OULUN YLIOPISTO
UNIVERSITY of OULU

Evaluating Performance of Serverless Virtualization

University of Oulu
Department of Information Processing Science
Master’s Thesis
Perttu Kärnä
6.5.2021

Abstract

The serverless computing has posed new challenges for cloud vendors that are difficult to solve with existing virtualization technologies. Maintaining security, resource isolation, backwards compatibility and scalability is extremely difficult when the platform should be able to deliver native performance. This paper contains a literature review of recently published results related to the performance of virtualization technologies such as KVM and Docker, and further reports a DESMET benchmarking evaluation against KVM and Docker, as well as Firecracker and gVisor, which are being used by Amazon Web Services and Google Cloud in their cloud services.

The context for this research is coming from education, where students return their programming assignments into a source code repository system that further triggers automated tests and potentially other tasks against the submitted code. The used environment consists of several software components, such as web server, database and job executor, and thus represents a common architecture in web-based applications.

The results of the research show that Docker is still the most performant virtualization technology amongst the selected ones. Additionally, Firecracker and gVisor perform better in some areas than KVM and thus are viable options for single-tenant environments. Lastly, applications that run untrusted code or have otherwise really high security requirements could potentially leverage from using either Firecracker or gVisor.

Keywords

serverless, firecracker, gvisor, docker, kvm, container, virtual machine, performance

Supervisor

University Lecturer, Jouni Lappalainen

Foreword

Even though the final years of my studies turned out to be more or less result oriented, I am glad that I made it to the end. During the past five years I got to meet lots of interesting and talented people both on the industry and as fellow students, which truly made the journey worthwhile.

I would like to thank my family and my girlfriend, Aurora, for continuous support and cheering throughout my studies. I also want to express my greatest gratitude to my supervisor, University Lecturer Jouni Lappalainen, for providing me with professional guiding and compassionate support in writing this master's thesis. Lastly, I want to thank my friends Hamish and Angus, for proofreading the paper, and suggesting grammatical fixes. Without the input of afore mentioned people, this thesis would most likely never have been completed.

Perttu Kärnä

Oulu, May 6, 2021

Contents

1. Introduction

Cloud computing has made virtualization an ordinary thing in software engineering. Services are more and more built cloud native, which enables leveraging operational features provided by cloud vendors. One of the emerging trends during the past decade has been serverless computing. While the term might suggest that there are no servers used, its official definition is not well established. Within this paper, the definition by Jonas et al. (2019) is used: "for a service to be considered to be serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage". Some of the common serverless services are FaaS (Function as a Service) services such as AWS (Amazon Web Services) Lambda (Amazon Web Services, 2021d), Microsoft Azure Functions (Microsoft, 2019) and Google Cloud Functions (Google, 2016). Typically, those services allow developers to write the application code in terms of functions that are triggered by events, such as HTTP requests, whereas running the code and managing the infrastructure is fully done by the cloud provider. While serverless is often used as a synonym with the term FaaS, they are not the same thing, and there exists other serverless services as well, such as AWS Fargate (Amazon Web Services, 2021c) and AWS DynamoDB (Amazon Web Services, 2021a).

Multitenant cloud computing and serverless computing has brought new requirements to virtualization. The first public serverless service (Baldini et al., 2017), AWS Lambda, was initially running function workloads on container-per-function basis, in dedicated EC2 instances for each customer (Agache et al., 2020). While traditional hypervisor-based virtual machines (VM) are working as virtual private servers, Agache et al. (2020) state that their long boot time and resource overhead is not very efficient and scalable for serverless computing, where workloads are often small. They mention also that container-based solutions are efficient on resources and boot time, but lack in security by default. In order to overcome this, AWS developed a new "MicroVM" called Firecracker, which they use for serverless services such as AWS Lambda and AWS Fargate (Barr, 2018). Google addresses the same issue by extending the security of containers with gVisor (Manor, 2018).

Previous literature, as presented in section 3, contains lots of studies regarding the performance and resource utilization of hypervisor-based virtualization and both LXC- (Linux container) and Docker container-based virtualization. While it is obvious that in most cases container-based virtualization overperforms the heavier hypervisor-based virtualization, it is not clear where the virtualization technologies used in serverless services, such as AWS Lamda and Google Cloud Functions, stand, when compared to the other technologies. Caraza-Harter and Swift (2020) compared Firecracker and gVisor to bare metal host, and found out that Firecracker has near-native performance in networking, CPU speed, file access and memory management, whereas gVisor seems to have significant overhead in memory management and networking. Young, Zhu, Caraza-Harter Arpaci-Dusseau and Arpaci-Dusseau (2019) compared gVisor runsc runtime for Docker containers to the default runc runtime, and concluded that "system calls are 2.2x slower, memory allocations are 2.5x slower, large downloads are 2.8x slower and file opens are 216x slower" when using runsc instead of runc for Docker containers.

Within this research, the performance differences between traditional KVM (kernel-based virtual machine), Docker containers both with runc (default) and runsc (gVisor) runtimes, Firecracker, and bare metal host, were investigated. In order to achieve that, a set of systematic benchmarks were conducted against each virtualization solution, as well as a simple end-to-end use case test was done and relevant metrics collected. The end-to-end test case was an imitation of a programming course assignment submission environment used at the University of Oulu, in which the code is submitted into a pipeline that runs automated tasks, such as building the source code and running a set of unit tests against it. As the existing literature does not describe the ideal features of a serverless computing hardware, the reasoning behind the development of Firecracker provided by Agache et al. (2020) was used as a base for evaluation. More specifically, the research questions that are addressed in this paper are:

1. Which virtualization technology is the most performant in terms of
   1. CPU events?
   2. Filesystem I/O?
   3. Network bandwidth and jitter?
2. Which virtualization technology provides the fastest launch time?

The context for the research questions is simple single-instance deployment without application-level multithreading. The CPU and filesystem performance were measured with sysbench benchmarking tool, and networking performance was measured with iperf3. Further constraints regarding the benchmarking are presented in section 5. Arguably also security and isolation are extremely important for serverless computing, but unfortunately the security has been left out of this paper due to time constraints.

The rest of the paper is organized as follows: section 2 contains related background information, section 3 presents a systematic literature review which describes the existing scientific knowledge about performance differences between containers and VMs, section 4 describes the research method, section 5 describes the empirical benchmarking evaluation, section 6 presents the results of the benchmarking, section 7 contains an analysis of the results, section 8 contains discussion and implications of the research and section 9 contains conclusions.

2. Background

The virtualization is a vague term that is frequently used in the domain of software engineering and computer science, regardless of the specific area. However, there is actually several different kinds of virtualization methods out there, that all work slightly differently and solve different problems. This section describes the background and key features of the virtualization technologies discussed in this paper, as well as how they are used on the industry.

2.1. Virtual Machine

According to Denning (2001), the origins of virtual machines (VM) can be traced back to 1960s, when the first projects around virtualization, such as M44/44X at IBM Yorktown Research Center, were started. Denning states, that the term "virtual" was coined from the field of optics, where "images in mirrors or at the focal points of lenses can be analysed as if the object's reversed or inverted copy were present". Since the 1960s, virtualization has become a major part of computer science and software engineering, not only in academia but also on the industry.

To the field of computer science, the term "virtual" refers to operating system components that simulate machines or devices (Denning, 2001). In practice, the machine needs an additional layer of software implemented to support virtualization (Smith & Nair, 2005), and it is possible to virtualize only parts of the system instead of full system. Today, there exists two different types of virtual machines: process VMs (or application VMs), and system VMs.

Process VMs are created and used solely as a runtime for a single application process (Smith & Nair, 2005). While the additional layer of VM brings some overhead to the application, one of the key objectives of process VM is to give cross-platform compatibility, meaning that programs written for the process VM can be run on top of different platforms (Smith & Nair, 2005). The process VMs become popular in early 1990s when the first implementation of Java was implemented along with Java Virtual Machine (JVM) (Binstock, 2015), and these days various languages are run on top of process VMs, such as .Net runs on top of Common Language Runtime (CLR) (Dykstra et al., 2020).

System VMs on the other hand offer a complete environment with guest operating system and multiple processes (Smith & Nair, 2005), and requires a hypervisor, which is the software layer that implements the required abstraction for system virtualization. Hypervisor can be either type-1 hypervisor (see Fig. 1), which is running directly on top of a host machine hardware, or type-2 hypervisor, which is running as a software on top of host machine operating system (Morabito, Kjällman, & Komu, 2015). Some examples of virtual machines are Xen (The Linux Foundation, 2016), Kernel-based Virtual Machine (KVM) (KVM contributors, 2016) and VMWare ESXi (VMWare Inc, 2021) with type 1 hypervisor, and Oracle VM VirtualBox (Oracle, 2021) and VMWare Workstation Pro (VMWare Inc, 2020) with type 2 hypervisor. Within this paper, KVM, which is built directly into Linux kernel, is used as a case example of VM.

Virtual machines are often used in modern software development, as practically all of cloud-based workloads are run on top of them. Typical use-cases are, for example, private virtual servers such as Digital Ocean Droplets (Digital Ocean, 2021) or AWS EC2 (Amazon Web Services, 2021b). Further use-cases of VMs are described in section 2.6.

2.2. Containers

Control groups (cgroups), that has been a part of Linux kernel since the version 2.6.24, is a feature that allows users to run processes and tasks with limited or shared set of resources such as memory, CPU and I/O bandwidth (Bellasi, Massari, & Fornaciari, 2015). Linux container (LXC) is a technology that utilizes cgroups and Linux namespaces among other things to provide a "environment as close as possible to a standard Linux installation but without the need for a separate kernel" (LXC contributors, 2021). What this means is, that whereas virtual machines virtualize the underlying hardware, Linux containers virtualize only the operating system.

According to Boettiger (2015), Docker is a popular open source technology for managing and running Linux containers. In addition to be able just run containers, Docker offers a platform to share, reuse, version and archive containers (Boettiger, 2015). Docker containers can be used in similar manner as virtual machines, but they work slightly differently. According to Docker Inc (2021), "a container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another". The users can build Docker images by defining build steps in a file called Dockerfile, and the built image is used by a container when it's running. Figure 1 describes the high-level architecture of Docker Engine and type-1 virtual machine for reference.

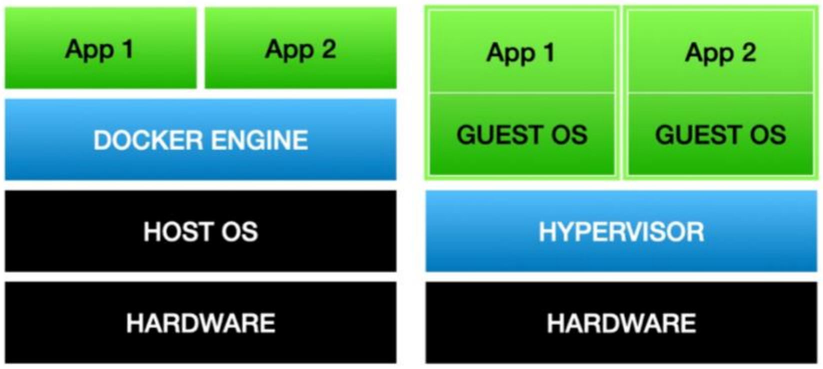

Figure 1. Docker Engine and type-1 hypervisor virtual machine.

Docker, CoreOS and some other container industry leaders launched a project called Open Container Initiative (OCI) in 2015 in order to come up with an open standard for container images and runtimes (The Linux Foundation, 2021). Docker donated their container runtime implementation, called runc, to the OCI as a reference implementation of a container runtime (The Linux Foundation, 2021), and that is the default runtime containers have when initiated with Docker's command line tools. Further in this paper, when Docker is referred to as virtualization or container technology, it means a container that uses runc as a runtime.

2.3. Sandboxed containers

One of the criticized things in Docker containers is their poor security, which is at least partially caused by the architecture of sharing the host kernel between containers (Combe, Martin, & Di Pietro, 2016; Yasrab, 2018) instead of using a separate guest kernel for each. One of the methods to fight this issue is to prohibit direct access from the container to the host kernel by introducing an additional layer that acts as a protective proxy between the host kernel and the container runtime. One of the technologies that uses this approach is gVisor.

According to Google LLC (2021a), "gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere". Being an open source technology initially developed by Google, it has been in use at least in Google App Engine and Google Cloud Functions for several years now (Manor, 2018). Whereas default Docker containers are traditionally seen as rather unsecure for multitenant cloud, gVisor is at least part of the solution Google has come up with.

Since gVisor implements the OCI model, Docker users can switch from the default runc runtime engine to gVisor runtime engine, called runsc (Young et al., 2019). Similarly as runc, runsc is implemented in Go language, and it contains several important components; Sentry, which receives the system calls from application, and Gofer, which "is a standard host process which is started with each container and communicates with the Sentry via the 9P protocol over a socket or shared memory channel" (Google LLC, 2021). This architecture makes it more secure than the default Docker runc runtime. The architecture of gVisor is presented in Figure 2.

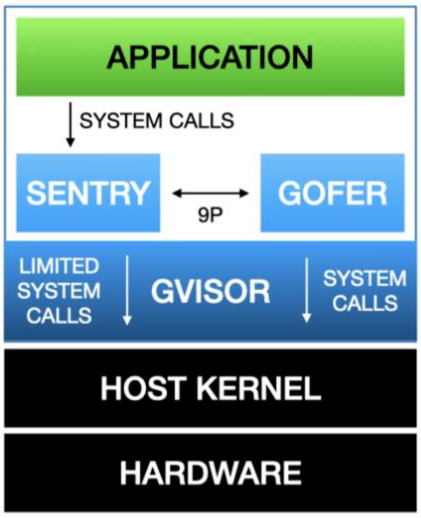

Figure 2. Architecture of gVisor.

Further in this paper, when gVisor is referred to as virtualization or container technology, it means a container that uses runsc as a runtime.

2.4. Firecracker

Firecracker is an open source Virtual Machine Monitor (VMM) developed by AWS, which uses KVM to provision lightweight VMs, called microVMs (Agache et al., 2020). According to Agache et al. (2020), Firecracker was developed for serverless cloud as a replacement of their first production implementation for AWS Lambda, which used per-customer EC2 instances to run functions in containers. The requirements that serverless computing and especially function workloads pose are quite extensive: 1) it must be safe in multi-tenant cloud, which essentially means that it must be safe to run workloads from different customers on the same hardware, 2) it must be possible to oversubscribe the resources of the host machine to maximize the ability to use resources efficiently, which means that the guest VMs must come with minimal overhead, and launching and cleaning them up must be fast, 3) it must be able to run workloads in native performance and 4) it must be able to support arbitrary Linux libraries and binaries (Agache et al., 2020). From the two available options, existing VMs and containers, neither were able to fulfil the requirements properly. Container-based solution with enhanced security, such as gVisor, were relatively fast and safe, but they were unable to support arbitrary Linux libraries and binaries. Existing VMs on the other hand were safe and able to support code in backward compatible manner, but they were either too immature or heavy for this kind of use-case (Agache et al., 2020). Thus, AWS started the development of Firecracker by re-using some of the components of Google's Chrome OS VMM (Agache et al., 2020). Firecracker has been used for several years in serverless services such as AWS Lambda and AWS Fargate, and according to Agache et al. (2020), it offers launch times even down to 125ms, low overhead as single microVM consumes only 5MB of memory, and enhanced security.

The security of Firecracker is based on simple guest model, which means that the guest machines have access only to limited device model which reduces attack surface, and "the Firecracker process is jailed using cgroups and seccomp BPF (SECure COMPuting with filters; Berkley Packet Filter), and has access to a small, tightly controlled list of system calls" (Barr, 2018). Seccomp filtering allows reducing kernel call surface of userland applications by using a filter program language, that can be used to define a set of rules for each specific system call (The kernel development community, 2020). For example, it would be possible to define a rule that causes the kernel to kill the thread or even the entire process making a forbidden system call to the kernel without executing the call.

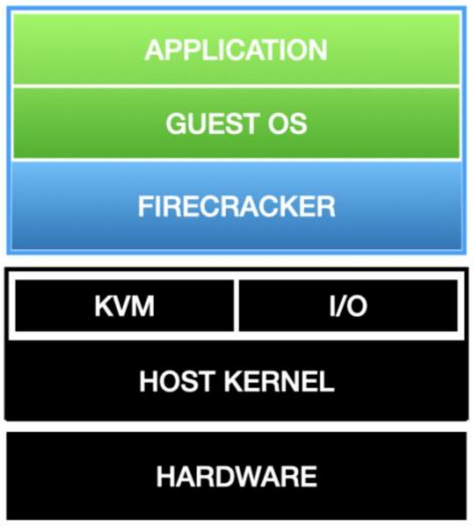

Figure 3. High-level architecture of Firecracker.

This allows AWS to run workloads from multiple customers in the same host machine with sufficient security, which is vital for multi-tenant cloud.

Firecracker VMM itself contains several components, such as a metadata service, networking and filesystem storage, rate limiting for fine grained resource management, and RESTful API that can be used to control the guest machines and, for example, rate limiting. The high-level architecture of Firecracker as a virtual machine is presented in Figure 3.

2.5. Serverless

The "serverless" as a term seems still to be a bit vague term on the industry. While there has been various suggestions for the definition of the term, generally accepted consensus is still missing. McGrath and Brenner (2017) define serverless computing as "a partial realization of an event-driven ideal, in which applications are defined by actions and the events that trigger them", and Adzic and Chatley (2017) define serverless as "a new generation of platform-as-a-service offerings where the infrastructure provider takes responsibility for receiving client requests and responding to them, capacity planning, task scheduling and operational monitoring". However, in this paper the definition is not limited to Function-as-a-Service computing, and hence a definition by Jonas et al. (2019) is used: in order to be a serverless service, it "must scale automatically with no need for explicit provisioning, and be billed based on usage". A few of the services from different cloud vendors that fall under that definition are:

- Amazon Web Services
    - AWS Lambda
    - AWS Fargate
    - AWS DynamoDB
- Google Cloud
    - Google Cloud Functions
    - Google App Engine
- Microsoft Azure
    - Azure Functions
    - Azure App Service

Serverless computing brings new challenges to the cloud. As the definition explains, the vendors should be able to monitor the resources carefully in order to be able to bill based on usage, as well as enable automatic scaling for the service. In many services, such as FaaS, the deployed code is often run in ad-hoc manner and shut down when not needed. Yet still, cloud vendors that provide serverless services are responsible for providing efficient, isolated and secure environments for their customers. According to Agache et al. (2020), to address this kind of need for dynamic and multitenant environment, the cloud vendors have to choose from two: either use traditional hypervisor-based virtualization with heavy overhead, or rely on containerized runtime environment with limited support for system calls, which might break support for many applications.

AWS came up with a solution to this problem by developing Firecracker, a new kind of virtual machine that does not contain as much overhead as traditional VM does, but still provides strong isolation and security for the host. Before Firecracker AWS was running AWS Lambda functions in Docker containers, and each container was deployed to an EC2 instance that was dedicated to the customer. These days their AWS Lambda service runs on top of EC2 bare metal instances, where one Firecracker microVM instance is allocated for each function (Agache et al., 2020). Since Google does not reveal their serverless platform architecture as openly as AWS does, the public knows only that many of their serverless services, such as Google Cloud Functions and Google App Engine uses gVisor as a runtime environment (Manor, 2018).

2.6. Virtualization in modern software development

Modern software development relies extensively on virtualization technologies. Virtualization, both lightweight and heavy, can be a part of development workflow all the way from developer's machine to the production system. For example, during development on a local machine, the developers can run external dependencies and components from pre-built VM or Docker images instead of installing it all individually (Boettiger, 2015), or they can launch multiple VMs to replicate a distributed production system locally (Duenas, Ruiz, Cuadrado, & Garcia, 2009). After the code changes have been committed, a set of automated tasks can be done to them with tools such as Jenkins (Lwakatare et al., 2019), which is also capable of running tasks in Docker containers (CD Foundation, 2020). When running services in production, it is common to rely on cloud service providers as their services are flexible, available on-demand, scalable and for many cases, affordable (Armbrust et al., 2010).

According to Kang, Le and Tao (2016), Development Operations (DevOps) "is a set of techniques for streamlining and integrating the software development process with the deployment and operations of said software". Practically it means wiring up the required pieces in the software development and operations in a way, that communication between the two is as easy as possible, and testing, deploying and maintenance of software is as automated as possible. Lwakatare et al. (2019) also point out that one of the objectives of DevOps is to enable rapid releases which makes it easier to gain feedback from users faster.

Whether to use, and what kind of virtualization to use in local development environment or in production is one of the DevOps decisions. According to Ebert, Gallardo, Hernantes and Serrano (2016), "virtualization technologies focus on security and multitenancy, whereas in application development, environmental consistency is the main goal". Moreover, they point out that usage of virtual machines can lead into compatibility issues when migrating between cloud vendors, while containers are practically cloud platform agnostic. Hence, it is important to understand the consequences of choices related to virtualization.

3. Literature review

In order to understand the current state of the art in research of performance differences between different virtualization technologies, a simplified version of systematic literature review (Kitchenham & Charters, 2007) was conducted. In the following sections the methodology and findings of the systematic literature review are reported.

3.1. Review methods

The method of the conducted literature review is based on the systematic literature review method introduced by Kitchenham and Charters (2007), with a few exceptions. The literature review helps us to understand the current state of art in research of computer virtualization performance and gives us a chance to reflect our own empirical research results against previous research.

The following subsections cover the steps, that are 1) Data sources and strategy, 2) Study selection, 3) Data extraction and 4) Data synthesis. Study quality assessment has been left out of this study due to the fact that this literature review was not conducted as a primary study.

3.1.1. Data sources and strategy

The data source used in this study was the Scopus database. The objective of the literature review was to find out how the container and hypervisor-based virtualization technologies have performed in previous studies. In order to limit the search into relevant studies, a set of inclusion criteria was developed in iterations based on research results. The initial search string was:

> *”performance” AND (”evaluation” OR ”comparison” OR “assessment”) AND (“virtual machine” OR “kernel virtual machine” OR “kvm” OR “micro virtual machine” OR “microvm” OR “firecracker”) AND (“container” OR “lxc” OR “docker” OR “gvisor”)*

The search string requires that the result contains word “performance”, as well as “evaluation” or a synonym for it, and a word referencing to VM or microVM, and container technology. With the initial search string, the results consisted of 157 papers when searching from title, abstract and keywords. After applying the inclusion criteria to the search string, the final search string was:

> *TITLE-ABS-KEY ("performance" AND ("evaluation" OR "comparison" OR "assessment") AND ("virtual machine" OR "kernel virtual machine" OR "kvm" OR "micro virtual machine" OR "microvm" OR "firecracker") AND ("container" OR "lxc" OR "docker" OR "gvisor")) AND (LIMIT-TO (PUBSTAGE , "final")) AND (LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019) OR LIMIT-TO (PUBYEAR , 2018) OR LIMIT-TO (PUBYEAR , 2017) OR LIMIT-TO (PUBYEAR , 2016) OR LIMIT-TO (PUBYEAR , 2015)) AND (LIMIT-TO (DOCTYPE , "cp") OR LIMIT-TO (DOCTYPE , "ar")) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (LANGUAGE , "English"))*

The final inclusion criteria for the search string is as follows:

- Study published between 2015 and 2020
- Subject area is either computer science or software engineering
- Document type is either conference paper or article
- Language is English
- Publication stage is final

As the technologies involved in the study are quickly evolving and the purpose of the study is not to compare the results of literature over long period of time but rather understand how the technologies perform today, papers older than five years were not considered to be relevant. The count of results from the query was 123 articles, which were exported for further selection.

3.1.2. Study selection

Within the study selection phase, another set of inclusion criteria was applied to the set of articles exported from Scopus. The study selection was conducted manually, and contained three phases:

- Inclusion based on title
- Inclusion based on abstract
- Inclusion based on availability

Within the first phase, the authors made judgements based on article titles whether or not they are relevant to this particular literature review. 67 papers were excluded based on the title, and after conducting similar process for abstracts, another 16 papers were excluded. Thus, the set of accepted articles contained 50 papers, of which 44 were fully available for the authors. Finally, there was 44 articles that went through the final steps of the review, which are data extraction and data synthesis.

3.1.3. Data extraction

The data extraction was a manual process where each paper was examined by the author and relevant information was extracted into pre-defined data form. The exported data was then used to in data synthesis, which in turn reveals how the virtualization technologies have performed in previous studies.

Due to resource constraints and the fact that the literature review conducted here is not a primary study, no cross-checking was done. Table 1 presents an example of data extraction form.

3.1.4. Data synthesis

The data synthesis was conducted manually, after the data extraction. During the synthesis, the extracted data was combined and systematically reviewed. This allows finding similarities and anomalies within the existing research. The findings were further labelled and categorized in order to be able to find relevant comparison points for reporting purposes, as well as to find suitable tooling for the empirical part of this research.

Table 1. Sample from data extraction forms.

Data item	**Value**	**Notes**
Study ID	S1	
Authors	Shirinbab, S., Lundberg, L., Casalicchio, E.	
Year	2020	
Title	Performance evaluation of containers and virtual machines when running Cassandra workload concurrently	
Data extractor	Perttu Kärnä	
Date of data extraction	12.10.2020	
Context	Databases, concurrent workload	
Domain	Telecommunication (Ericsson)	
Research method	Experimental research	Assumed, not explicitly mentioned
Findings	Docker has near native performance and outperforms VM. Security is better in VM, as Docker containers share the underlying kernel and thus a trigger on single bug in the kernel will impact all containers.	

The data synthesis revealed over 10 different candidates for performance evaluation purposes, based on which the tools used in the empirical section of this research were selected. The results of the data synthesis are documented in section 3.3, and the tools used in the empirical part of this research are presented in section 5 and Table 4.

3.2. Included and excluded studies

From the original set of 123 articles, 44 were selected to the literature review. The inclusion was done based on inclusion criteria (see section 3.1.2), and all included studies are both used in the results and presented in references.

3.3. Results

The results of the literature review were categorized into several different categories, which are presented in the following subsections.

3.3.1. File I/O

Based on the results, the existing literature contains several studies that are focusing partially or fully into databases, big data and disk I/O, all of which are now presented under the same category for the sake of simplicity. Shirinbab, Lundberg and Casalicchio (2018; 2020) studied the performance of containerized and virtualized Cassandra workloads, and concluded that containerized solutions consume less resources than fully virtualized solutions, and perform nearly as good as native solutions. Seybold, Hauser, Eisenhart, Volpert and Domaschka (2019) concluded that host filesystem outperforms container filesystems, and that running databases in containers on top of VM results in significant overhead.

In the domain of big data applications, Chung and Nah (2017), Jlassi and Martineau (2016), and Zhang et al. (2018) reported containers in general perform better under big data workloads when compared to virtual machines, albeit Chung and Nah (2017) note that Xen -based server outperforms containers for write-bound applications where block size is 128MB or 64MB. Bhimani, Yang, Leeser and Mi (2017) claims that containers suit map and calculation intensive applications, whereas they do not perform well under shuffle intensive Spark applications. Ruan, Huang, Wu and Jin (2016) claim that system containers suit I/O-bound applications better, and that additional layer of virtual machine can "result in severe disk I/O performance degradation up to 42,7%". Morabito (2015) state that Docker containers beat LXC and KVM in disk I/O, and that KVM in general performs worse than the other two in disk I/O.

3.3.2. Networking

In the area of networking, different virtualization methods have been studied under various applications. For Network Function Virtualization (NFV) usage, containers beat unikernels in memory usage, CPU usage and image size (Behravesh, Coronado, & Riggio, 2019), and VMs in memory consumption, throughput and provision time (Gedia & Perigo, 2018). Eiras, Couto and Rubinstein (2017) suggest that containers are up to two times faster than KVM for NFV, whereas Bonafiglia, Cerrato, Ciaccia, Nemirovsky and Risso (2015) state that Docker containers are unsuitable for "VNFs (Virtualized Network Functions) implemented as callbacks to be implemented in the kernel".

While Felter, Ferreira, Rajamony and Rubio (2015), Ramalho and Neto (2016), and Ruan et al. (2016) state that containers perform better in network I/O than hypervisor-based VMs, Felter et al. (2015) point out that Docker NAT introduces significant overhead that can be avoided with host networking. Ramalho and Neto (2016) made a note that both container-based and hypervisor-based virtualization perform poorly on TCP/UDP request/response benchmark. Chang et al. (2018) ran some experiments in the domain of 5G networks, and state that virtualization of Evolved Packet Core is possible especially with Docker "because of low virtualization overhead". Weerasinghe, Abel, Hagleitner and Herkersdorf (2016) studied the suitability of virtualization technologies for Field Programmable Gate Arrays (FPGAs), and while they found out that disaggregated FPGA outperforms everything else on the table, containers are more performant than VMs and not far behind bare metal.

3.3.3. High Performance Computing

Beserra et al. (2015), Beserra et al. (2017), Kovács (2017), and Sande Veiga et al. (2019) conclude all that containers are more suitable for High Performance Computing (HPC) than VMs. Chung, Nguyen, Nguyen-Huynh, Thong and Thoai (2017) confirm similar results by stating that for VM it takes 1,6 times longer than for Docker container to run their HPC simulations. Beserra, Moreno, Endo and Barreto (2017) state that containers suit I/O-bound HPC applications better than KVM, and that the overhead of VM grows when the number of instances scale up. Zhang, Lu and Panda (2016) claim that "container-based solutions can deliver better performance than the hypervisor-based solution (VM-PT and VM-SR-IOV) overall", even if their study shows that Docker-based solution has up to 9% of overhead to the native solution for HPC applications. Shirinbab, Lundberg and Casalicchio (2020) argue that "increasing the level of multitasking amplifies the overhead of containers", and Abuabdo and Al-Sharif (2019) point out that there is no point in using threading for 1-CPU virtual machines.

3.3.4. Cloud computing and auto-scaling

In terms of generic cloud computing, several studies (Chae, Lee, & Lee, 2019; Felter et al., 2015; Jaikar, Shah, Bae, & Noh, 2016; Lingayat, Badre, & Gupta, 2018; Maliszewski et al., 2018; Poojara, Ghule, Birje, & Dharwadkar, 2018; Potdar, Narayan, Kengond, & Mulla, 2020; Xie, Wang, & Wang, 2018) show that container technologies (Docker or LXC) beat hypervisor-based virtualization in resource usage and performance. Joy (2015) found out that containers in Kubernetes cluster scale up 22 times faster than virtual machines when using AWS EC2 Auto Scaling. Also Chae et al. (2019) and Khalid, Ismail and Mydin (2017) made a conclusion that containers boot up faster than VMs. Abdullah, Iqbal and Bukhari (2019) found out that "the number of requests rejected during the auto-scaling of multi-tier application deployed on containers is significantly less than using VMs". Beserra, Moreno, Endo and Barreto (2017) state that the overhead of hypervisor-based virtualization grows significantly when scaling up instances. Opposite to that, Chung and Nah (2017) argue interestingly that increasing the number of nodes in Xen-based cluster for big data processing has a positive effect, whereas for Docker it has negative effect due to increased need for resource management and scheduling in time-sharing environment.

Ruan et al. (2016) evaluate the performance of cloud services for containers (Google Container Engine, renamed to Google Kubernetes Engine later, and Amazon Elastic Container Service), and conclude that the additional layer of VM can "result in severe disk I/O performance degradation up to 42,7%, and network latency up to 233%". Naik (2016) did some performance evaluation for distributed systems and found out that container-based distributed system Docker Swarm consumed less resources than type-2 hypervisor VM -based distributed system, and has built-in load balancing and networking. Salah, Zemerly, Yeun, Al-Qutayri and Al-Hammadi (2017) tested cloud-based services and discovered that "VM-based web services outperform container-based web services with respect to all performance metrics", although they mention that the container service they use, Amazon Elastic Container Service, runs containers on top of VMs. Barik, Lenka, Rao and Ghose (2017) found out also that containers beat VMs with minimal difference, although their weakness is security, which is not covered in this paper.

3.3.5. Other results

Existing literature contains also a few studies that compare unikernels and containers. Mavridis and Karatza (2019b) found out that containers have higher request rate than unikernels in most cases, whereas they state that "for latency, MirageOs, IncludeOS and Rumprun showed significantly lower and more stable values compared to containers". Behraves et al. (2019) and Goethals, Sebrechts, Atrey, Volckaert and De Turk (2018) argue that containers beat unikernels under heavy workload that does not contain lots of context switching, since unikernels suit better applications that contain lots of context switching.

Lin, Pai and Chou (2018) compared containers and VMs when using Tensorflow for image classification and concluded that without network and filesystem I/O virtualization, even when using container on top of VM, no significant overhead was observed. However, they state that for distributed usage of Tensorflow, container on top of VM can render over 35% throughput degradation.

Morabito (2015) found out that power consumption seems to be quite similar between all virtualization technologies, except for networking, which causes hypervisor-based virtualization to consume significantly more energy than containers. Mavridis and Karatza (2019a) found out that between hypervisor-based virtualizations, “Xen hypervisor is more energy demanding compared to KVM” under stress.

4. Research method

The selected empirical research method for this study is DESMET (Kitchenham, Linkman, & Law, 1997), which provides a systematic framework for evaluating software engineering tools and methods. The chosen DESMET evaluation method for this particular study was benchmarking, which supports the research problem well as the objective was to conduct systematic benchmarks against the candidates. The research method is described in detail in the following sections.

4.1. Selecting an appropriate evaluation method

According to Kitchenham et al. (1997), conducting a research with the DESMET method starts by selecting an appropriate evaluation method. DESMET describes quantitative evaluation methods, case studies and surveys, and qualitative evaluation methods, qualitative experiment, qualitative case study and qualitative survey. In addition to those methods, they mention two hybrid evaluation methods: collated expert opinion and benchmarking. In order to find proper evaluation method for the case in research, Kitchenham et al. (1997) propose that the following items should be considered:

1. The evaluation context.
2. The nature of the expected impact of using the method/tool.
3. The nature of the object (i.e. method/tool/generic method) to be evaluated.
4. The scope of impact of the method/tool.
5. The maturity of the method/tool.
6. The learning curve associated with the method/tool.
7. The measurement capability of the organisation undertaking the evaluation.

The evaluation context in the case of this research was generic, as this work was not done for any organization, but rather for anyone interested. When it comes to the nature of the expected impact of using the tool(s), the results are directly measurable and thus it has quantitative impact. Since the objects under evaluation are tools, and for conducting the evaluation no user intervention was needed, benchmarking evaluation was the best fit. The scope of the impact of the tool affects at module level, and single process stage at a time, and thus the effects can be measured as an output of the stage. The tools under the evaluation are widely used on the industry, and since the tools do not require user intervention in the evaluation, the learning curve was not relevant. Lastly, the measurement capability of the organization undertaking the evaluation was not relevant, as the objective was not to make direct changes to software engineering processes.

As a summary, the hybrid evaluation model benchmarking was the best fit for this purpose. As the original description of DESMET by Kitchenham et al. (1997) do not provide guidelines for conducting a benchmarking evaluation, the steps of feature analysis are applied into this study, with minimal changes.

4.2. Feature analysis

According to Kictchenham et al. (1997), “a feature analysis type of evaluation is an attempt to put rationale behind a ‘gut feeling’ for the right product”. It provides a flexible, but systematic method for evaluating different kind of software engineering tools or methods. Within this research, the following steps presented by Kitchenham et al. (1997) with minimal changes for benchmarking evaluation were applied:

1. Select a set of candidate method/tools to evaluate.
2. Decide the required properties or features of the item being evaluated.
3. Prioritise those properties or features with respect to the requirements of the method/tool users.
4. Decide the level of confidence that is required in the results and therefore select the level of rigour required of the feature analysis.
5. Agree on a scoring/ranking system that can be applied to all the features.
6. Allocate the responsibilities for carrying out the actual feature evaluation.
7. Implement the benchmarking test suites for evaluation.
8. Carry out the evaluation to determine how well the products being evaluated meet the criteria that have been set.
9. Analyse and interpret the results.
10. Present the results to the appropriate decision-makers.

The only notable change to the original list of steps provided by Kitchenham et al. (1997) is step 7, "implement the benchmarking test suites for evaluation", which was a completely new step. It should be also noted, that as this research was not done for any specific organization, step 10 was achieved by constructing this report.

Table 2. The selected features, and their importance and measurement for feature evaluation.

Feature	Description	Importance	Measurement
Disk – File I/O	Filesystem read and write operations.	High	MiB / sec, higher is better
Networking – Jitter	Variance in network latency, i.e. how much the round-trip time changes during the network transmission.	High	Milliseconds, lower is better
Networking - Bandwidth	How much data can be transmitted (sent and received) over the network per second.	High	Mbps, higher is better
CPU Events	Calculate prime numbers as fast as possible.	High	Events / sec, higher is better
Boot up time	Booting up the system to a state where it's responsive.	High	Seconds, lower is better
Database	Relational database operations, both read and write.	Low	Transactions / sec, higher is better
Pipeline turnover time	Continuous integration pipeline runs with case application setup.	Low	Seconds, lower is better
HTTP response time	HTTP request benchmarking with simultaneous HTTP calls against the system.	Low	Mean response time (ms / req), lower is better

In this particular study, the selected tools, or virtualization technologies, are selected based on their usage on the industry. Thus, the selected technologies are Docker, gVisor (with Docker), Firecracker and KVM. As a baseline result, the evaluations are run against a bare metal server, which is described in detail in section 5.1.1.

The required features for the benchmark of the virtualization technologies are decided based on the example usage case, which is described in section 5. In order to find out typical low-level resource constraints of these technologies, networking, disk I/O and CPU performance were selected as the first set of so-called low-level features. More specifically, the measured features contain networking jitter and bandwidth, disk file I/O read and write, CPU prime number calculation performance, and system boot up time. Their importance in this study is high because they are less complex than real applications, and thus are likely to be less prone to resource contention or other side-effects in the system. In addition, several high-level features were selected to help the evaluation of performance of these technologies closer to real world usage scenarios. Those features are database performance and a case application, which contains HTTP server benchmarking and a continuous integration pipeline job benchmarking. The high-level features are described in detail in section 5.1.6 and 5.1.7, and their importance is considered to be low due to the high complexity and increased likelihood of uncontrollable resource contamination in the system. All tests are repeated 10 times in order to mitigate the possibility of circumstantial bias. See Table 2 for the final set of features, and their respective prioritization. Since this report is a master's thesis work, all steps are carried out by the author.

4.3. Benchmarking

In benchmarking, the selected features are evaluated by running a set of custom benchmarks. Since Kitchenham et al. (1997) do not describe how the benchmarking should be carried out, the benchmarking in this study contains three steps: 1) preparing the tests and gathering all required resources, 2) running the tests, and 3) collecting the results. The benchmarking is described in detail in section 5.

4.4. Benchmarking analysis

Within the analysis, the results are analysed in order to find out which candidate fits the requirements the best. Typically, in DESMET the results are collected into a score sheet called an evaluation profile, which can be used to determine the final score of the candidates (Kitchenham et al., 1997). The score sheet contains the priority of a feature, the minimal acceptance threshold scores and the scores the candidate received, which allows the comparison of results in a coherent manner.

Table 3. Sample of an evaluation profile score sheet for gVisor.

Feature	**Priority**	**Score**
File I/O read	High	2
File I/O write	High	0
Network jitter	High	4
Network bandwidth received	High	4
Network bandwidth sent	High	4

However, this research does not have minimal acceptance thresholds as the goal was to do comparative benchmarking of the candidates, and that is why the scoring sheet is missing the minimal acceptance threshold column in this research. Table 3 presents a sample of an evaluation profile scoring sheet. The benchmarking test results are reported in section 6 and the benchmarking analysis is reported in section 7.

5. Benchmarking

Benchmarking consists of preparing and running a set of automated tests and collecting the results. The tests are always designed for the particular case in study, and thus no general-purpose tests are available. The following subsections describe the preparations and environmental details regarding the benchmarking, as well as how the individual tests were run and the results collected after the benchmarking.

5.1. Preparing the tests

The test preparation was the first thing that was done in feature evaluation phase. Being arguably one of the most critical parts of the research, test preparation was a heavy process including constant decision making and trial and error. The scripts used for the benchmarking are stored in a publicly available repository[1]. This section covers the details of the benchmarking environment setup and each test, what it does, what it measures, and the motivation behind using it.

5.1.1. Environment setup

The environment used for running the benchmarking was built on top of commodity hardware available due to heavy resource constraints. The setup consists of 2010-era desktop computer with Intel® Core™ 2 Quad Q6600 2.40GHz processor, 4GiB of memory and 500GB SSD disk, running Ubuntu Server 20.04.1 with Linux Kernel version 5.4, and a MacBook Pro 2017 with Intel® Core™ i7 2.9GHz processor and 16GB of

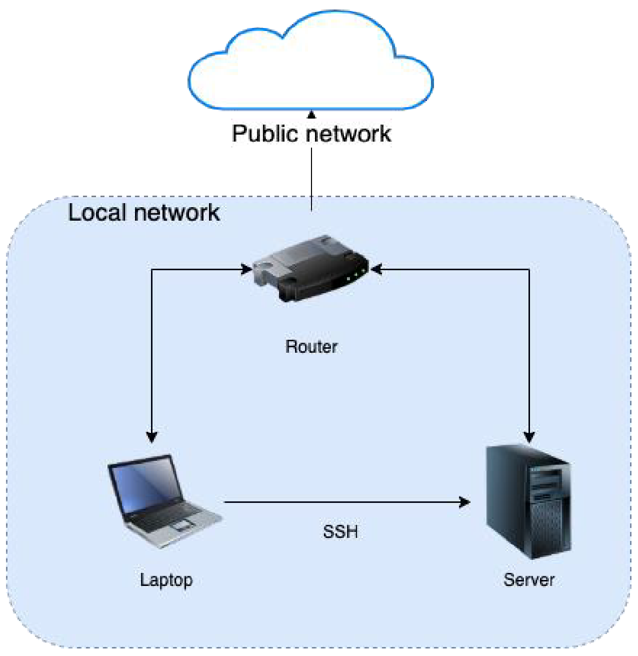

Figure 4. Environment setup for benchmarking.

[1] https://github.com/ppeerttu/virtualization-benchmarks

memory. The Ubuntu Server was used as a host machine for virtualization platforms in all tests, while the MacBook was acting as a client in some of the tests, such as database and HTTP benchmarking tests. In between these computers was a cable connected Asus router with 1 Gb transfer rate. In order to mitigate possible resource contention, the host machine was not running anything else besides what was required for conducting the tests. Figure 4 describes the high-level setup of the testing environment from networking perspective.

Table 4. Versions of tools, applications and operating systems used in the benchmarking.

Tool / Application / Operating System	Version
Ubuntu Server	20.04.1
Docker	20.10.2, build 2291f61
gVisor / runsc	release-20201208.0
Firecracker	0.23.1
MySQL	8.0.22
sysbench	1.0.20
iperf3	3.9
ab	2.3
GitLab Server Enterprise Edition	13.2.0
GitLab Runner	13.2.4
macOS	11.1 Big Sur

All containers that were used were based on official Ubuntu 20.04 image. KVM guest was installed similarly from Ubuntu 20.04 Server installer, but the operating system for Firecracker guest machine was built with Debian bootstrap tool as it was easier to come up with suitable setup that way. The baseline operating system was still Ubuntu 20.04, and Linux Kernel for Firecracker guest was built manually from version 5.4 source files, with similar configuration as in the host machine. Table 4 describes detailed versions of each tool, application and operating system used in the benchmarking.

5.1.2. Boot time

The purpose of the boot time test was to find out how quickly and reliably the machine can launch up from an existing image. The motivation for this test comes from the definition of serverless computing by Jonas et al. (2019): the serverless services need to be able to scale automatically and “be billed based on usage”. Both of those requirements advocate the idea of runtimes that are capable of launching and shutting down rapidly in order to match the load with minimal cost.

The boot time test is heavily debatable from two viewpoints: 1) internal and external factors, such as both host and guest operating system, pre-installed software, and system configuration, can affect heavily the boot time of a machine, and 2) it’s not easy to determine at which point the machine is considered to be fully launched. For example, Agache et al. (2020) consider the boot time of a Firecracker microVM to be the time between “VMM process is forked and the guest kernel forks its init process”. As this kind of detailed low-level measurement would require custom implementation in underlying virtualization platforms, the boot time in this study was measured with existing tools without any modifications in any of the virtualization platforms. As such, the results are not highly accurate and should be treated only as approximate guidelines.

For container technologies, the container was considered to be fully launched once “docker run” -command has returned to the foreground, and the time is measured with the Unix “time” -command. In the case of other platforms, including both virtual machines and bare metal, the machine was considered to be up once its network stack was up and able to respond to the Unix “ping” -command.

5.1.3. CPU events

The purpose of the CPU test was to measure the performance of the platform under generic computing-intensive tasks. In order to simplify the test setup, a generic benchmarking tool called sysbench was used as it supports testing properties other than just CPU performance. However, more extensive tools such as High Performance Linpack have been used at various times in previous studies (Beserra et al., 2015; Beserra et al., 2017; Mavridis & Karatza, 2019; Ramalho & Neto, 2016), whilst sysbench has been used for CPU benchmarking several times (Morabito, 2015; Potdar et al., 2020; Ramalho & Neto, 2016).

The test counts how many events the system is able to process during the period of test. Within an event, the system calculates and stores in a list all prime numbers between one and 10 000, and the duration for one run was 10 seconds. The test was run on each platform directly by using the sysbench command.

5.1.4. File I/O

The file I/O test is measuring the system under generic file read and write workload. Within the previous literature, tools such as Bonnie++ (Morabito et al., 2015; Ramalho & Neto, 2016; Xie et al., 2018) and IOzone (Barik et al., 2017; Mavridis & Karatza, 2019; Potdar et al., 2020) have been used, but since sysbench was already selected for benchmarking several other features, it was chosen for file I/O benchmarking as well.

The file I/O benchmarking conducted in this research contains two sysbench tests: seqrd, which means sequential read, and seqwr, which means sequential write. In both tests the total size of files within the test was five gigabytes, and the test time 30 seconds. File operation mode was synchronous, which is default, and number of files was also the default, 128. Within the test, the files are either written or read in sequential order, and the speed of reading or writing was measured and reported as Mebibytes per second. The tests were conducted in similar manner as CPU tests, which was command line invocation directly on the virtualization platform.

5.1.5. Network bandwidth and jitter

The network bandwidth and jitter tests are measuring the network performance of the virtualization platforms. The bandwidth means how much data can be transmitted within specified period of time over the network, while the jitter means the variance of latency in the network.

The selected tool for this task was iperf version 3 (often called iperf3), because it has been used in previous studies (Barik et al., 2017; Morabito, 2015), as well as it is capable of measuring both bandwidth and jitter. More specifically, it supports both TCP and UDP protocol, both of which are relevant in this case.

In order to be able to measure both bandwidth and jitter, the same benchmarking suite was run twice, but TCP was used for measuring the bandwidth, and UDP was used for measuring the jitter. Within the test, the iperf3 server was launched on the virtualization platform, and then the iperf3 client on another machine connected to the server and started the actual test. The results are visible for both the client and the server, but in this case they were collected from both.

5.1.6. Database

In order to approach the performance from more practical perspective, database benchmarking was conducted. Databases are a common part of web applications, and they heavily utilize memory, CPU, filesystem and network. Similarly as in several previous studies (Felter et al., 2015; Mavridis & Karatza, 2019; Ruan et al., 2016), sysbench was used to run a set of OLTP-like transactions against MySQL relational database.

Setting up the test contained two steps: first preparing the database by launching it and seeding it with the initial data, and then running the benchmarking test suite. The initial seed data contained eight identical tables containing two integer columns and two char columns, and they were all filled with a million rows of data. The test suite contained three different kind of tests: read only, write only, and read write. Read only test consists of transactions that contain five different SELECT queries. Write only test consists of transactions that contain two UPDATE statements, one DELETE statement and one INSERT statement. Read write test suite consists of transactions that contain both read only and write only statements. For each virtualization platform, the database was launched on the virtualized platform first, and then the sysbench client performing the queries was launched on another machine. Each individual test duration was 20 seconds, and they were run on concurrency levels of 10, 20, 30, 40, 50, 60, and 70. In order to mitigate potential bias, each concurrency level was repeated 10 times.

Due to obscure network issues when reaching out to Ubuntu repositories from the Firecracker guest, it was not possible to install MySQL to the Firecracker guest machine this time. This problem was most likely caused by the networking setup, but as the time for benchmarking was limited, it was not possible to spend more time on investigating that issue. As a result, database benchmarking was not conducted against Firecracker.

5.1.7. Case application

In order to simulate potential real-world usage of these virtualization technologies, a case application was selected for benchmarking. After brief investigation, a suitable use-case was found: a source code submission environment for programming courses at the University of Oulu. In this case, GitLab (GitLab, 2021) was selected as the case application which serves as source code repository host, as well as pipeline runner for automated programming assignment evaluation. This setup allows students to test their code against pre-defined testbed in automated pipelines and return their assignment by tagging their final commit, for example.

Based on this use-case, two different kind of tests were seen suitable for benchmarking evaluation. First of them was traditional HTTP request benchmarking, as it is common to see increased HTTP traffic in the server near assignment deadlines. For this purpose, a HTTP benchmarking tool called ApacheBench (also known as ab), that has seen use in previous studies (Barik et al., 2017; Behravesh et al., 2019; Poojara et al., 2018; Potdar et al., 2020), was used to send simple HTTP GET requests to the server with concurrency levels of 10, 30, 60, 90, 120, 150, 180, and 200, whereby average response times were collected. This test was repeated 10 times for each concurrency level in order to mitigate potential bias.

The second potential target identified for testing was pipeline turnover time. Pipeline turnover time means the time it takes for an automated pipeline to run series of tasks for the source code after a student has submitted their assignment. In this particular test, the assignment contained a simple matrix calculator written in C++, and the pipeline tasks contained code compilation, running unit tests, collecting code coverage and a static code analysis. The pipeline runner was a single instance of GitLab runner, which was running on the same guest machine with the GitLab server, except for Docker platform, in which case the server and runner were running in different containers. The pipeline was manually re-launched 10 times in order to mitigate potential bias.

Unfortunately, due to the way GitLab server was installed on Linux containers, it was not possible to get GitLab working with gVisor runtime. As a result, gVisor was not benchmarked in case application tests.

5.2. Running the tests

The tests were run in sequential order, so none of the tests were run concurrently. Primarily, the following order of platforms were used: bare metal, KVM, Docker and gVisor, and Firecracker. Because Docker CLI tool was used to run containers with runc and runsc runtimes, each test was run for both runtimes before moving to the next one. Exceptions were file I/O test and boot time test, both of which had to be rerun for all platforms afterwards due to bug in the testing script.

The tests were run for each platform in the following order: boot time, file I/O, CPU, network bandwidth (TCP), network jitter (UDP), database, case application HTTP and case application pipeline. An exception for this was Firecracker, for which some of the tests required a bit more preparations than initially anticipated, and thus they were run in the following order: CPU, network bandwidth (TCP), network jitter (UDP), case application HTTP, case application pipeline and file I/O.

During benchmarking, it was noticed that the GitLab server was unable to be installed on containers using gVisor runsc runtime. After investigating the issue further, it was concluded that the issue was related to a step where the GitLab installation script attempts to change directory permissions in certain locations of the filesystem, which was assumably prevented by the runsc runtime. Due to resource constraints, it was decided to drop gVisor from case application benchmarking altogether due to this issue. Similarly, when installing the tooling to Firecracker guest VM in order to conduct the benchmarking, it was noticed that the guest machine was not able to reach out to Ubuntu repositories. Even after several hours of further investigating this issue, it was never understood what caused this to happen. As a result, the tools that had to be installed from Ubuntu repositories were pre-installed during disk image creation instead of when running the actual guest VM. However, this solution proved to be inadequate for installing MySQL, which meant that database benchmarking could not be done for Firecracker at this time.

5.3. Collecting the results

The results were collected from logfiles generated during the test run. In most of the tests, the relevant log output was directly piped to a known directory that was then submitted to version control along with the test scripts in order to guarantee persistence and transparency. However, there were two tests that required also manual work for collecting results: 1) running the tests with iperf3 server required fetching the server log output from the remote machine, and 2) running the pipeline test with GitLab required manually collecting results from the GitLab web user interface.

After collecting the results, the results were pre-processed before further analysis. Some of the data was collected in JSON format, but most of it was generic, non-structured, free text output. During the pre-processing, the results were parsed and formatted into a set of CSV files, which were then used for the analysis.

6. Results

Within this section, the results of the benchmarking tests are presented by each feature, and the further feature evaluation analysis is presented in section 7. All the results for each individual test including mean values and standard deviations are included as Appendix A. The results of tests where the level of concurrency was variable are aggregated into mean value across all concurrency levels.

6.1. Boot time

The mean boot time and standard deviation for each platform is presented in Figure 5. The benefits of container technologies are clearly visible here; both Docker and gVisor reach to sub-second boot times with average result of 0,69 and 0,83 seconds respectively, whereas KVM and Firecracker boot times are in the order of seconds rather than milliseconds. Nevertheless, they both still perform still better than the bare metal, as the metal takes 58,46 seconds to boot up on average, while KVM reached to 19,34 seconds and Firecracker to 5,13 seconds. An interesting result here is the standard deviation of

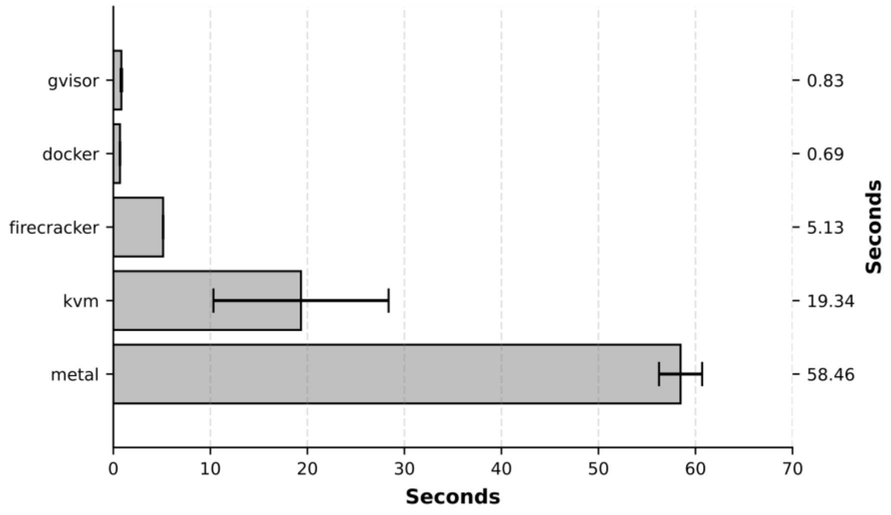

Figure 5. Mean time to boot up a machine (lower is better) with standard deviation.

values, because both KVM and metal have a lot of variation in their bootup time, whereas Firecracker, gVisor and Docker are much more stable. Based on the results of the boot time benchmark, the answer to the RQ2 is: the fastest virtualization technology in terms of boot up time was Docker.

6.2. CPU events

The mean count of events per second for all platforms is presented in Figure 6. The results show that none of the virtualization platforms can reach up to the level of bare metal. Firecracker and gVisor are both 19% slower than the metal while Docker and KVM are 16% and 9% slower, respectively. It is also interesting finding that the standard deviation of all virtualization platforms is roughly at least five times as high as the one of bare

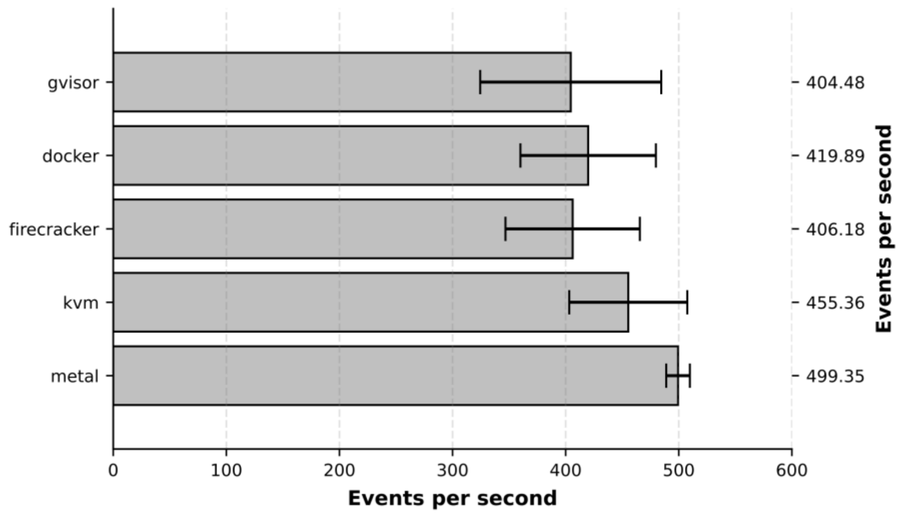

Figure 6. Mean count of CPU events per second (higher is better) with standard deviation.

metal. Based on these results, the answer to the RQ1.1 is: KVM is the most performant virtualization technology in terms of CPU events.

6.3. File I/O

The file read and write performance is presented in Figure 7 and Figure 8. Based on the results of file reading, Firecracker performs the worst of all platforms. Its result is dramatic 66% worse than the bare metal, which reaches up to 318,82 MiB/s. Contrary to that, Firecracker is the most performant virtualization platform in file writing, being only 12% percent less performant than the bare metal.

KVM is 47% slower than the bare metal in file reading and 30% slower in file writing. Interestingly KVM is faster than Firecracker in file reading, but slower in file writing, which seems counterintuitive when considering the seccomp filtering Firecracker uses.

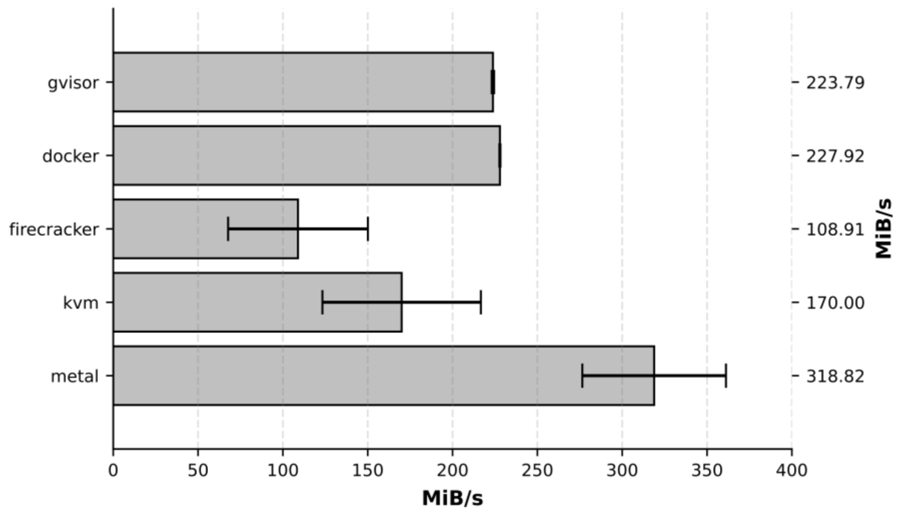

Figure 7. Mean file read performance (higher is better) with standard deviation.

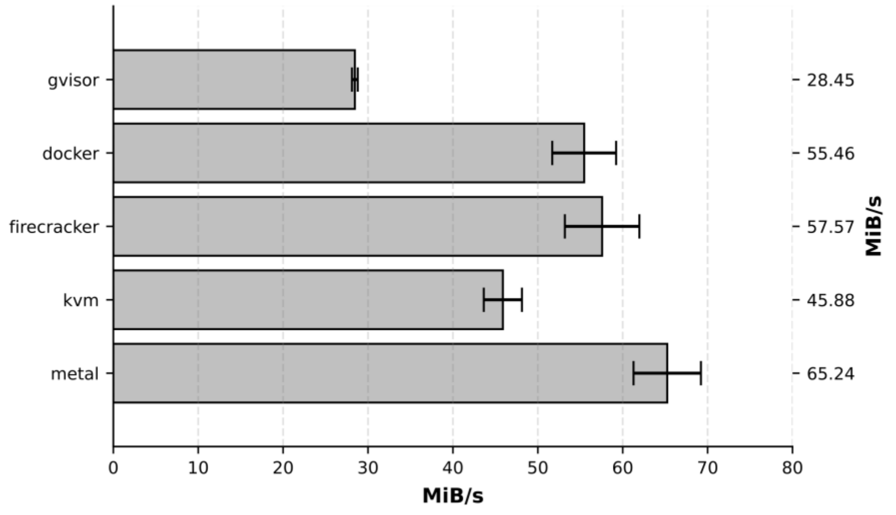

Figure 8. Mean file write performance (higher is better) with standard deviation.

Docker performs adequately in both filesystem operations, losing 29% in file reading and 15% in file writing to the bare metal host. While the cost of enhanced security in gVisor is insignificant in file reading, it becomes clearly visible in the file writing, which is 56% less performant than in bare metal. Based on the results, Firecracker is the fastest in file writing but the slowest in file reading, whereas gVisor is almost the fastest in file reading but the slowest in file writing. Docker seems to be the most stable platform, leaving KVM behind in both tests. Overall, these results reveal that as an answer for the RQ1.2, Docker is the most performant virtualization technology in file reading, whereas Firecracker is the most performant in file writing.

6.4. Network

The network TCP bandwidth is presented in Figure 9, and UDP jitter is presented in Figure 10. When looking at the bandwidth, gVisor performs the worst by losing 7% to the bare metal in both sending and receiving data. Network bandwidth performance

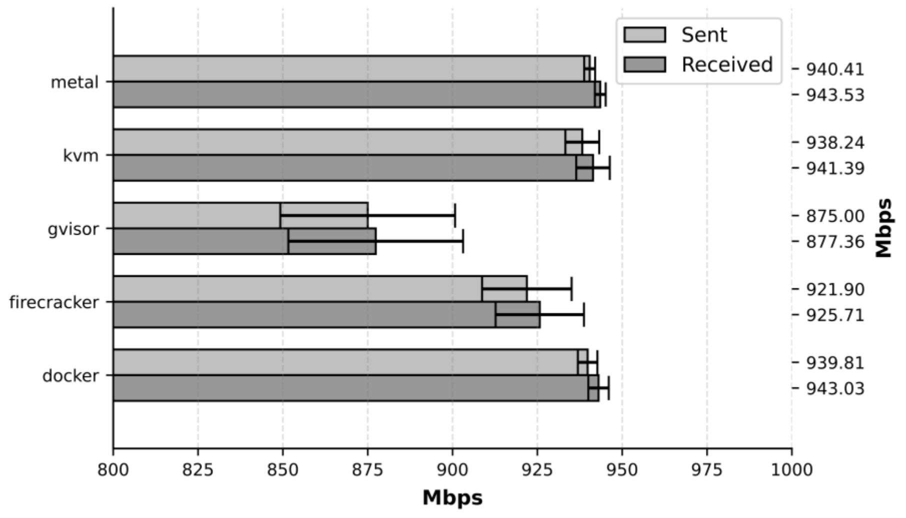

Figure 9. Network bandwidth (higher is better) using TCP traffic.

overhead of the other virtualization platforms is almost negligible, as Firecracker performs only 2% worse than the bare metal, and both Docker and KVM lose to bare metal by less than 1%.

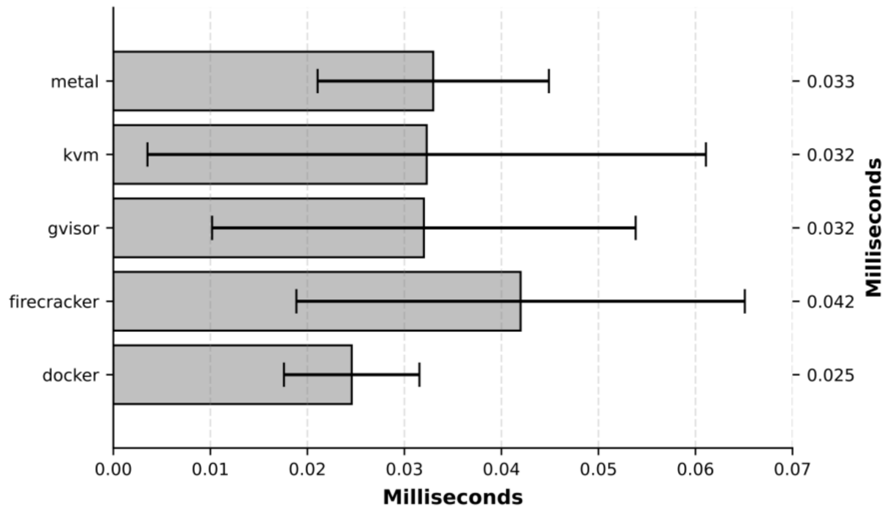

Figure 10. Network jitter (higher is better) using UDP traffic.

When looking at the UDP jitter results, it becomes evident that Docker, gVisor and KVM all have smaller jitter than bare metal, which is a surprising result. The best result was achieved by Docker with its 0,025ms jitter, which is 26% smaller than bare metal's 0,033ms jitter. The worst jitter was on Firecracker, which achieved 0,042ms jitter on average, which is 27% bigger than bare metal's jitter. Overall, these results tell that as an answer to the RQ1.3, Docker is the most performant virtualization technology in terms of network I/O.

6.5. Database

The database benchmarking contained three different kind of tests. The first of them was a test containing read-only transactions, the second one contained write-only transactions, and the third one contained both read-only and write-only operations within a transaction.

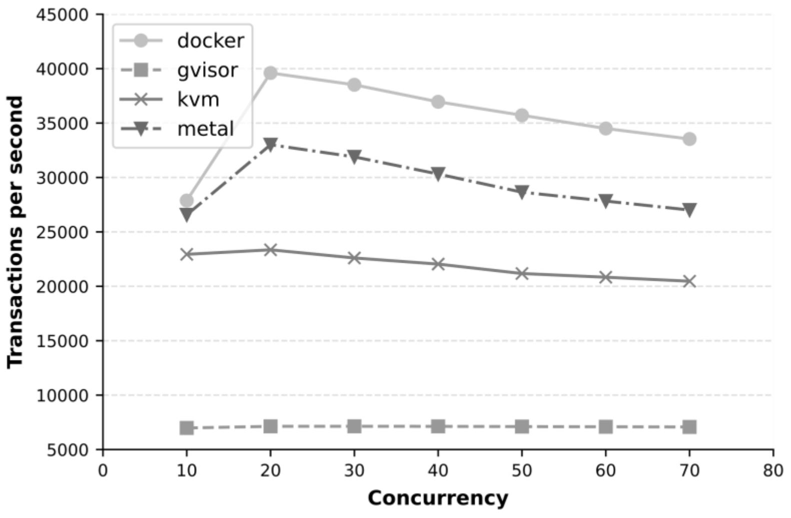

Figure 11. Count of database read transactions per second (higher is better) vs. concurrency.

As stated earlier in this report, database benchmarking was not possible to run on Firecracker due to technical issues.

The clear winner of the read-only test is Docker, reaching 39 602 transactions per second at concurrency level of 20, beating the bare metal by 20%, while KVM loses roughly 29% to the bare metal at the same concurrency level. Based on the read-only results, the differences in transactions per second between Docker, bare metal and KVM are biggest at the concurrency level of 20. When the concurrency is 10, Docker and bare metal have almost even results: 27878 and 26568 transactions per second, respectively. The worst performer was gVisor, which was able to process around 7000 transactions per second on every concurrency level, which means over 70% less transactions than with the bare metal. Figure 11 describes read-only transaction test results.

The order of platforms does not change when looking at the write-only test results. Again, Docker beats the bare metal at every concurrency level, but by smaller margin than in the read-only test. At most, Docker beats bare metal by 13% at the concurrency level of 40, where it is able to reach 7157 transactions per second. Similarly, KVM is performing around 22% worse than the bare metal at concurrency level of 20, although the relative difference between the two is dropping when moving towards bigger concurrency levels. The worst performer is again gVisor, but contrary to what was seen with read-only tests, gVisor is able to increase the number of transactions per second when the concurrency grows. It is able to reach up to 3646 transactions per second at the concurrency level of 70, which is around 55% less than with the bare metal. Figure 12 describes write-only transaction test results.

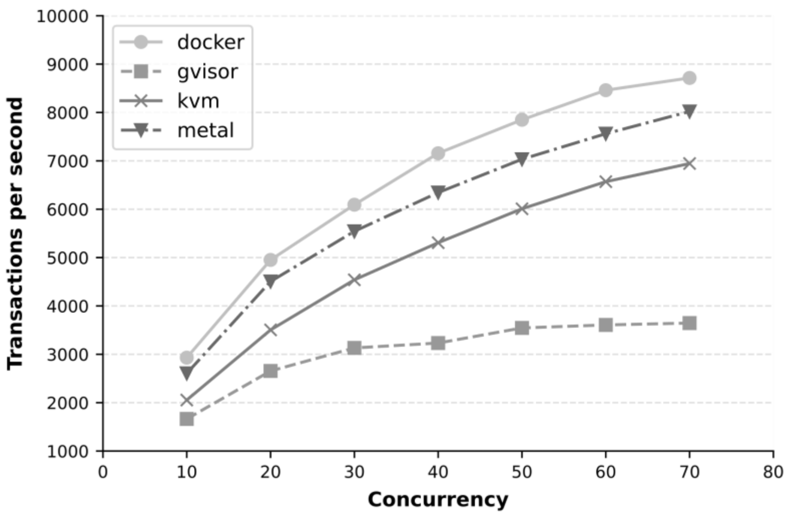

Figure 12. Count of database write transactions per second (higher is better) vs. concurrency.

When running the read-write test, the results of bare metal, Docker and KVM are closer to each other. Docker and KVM perform actually better than the bare metal on every concurrency level, which is a surprising result. At concurrency levels of 10, 30, 40 and 50, KVM is the most performant, while Docker is slightly better at concurrency levels of 20, 60 and 70. At concurrency level of 20, KVM beats bare metal by 22% while reaching up to 2568 transactions per second, whereas Docker gains the lead with higher concurrency levels, being 16% faster than bare metal at concurrency level of 70. Similarly as with the previous database tests, gVisor is the worst performer, having a difference of 29% to bare metal at concurrency level of 20, which grows up to 76% when the concurrency level increases to 70. Figure 13 describes read-write transaction test results.

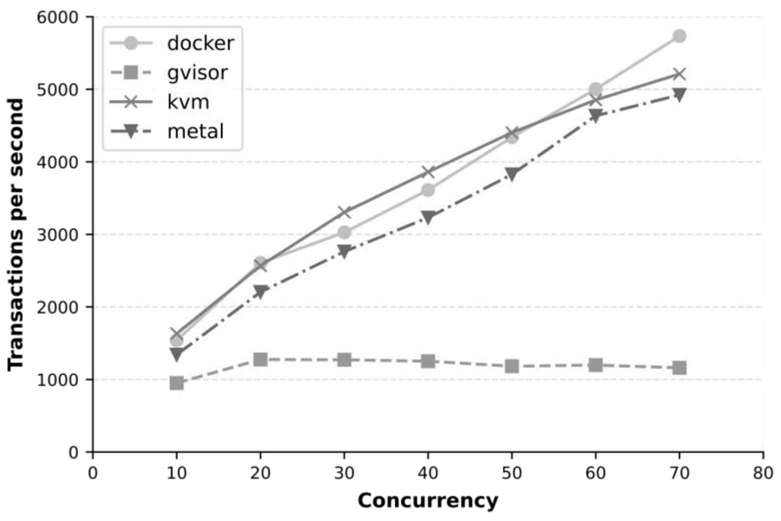

Figure 13. Count of database read-write transactions per second (higher is better) vs. concurrency.

An interesting observation regarding the database benchmarking results is that the read-only performance of every platform begins to drop when the concurrency level grows over 20, while the write-only performance keeps growing throughout all concurrency levels tested, up to 70. It should be also noted that the count of transactions is much higher in read-only tests than it is in write-only tests; Docker is able to perform 39602 transactions at concurrency level of 20 in read-only test, while the best write-only result, 8714 transactions per seconds, was achieved similarly by Docker, but at concurrency level of 70. This is most likely happening due to the in-memory cache that the database uses heavily for read-only queries. As the database does not need to wait for disk I/O as often with read-only queries that is required with write-only queries, it is able to process queries faster, and thus reach much higher transaction rates even at lower concurrency levels. In this particular case, the benefit of increased concurrency begins to drop as the scheduling becomes the bottleneck of the performance after the concurrency level increases over 20.

6.6. HTTP Benchmark

The results for HTTP benchmarking are described in Figure 14. As the HTTP benchmarking test was conducted against a GitLab server instance, it was not possible to run that on top Docker with runsc runtime. Hence, no results for gVisor were measured.

Based on the results, Firecracker and KVM form their own group, and Docker and bare metal form their own group. Contrary to what was seen with database benchmarking results, none of the virtualization platforms perform better than the bare metal in HTTP benchmarking. At the concurrency level of 10, the bare metal is able to respond to a HTTP request in 136 milliseconds on average, whereas for Docker, KVM and Firecracker it takes 145, 212 and 227 milliseconds, respectively. The mean response time grows linearly for each platform, so the relative difference between the results has no big changes over different concurrency levels. For Docker, the relative difference to bare metal ranges from 4% to 9%, for KVM it ranges from 51% to 57%, and for Firecracker it ranges from 58% to 67%.

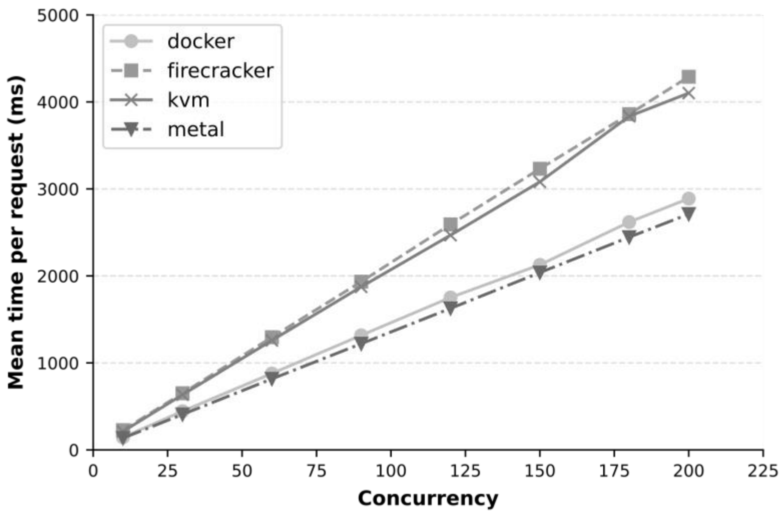

Figure 14. Mean response time for HTTP requests (lower is better) vs. concurrency.

6.7. Pipeline

The pipeline results are presented in Figure 15. Based on the results, the bare metal was the most stable and performant in running the pipeline, taking around 291 seconds on

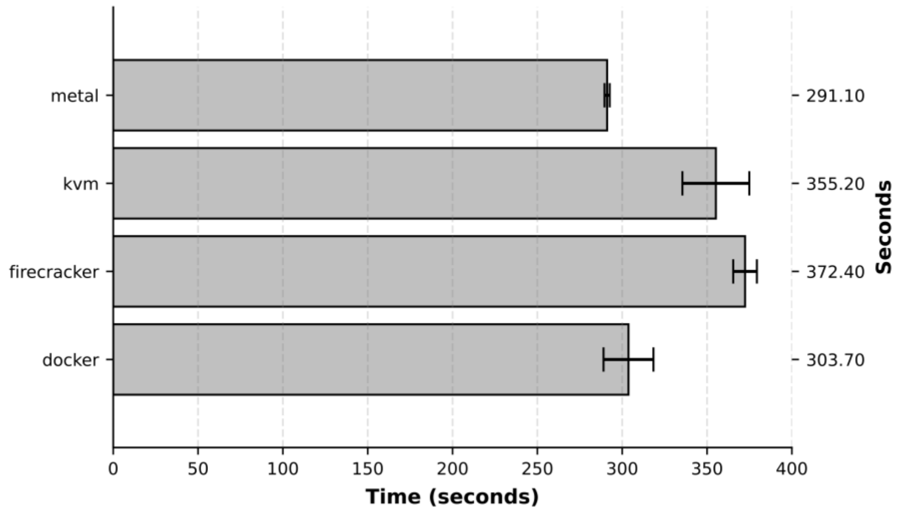

Figure 15. Mean time (lower is better) and standard deviation for running the GitLab pipeline job.

average to complete. Docker was just 4% slower than the bare metal, taking around 304 seconds to complete on average. KVM was 22% slower than the bare metal, and Firecracker was the worst performing platform, taking around 372 seconds to complete on average, which translates to 28% slower than the bare metal. On the other hand, Firecracker has clearly smaller standard deviation (6,98 seconds) when compared to Docker (14,71 seconds) and KVM (19,71 seconds), which indicates that it is more stable in this test.

7. Analysis

The first purpose of the analysis section is to report the benchmarking analysis, according to the description of the DESMET research method by Kitchenham et al. (1997). The benchmarking analysis will make use of scoring sheets, where each candidate will be assigned a score per benchmarking test, according to defined grading rules. This allows determining which candidate best suits the given use-case, albeit it should be noted that this is not directly a generalizable result.

The scoring scheme in benchmarking analysis is done by using relative metrics, where the bare metal platform is used as a baseline. The total score (higher is better) is calculated from individual scores that are granted based on the results of each platform for each feature. Table 6 displays the required result for each feature and score. As no previous DESMET benchmarking evaluations were found in existing literature, the grading schema is not based on any previous references. Most of the features require at least 60% of the result of the baseline in order to grant any points in grading, but sometimes the grading rules are reversed. For example, in order to get score one (1) from boot time feature, a platform mean boot up time must be less than or equal to 40% of the baseline boot up time. This is because the boot time with bare metal is not expected to be good, and almost every virtualization platform should be able to boot up faster. This means that for some features, higher result means higher score, while for others, lower result means

Table 6. Grading scheme for the benchmarking evaluation analysis. Requirements for scores are relative to the bare metal result.

Feature	**Baseline**	**Score 1**	**Score 2**	**Score 3**	**Score 4**
Boot time	58,46 s	≤ 40%	≤ 30%	≤ 20%	≤ 10%
CPU events	499,35 e/s	≥ 60%	≥ 70%	≥ 80%	≥ 90%
File read	318,82 MiB/s	≥ 60%	≥ 70%	≥ 80%	≥ 90%
File write	65,24 MiB/s	≥ 60%	≥ 70%	≥ 80%	≥ 90%
Network sent	940,407 Mbps	≥ 60%	≥ 70%	≥ 80%	≥ 90%
Network received	943,532 Mbps	≥ 60%	≥ 70%	≥ 80%	≥ 90%
Network jitter	0,033 ms	≤ 140%	≤ 130%	≤ 120%	≤ 110%
Database read	29325 t/s	≥ 60%	≥ 70%	≥ 80%	≥ 90%
Database write	5945 t/s	≥ 60%	≥ 70%	≥ 80%	≥ 90%
Database r/w	3274 t/s	≥ 60%	≥ 70%	≥ 80%	≥ 90%
Pipeline turn.	291,10 s	≤ 140%	≤ 130%	≤ 120%	≤ 110%
HTTP res.	1425,18 ms	≤ 140%	≤ 130%	≤ 120%	≤ 110%

higher score. As the grading schema is highly subjective and not based on previous work, it should not be treated as anything else than a tool that allows comparison of the virtualization platforms.

The following subsections will cover the details of scoring each feature, as well as present the final scores of each platform.

7.1. Boot time

The boot time is relevant in modern systems where the infrastructure needs to be able to scale, sometimes in matter of minutes or seconds, to be able to cope with increased load. This is a relevant feature for serverless computing, because the underlying infrastructure must be provided as it is needed, "without explicit provisioning" (Jonas et al., 2019).

Table 7. Benchmarking analysis scores for low-level features.

Feature	KVM	Firecracker	gVisor	Docker
Boot time	1	4	4	4
CPU events	4	3	3	3
File read	0	0	2	2
File write	2	3	0	3
Network sent	4	4	4	4
Network received	4	4	4	4
Network jitter	4	3	4	4
Sum of low-level features	19	21	21	24

The scoring for each platforms' low-level features is presented in Table 7. Based on the scoring used in this paper, KVM is the only platform that does not receive full score for boot up time. Deeper look into the results reveals that both Docker and gVisor achieved sub-second boot up times, while for Firecracker it takes around five seconds to boot up. The interesting observation in these results is that Firecracker is still several times faster than KVM, even if it uses partially same underlying APIs as KVM does. This is indeed one of the very reasons Firecracker was developed in the first place – speed and resource efficiency. Nevertheless, in this benchmarking evaluation, Firecracker is still far behind the boot up times of containers. The results might have been more in favour of Firecracker if the setup for the benchmarking would have been selected differently. Smaller, preferably Alpine Linux image would most likely be faster to boot up, and measuring the boot up time by writing to disk once the user space has been loaded might prove different results than relying on the network interfaces and devices. It is also likely that the modest commodity hardware used in this research does not give similar boot up times as compared to modern cloud servers.

7.2. CPU events

Interestingly, the only virtualization platform that receives full score for CPU speed is KVM, which was 9% slower than the bare metal on average. This was a surprising result as the results of the literature review revealed that several previous studies have found containers faster in HPC, for example, than hypervisor-based virtualization. All the other platforms received the second-best score and were between 16 and 19% slower than the bare metal. Nevertheless, none of the platforms was dramatically slower than the others, which is a reasonable result as many of the decisions made regarding the architecture of these virtualization platforms are related to system calls and I/O. Moreover, all the virtualization platforms had more variation in their results than the bare metal, which is an interesting result.

It should be noted, that the used hardware, especially CPU, was relatively old when compared to what is available today. Hence, the results of the CPU benchmarking may not reflect to what is experienced in reality with modern CPUs today.

7.3. File I/O

The file I/O benchmarking gave interesting results. The best performer was Docker, which was able to reach five out of eight available points for file I/O scoring. Both Firecracker and KVM were unable to get any points from file I/O read benchmark, whereas gVisor was unable to get any points from file I/O write benchmark.

The results reveal that gVisor's runsc runtime comes with clear overhead when compared to the default runc runtime, especially when doing file writing. This is most likely due to the implementation costs that come with the Virtual File System (VFS) that the runsc uses (Google LLC, 2021b). The runsc runtime got two points from the file reading, and as stated before, zero points from the file writing.

KVM and Firecracker on the other hand perform poorly on filesystem reading, receiving zero points from it. Especially the result of Firecracker's read performance was surprising, as it is 66% slower than bare metal, and even 36% slower than KVM. While the reason for this is not clear, it is likely that the security model Firecracker uses has an impact on this. Considering the results of file reading, it is equally surprising to see that Firecracker performs extremely well on file writing when compared to other virtualization platforms. Firecracker beats even Docker's default runc runtime by a margin of 4%. Firecracker and Docker received both three points from file writing, whereas KVM received two.

7.4. Network

The network results reveal that both KVM and Docker are able to reach close to the performance of bare metal. In fact, all virtualization platforms receive full score from the network bandwidth features, both sent and received. gVisor seems to be the only platform that stands out from the rest with its 7% smaller bandwidth than the bare metals. Nevertheless, the differences are still relatively small. The overhead of virtualization is more visible in the variation of the results, as their standard deviation is clearly bigger than with bare metal.

Interestingly all virtualization platforms except Firecracker beat the bare metal in UDP jitter. Again, the differences between the platforms are small, and the variation in the jitter is certainly larger with the virtualized platforms than it is with the bare metal, except when using Docker, which has smaller standard deviation than the bare metal. Firecracker received three points from the network jitter benchmark, while the others received the full four points.

7.5. Database

The database benchmarking is one of the tests where the platforms were benchmarked using a real application. The scores of high-level features are presented in Table 8. The importance of the database in benchmarking is low because of two reasons: 1) the resource contention becomes more difficult to control when the complexity of the application grows (compare for example to file I/O benchmark), and 2) databases are often available as managed services, which often makes it impractical to deploy them similar to the application code.

Table 8. Benchmarking analysis scores for high-level features.

Feature	KVM	Firecracker	gVisor	Docker
Database read	1	-	0	4
Database write	2	-	0	4
Database r/w	4	-	0	4
Pipeline turnover	2	2	-	4
HTTP res.	0	0	-	4
Sum of high-level features	9	2	0	20

Nevertheless, the results of database benchmarking reveal interesting details about the virtualization platforms. First of all, the performance of gVisor was surprisingly poor, and thus it received no points from the database benchmarking. On average across all concurrency levels in read-only benchmarking, it was around three times slower than KVM, which was the second slowest. Docker on the other hand received a full score from all sub-features of the database benchmarking. KVM received one point from the database read feature, two points from the database write feature, and a full four points from the database read-write feature. KVM was in fact the fastest database in read-write benchmarking on most concurrency levels, which was surprising as it was not very close to the bare metal in either of read-only or write-only benchmarks. As stated before, unfortunately Firecracker was unable to be benchmarked due to technical problems.

7.6. HTTP Benchmark

The HTTP benchmark was another test where the platforms were benchmarked using a real application, and it represents a part of the case-application. In this particular case, the HTTP server consisted of a GitLab server instance, and the HTTP client used for the benchmarking was ApacheBench. The HTTP benchmark is treated as a low priority benchmark here as it is harder to control the resource contention than with the low-level benchmarks.

The results reveal that the only platform comparable to the bare metal is Docker, which received a full score, and was 7% slower than the bare metal on average across all concurrency levels. KVM and Firecracker were both left far behind, both being over 50% slower than the bare metal at all concurrency levels, thus receiving no points at all from the HTTP benchmarking. As explained earlier, the GitLab server was unable to be installed to gVisor, which is why it was not part of the HTTP benchmark.

7.7. Pipeline

The pipeline benchmark was another part of the case-application, where a real application usage is demonstrated and used as a benchmark for the virtualization platforms. The pipeline benchmark uses GitLab runner as an application to run the pipeline tasks. Similarly as with the HTTP benchmark, the pipeline benchmark is considered to be low priority as the resource contention is difficult to manage due to the high complexity of the application.

The pipeline turnover time shows that all of the virtualization platforms have some overhead to the bare metal. Docker was the fastest virtualization platform, being only 4% slower than the bare metal on average, and thus receiving full score. Both KVM and Firecracker received two points from the pipeline benchmark, as they were 22% and 28% slower than the bare metal. Between all the platforms, the bare metal had clearly the smallest standard deviation, 1,52 seconds. Between the virtualization platforms, Firecracker had the smallest standard deviation of 6,98 seconds, leaving Docker and KVM far behind with their results of 14,71 and 19,71 seconds, respectively. Based on these results, the overhead of hypervisor-based virtualization is clearly visible, but interestingly Firecracker is still the most stable performer.

8. Discussion

Within the following subsections, suggestions regarding choosing a virtualization platform are made, and further implications of this research are presented.

8.1. Choosing between virtualization technologies

When looking at the total score, KVM introduced the most overhead to performance from low-level feature point of view. It received two points less than gVisor and Firecracker, and the only thing where it lost in points, for those two technologies, was the boot up time. If the boot time scores are ignored, KVM received a total of 18 from low-level benchmarking scores, whereas Firecracker and gVisor got both 17. This tells that the overhead of type-1 hypervisor is not far from sandboxed containers, and in some cases the KVM performed even better than non-sandboxed Docker container, such as in CPU benchmark.

Based on these results, the KVM is most suitable for long-living deployments, for example, where rapid auto-scaling is not critical. Spinning up a new instance of KVM, for example, in automated continuous integration pipelines might not be wise, unless they can be re-used without the overhead of re-creating a new instance every time. Moreover, it was not investigated in this research how much resources the KVM instances consume while sitting idle, which is relevant in cases where one of the objectives of isolating applications into containers or VMs is cost efficiency by subscribing the host machine resources as much as possible. For example, in cases where the guest instance density is prioritized over host kernel-level security, it might be better to consider sandboxed or regular containers such as gVisor or Docker instead of KVM.

Firecracker was a stable performer in all of the low-level feature benchmarks, except for disk file reading, in which it was clearly the worst performer. The reason for such a result may be caused by the current implementation of block device emulation in Firecracker, as it does not use asynchronous model, but instead waits for each I/O call to complete before issuing the next one (Firecracker Contributors, 2020). Contrary to the disk file reading, Firecracker was the most performant virtualization platform in disk file writing. In addition to that, Firecracker was almost four times faster in launching a guest machine than the KVM. This result was expected as one of the design principles of Firecracker was a lightweight device model that both leaves minimal attack surface and introduces minimal performance overhead.

Due to the short boot up time, Firecracker suits short-term deployments almost as well as container technologies. Because this research did not contain any multi-instance benchmarks, it is not possible to estimate how well it supports, for example, dense deployments where the resources of the host system are over-subscribed in order to reach maximal resource utilization. The APIs that Firecracker offers for guest machine management makes it easy to automate the management of guest VMs, but it does have some limitations as well. For example, both Docker and KVM comes with pre-configured network devices and configuring either bridged or NAT networking for guests is relatively easily, whereas for this research the network configuration for Firecracker guest machine had to be self-configured by using IP tables, which is error prone task and requires quite extensive knowledge in networking. Another limitation for the utilization of Firecracker is missing support for hardware-accelerated GPUs, which is still a work in progress (Firecracker Contributors, 2019a; Firecracker Contributors, 2019b).

Sandboxed container runtime gVisor provided varying results. It was both capable of launching an instance and performing disk file reading almost as quickly as the default Docker runtime runc, but in many tests it was the worst performer. It was the worst in disk file writing, network bandwidth and CPU events, which means that the overhead of additional security is a concern in terms of performance. Moreover, the fact that gVisor was unable to get any points from the database benchmarking was unexpected result when considering that it received the same total score from low-level benchmarking than Firecracker did. On the other hand, because the database benchmarking was not conducted for Firecracker, it is not possible compare the two in that sense. One reason for the performance issues of gVisor, especially regarding the database benchmarking, might be caused by the default ptrace platform it uses. According to the documentation of gVisor (Google LLC, 2021c), "the ptrace platform has high context switch overhead, so system call-heavy applications may pay a performance penalty". This can be mitigated by switching to the KVM platform that gVisor also supports, but testing beyond default configurations was out of scope in this research.

The optimal use-cases for gVisor are deployments which do not contain system call-heavy applications and can leverage the OCI implementation of gVisor. Such environments can be, for example, Kubernetes clusters, which are running applications that control business logic without need for heavy I/O operations, such as media streaming or data storing on local file system. Another good use-case for gVisor could be, for example, arbitrary workloads that run untrusted code, such as in continuous integration pipelines. Using the runsc runtime instead of Docker's default runc allows using the same container orchestration tools with extended security where it is needed. However, gVisor comes with limitations as well. It is not compatible with any arbitrary existing Docker image; for example, in this research, the installation of GitLab Server failed most likely due to failing system calls. In addition to that, similarly as Firecracker, gVisor is lacking support for hardware-accelerated GPUs (gVisor Contributors, 2018).

The default Docker runtime, runc, received expected results in this research. It was the most performant virtualization platform in both low-level feature benchmarks, and application benchmarks, and thus it can be concluded that the results of this study are aligned with the findings in the literature review. The only test where Docker was clearly worse than some other platform was the CPU event benchmark, where KVM was able to reach up to 455,36 events per second when Docker got only 419,89 events per second. The literature review did not confirm this particular case, so the reason for this result remains unknown. On the other hand, this study does not confirm the result provided by Felter et al. (2015), where they state that Docker NAT networking introduces significant overhead. These results show that Docker had network throughput almost equal to what bare metal had, and the network latency was in fact better than what the bare metal had. The different result received in this research may be due to differences in the hardware as well as it is possible that Docker's NAT networking has been improved.

Due to the low performance overhead, non-sandboxed container technology Docker is an appealing alternatives for any kind of application that does not need to be isolated on kernel level from the rest of the host machine. For example, Docker is suitable for running trusted code in systems that do not have extremely high reliability requirements, such as hospital or banking systems do. Depending on the environmental constraints of the application, it may be possible to increase the level of security with infrastructural choices, such as by using a network proxy in front of the containers. Another use-case for both KVM and Docker is utilization of GPUs. This means that Docker is a good candidate, for example, for training machine learning models.

8.2. Implications

The results received from the empirical research reported in this paper gives high-level guidelines for choosing proper virtualization technologies for industry practitioners from the performance point of view. The area of concern in this research was focused to performance features, whereas other features, such as usability, reliability and security, should be considered when making decisions. Information related to those features should be sourced from other research papers and industry publications, and there might be room for new research in some of those areas as well.

The following subsubsections covers the implications of this study for modern software development, as well as for research.

8.2.1. Modern software development

As described in subsection 2.5, serverless computing is a recent emerging trend for cloud-native applications. The possibility to run applications on top of a fully managed, auto-scaling infrastructure that is billed based on usage, is clearly an appealing package for the industry practitioners. However, the objective of this study is not to help in choosing between different serverless services or platform providers, but to help in understanding the differences between the virtualization technologies used underneath those serverless services, as well as choosing between different virtualization technologies.

As the results of this study merely address the performance differences between the virtualization technologies, the results do not help in choosing the right tool for local development purposes, at least in most cases. As described in subsection 2.6, the objective of using virtualization in software development is usually to achieve environmental consistency, which is not related to performance. Any virtualization method investigated in this research could fulfil the need for environmental consistency, so one should focus on more practical issues instead. For example, considering the usability, tooling and support for the virtualization technology is more relevant in that case. However, those things are out of scope of this research.

The results of this research are applicable to situations where the infrastructure management is not provided by a service provider, for example, when using a on-premises cloud. The domains where this scenario might be applicable are, for example, health care and banking, both of which are highly critical elements in the modern society, and thus sensitive for any kind of technical problems. In these circumstances, the application-specific requirements determine the critical features required from the underlying infrastructure, but it is likely that the choice is being made by comparing the requirements for performance and scaling to the requirement for security. The boot up time difference between hypervisor-based virtualization and containers raises an interesting paradox here when considering availability requirements. The containers are able to scale up faster than VMs, and thus they may be able to satisfy peak-time load better. On the other hand, VMs are better in isolating critical operational failures, such as kernel-level bugs, meaning that an issue in one guest VM does not affect to other guest VMs running on the same host, which in turn may improve availability.

Similar to running applications in on-premises clouds, these results can be applied when running applications in hybrid clouds. For example, container orchestration tool Kubernetes helps in running container-based workloads reliably without forcing to a vendor lock-in. Kubernetes introduced a Container Runtime Interface (CRI) in 2016 (Y.-J. Hong, 2016), and thus is able to use various kinds of container runtimes that implement the OCI standard. This means, that Kubernetes users can choose, for example, between the default Docker runc runtime, and the gVisor runsc runtime. Moreover, it is possible to switch run Kubernetes pods within both KVM and Firecracker provisioned guest VMs, thanks to the OCI-compliant Kata Containers (E. Ernst & G. Whaley, 2019; Kata Containers, 2020). However, this research did not consider the performance when using nested virtualization, and hence the results of this study may not be directly applicable to such deployments.

8.2.2. Research

As these technologies are in constant development, and especially as young technologies see improvements in rapid pace, it is possible that these results will become obsolete within the following five years. This means that in order to have up-to-date information regarding performance differences between different virtualization technologies in the future, this kind of benchmarking evaluation would need to be done regularly, for example, every other year. Furthermore, it can be seen from the results of the literature review, that the existing literature contains a lot of studies in the area of evaluating the performance of different virtualization technologies, but it was difficult to find similarities amongst the papers regarding the methods used for benchmarking. In order to provide a systematic and comparable method for conducting this kind of benchmarking, it might be good to develop a systematic framework for benchmarking. That would give the researchers a clear set of tools for performing the benchmarking, and also rules for experimenting or extending the benchmarking into new areas.

Lastly, the empirical research conducted in this paper revealed a documentation gap in the DESMET benchmarking research method. While the other DESMET evaluation methods, such as case studies and surveys, are extensively covered by Kitchenham et al. (1997), the hybrid evaluation methods, namely benchmarking and collated expert opinion, are not. Kitchenham et al. (1997) state, that there can be even more hybrid benchmarking evaluation methods than the previously mentioned ones, but they were not able to think about any others. This may be also the reason why the instructions do not cover hybrid evaluation methods; the amount of possible hybrid methods is so high, that it makes it difficult to give guidelines for all of them.

9. Conclusions

The serverless computing in cloud has posed new problems for cloud vendors that traditional means of virtual infrastructure provisioning do not solve well. Requirements such as resource isolation, instance density, security and scalability are difficult to fulfil in multi-tenant cloud while the underlying platform should still maintain performance comparable to native. Within this research, a literature review was conducted to see what other authors have reported regarding the performance of containers and VMs recently, and then a DESMET benchmarking evaluation was performed against four different virtualization technologies, namely Firecracker, gVisor, KVM and Docker. The research questions were:

1. Which virtualization technology is the most performant in terms of
    1. CPU events?
    2. Filesystem I/O?
    3. Networking bandwidth and jitter?
2. Which virtualization technology provides the fastest launch time?

The results are aligned with findings of the literature review, and they show that Docker is still the most performant virtualization technology amongst these selected technologies in terms of CPU events (RQ1.1), file reading (RQ1.2), networking I/O (RQ1.3), as well as the fastest in boot up time (RQ2). Firecracker was the fastest virtualization technology in file writing (RQ1.2), and generally speaking both Firecracker and gVisor were able to reach better total score than KVM when considering the before mentioned low-level features.

It should be noted that this research has some limitations, and those limitations should be taken into account in order to be able to look at the results objectively. The research method description for DESMET provided by Kitchenham et al. (1997) does not describe detailed steps of how the hybrid evaluation method benchmarking should be conducted. Along the way, there had to be some choices made without any kind of supporting information, for example, when mapping the default DESMET feature evaluation into benchmarking evaluation, or when coming up with the grading schema for the virtualization platforms. Hence, this research should be treated as a starting point for further DESMET benchmarking evaluations instead of an absolute reference implementation.

Secondly, as this research was conducted as a master’s thesis work, and was not financially supported, the resources were highly limited. More specifically, the hardware used for the benchmarking evaluation was not as modern as the author would have hoped, and the substance knowledge required for operating with the virtualization technologies was higher than expected. Especially the effort required for running the benchmarks against Firecracker was surprisingly high due to lack of expertise in operating systems and network configuration, and the immaturity of the documentation Firecracker has. Moreover, the inability to run all the application benchmarks against all virtualization technologies was most likely caused by the lack of expertise in these technologies.

In the future, it would be good to run similar benchmarks with more modern hardware and focus, for example, to the qualities of multi-tenant cloud, such as resource limiting and stability when running multiple guests on the host. Another interesting topic for further research is the security of Firecracker and gVisor, and the effects in performance when their security configuration profiles are adjusted. Furthermore, when these technologies mature and potentially other alternatives emerge, their usability could be studied to find out which of the tools would be easiest to adapt in modern cloud setups.

References

Abdullah, M., Iqbal, W., & Bukhari, F. (2019). *Containers vs virtual machines for auto-scaling multi-tier applications under dynamically increasing workloads* doi:10.1007/978-981-13-6052-7_14

Abuabdo, A., & Al-Sharif, Z. A. (2019). Virtualization vs. containerization: Towards a multithreaded performance evaluation approach. Paper presented at the *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA, , 2019-November.* doi:10.1109/AICCSA47632.2019.9035233

Adzic, G., & Chatley, R. (2017). Serverless computing: Economic and architectural impact. Paper presented at the *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,* pp. 884-889.

Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., et al. (2020). Firecracker: Lightweight virtualization for serverless applications. Paper presented at the *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20),* pp. 419-434.

Amazon Web Services. (2021a). *Amazon DynamoDB.* Retrieved Jan 26, 2021, from https://aws.amazon.com/dynamodb/

Amazon Web Services. (2021b). *Amazon EC2.* Retrieved Jan 30, 2021, from https://aws.amazon.com/ec2

Amazon Web Services. (2021c). *AWS fargate.* Retrieved Jan 26, 2021, from https://aws.amazon.com/fargate/

Amazon Web Services. (2021d). *AWS lambda.* Retrieved Jan 26, 2021, from https://aws.amazon.com/lambda/

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., et al. (2010). A view of cloud computing. *Communications of the ACM, 53*(4), 50-58.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., et al. (2017). Serverless computing: Current trends and open problems. *Research advances in cloud computing* (pp. 1-20) Springer.

Barik, R. K., Lenka, R. K., Rao, K. R., & Ghose, D. (2017). Performance analysis of virtual machines and containers in cloud computing. Paper presented at the *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2016,* pp. 1204-1210. doi:10.1109/CCAA.2016.7813925

Barr, J. (2018). *Firecracker - lightweight virtualization for serverless computing.* Retrieved October 19, 2020, from https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/

Behravesh, R., Coronado, E., & Riggio, R. (2019). Performance evaluation on virtualization technologies for NFV deployment in 5G networks. Paper presented at the *Proceedings of the 2019 IEEE Conference on Network Softwarization: Unleashing the Power of Network Softwarization, NetSoft 2019,* pp. 24-29. doi:10.1109/NETSOFT.2019.8806664

Bellasi, P., Massari, G., & Fornaciari, W. (2015). Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Transactions on Embedded Computing Systems (TECS), 14*(2), 1-17.

Beserra, D., Moreno, E. D., Endo, P. T., & Barreto, J. (2017). Performance evaluation of a lightweight virtualization solution for HPC I/O scenarios. Paper presented at the *2016 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2016 - Conference Proceedings,* pp. 4681-4686. doi:10.1109/SMC.2016.7844970

Beserra, D., Moreno, E. D., Endo, P. T., Barreto, J., Fernandes, S., & Sadok, D. (2017). Performance analysis of linux containers for high performance computing applications. *International Journal of Grid and Utility Computing, 8*(4), 321-329. doi:10.1504/IJGUC.2017.088266

Beserra, D., Moreno, E. D., Endo, P. T., Barreto, J., Sadok, D., & Fernandes, S. (2015). Performance analysis of LXC for HPC environments. Paper presented at the *Proceedings - 2015 9th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015,* pp. 358-363. doi:10.1109/CISIS.2015.53

Bhimani, J., Yang, Z., Leeser, M., & Mi, N. (2017). Accelerating big data applications using lightweight virtualization framework on enterprise cloud. Paper presented at the *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017,* doi:10.1109/HPEC.2017.8091086

Binstock, A. (2015, May 20th,). Java's 20 years of innovation . Retrieved from https://www.forbes.com/sites/oracle/2015/05/20/javas-20-years-of-innovation/

Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review, 49*(1), 71-79.

Bonafiglia, R., Cerrato, I., Ciaccia, F., Nemirovsky, M., & Risso, F. (2015). Assessing the performance of virtualization technologies for NFV: A preliminary benchmarking. Paper presented at the *Proceedings - European Workshop on Software Defined Networks, EWSDN,* pp. 67-72. doi:10.1109/EWSDN.2015.63

Caraza-Harter, T., & Swift, M. M. (2020). Blending containers and virtual machines: A study of firecracker and gVisor. Paper presented at the *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments,* pp. 101-113.

CD Foundation. (2020). *Using docker with pipeline.* Retrieved October 30th, 2020, from https://www.jenkins.io/doc/book/pipeline/docker/

Chae, M. S., Lee, H. M., & Lee, K. (2019). A performance comparison of linux containers and virtual machines using docker and KVM. *Cluster Computing, 22*, 1765-1775. doi:10.1007/s10586-017-1511-2

Chang, H. -., Qiu, B. -., Chiu, C. -., Chen, J. -., Lin, F. J., De La Bastida, D., et al. (2018). Performance evaluation of Open5GCore over KVM and docker by using Open5GMTC. Paper presented at the *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018,* pp. 1-6. doi:10.1109/NOMS.2018.8406141

Chung, H., & Nah, Y. (2017). *Performance comparison of distributed processing of large volume of data on top of xen and docker-based virtual clusters* doi:10.1007/978-3-319-55753-3_7

Chung, M. T., Nguyen, M. -., Nguyen-Huynh, N. -., Thong, N., & Thoai, N. (2017). Performance evaluation of environmental applications using TELEMAC-MASCARET on virtual platforms. Paper presented at the *Proceedings of the 2016*

IEEE 12th International Conference on E-Science, E-Science 2016, pp. 441-448. doi:10.1109/eScience/2016.7870934

Combe, T., Martin, A., & Di Pietro, R. (2016). To docker or not to docker: A security perspective. *IEEE Cloud Computing, 3*(5), 54-62.

Denning, P. J. (2001). Anecdotes [virtual machines]. *IEEE Annals of the History of Computing, 23*(3), 73.

Digital Ocean. (2021). *Droplets - scalable virtual machines.* Retrieved Jan 30, 2021, from https://www.digitalocean.com/products/droplets/

Docker Inc. (2021). *What is a container?* Retrieved Jan 30, 2021, from https://www.docker.com/resources/what-container

Duenas, J. C., Ruiz, J. L., Cuadrado, F., & Garcia, B. (2009). System virtualization tools for software development. *IEEE Internet Computing, 13*(5), 52-59.

Dykstra, T., et al. (2020). *Common language runtime (CLR).* Retrieved Jan 30, 2021, from https://docs.microsoft.com/en-us/dotnet/standard/clr

E. Ernst, & G. Whaley. (2019). *Initial release of kata containers with firecracker support* . Retrieved Apr 19, 2021, from https://github.com/kata-containers/documentation/wiki/Initial-release-of-Kata-Containers-with-Firecracker-support

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *IEEE Software, 33*(3), 94-100.

Eiras, R. S. V., Couto, R. S., & Rubinstein, M. G. (2017). Performance evaluation of a virtualized HTTP proxy in KVM and docker. Paper presented at the *2016 7th*

International Conference on the Network of the Future, NOF 2016, doi:10.1109/NOF.2016.7810144

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. Paper presented at the *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software,* pp. 171-172. doi:10.1109/ISPASS.2015.7095802

Firecracker Contributors. (2019a). *[Devices] offer support for hardware-accelerated inference in firecracker.* Retrieved Apr 17, 2021, from https://github.com/firecracker-microvm/firecracker/issues/1179

Firecracker Contributors. (2019b). *GPU support.* Retrieved Apr 17, 2021, from https://github.com/firecracker-microvm/firecracker/issues/849

Firecracker Contributors. (2020). *[Perf] implement async IO for the block device using io_uring.* Retrieved Apr 17, 2021, from https://github.com/firecracker-microvm/firecracker/issues/1600

Gedia, D., & Perigo, L. (2018). Performance evaluation of SDN-VNF in virtual machine and container. Paper presented at the *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2018,* doi:10.1109/NFV-SDN.2018.8725805

GitLab. (2021). *The complete DevOps platform.* Retrieved Feb 23, 2021, from https://about.gitlab.com

Goethals, T., Sebrechts, M., Atrey, A., Volckaert, B., & De Turck, F. (2018). Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications. Paper presented at the *Proceedings - 8th IEEE*

International Symposium on Cloud and Services Computing, SC2 2018, pp. 1-8. doi:10.1109/SC2.2018.00008

Google. (2016). *Cloud functions.* Retrieved Jan 26, 2021, from https://cloud.google.com/functions

Google LLC. (2021). *What is gVisor?* Retrieved Jan 30, 2021, from https://gvisor.dev/docs/

Google LLC. (2021a). *gVisor.* Retrieved Jan 16, 2021, from https://gvisor.dev

Google LLC. (2021b). *Performance guide.* Retrieved Mar 19, 2021, from https://gvisor.dev/docs/architecture_guide/performance/#file-system

Google LLC. (2021c). *Platform guide.* Retrieved Apr 12, 2021, from https://gvisor.dev/docs/architecture_guide/platforms/

gVisor Contributors. (2018). *Support for nvidia-docker GPU container sandboxing.* Retrieved Apr 17, 2021, from https://github.com/google/gvisor/issues/14

Jaikar, A., Shah, S. A. R., Bae, S., & Noh, S. -. (2016). *Performance evaluation of scientific workflow on OpenStack and OpenVZ* doi:10.1007/978-3-319-38904-2_13

Jlassi, A., & Martineau, P. (2016). Virtualization technologies for the big data environment. Paper presented at the *Proceedings of the ACM Symposium on Applied Computing, , 04-08-April-2016.* pp. 542-545. doi:10.1145/2851613.2851881

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., et al. (2019). Cloud programming simplified: A berkeley view on serverless computing. *arXiv Preprint arXiv:1902.03383,* Joy, A. M. (2015). Performance comparison between linux containers and virtual machines. Paper presented at the *Conference Proceeding - 2015 International Conference on Advances in Computer Engineering and Applications, ICACEA 2015,* pp. 342-346. doi:10.1109/ICACEA.2015.7164727

Kang, H., Le, M., & Tao, S. (2016). Container and microservice driven design for cloud infrastructure devops. Paper presented at the *2016 IEEE International Conference on Cloud Engineering (IC2E),* pp. 202-211.

Kata Containers. (2020). *Kata containers architecture* . Retrieved Apr 19, 2021, from https://github.com/kata-containers/documentation/blob/master/design/architecture.md

Khalid, M. F., Ismail, B. I., & Mydin, M. N. M. (2017). Performance comparison of image and workload management of edge computing using different virtualization technologies. *Advanced Science Letters, 23*(6), 5064-5068. doi:10.1166/asl.2017.7310

Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering.

Kitchenham, B., Linkman, S., & Law, D. (1997). DESMET: A methodology for evaluating software engineering methods and tools. *Computing & Control Engineering Journal, 8*(3), 120-126.

Kovács, Á. (2017). Comparison of different linux containers. Paper presented at the *2017 40th International Conference on Telecommunications and Signal Processing, TSP 2017, , 2017-January.* pp. 47-51. doi:10.1109/TSP.2017.8075934

KVM contributors. (2016). *Main page.* Retrieved Jan 30, 2021, from https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792

Lin, C. -., Pai, H. -., & Chou, J. (2018). Comparison between bare-metal, container and vm using tensorflow image classification benchmarks for deep learning cloud platform. Paper presented at the *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science, , 2018-January.* pp. 376-383. doi:10.5220/0006680603760383

Lingayat, A., Badre, R. R., & Gupta, A. K. (2018). Performance evaluation for deploying docker containers on baremetal and virtual machine. Paper presented at the *Proceedings of the 3rd International Conference on Communication and Electronics Systems, ICCES 2018,* pp. 1019-1023. doi:10.1109/CESYS.2018.8723998

Lwakatare, L. E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., et al. (2019). DevOps in practice: A multiple case study of five companies. *Information and Software Technology, 114*, 217-230.

LXC contributors. (2021). *What's LXC?* Retrieved Jan 30, 2021, from https://linuxcontainers.org/lxc/introduction/

Maliszewski, A. M., Griebler, D., Schepke, C., Ditter, A., Fey, D., & Fernandes, L. G. (2018). The NAS benchmark kernels for single and multi-tenant cloud instances with LXC/KVM. Paper presented at the *Proceedings - 2018 International Conference on High Performance Computing and Simulation, HPCS 2018,* pp. 359-366. doi:10.1109/HPCS.2018.00066

Manor, E. (2018). *Bringing the best of serverless to you.* Retrieved October 18th, 2020, from https://cloudplatform.googleblog.com/2018/07/bringing-the-best-of-serverless-to-you.html

Mavridis, I., & Karatza, H. (2019a). Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems, 94*, 674-696. doi:10.1016/j.future.2018.12.035

Mavridis, I., & Karatza, H. (2019b). Lightweight virtualization approaches for software-defined systems and cloud computing: An evaluation of unikernels and containers. Paper presented at the *2019 6th International Conference on Software Defined Systems, SDS 2019,* pp. 171-178. doi:10.1109/SDS.2019.8768586

McGrath, G., & Brenner, P. R. (2017). Serverless computing: Design, implementation, and performance. Paper presented at the *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW),* pp. 405-410.

Microsoft. (2019). *Azure functions.* Retrieved Jan 26, 2021, from https://azure.microsoft.com/en-us/services/functions/

Morabito, R. (2015). Power consumption of virtualization technologies: An empirical investigation. Paper presented at the *Proceedings - 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015,* pp. 522-527. doi:10.1109/UCC.2015.93

Morabito, R., Kjällman, J., & Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. Paper presented at the *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015,* pp. 386-393. doi:10.1109/IC2E.2015.74

Naik, N. (2016). Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems. Paper presented at the *Proceedings - 2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments, MESOCA 2016,* pp. 1-8. doi:10.1109/MESOCA.2016.9

Oracle. (2021). *Oracle VM virtualbox.* Retrieved Jan 30, 2021, from https://www.virtualbox.org/

Poojara, S. R., Ghule, V. B., Birje, M. N., & Dharwadkar, N. V. (2018). Performance analysis of linux container and hypervisor for application deployment on clouds. Paper presented at the *Proceedings of the International Conference on Computational Techniques, Electronics and Mechanical Systems, CTEMS 2018,* pp. 24-29. doi:10.1109/CTEMS.2018.8769146

Potdar, A. M., Narayan, D. G., Kengond, S., & Mulla, M. M. (2020). Performance evaluation of docker container and virtual machine. Paper presented at the *Procedia Computer Science, , 171.* pp. 1419-1428. doi:10.1016/j.procs.2020.04.152

Ramalho, F., & Neto, A. (2016). Virtualization at the network edge: A performance comparison. Paper presented at the *WoWMoM 2016 - 17th International Symposium on a World of Wireless, Mobile and Multimedia Networks,* doi:10.1109/WoWMoM.2016.7523584

Ruan, B., Huang, H., Wu, S., & Jin, H. (2016). *A performance study of containers in cloud environment* doi:10.1007/978-3-319-49178-3_27

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2017). Performance comparison between container-based and VM-based services. Paper presented at the *Proceedings of the 2017 20th Conference on Innovations in Clouds, Internet and Networks, ICIN 2017,* pp. 185-190. doi:10.1109/ICIN.2017.7899408

Sande Veiga, V., Simon, M., Azab, A., Muscianisi, G., Fernandez, C., Fiameni, G., et al. (2019). Evaluation and benchmarking of singularity MPI containers on EU research e-infrastructure. Paper presented at the *Proceedings of CANOPIE-HPC 2019: 1st International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC - Held in Conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis,* pp. 1-10. doi:10.1109/CANOPIE-HPC49598.2019.00006

Seybold, D., Hauser, C. B., Eisenhart, G., Volpert, S., & Domaschka, J. (2019). *The impact of the storage tier: A baseline performance analysis of containerized DBMS* doi:10.1007/978-3-030-10549-5_8

Shirinbab, S., Lundberg, L., & Casalicchio, E. (2018). Performance evaluation of container and virtual machine running cassandra workload. Paper presented at the *Proceedings of 2017 International Conference of Cloud Computing Technologies and Applications, CloudTech 2017, , 2018-January.* pp. 1-8. doi:10.1109/CloudTech.2017.8284700

Shirinbab, S., Lundberg, L., & Casalicchio, E. (2020). Performance evaluation of containers and virtual machines when running cassandra workload concurrently. *Concurrency Computation, 32*(17) doi:10.1002/cpe.5693

Smith, J. E., & Nair, R. (2005). The architecture of virtual machines. *Computer, 38*(5), 32-38.

The kernel development community. (2020). *Seccomp BPF (SECure COMPuting with filters)* . Retrieved Jan 16, 2021, from https://www.kernel.org/doc/html/v5.10/userspace-api/seccomp_filter.html

The Linux Foundation. (2016). *Home - xen project.* Retrieved Jan 30, 2021, from https://xenproject.org/

The Linux Foundation. (2021). *About the open container initiative.* Retrieved Apr 4, 2021, from https://opencontainers.org/about/overview/

VMWare Inc. (2020). *Workstation pro.* Retrieved Jan 30, 2021, from https://www.vmware.com/products/workstation-pro.html

VMWare Inc. (2021). *What is ESXI?* Retrieved Jan 30, 2021, from https://www.vmware.com/products/esxi-and-esx.html

Weerasinghe, J., Abel, F., Hagleitner, C., & Herkersdorf, A. (2016). Disaggregated FPGAs: Network performance comparison against bare-metal servers, virtual machines and linux containers. Paper presented at the *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom, , 0.* pp. 9-17. doi:10.1109/CloudCom.2016.0018

Xie, X. -., Wang, P., & Wang, Q. (2018). The performance comparison of native and containers for the cloud. Paper presented at the *Proceedings - 2018 International Conference on Smart Grid and Electrical Automation, ICSGEA 2018,* pp. 378-381. doi:10.1109/ICSGEA.2018.00100

Y.-J. Hong. (2016). *Introducing container runtime interface (CRI) in kubernetes.* Retrieved Apr 19, 2021, from https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/

Yasrab, R. (2018). Mitigating docker security issues. *arXiv Preprint arXiv:1804.05039,*

Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2019). The true cost of containing: A gVisor case study. Paper presented at the *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19),*

Zhang, J., Lu, X., & Panda, D. K. (2016). Performance characterization of hypervisor- and container-based virtualization for HPC on SR-IOV enabled infiniband clusters. Paper presented at the *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016,* pp. 1777-1784. doi:10.1109/IPDPSW.2016.178

Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L., & Zhou, W. (2018). A comparative study of containers and virtual machines in big data environment. Paper presented at the *IEEE International Conference on Cloud Computing, CLOUD, , 2018-July.* pp. 178-185. doi:10.1109/CLOUD.2018.00030

Appendix A. Results of the benchmarking

Test	**Metal**		**KVM**		**Firecracker**		**gVisor**		**Docker**	
	Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.	Mean	Sd.
Boot. (s)	58,46	2,22	19,34	9,03	5,13	0,02	0,83	0,08	0,67	0,03
CPU (event/s)	499,35	10,45	455,36	52,18	406,18	59,40	404,48	80,02	419,89	59,87
File r. (MiB/s)	318,82	42,33	170,00	46,70	108,91	41,19	223,79	0,70	227,92	0,71
File w. (MiB/s)	65,24	3,97	45,88	2,25	57,57	4,39	28,45	0,34	55,46	3,76
Sent (Mbps)	940,41	1,61	938,24	4,99	921,90	13,19	875,00	25,76	939,81	2,88
Received. (Mbps)	943,53	1,59	941,39	4,94	925,71	13,02	877,36	25,75	943,03	3,00
Jitter (ms)	0,033	0,012	0,032	0,029	0,042	0,023	0,032	0,022	0,025	0,007
Pipe. (s)	291,10	1,52	355,20	19,71	372,40	6,98	-	-	303,70	14,71
	Mean		Mean		Mean		Mean		Mean	
Db read (trans./s)	29325,04		21916,47		-		7094,03		35243,14	
Db write (trans./s)	5944,71		4989,96		-		3068,93		6593,74	
Db rw (trans./s)	3274,96		3690,49		-		1184,50		3694,86	
HTTP (s/req.)	1425,18		2183,10		2259,84		-		1521,48	