Performance Evaluation of Lightweight Virtualization Technologies

Nitin Goyal
Electrical and Computer Engineering
University of Florida
Gainesville, FL
nitin.goyal@ufl.edu

***Abstract*—Modern computing environments demand agile and portable solutions to maximize resource utilization and application deployment flexibility. Today we have lightweight virtualization technologies options available to achieve this. System containers, application containers and micro-virtual machines hold potential in this domain. These technologies are used depending on the level of isolation and how it fits with the use-case in the broader system. Understanding their features and performance characteristics is crucial for selecting the most suitable technology, ensuring optimal application performance and resource efficiency across diverse environments. As a part of this project, I perform CPU, Memory and IO performance benchmarks using sysbench and compare the results.**

Keywords— benchmark, containers, micro-VMs, docker, oci runtime-specs, runc, runsc, incus, firecracker

1. Introduction

Serverless computing have become popular in recent years and being offered by multiple cloud providers. Virtualization technologies, e.g. containers and VMs (virtual machines), are building blocks to enable multi-tenant cloud environments. Depending on the type of service users get different isolation. In FaaS (Function as a Service) for instance, isolation takes priority to enable secure access to customer data. This requires full system virtualization with VMs. A full VM would not be suitable for the requirements of serverless applications which demands for low start-up time and low performance overhead. Lightweight virtualization technologies are designed and built to serve this purpose. Quick start-up times and low performance overheads make them well suited for these applications. Of course, the choice of the technology would vary with the requirements of the service.

A full VM creates a complete, isolated environment with its own kernel, operating system, memory and resources. This provides excellent security, making it ideal for sensitive applications and running software incompatible with the host system kernel. They boot slowly and require a significant amount of available resources; and not all of the applications need this isolation. Containers share host's operating system kernel and are lighter on resource consumption compared to VMs. With a very little start-up time they provide excellent scalability. A container could be an application container which are designed to run a specific application and its dependencies, or a system container which allows multiple applications to run inside a single container but comes with additional overhead. Lastly, A microVM, essentially lightweight VM are designed for containerized workloads. It is the best of both worlds: the security and isolation of VMs with the efficiency and speed of containers.

In this study, I evaluated: system containers with Incus, application containers with runc (default for most container runtimes) and gVisor (highly secure alternative to runc) OCI (open container initiative) runtimes and firecracker microVM.

In the rest of the section of this report - Section 2 provides the basic background knowledge for the paper. Section 3 discusses the related works. Section 4 provides details of testing environment and benchmark results. Section 5 summarizes the evaluation results. Finally, section 6 concludes the work.

2. Background

The technologies that have been evaluated vary significantly w.r.t. to their underlying architecture and the features they provide to the user. Let's look at high level designs of each of these technologies.

2.1 Evaluated OCI Runtimes

Open Container Initiative (OCI) is a Linux Foundation project to design open standards for containers. The OCI defines two standards – the *image-spec* for OCI images and the *runtime-spec* for OCI runtimes. Often, container runtimes are divided into two categories – low-level (e.g. runc, gVisor) and high-level (e.g. containerd, docker). The difference is in the amount of consumed OCI specification and additional features. Figure – 1 shows an *image-spec* and *runtime-spec* job sequence.

A typical job sequence would be that a high-level container runtime downloads an OCI image, unpacks it and prepares an OCI runtime filesystem bundle on local disk. After that, the OCI runtime bundle will be run by an OCI Runtime. The *runtime-spec* defines how to run the OCI filesystem-bundle. It specifies the configuration, execution environment and

lifecycle of a container. This subsection explores two low-level container runtimes aka OCI runtimes - *runc* and gVisor (*runsc*)

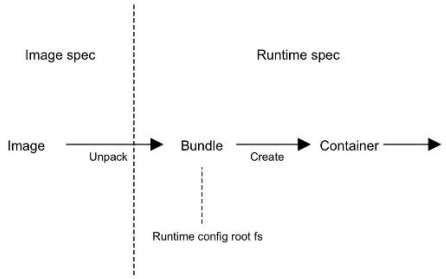

Figure 1. OCI image-spec and runtime-spec

2.1.1 Runc

Runc is a standardized runtime for spawing and running containers on Linux according to the OCI specifications. However, it does not follow the image-spec specification of the OCI. The higher level runtime interfaces (e.g. docker, containerd) implement those requirements on top of runc.

2.1.2 gVisor

Typical low-level container runtimes like *runc* limit system access using different capability models. If an adversary has access to a limited set of system calls and manages to exploit a vulnerability in the Linux kernel, then they may be able to escape the sandbox and gain privileged access. This is one of the disadvantages of using containers. This is especially a major risk when running untrusted in FaaS like services. A VM avoids this by having its own OS kernel but has significant performance overhead.

gVisor's approach to container isolation is in between this typical low-level container and of a fully virtualized machine. It uses UML like user-space kernel implementation - *Sentry*, to intercept application system calls, without the need for translation through virtualized hardware, see figure 2. This brings higher per system call overhead. To reduce the overhead, it does not virtualize the system hardware and leaves scheduling and memory to the host kernel. gVisor has runtime OCI *runtime-spec* – *runsc* and can be easily plugged into higher level container runtimes like *containerd* or *docker*, see figure 3.

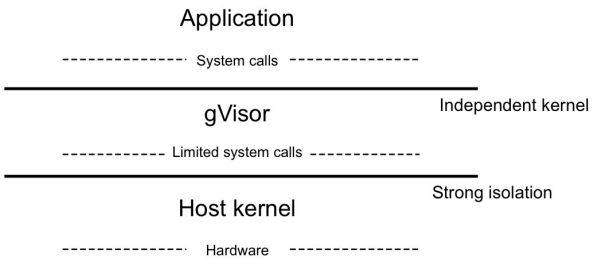

Figure 2. gVisor architecture.

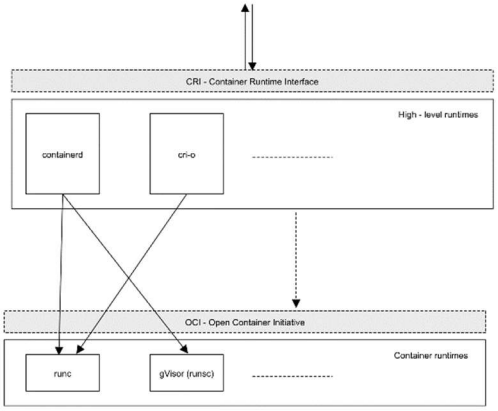

Figure 3. Container runtimes.

2.2 System Container Runtime – Incus

Application containers package a single process application. System containers simulate a full operating system like a virtual machine. System containers provide a full solution of libraries, applications, databases and so on. Incus is a system containers and virtual machine manager. It provides a way to run and manage full Linux systems inside containers and VMs. Incus is an extension of project LXD, which is based upon LXC – Linux containers, figure 4.

2.3 Firecracker microVM

Firecracker helps to deploy workloads in lightweight virtual machines, called microVMs. They provide complete workload isolation without compromising on the speed and resource

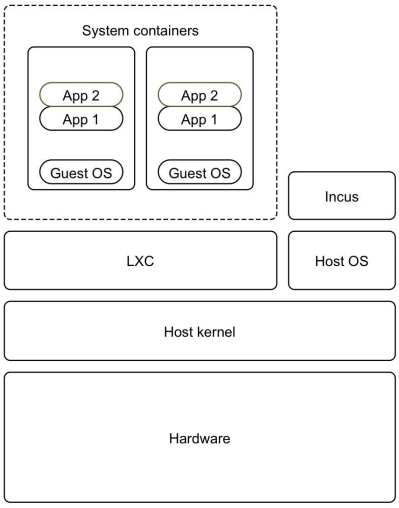

Figure 4. System containers.

efficiency. Firecracker runs in user space and uses Linux KVM to create MicroVMs. Each microVM runs a stripped down version of the Linux kernel with only necessary drivers loaded. It avoids unnecessary devices and guest functionality, which reduces memory footprint and the attach surface for each microVM, see figure 5.

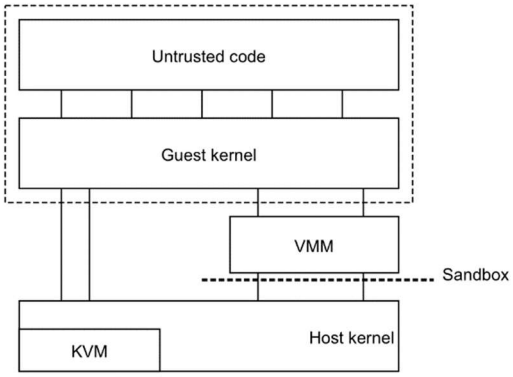

Figure 5. Firecracker architecture

3. RELATED WORK

Everarts et al. in [1] present a benchmarking tool to analyze the performance of container runtimes. Their focus lies on container startup time and scalability along with general performance characteristics of various container and OCI runtime configurations. The study reveals performance variations between container runtime environments, highlighting the need for careful configuration for optimal performance. Mieden et al. in [2] explore the performance of microVMs. They compare Firecracker and QEMU, two hypervisors supporting microVMs. Their findings demonstrate that Firecracker outperforms QEMU in terms of boot time and exhibits better scalability when running multiple microVMs concurrently. Lastly, Anjali et al. in [3] address the trade-off between isolation and performance in serverless computing platforms. They compare three virtualization approaches: Linux containers, gVisor, and Firecracker microVMs. The study concludes that while both Firecracker and gVisor achieve strong isolation, they incur some performance overhead due to their reliance on the host kernel. Linux containers offer the best performance but provide weaker isolation compared to the other two approaches.

4. TESTING AND RESULTS

4.1 ENVIRONMENT SETUP

A single Cloudlab [4] x86_64 bare metal machine with Ubuntu 22.04 as the host operating system has been used to perform all of the experiments.

Tool	Version
Docker	Docker version 24.0.5build 24.0.5-0ubuntu1~22.04.1
gVisor	version release-20240408.0 spec: 1.1.0-rc.1
Firecracker	v1.7.0
Incus	v6.0.0

Table 1: Tools installed for testing.

Bash scripts have been written to install and setup the tools, listed in table – 1, to produce consistent environments over successive benchmark runs. We use sysbench to evaluate general aspects of these virtualized systems.

4.2 PERFORMANCE TESTING

4.2.1 CPU

The benchmark measures the time needed to compute all prime numbers up to 20000. Figure 6 shows that the incus runtime had the lowest performance overhead and docker with runc runtime closely matches its performance.

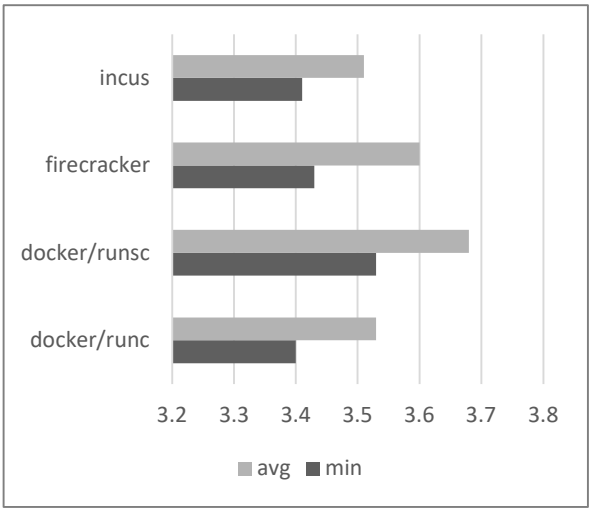

Figure 6. CPU benchmark: average and minimum latency in milliseconds

4.2.2 I/O

Figure 7 compares the sequential read throughput of *fileio* benchmark. We could see a large difference between runc and *runsc* container runtimes. This is due to the additional overhead of gVisor setup in runsc. Interestingly, firecracker despite being a virtual machine, its performance has been better than the any other tool.

In the case of write and rewrite benchmarks the results, as shown in figure 8, are slightly different. The default runc OCI runtime and incus performance has been the worst

compared to the firecracker or the secured containers (gVisor). Which is a surprising and yet interesting finding. The difference was expected but not so much.

The system containers in all the benchmarks have had poor throughput. This is quite the opposite of what we have inferred from the high-level understanding of the Linux containers. Linux containers are expected to perform better but as we can see they bring significant overhead. It would be interesting to benchmark the tools in system containers space – LXC, LXD and the most recent runtime, Incus.

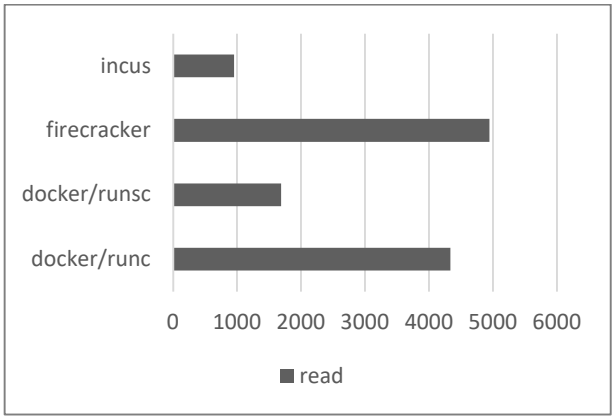

Figure 7. IO Benchmark: SeqRead throughput (MiB/s)

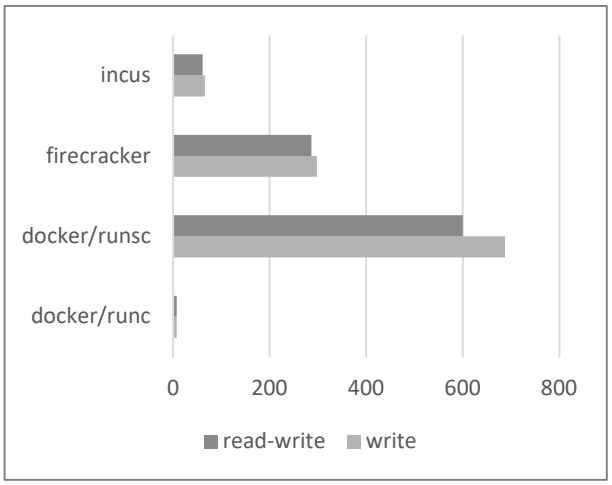

Figure 8. IO Benchmark: SeqWrite and SeqRewrite throughput (MiB/s)

4.2.3 MEMORY

Figure 9 shows the comparison of time spent in memory benchmark of *sysbench*. The Docker container with default runc OCI runtime clearly outperforms every other tool. Firecracker performance, with subtle difference, is apar with the docker container. Incus spent the most amount of time in the memory.

When looking at the average and minimum latency, Figure 10, we could see that runsc, firecracker and incus, introduce an overhead when accessing the memory. On average, docker containers perform better.

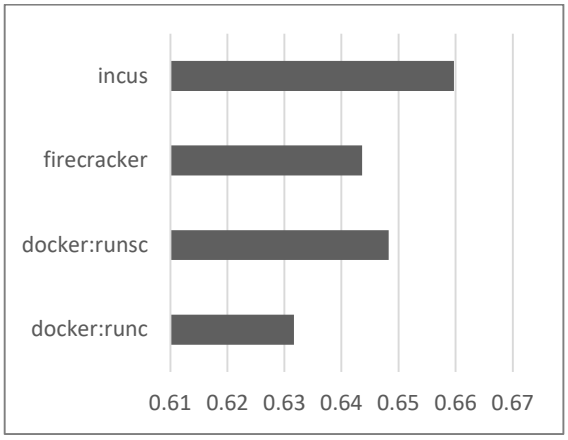

Figure 9. Memory benchmark: Total Time in seconds

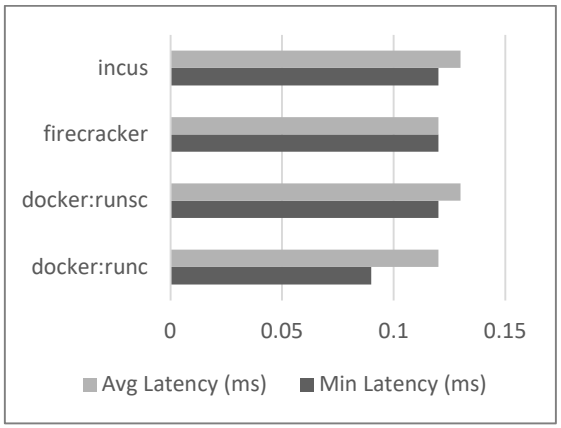

Figure 10. Memory Benchmark: Average and minimum latency in milliseconds

5. RESULTS

To summarize, docker container with default OCI runtime performed better due to lower overhead in CPU and Memory benchmarks. Firecracker comes second. IO performance benchmark has slightly unexpected results; when considering all – read, write and rewrite – firecracker does better. Docker container performed poorly in multiple write and rewrite benchmarks consistently.

Firecracker despite being a VM does not have significant overhead and its performance is reasonably and consistently better than the rest.

Incus, running a system container, suffers the most due to the additional overhead of running its own OS. Interestingly, runsc,

despite running also a user space kernel and providing a higher level of isolation compared to system container, performs better in all of the tests.

6. Conclusion

In this project I compared 4 different lightweight virtualization techniques (in their decreasing level of isolation features) – Firecracker, gVisor/runsc, Incus and docker/runc. Each provide different degree of isolation to the process and application running inside them. Enhanced isolation and security comes with additional performance overhead. This could be clearly seen by the results of the benchmarks. With this understanding it becomes important to carefully consider the requirements of the application and then choose the virtualization technology. Future work of this could include understanding the deployment aspects of these techniques. This could include benchmarking their startup and shutdown times, scaling capacity, ease of deployment in cluster mode and administration. We could further benchmark on the basis of workloads such as over the network communication, streaming and inter-communication.

References

[1] Espe, L., Jindal, A., Podolskiy, V., & Gerndt, M. (2020). Performance evaluation of container runtimes.

[2] Guillaume Everarts de Velp, Etienne Rivière, and Ramin Sadre. 2021. Understanding the performance of container execution environments.

[3] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending containers and virtual machines: a study of firecracker and gVisor.

[4] 2024. Cloudlab. https://www.cloudlab.us/.

[5] 2024. Firecrackr documentation. https://github.com/firecracker-microvm/firecracker/tree/main/docs.

[6] 2024. gVisor documentation. https://gvisor.dev/docs/.

[7] 2024. Docker documentation. https://docs.docker.com/.

[8] 2024. Incus documentation. https://linuxcontainers.org/incus/docs/main/.

[9] 2024. Open Container Initiative. https://www.opencontainers.org/.

[10] 2024. Sysbench Manual. https://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html.