83	86
84	85	85	86	84
83	86	85	86	84
85	87	85	86	85
86	85	87	86	84
83	86	87	87	85
83	86	87	87	86
85	87	85	84	85
86	86	86	87	84
85	87	87	87	84
84	87	86	85	84
84	86	87	84	86
86	86	85	87	86
84	87	85	83	85
84	86	85	86	84
85	85	85	83	84
83	87	87	86	86
83	86	87	87	84
85	85	85	85	87
86	87	87	86	84
84	86	85	85	85
84	86	86	87	85
86	87	87	85	85
85	86	86	87	87
87	86	86	85	84
85	87	87	83	87
84	87	85	83	87
84	86	86	86	86
84	87	86	83	86
87	87	87	86	85
87	86	85	83	86

87	87	86	83	85
83	85	87	85	85
87	85	86	87	87
87	85	86	87	87
84	86	85	85	87
85	85	87	85	85
84	86	85	85	86
83	87	86	85	87
83	87	87	85	86
86	86	85	84	85
84	87	86	83	87
87	86	85	85	87
84	85	86	85	86
83	86	85	87	87
86	87	85	87	85
83	87	85	85	85
85	86	87	84	85
87	85	85	83	85
87	87	87	84	84
85	87	86	87	84
84	85	86	84	87
86	87	86	85	85
87	86	85	87	86
85	86	85	87	87
87	86	87	85	86
85	85	87	87	85
87	85	86	83	84
1,423903	0,764918	0,847585	1,468469	1,085311
84	86	86	85	85

Table 6 RAM Percent, Std dev and Median

Appendix C:

Computer specifications:

MSI GP62MVR i5 8GB 256GB SSD GTX 1060

Intel Core i5 7300HQ

8 Gb DDR4 RAM at 2133 MHz

256 Gb SSD

Appendix D:

Example Sysbench statistics before regex:

sysbench 1.0.17 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time

Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
  events per second: 473.56

General statistics:
  total time: 10.0014s
  total number of events: 4737

Latency (ms):
  min: 2.08
  avg: 2.11
  max: 2.91
  95th percentile: 2.22
  sum: 10000.38

Threads fairness:
  events (avg/stddev): 4737.0000/0.00
  execution time (avg/stddev): 10.0004/0.00

<u>Example after regex:</u>

total number of events: 4737

Appendix E:

T-tests: All t-tests are measured against bare metal that serves as the baseline.

	Variable 1	*Variable 2*
Mean	2982,15	2977,23
Variance	21,09848	18,78495
Observations	100	100
Pearson Correlation	0,113932	
Hypothesized Mean Difference	0	
df	99	
t Stat	8,275388	
P(T<=t) one-tail	3,07E-13	
t Critical one-tail	1,660391	
P(T<=t) two-tail	6,15E-13	
t Critical two-tail	1,984217	

Table 7 Docker, Bare-metal

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	2980,81	2977,23
Variance	23,79182	18,78495
Observations	100	100
Hypothesized Mean Difference	0	
df	195	
t Stat	5,486516	
P(T<=t) one-tail	6,31E-08	
t Critical one-tail	1,652705	
P(T<=t) two-tail	1,26E-07	
t Critical two-tail	1,972204	

Table 8 LXD, Bare-metal

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	3199,51	2977,23
Variance	255,6464	18,78495
Observations	100	100
Hypothesized Mean Difference	0	
df	113	
t Stat	134,1787	
P(T<=t) one-tail	9,8E-127	
t Critical one-tail	1,65845	
P(T<=t) two-tail	2E-126	
t Critical two-tail	1,98118	

Table 9 Podman, Bare-metal

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	3198,35	2977,23
Variance	252,1288	18,78495
Observations	100	100
Hypothesized Mean Difference	0	
df	114	
t Stat	134,3422	
P(T<=t) one-tail	1,1E-127	
t Critical one-tail	1,65833	
P(T<=t) two-tail	2,2E-127	
t Critical two-tail	1,980992	

Table 10 Buildah, Bare-metal