Thesis

Benchmarking Container Engines with a Networking Perspective

Author: Albert Ärleskog, Daniel Ekström
Supervisor: Mattias Davidsson
Examiner: Arianit Kurti
Semester: Spring 2023
Subject: Computer Science
Level: First level
Course code: 2DV50E, 15 credits

Linnæus University
Sweden

Abstract

The growth of distributed applications stand on a foundation of containers and their communication and have seen the rise and fall of many implementations throughout the years with a mix of proprietary and open sources. Today there are two implementations widely used as a result of the popularity of the huge project Kubernetes: CRI-O and Containerd. Both with the edge responsibility of managing containers using similar underlying software raising the question; do they have any implications on the containers they spawn? This thesis investigate these implementations from a performance perspective with a custom developed tool for direct communication to them and run a suite of benchmarks within the containers created by each. The suite consists of tests for throughput, latency, cpu, memory, random file read/write and sequential file read/write. Results conclude they perform similarly in all, but the file tests which showed overall CRI-O dominating in write speed and Containerd dominating the read speed.

Keywords: container, container engine, benchmark, CNI, CRI-O, containerd, network, Kubernetes, performance, cloud

Linnæus University
Sweden

Contents

List of Figures

List of Tables

1 Introduction

This is a 15 credits bachelor thesis in computer science aiming to contribute knowledge to the rapidly growing containerization world with a primary focus of comparing different container engines and introduce a networking perspective in said benchmarks. Labor was split as evenly as possible, and each members programming area reflected who wrote the different parts of the thesis. Overall, programming of the tool focusing on communication with the container engines, log retrieval and the first half of the report was one part of the labor and the other being container configuration, log parsing, running the benchmarks and the second half of the report.

1.1 Background

Since the introduction of the computer, the creation nature of the human have inspired companies and collaborators to build programs of many different kinds with many different areas of application to solve challenges and effectivize among others. Software development schemes and designs have developed alongside as programmers gained knowledge on how different application areas may require different approaches and focus areas in their implementation, such as extendability and modularity. Sysadmins and other users alike utilize these programs and together with the documentation gain knowledge on its quirks and features as well as its recommended deployment approach to serve it for any type of end user. As programs grow in functionality and stability, so does their advancedness and the sheer size makes tasks such as testing and compiling an ever-growing black hole where resources disappear. A description of this phenomenon is "outgrowing the monolithic architecture” as Chris Richardson explains it in his book Microservices Patterns [1]. In general, a common solution to handling larger projects is dissecting and splitting it into smaller, more manageable parts which can be handled each by themselves. This is also the direction programmers have taken their software, towards a loosely coupled architecture, but new challenges arise with these new distributed fractions and their difficulty are multiplied by their quantity. The software evolves into a set of distributed applications, each with their own smaller stack, and one of the new challenges is the communication between them. The previous ties of shared memory between the internal software components have been exchanged for the IP-stack, thus emphasizing on the importance of networking. Another challenge is a framework fit for the task of handling these scattered components, which an emerging technology called containers appears to have a core role in.

The containerization era is arguably still in its infancy with multiple projects and methods aiming to solve similar challenges in different approaches, but an undisputed orchestrator of containers is Kubernetes and the developers of the Kubernetes project have begun standardizing the interfaces between the multiple components. With a huge number of adopters, it seems Kubernetes will have a core role in future IT-infrastructure, but similarly to the orchestrator of an orchestra, it is dependent on the performance of its musicians [2]. Kubernetes musicians are container engines and it communicates with them through an Application Programming Interface (API) called the Container Runtime Interface (CRI). Container engines that speak CRI have functionality to manage images, pods, networks and much more making them the implicit base of modern and the foreseeable future of IT-infrastructure, thus a very important part of the environment.

1.2 Related work

Our research area have seen an increased amount of articles with recent years, but mostly focused on comparison between techniques rather than different implementations within the technique. "Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes" by Wang et al. (2022) describe how a more secure approach to containers runtimes may have performance implications [3]. The authors conclude that both RunC and Kata have less performance overhead while gVisor suffers from significant performance loss in I/O and system calls, but feature better isolation. In "Performance Evaluation of Container Runtimes", Espe and Jindal (2020) benchmark and compare the container engines containerd and CRI-O with the different runtimes runc and gVisor from the perspective of performance regarding running containers and container runtime operations as well as scalability. [4]. The authors created their own tool called TouchStone to evaluate these perspectives and concluded that CRI-O has better file system operations (read/write) while containerd feature better CPU usage, memory latency and scalability aspects. "Container Hosts as Virtual Machines - A performance study" by Aspernäs and Nensén (2016) compare container performance between virtualized and bare-metal hosts running the operating systems CentOS, CoreOS and Photon OS [5]. Their tests were based on the Linux, Apache, Mysql and Php (LAMP) stack with a suite of macro-benchmarks. They concluded an overall decrease in performance for virtualized hosts, but the type of containerized applications, which operating system and type of operation differed between the performance gains. Manfredsson and Nyquist (2013) presented in "Jämförelse av Hypervisor & Zoner" a comparison between a hypervisor and a container implementation called Solaris™ Zones within the Solaris™ operating system by running a benchmarking suite on two scenarios [6]. The suite featured the programs Apache and Httperf which were tested on the scenarios of a single virtual server and three virtual servers. Virtual servers is either a container or a virtual machine. They concluded the reduced overhead of the container implementation performed better from a load perspective, but a restriction with containers only being able to run the same operating system as its host could be a limit in heterogeneous environments. In "Container-Based Cloud Virtual Machine Benchmarking", Varghese et al. (2016) investigate the difficulties of benchmarking virtual machines in the cloud with a primary issue of the time it consumes [7]. They propose a technique of only benchmarking a part of the virtual machine utilizing a custom developed tool to restrict available resources of the container which reduces benchmarking time by up to 91 times. Results showed an accuracy of 90% and 86% for sequential and parallel execution modes of the benchmarking program respectively. Kozhirbayev and Sinnott (2017) compared bare-metal, Docker and Flockport by running tests for cpu, memory, fileIO and networking [8]. Results showed a negligible difference in cpu and memory, but the container implementations saw curtailed performance with networking and fileIO. Notably Docker saw a huge reduction in total amount of random seeks during the fileIO test, about 90% lower compared to the other two targets, but retained a reasonable write throughput nonetheless within 10% of the other targets. Kushwaha's (2017) study on the performance of VM-based and OS-level container runtimes such as runc, and engines containerd and CRI-O concluded that CRI-O performs worse in comparison to containerd due to different file system driver interface design though CRI-O has very low container start-up latency in comparison to containerd [9].

1.3 Problem formulation

There are research closely related to our subject, but the main area either differs or make the wrong comparisons. The rather undefined nature of this area may be at fault such as terminology which has sparked multiple problems where researchers ask questions, formulate problems and draw irrelevant conclusions regarding completely different technologies. Wang et al. (2022) for example compares container engines with a container runtime and include evaluations of networking performance in a very similar manner to our planned testing, but does so unintentionally [3]. Runc is a container runtime reference implementaion of the Open Container Initiative (OCI) runtime specification which the authors compare to the container engine Docker [10]. The OCI runtime specification does not include networking configuration at all and since the authors utilize Docker for testing Runc, they are actually using Containerds network setup failing to realize they compare and draw conclusions between apples and pears [11]. Espe and Jindal (2020) does test performance also in a similar manner to the previous article with correct comparisons between container engines as opposed to mixed between container runtimes/engines, but completely omits networking from their testing [4]. By doing a proper comparison where we test the two container engines against each other, we get the best of both worlds where the correct engines are compared to each other and we have the additional metric of networking performance.

We aim to answer these questions:

- How do Container Runtime Interface compliant container engines compare to each other in relation to performance?
- What is the procedure to pick a Container Runtime Interface compliant container engine when factoring in different workloads?

1.4 Motivation

This thesis aims to clarify and formulate a base for companies and individuals when evaluating different container engines depending on different workloads as adapting technologies to their greatest strengths can help from multiple perspectives such as economical and utilize resources to their fullest.

1.5 Results

The result of this thesis will include two parts: firstly a judgement and verification of previous research technique and results while answering the first question and secondly extending said technique to comprise a networking perspective and tie the bag together answering the second question.

1.6 Scope/limitation

We limit ourselves to developing for the CRI compliant engines CRI-O and Containerd as Kubernetes is a major driver within the industry for these engines and the OCI compliant Runc as container runtime [2].

1.7 Target Group

The target group is any user of container engines within their infrastructure, from large companies to actors with limited resources looking to get the most out of their hardware.

1.8 Outline

This report is structured in the following way: Chapter 2 explains and discusses methodologies used, Chapter 3 describes the technical and theoretical knowledge, and Chapter 4 includes the objectives and their specific implementations. In Chapter 5 the results are shown and analysis of them, discussions are conducted in Chapter 6 and finally conclusions and future work in Chapter 7.

2 Method

The method of this thesis builds on the lack of benchmark categories from Espe and Jindal (2020) as the original developers of the container engine testing tool [4]. The tool is developed for directly communicating with container engines and running controlled benchmarks, called Touchstone.

2.1 Research Project

For starters, information needs to be gathered through a document study regarding the current state of how Touchstone uses the CRI, relevant programs, standards and interfaces. A stocktaking of sorts, as new developments since Espe and Jindal (2020) published their research may result in entirely different outcome compared to their older versions, thus needing to be highlighted.

Secondly, continued development of Touchstone will be conducted with the design science method [12]. The exploratory nature of the method will be beneficial when creating the extended functionality. Early identification of problems with Touchstone is it is currently a single container oriented tool, so there may be a need for large restructuring when testing with two containers, as configuration set at runtime for the first container may be needed in the second. For example, the Internet Protocol (IP) address of the second container for testing network communication, but this depends entirely on the findings in the previous paragraph. Another problem is the complete lack of testing any kind of networking.

Lastly, an experiment will be used to conduct the tests and compare with previous research. There are lots of variables to consider when comparing benchmarks between vastly different computers, so our primary target is looking for similar patterns in our tests and drawing conclusions from those patterns. Testing programs are picked based on previous research and personal familiarity.

2.2 Research methods

The gathering of initial information have but one relevant method; the document study. Experimenting to gather relevant knowledge is generally a valid approach, but in this case would be very time-consuming, so a much more effective method is using the respective projects documentation as it also provides otherwise missed in-depth knowledge. The Interview method could also be a viable option, but have similar flaws to experiments. Other common methods are ill fit for this goal. The information will be presented in chapter 3.

Design science is well fit for research producing artifacts and also for our particular challenge. The artifact will be the extended functionality of Touchstone and with the intention of reusability for future CRI compatible container engines. Considering design science is a big part of the experimenting section of this thesis, other methods that do not produce a tool for testing are not applicable. A more interesting perspective is the necessity of including this step at all or rather why the use of Touchstone, as this method is specifically derived from the lack of functionality in the tool. Benchmarking computers have a long history of different solutions and programs, but none of them, as <far as the authors know, target the CRI specifically. This step could have been replaced by a series of scripts or automation tools since both projects have their own Commandline Interface (CLI) applications to interact with the engine, but a large part of the rationale of a custom tool is the future reusability since the CRI is standardized.

The experiment will act as a base to validate and draw conclusions about the differences of the container engines. For each benchmark, the dependent variable is the benchmark itself and independent variables are the container engines and the amount of threads. The first part of the Research Project may influence the independent variables when mimicking previous tests.

2.3 Reliability and Validity

Reliability regarding benchmarks in computer systems is an everlasting threat with few mitigations. The combinations of different software, software versions and hardware is enough to uniquely define a system, thus also making it hard to exactly replicate benchmarks. An example of this is using HTML5s canvas element to reliably fingerprint systems without any type of cookie or tracker because the underlying system is unique enough that the pixels in the resulting render mirrors the systems unique properties [13]. As such, we will not directly take any measures to reduce the impact of the underlying system, but run the tests multiple times on multiple machines and inside and outside of a virtual machine to generate a broad view of how these engines behave and look for similar patterns. Essentially looking at the relative numbers of the benchmarks between the engines rather than absolute numbers. For example the disk read and write speed of an Secure Digital (SD) card in a Raspberry Pi is not comparable to an Non-Volatile Memory Express (NVMe) Solid State Drive (SSD) in a desktop computer in terms of raw numbers, but there may be a similar pattern where one engine performs 10% better or worse compared to the other in both of these environments, creating a base to draw conclusions from indicated by their relativeness rather than absoluteness. Version pinning relevant software also contributes to the reliability.

The validity of the thesis is increased by a collection of actions. Previously mentioned randomness of hardware contributes to the validity by reducing selection bias. Furthermore the results were compared primarily with the results of Espe and Jindal (2020), the previous developers of Touchstone, representing a control group.

The extended functionality however have a unique stance in the research community with extremely few results to compare with. Wang et al. (2022) provided networking benchmarks in their research and since they used Docker, more specifically Containerd, they were used as a second control group to provide a broader picture of how the engines should perform.

2.4 Ethical Considerations

There are no ethical concerns with this thesis since no personal or private data is used.

3 Technical Background

This chapter intends to give a description of the technologies used in this thesis project. It aims to create theoretical background for the technologies and subject which are addressed in this report. The purpose it to give the reader a understanding of the technologies, tools and concepts the thesis uses to answer the research questions. The new technologies we chose to work was from previous positive experience and some technologies was used since they were used in the previous research.

3.1 Containers and pods

The container concept have multiple implementations, but a core concept of utilizing a single kernel with functionality for grouping runtime properties such as filesystem and process cputime allocation to create a tailored sandboxed environment for the specific process(es). The lower overhead by using this method allows for close to native performance compared to virtualization, another type of sandboxing, where virtual interfaces are created to be consumed by another kernel on top. Containers can be grouped together by sharing one or multiple runtime properties creating a pod. A pod depends on the grouped runtime properties provided by the kernel, thus cannot be ran on two machines at the same time by, for example creating a container on each and joining them. Communication between pods are done over a network which is usually setup automatically by the tools utilizing these concept such as Docker [14].

An implementation of these concepts are namespaces in Linux. It features the isolation of runtime properties: network, mounts, process ids and user to mention a few [15]. A process have lots of properties and configuration and in relation to namespaces, ids for each of the namespaces it belongs to. For example two processes can have network namespace id two, thus sharing the network stack with the same interfaces for ip communication. They may also have different user namespaces where one of the processes is ran as user 1000 outside the namespace, but the user inside the namespace is 0, the root user, tricking forked processes inside that they are ran as root. Processes inside the namespace cannot access any resources requiring privileged access, even if they are shared a file from the mounts namespace since it is a "fake root" only existing inside the namespace. Outside, and the user actually trying to access the privileged resource is user 1000, which may or may not have access. This is a feature of the user namespace by creating a mapping scheme for user and group ids outside and inside the namespaces.

3.2 Container Runtime Interface

The Container Runtime Interface (CRI) is a specification created by the Kubernetes project as an Application Programming Interface (API) over gRPC [16]. gRPC is a "A high performance, open source universal RPC framework" over Remote Procedure Call (RPC) which the specification is both defined as and have a reference implementation in Golang [17]. The specification features structured protocol to among others a create pods, containers and pull images that can be understood by the receiving container engines.

3.3 Open Container Initiative specification

The Open Container Initiative (OCI) is an "open governance structure for the express purpose of creating open industry standards around container formats and runtimes." with members such as Facebook, Google, Microsoft, AWS, Oracle and Redhat to name a few [18]. It was established in 2015 by Docker, CoreOS and other leaders in the container industry and is responsible for three specifications: the Image specification, Runtime specification and Distribution specification. OCI describes a sample workflow as "At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime." [18]. We work mainly with the image spec and runtime spec in this thesis.

3.4 Container Runtime

The container runtime is the lowest level of the components making up a container stack. It talks directly to the kernels sandboxing features.

3.4.1 Runc

Runc is a lowlevel commandline tool for running images according to the OCI specification. A relation to the Linux namespaces can be viewed when running a process manually with Runc as it creates a new namespace for each of the available properties.

3.5 Container Engines

Container engines act as a middle-man with default drivers for networking and storage and in our case using Runc as runtime. Their responsibility include among others container life cycle management and pulling images.

3.5.1 CRI-O

CRI-O is specifically designed as the path between the Kubelet and OCI conformant runtimes by following the CRI closely and also sharing the scope [19]. One could say it is a complete reference implementation of the CRI to provide a simple engine and focus on picking a OCI compliant runtime.

3.5.2 Containerd

Containerd was historically a part of Docker as its container engine, but was donated to Cloud Native Computing Foundation (CNCF) [20]. It is still in use by Docker today and provides its own library in Golang to interact with it. The CRI compliance is achieved through an official plugin and is easily enabled in the configuration file.

3.6 Kubernetes

Kubernetes was originally developed internally at Google, but released under an open source license in 2014 for the rest of the world to use [21]. It is defined as an orchestrator i.e it does not run any containers itself, but instructs the container engine(s) what configuration to run the pods and containers with. This is an interesting piece of the puzzle when distributing programs across multiple hosts, hosts running container engines. For example deploying a database; we tell Kubernetes to always keep three replicas of the database container up at all times, they should not run on hosts with "loadbalancer" in the hostname and they should use fast storage. Kubernetes brings up three replicas, ignores hosts with "loadbalancer" in the name and use a storageclass called fast which we have predefined. Kubernetes notices if a host goes offline and tells a container engine on another host to create a database container with the same specification as the failed one. In its essence, Kubernetes is just a couple of programs also running as containers on one or multiple hosts interacting with a specific agent on each host called kubelet [22].

3.7 Touchstone

Touchstone is the target for our continued development efforts. It acts as a simple middle hand between the user and the container engine with a declarative configuration for what and how a benchmark should run. Its architecture is outlined by Espe and Jindal (2020) and consists of six modules called benchmark, runtime, visual, suites, cmd and config [4]. The benchmark module acts as the controller in the Model, View and Controller (MVC) design pattern and communicate with runtime which wraps the gRPC client. Visual is the report output after benchmark runs and cmd provides the CLI frontend. Last but not least are the benchmarks themselves which creates in suites with configurations from config.

3.8 Testing software

Testing software are the programs running inside the containers. The networking tools were chosen because of previous familiarity while Sysbench was used in previous research. Ping is a common networking tool for sending Internet Control Message Protocol (ICMP) request packets and waiting for an ICMP response packet. Ping have many implementations, but we will use the default command in the Alpine Linux container image which is contained in the Busybox binary [23]. iPerf3 is a bandwidth testing tool over IP networks with lots of options for tuning parameters. It features UDP, TCP over either IPv4 and IPv6 to mention a few. The original iPerf was developed by a joint effort including in selection representatives from National Laboratory for Applied Network Research (NLANR), Internet2 and University of Nebraska-Lincoln (UNL) [24]. The latest iteration, which we will be using, is also a joint effort, but this time with representatives from Energy Sciences Network (ESnet) and Lawrence Berkeley National Laboratory [24]. Sysbench is a popular multi-purpose benchmarking tool that is commonly used to evaluate the performance of various system components, including the CPU, memory, fileIO, and database systems. It provides a standardized and extensible framework for running different types of benchmark tests and generating performance metrics [25].

4 Research project – Implementation

4.1 Touchstone(2)

After careful evaluation of Touchstone, we decided to redesign the application from the ground up, as we feared would be required when we established the method. Creating the second container, a functionality that was not present in the original version of touchstone, required waiting for the first container to be in a running or sleeping state before retrieving of its configuration, halting the execution till that point. The request and response method of communicating with the engine does not have an event feature, making waiting for containers changing state require sending multiple requests until state changes or a timeout. The underlying abstractions did not intend for this functionality requiring a large rewrite of those parts. Furthermore, which tests to run were specified in external YAML files making the tool harder to distribute to multiple systems. Thus we decided to create a new tool we call Touchstone2. The objectives for our tool in relation to the design science method and close remark to software requirements specification (SRS):

1. Extensible for different workloads/testing programs.
2. Run multiple containers at the same time with configuration dependencies between them.
3. Single binary for easy distribution.
4. Results are written in a general format such as JSON or YAML.

Solving an extensible approach with our new tool required a common interface all benchmarking programs utilize; the standard output (stdout). Starting a program or a program inside a container attaches a file descriptor to the process's stdout where log and common print functions usually write their output unless configured otherwise. Capturing output in Touchstone2 is thus an important part for extensibility and can be done in multiple ways. One way is to start the container process with the CLI applications for the respective engines, but that would contradict the point of utilizing the CRI and require specific reimplementation with new engines as their CLI applications does not necessarily conform to any standard with arguments and order of arguments etc. A feature of the CRI is specifying a log path where stdout of the container is written which solved the issue as long as the container attaches to the stdout of the benchmarking process which can be specified in the image.

Running multiple containers at the same time is simply done by issuing multiple requests to the engine, however the challenge arises when there are dependencies on one of the containers resulting in halting the request queue for the following containers. There is no way to reliably wait for the first container, as mentioned, as lack of an event interface is a general limitation of the request-response method. A webhook is a common solution among webapplications, but no such functionality is implemented in the container engines. Thus a simple wait loop is required to request the config of the created container.

Single binary and results written in a general format are both experiences we bring from the original Touchstone. To contribute to the reliability of our thesis, simple distribution is a cornerstone for our new tool where we minimize external dependencies. Similarly we also restrict internal dependencies on libraries as these would require pulling from multiple repositories when building the tool.

4.2 Systems

CPU	Intel I7 10700k @5.0 GHz
MEMORY	DDR4 32 GB 3600 MHz
STORAGE	256 GB SSD
DISTRIBUTION	Fedora 36
KERNEL	5.17

TABLE 4.1: System 1, x86

CPU	AMD 5700G
MEMORY	DDR4 32 GB 3200 MHz
STORAGE	2x 1000GB NVME SSD PCIe 3.0 with BTRFS RAID1
DISTRIBUTION	Arch
KERNEL	6.3

TABLE 4.2: System 2, x86

CPU	Broadcom BCM2711
MEMORY	LPDDR4 4 GB 3200 MHz
STORAGE	MicroSD C10 32GB
DISTRIBUTION	Debian 11
KERNEL	5.10

TABLE 4.3: System 3, a Raspberry Pi 4 B rev. 1.1

4.3 Environment

Each x86 system ran the tests baremetal and in a VM of size 4 cpu and 8GB ram allocated with Debian 11 as operating system while the tests were only ran baremetal on system 3. Setting up the environments and installing software were automated using Ansible.

Application	Version
CRI-O	1.27
Containerd	1.6
Runc	1.1
CNI	1.1
Sysbench	1.0
Ping	1.36[1]
iPerf3	3.13

TABLE 4.4: General version table

[1]Version of the Busybox binary

The patch version in 4.4 is intentionally excluded as the specific version at the time of benchmarking should be insignificant since projects following the Semantic Versioning Specification, where the last number is specified as "Patch version Z (x.y.Z | x > 0) MUST be incremented if only backward compatible bug fixes are introduced." should not have any implications on the test results [26].

A rough outline of how our testings were ran in each environment:

1. Run Ansible on target environment. Optional if correct software was already installed as per 4.4.
2. Build Touchstone2 for intended tests and architecture.
3. Run Touchstone2. Results are saved in a JSON file.

5 Results And Analysis

In this section, the result of our experiment with our tool presented and analyzed. information will be given in for each graph. The graphs present the results, and surrounding text presents an analysis. The results shown were run on a virtual machine on system 1. We saw similar results across the other systems and virtual machines, so we decided to only include results from one environment in the report.

5.1 CPU

The Results seen in 5.1 showed the computational performance of CRI-O and containerd. It is to be expected that the difference between CRI-O and containerd is minimal since containerization generally introduces a minimal amount of overhead compared to bare metal environments. Both CRI-O and containerd share architecture and are built on similar underlying technologies. They share common design principles such as lightweight container isolation, minimal overhead and efficient resource utilization. This contributes to similar CPU performance. And the results show that is the case for computational performance. The results are evaluated with sysbench. The benchmark is configured to calculate 20,000 prime numbers with four threads and report the number of events executed per second as the performance metric. The test was run ten times. The figure shows that containerd computational performance is almost 0.1% faster the CRI-O which is equivalent to zero different in performance.

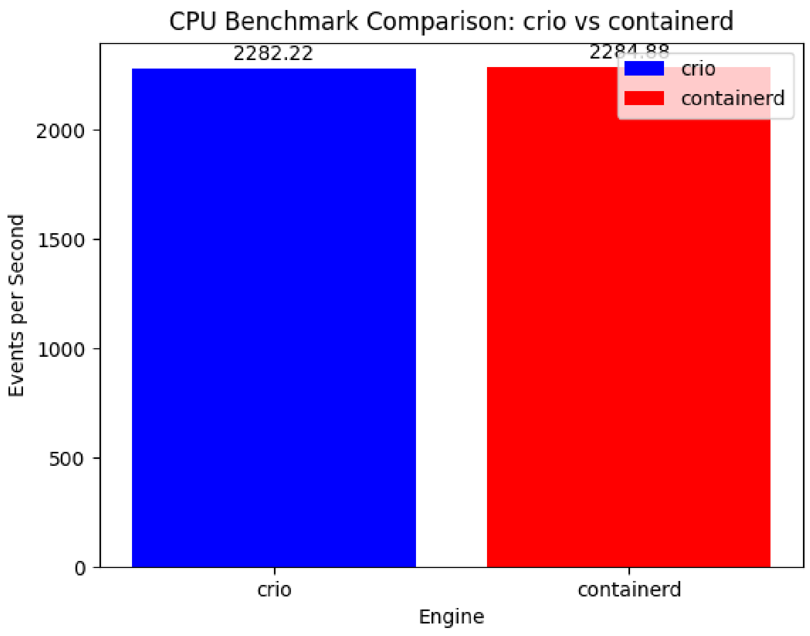

FIGURE 5.1: CPU computational performance (More is better)

5.2 Memory

The result seen in 5.2 showed the total operations per second using RAM of CRI-O and containerd. The test was performed with a tool called sysbench. The benchmark will allocate a memory buffer and then read or write from it until the buffer size has been from or written to. These results are from the benchmark using four threads and default memory buffer size 102400M. The results show that containerd did let operations per second then CRI-O. The CRI-O is about 1% faster the containerd. The total time average it took to run these benchmark are almost identical which vary from the results that the original touchstone developers got [4]. Running the exact same benchmark as they did, in these results they are closely matched, however they recorded a more noticeable difference. They got a difference of 3 seconds between containerd and CRI-O. The explanation could be a more optimization for the container engine. even running The same docker image as they did. Same results.

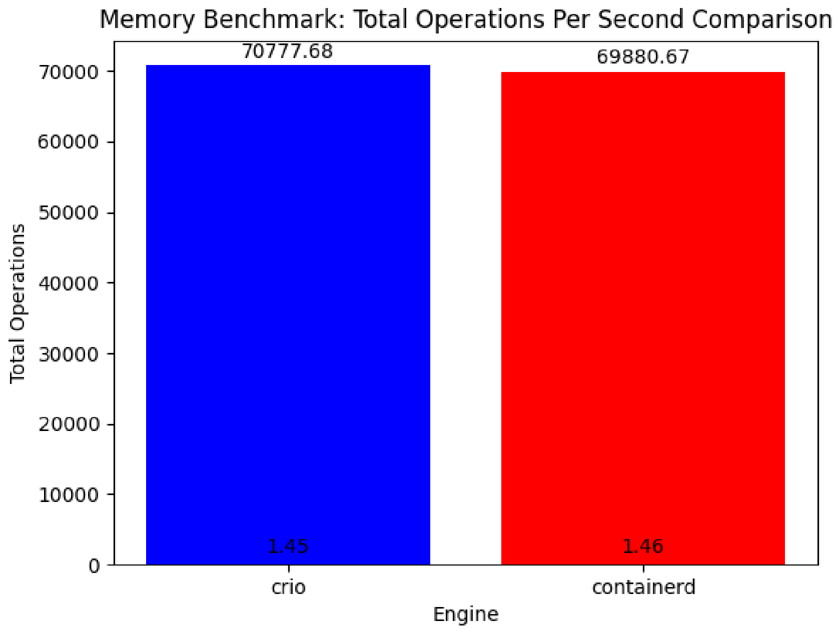

FIGURE 5.2: Total operations per second (More is better)

5.3 Storage

5.3.1 Random Read/write

The results seen in figure 5.3 showed the metrics from running sysbench FileIO test. The test consist of a prepare stage and a run stage. At the prepare stage, Sysbench creates a specified number of files with a specified total size. Sysbench performs checksums validation on all data read from the disk. On each write operation the block is filled with random values, then the checksum is calculated and stored in the block along with the offset of this block within a file. On each read operation, the block is validated by comparing the stored offset with the real offset, and the stored checksum with the real calculated checksum [25]. This random read and write is single threaded. The results show that CRI-O is superior when the operations are random read and write. Containerd crumbles when needed to perform both random reads and random writes at the same time. The expected difference is the underlying file system that these container engine uses.

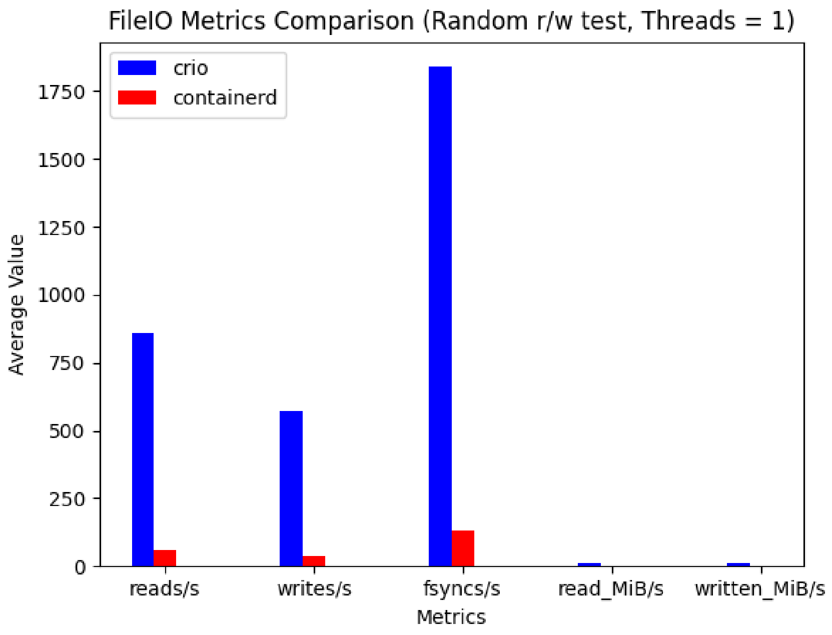

FIGURE 5.3: Random FileIO Read/write performance (Threads = 1, More is better)

The results in figure 5.4 is the same benchmark as the one above, however this time it uses four threads instead of just one. The results show that both CRI-O and Containerd uses multi threading in their underlying file system, nonetheless it doesn't help containerd. The difference between CRI-O and containerd is still there.

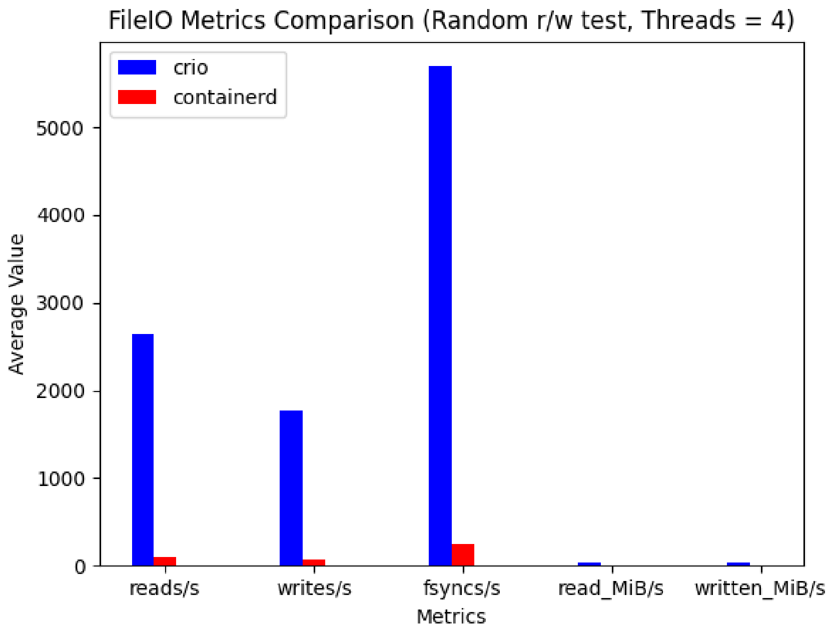

FIGURE 5.4: Random FileIO Read/write performance (Threads = 4, More is better)

5.3.2 Random Read and write

The results seen in figure 5.5 showed the metrics from running sysbench. The test consist of the same prepare and run stage. This result consist of running random read and random write individually and not simultaneously. The results show a huge increase in containerd random read performance. Containerd random read performance is nearly 10 times the performance of CRI-O. The results also show a huge loss for containerd when it comes to random writes performance. It almost looks like there is nothing, however the results show that it wrote about 50 writes/s.

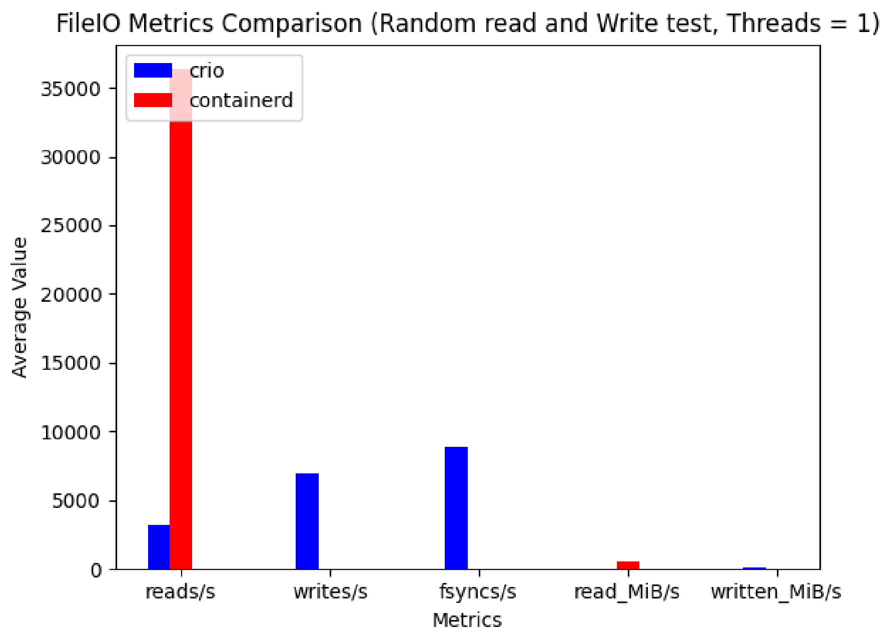

FIGURE 5.5: Random FileIO Read and write performance (Threads = 1, More is better)

The results shown in figure 5.6, is the same tests presented in figure 5.5. The only difference here is that the benchmark used four threads instead of one. The results showed an increase in performance, however the relative performance stayed the same.

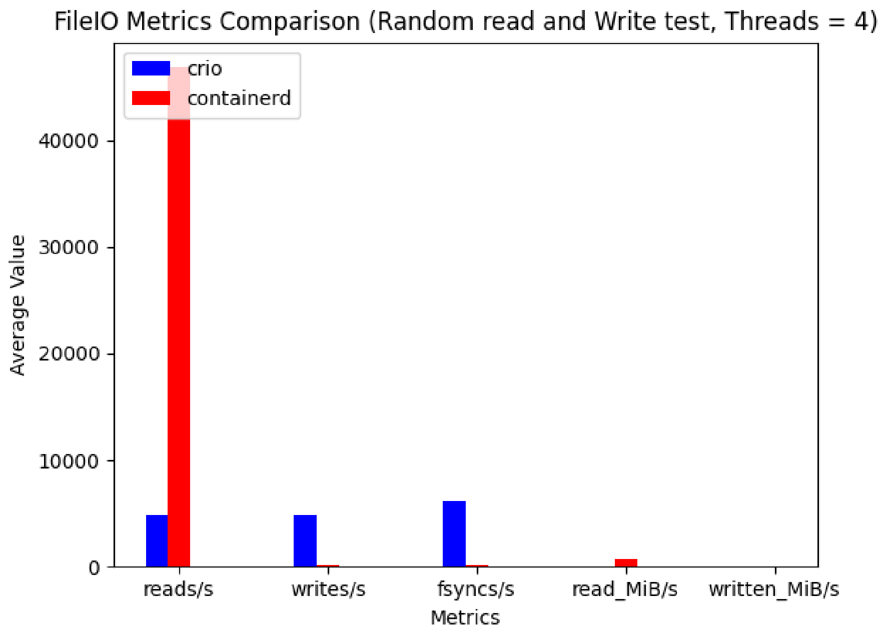

FIGURE 5.6: Random FileIO Read and write performance (Threads = 4, More is better)

5.3.3 Sequential Read and write

The results shown in figure 5.7 displays the metrics from running benchmark sysbench. This test consist of prepare and run stage. This result consist of running sequential reads and writes on a specific number of files using only one thread. The results show that containerd is superior in reads comparing to CRI-O. However, when comparing writes it is a clear win for CRI-O, it outperforms containerd by quite a margin. A possible explanation is the storage driver that containerd idealizes is especially optimized for fast read but slow writes [20].

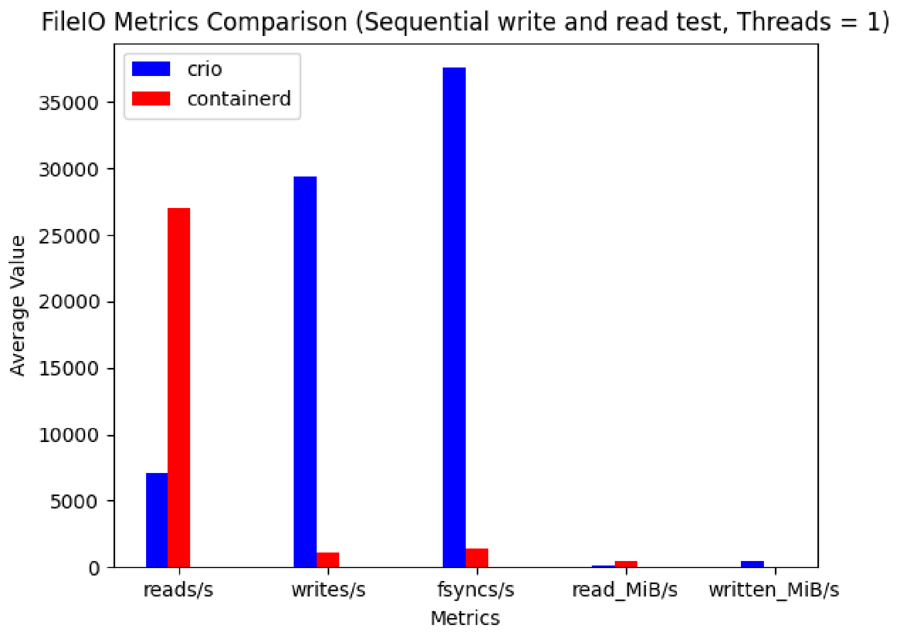

FIGURE 5.7: Sequential FileIO Read and write performance (Threads = 1, More is better)

The results shown in figure 5.8 is the same running benchmark as in figure 5.7, however it utilizes four threads instead of one. The results present an increase in for both containerd and CRI-O in reads department. Containerd and CRI-O are closely matched in the reads department, which means CRI-O really utilizes all four threads. However, at the same time the writes for both the container engines display a almost identical performance.

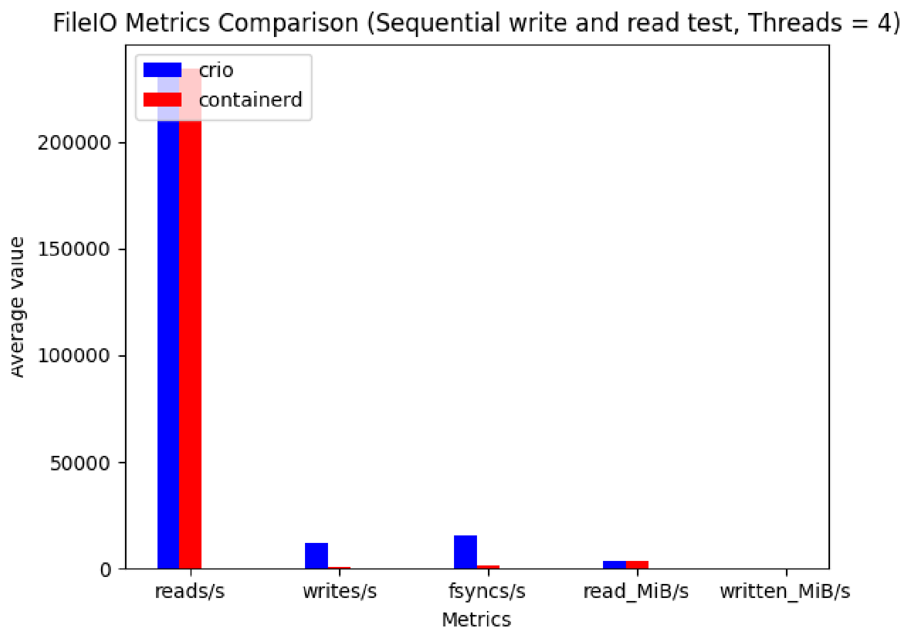

FIGURE 5.8: Sequential FileIO Read and write performance (Threads = 4, More is better)

5.3.4 Sequential Rewrite

The results shown in figure 5.9 displays the metrics from running benchmark sysbench. This test consist of prepare and run stage. This result consist of running sequential rewrites on a specific number of files using only one thread. CRI-O is a dominant container engine when it comes to write speed. The results show that compared to containerd, CRI-O is around 400% faster when it comes to writes.

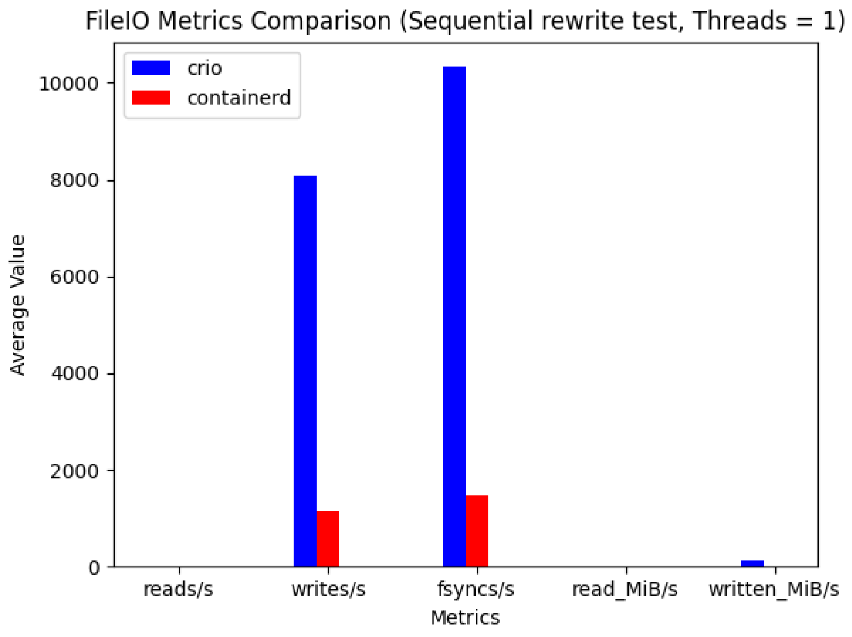

FIGURE 5.9: Sequential FileIO rewrite performance (Threads = 1, More is better)

The results shown in figure 5.10 is the same running benchmark as in figure 5.9, however it utilizes four threads instead of one. It shows a small decrease in performance for CRI-O engine, however containerd stayed the same. Nonetheless, CRI-O is relatively much faster in the writes department.

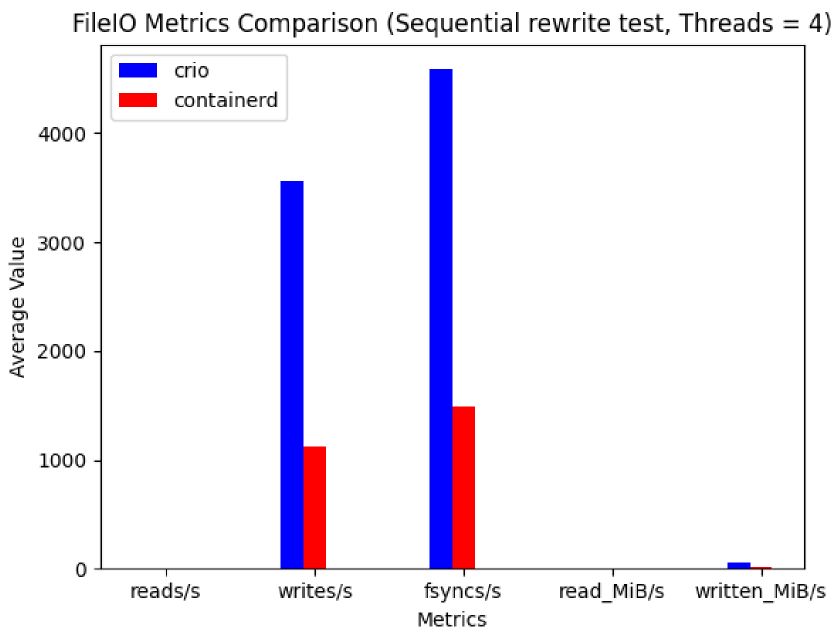

FIGURE 5.10: Sequential FileIO rewrite performance (Threads = 4, More is better)

5.4 Throughput

The results shown in figure 5.11 is the throughput from running a benchmark called iperf3. iPerf3 is a popular open-source tool for measuring network performance by generating TCP and UDP data streams between two endpoints. It helps to assess the bandwidth, throughput, and various other parameters of a network connection. The result is from a connection between two containers, one is a server and one is a client connected to see server. The results displayed is the throughput of data that is transferred between the two containers. The results show that the throughput between two containerd engines and two CRI-O engines is closely matched and can almost be said as the same performance.

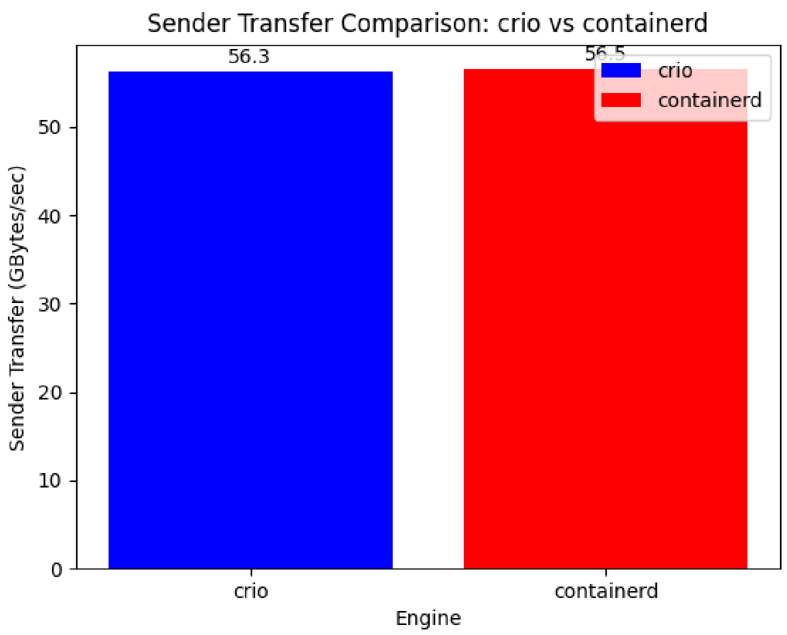

FIGURE 5.11: Throughput performance (More is better)

5.5 Network Latency

The results shown in figure 5.12 is the latency or how fast a data signal travels from one place to another, it determines round trip time. RTT is a measure of how long it took to receive a response. Measured in milliseconds (ms), the process starts when containerA in our case sends a request to containerB and is completed when a response from the containerB is received. RTT is a key performance metric. The result show a similar RTT for CRI-O and containerd. The average RTT for CRI-O is around 6% lower then containerd. For the max RTT recorded we measure a almost 50% increase in RTT for containerd compared to CRI-O.

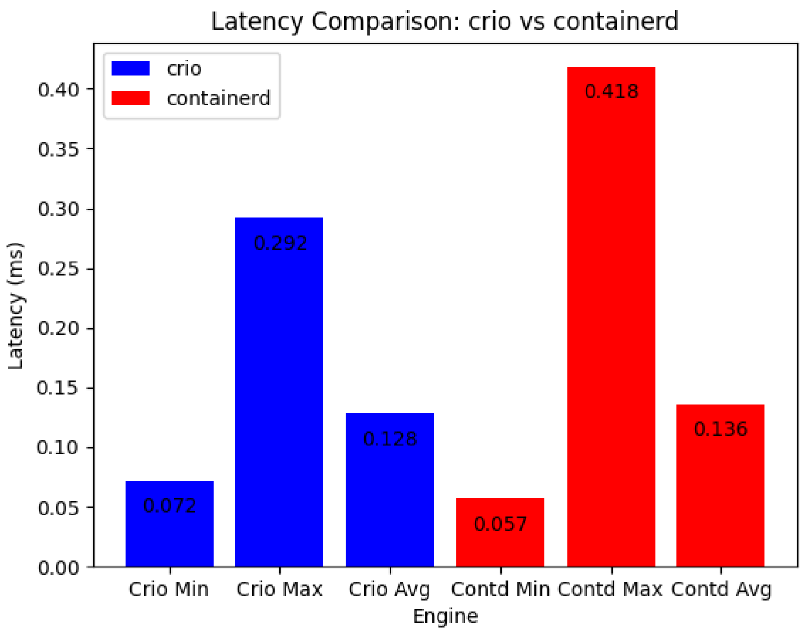

FIGURE 5.12: Latency performance (Less is better)

6 Discussion

What is the procedure to pick a Container Runtime Interface compliant container engine when factoring in different workloads?

Our research is a continuation of previous performance findings, and we use the current benchmarks presented in these research articles and extend them to cover not only performance of CPU, fileIO, memory, additionally include networking performance.

The purpose of this study was to answer the following research questions: 1) How do CRI compliant container engines compare to each other in relation to performance? In this study it was between containerd and CRIO 2) What is the procedure to pick a CRI (Container runtime interface) compliant container engine when factoring in different workloads?

The answers to these research questions were found through a multi-method scientific process. The process consisted of a document study to gain knowledge about the current state of how Touchstone uses CRI, relevant programs, standard and interfaces. The multi-method scientific process also contained the design science method. With this multi-method approach, we were able to adopt Touchstone from previous research that was used to evaluate a container engines performance [4]. We decided to use this tool as a foundation and to then further develop it to be able to create multiple containers in the same runtime. This functionality was necessary for the reason that we needed two containers to be able to talk to each other for our performance benchmarks to take place. With this tool developed, we were able to independently test the different container engines to ensure our results would not be influenced by other variables. In this study it was between containerd and
CRIO

The last part was to conduct an experiment to compare to previous research to see if there were differences in the performance. The experiments target was to collect relative performance since using absolute numbers would be useless since there are a lot of variables that affect the results. These variables range from computer to computer. The experiments results show some moderately different results from what we were expecting. When comparing our results to the results, [4] got using the same benchmarks. The two most notables differences we measured were memory and fileIO performance. They measured a three-second difference between CRI-O and containerd. However, we saw a very little difference of just 0.01 seconds, which is so close it’s almost a tie and definitely something you can’t notice unless you put them side to side. We used different variables to evaluate how the container engines fileIO performance compares. They used total time to compare the container engines. However, we decided to use a set time of ten seconds to compare how they perform under the same time interval and read the reads and writes instead. If we compare their results to ours, containerd and CRI-O has completely different upsides. We saw a dominance from containerd in almost every read only benchmark, compared to CRI-O. The results also show a dominance for CRI-O when having to write or needing to read and write at the same time. In real world examples, you would have a database that has to do a lot of reads, you would containerize that database with the containerd container engine.

With all these results, we could show the difference in performance between CRI-O and containerd, and see their differences. These results are an indication that there is a point to choosing a container based on the workload that is expected to be executed. This work also displays ways to test these container engines performance.

7 Conclusion

In this study, we examined the relative performance of running different container engines. This was done using a developed tool to run these benchmarks for us and log the benchmark results.

Our conclusion is that the two different engines we used to benchmark performance have their own workloads where they are superior to each other. Depending on the workload, it does make sense to look into the benchmarks to pick an engine that benefits the workload. We found that the network, CPU, and memory performance is nearly identical. However, when it comes to fileIO we found that containerd possess great read speed when it comes to random reads or sequential reads. However, from the writes and random reads and writes benchmark, the results show a relative noticeable performance difference. CRI-O possesses a great performance increase compared to containerd in this area. These benchmark show that read heavy scenarios benefit containerd and writes benefit CRI-O.

In conclusion, to answer the first research question. Our study and results show that comparing the container engines in relation to performance, there is indeed a difference in performance. And to answer the second research question, the procedure to pick a CRI compliant container based on workload. It totally depends on what the specific task the container in going to perform. Real world examples where the different container engine makes sense. For example Real-time Data Processing, if you have a high-frequency data steams coming in and need to process them containerd would be the preferred container engine. Another example would be a database that has to do a lot of transactions processing, which requires fast read and write speeds to serve data to applications and ensure data consistency.

7.1 Future work

Future work for this research area could be to conduct more test for other container engines, like LXC. It would also be interesting if more individual tools to test the performance like sysbench and iperf3 would be developed. To further collect metrics on which container engines are superior. Further metrics on how long it takes to create a container and destroy it would also be a important benchmark to run, one example could be to create a container, run some test or application then destroy it, and take the total time it took. Additionally, another variable to consider is to consider using more container runtimes instead of only using runc. A further area to explore is the security for these container engines.

References

[1] C. Richardson, *Microservices Patterns*. Manning Publications, 2018, ISBN: 9781617294549.

[2] *The state of kubernetes 2023*, Company released ebook, 2023. [Online]. Available: https://tanzu.vmware.com/content/ebooks/stateofkubernetes-2023.

[3] X. Wang, J. Du, and H. Liu, "Performance and isolation analysis of runc, gvisor and kata containers runtimes," *Cluster Comput*, vol. 25, pp. 1497–1513, 2022.

[4] L. Espe and A. Jindal, "Performance evaluation of container runtimes," *Cloud Computing and Services Science*, vol. 10, pp. 273–281, 2020.

[5] A. Aspernäs and M. Nensén, "Container hosts as virtual machines - a performance study," B.S Thesis, Linnaeus University, 2016.

[6] A. Manfredsson and J. Nyquist, "Jämförelse av hypervisor & zoner," B.S Thesis, Linnaeus University, 2013.

[7] B. Varghese, T. L. Subba, L. Thai, and A. Barker, "Container-based cloud virtual machine benchmarking," IEEE International Conference on Cloud Engineering, 2016.

[8] Z. Kozhirbayev and O. R. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017.

[9] K. Kushwaha, "How container runtimes matter in kubernetes?," 2017. [Online]. Available: https://events19.linuxfoundation.org/wp-content/uploads/2017/11/How-Container-Runtime-Matters-in-Kubernetes_-OSS-Kunal-Kushwaha.pdf (visited on 2023-06-06).

[10] "Runc." (2023), [Online]. Available: https://github.com/opencontainers/runc (visited on 2023-05-20).

[11] "Open containers initiative image specification." (2023), [Online]. Available: https://github.com/opencontainers/image-spec/blob/v1.0.2/spec.md (visited on 2023-05-20).

[12] K. Peffers, T. Tuunanen, M. Rothenberger A, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–78, 2007.

[13] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," M.S. thesis, University of California, La Jolla, California, USA, 2012.

[14] "Docker networking." (2023), [Online]. Available: https://docs.docker.com/network/ (visited on 2023-05-22).

[15] "Manpage namespaces." (2023), [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 2023-05-22).

[16] "Cri." (2023), [Online]. Available: https://github.com/kubernetes/cri-api/blob/v0.27.2/pkg/apis/runtime/v1/api.proto (visited on 2023-05-22).

[17] "Grpc." (2023), [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/ (visited on 2023-05-22).

[18] "Open containers initiative overview page." (2023), [Online]. Available: https://opencontainers.org/about/overview/ (visited on 2023-05-22).

[19] CRI-O. "Crio." (2023), [Online]. Available: https://github.com/cri-o/cri-o (visited on 2023-05-16).

[20] "Containerd." (2023), [Online]. Available: https://github.com/containerd/containerd (visited on 2023-05-16).

[21] "Kubernetes first release." (2023), [Online]. Available: https://github.com/kubernetes/kubernetes/tree/v0.2 (visited on 2023-05-15).

[22] "Kubelet reference." (2023), [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (visited on 2023-05-22).

[23] "Ping.c." (1999), [Online]. Available: https://git.busybox.net/busybox/tree/networking/ping.c?h=1_36_stable (visited on 2023-05-21).

[24] "Iperf authors." (2023), [Online]. Available: https://iperf.fr/contact.php#authors (visited on 2023-05-22).

[25] A. Kopytov. "Sysbench." (2023), [Online]. Available: https://github.com/akopytov/sysbench (visited on 2023-05-16).

[26] "Semantic versioning 2.0.0." (2023), [Online]. Available: https://semver.org/spec/v2.0.0.html (visited on 2023-06-05).