Containers Runtimes War: A Comparative Study

Ramzi Debab(✉) and Walid Khaled Hidouci

Ecole Nationale Supérieure D'Informatique (ESI), Algiers, Algeria
{r_debab,hidouci}@esi.dz

Abstract. Docker has revolutionized the cloud by popularizing containerization, changing the habits of developers and their tools, introducing new paradigms such as microservices and serverless computing, facilitating CI/CD pipelines and packaging techniques, inspiring Infrastructure as Code (IaC) and pushing the Software Defined Everything (SDx. Docker is omnipresent, on our laptops, servers, storage arrays and even on IoT devices. Docker had also the merit of defining standards such as the OCI initiative. Despite all this, can we trust this technology and deploy it widely in our private or public clouds? Given the few vulnerabilities of Docker but badly impactful, many Cloud players (Google, Amazon, RedHat, IBM, Intel, VMware among others) have worked on containerized house technologies made public. Four types of releases resulted: containerization based on OS virtualization, containerization based on micro VMs, syscalls virtualization and finally the Unikernels approach. In this article we will try to briefly present the recent containerization technologies respecting the OCI standards with a comparative study of performance. The purpose of this paper is to provide elements that help in decision-making for the choice of the most suitable container engine.

Keywords: Docker · Containerization · IaC · OCI · CNI · OS virtualization · Hardware virtualization · Unikernels

1 Introduction

The containerization is an old OS virtualization technology that appeared since about 20 years [1]. It aims to execute applications and their dependencies in sandboxed environments. Docker [2–5] becomes increasingly popular and is the de facto best choice for many scenarios. For developers, it is so easy to instantiate in couple of seconds an isolated development environment based on one or multiple images from the Docker Hub [6, 7]. This technology had led to think about new paradigms like the microservices [8, 9] and serverless computing [10, 11]. The containers are the best solution in order to deploy a microservices infrastructure [12]. Most of Cloud actors are using containers to host serverless workloads for costs optimization purposes (serverless).

For IT administrators, Docker is seriously an excellent replacement to the classic Hardware Virtualization in many scenarios because the containers are much lighter and more performant than the virtual machines with better density [13].

K. Arai et al. (Eds.): FTC 2020, AISC 1289, pp. 135–161, 2021.
https://doi.org/10.1007/978-3-030-63089-8_9

VMware has recently released the new vSphere Integrated Containers Engine that aims to deliver a unified solution to manage heterogeneous workloads; containers, VMs and applications [14]. Docker which was developed initially in order to solve the applications packaging problems, has really filled the gap between the developers that release packages and IT admins that should install and maintain them [15].

After the standalone container, different technics of orchestration have appeared like Docker Compose, Swarm and Kubernetes. With a yaml file, it is now possible to define declaratively an infrastructure based on containers. This is a big step in IaC where automation becomes easier [16].

Today, Docker is widely used and is everywhere, even in IoT devices. But is it really the best choice in Cloud environments under multitenancy and security constraints?

In order to support other containers engines development, the standardization is a must. Docker has massively participated in founding open standards:

- OCI (Open Container Initiative) [17]: OCI standards define images and runtimes specifications. Docker Images are OCI images and Docker runc is an OCI runtime. Most of the vendors are supporting this initiative. Thus, any OCI orchestrator can manage any OCI container runtime and any OCI runtime ca execute any OCI image.
- CNI (Container Network Interface): This is a project of the CNCF (Cloud Native Computing Foundation) that defines network interfaces specifications and APIs for Linux containers [18].
- Kubernetes CRI (Container Runtime Initiative): Kubernetes becomes by far the most used orchestrator. In order to be orchestrated by Kubernetes, the runtime has to be CRI compliant. This initiative consists of protobuff and gRPC APIs in addition to other libraries and tools [19].

A. Martin et al. [20] have analyzed different vulnerabilities that impact Docker containers by covering the different layers; the host kernel, the Docker engine, the content of images, etc. In Cloud environments where the tenants' workloads isolation and their security are an important concern, thinking about another container engine becomes mandatory. For this reason, different Cloud players have developed their own runtimes to overcome the different shortcomings. The resulted runtimes can be classified as follows (Table 1):

Table 1. Current containers runtimes with enhanced security

Solution	Examples
Micro VMs	Kata Containers (Intel, OpenStack Foundation) [21], Pacific Project (VMware) [14], Hyper-V Containers [22] & WSL2 (Microsoft) [24], Firecracker [23] (Amazon)
Syscalls Virtualization	WSL1 (Microsoft) [25], gVisor (Google) [23]
Library OS (Unikernels)	Nabla (IBM) [26]

In this paper, we will focus on OCI runtimes in order to support images and containers interoperability. To choose the most suitable containers runtime, a desirability function can be defined and based on some inputs like performance and security posture like in [29].

The rest of this paper is structured as follows: We start with a brief motivation for this comparative study (Sect. 2). We then present the Linux containers kernel background and the OCI runtimes that will be subjects of the study (Sect. 3). Following this, we present the benchmarking methodologies to assess different metrics incurred by the selected runtimes (Sect. 4). This is followed by a presentation and discussion of the experiments' results that have been performed (Sect. 5). We conclude the paper with final remarks and identify directions for future work (Sect. 6).

2 Motivation

Ramzi, D. et al. have presented a solution named MetaOS [27] and based on a custom orchestrator to select at any given time the most performant kernel to execute Docker domains containers. The principle was based on using heterogenous kernels to increase the resiliency of the whole platform. Thus, when detecting a failure or a threat, the system migrates automatically to a less performant kernel to mitigate the previous shortcomings and ensure the continuity. The system is consequently more resilient and robust.

We can improve the resiliency of the system by using more secured OCI runtimes. In addition to containers migration at the host level (containers migration [28] is just supported between the same containers engines and runtimes), we can use the workloads migration at the application level between different OCI runtimes. This is feasible because the OCI runtimes can leverage the same images.

The following figure simulates an application level migration scenario from Docker & runc selected at the first time as the most performant runtime to Kata Containers engine that is less performant because of the overhead of the hardware virtualization but more secure.

During our tests, we have succeeded to migrate simple applications from Docker to Kata using the DMTCP tool (Fig. 1):

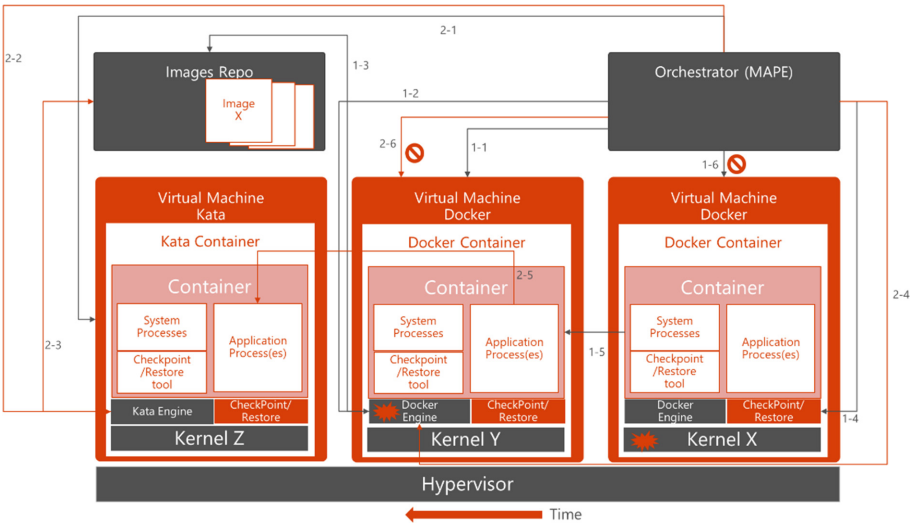

Fig. 1. Towards a more Robust heterogeneous MetaOS based on different Kernels and containers runtimes

The following table (Table 2) describes the steps applied by the orchestrator foreach scenario:

Table 2. MetaOS resiliency scenarios

Container migration scenario (Failure in Kernel X)	App migration scenario (Failure in Docker Engine)
1–1 Instantiate a new VM based on a different Kernel version	2–1 Instantiate a new VM based on a different runtime (Kata)
1–2 Invoke Docker Engine to download the App Image from the registry corresponding to the original container	2–2 Invoke Kata Engine to download the App Image from the registry corresponding to the original container
1–3 Download the App Image form the registry and run the container	2–3 Download the App Image form the registry and run the container
1–4 Invoke the Checkpoint/Restore tool to initiate the migration of the container from the source to the destination.	2–4 Invoke the Checkpoint/Restore tool to initiate the migration of the application from the source to the destination container.
1–5 Migrate the container	2–5 Migrate the application
1–6 Deallocate the VM to free resources	2–6 Deallocate the VM to free resources

3 Background and Related Work

In this section we will review some contemporary containers runtimes solutions with emphasis on the OCI runtimes.

3.1 Linux Kernel for Containers

Starting from the 2.6.24 Linux kernel release, containers are supported natively with the introduction of LXC in 2008 [1]. The following features were integrated in the kernel to support containers [1]:

- Namespaces: they were introduced for the first time by [30]. A namespace is a technique that provides isolation and abstraction of a given sandboxed container regarding other containers and the host. The following table (Table 3) depicts the most used namespaces in Linux:

Table 3. Linux Namespaces

Namespace	Isolation scope
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
User	Users and Groups IDs
UTS	Hostnames and Domain names
CGroup	CGroup directory

- CGroups (Control Groups): they were introduced by Google in 2007 in order to control resources (CPU, RAM, Disk IOs, Network IOs, etc.) consumption of processes' groups.
- Rootfs: they offer necessary files to the containers. They are based on the COW (Copy On Write) feature to create a writable layer based on a shared Read Only Image layer. Thus, the storage is globally optimized with small containers footprints.

3.2 Docker

The Docker project is by far the most well-known containers engine since its release in 2013. Docker is not only a simple engine or runtime, but it is a complete ecosystem [20]:

- Docker Engine [2–4, 6, 7]: It offers a CLI (client interface) that interacts with the **Docker Daemon** using Linux sockets and the Rest API. The Daemon plays the role of the gateway and forwards all Rest API calls into gRPC calls to the **containerd** daemon. Foreach container creation call, the **containerd** daemon instantiates a shim that makes a call to the **runc** runtime in order to create the container. The shims are the contact points between the containers and the **containerd**. The **runc** runtime is an OCI runtime. The **containerd** abstracts all the kernel features (Namespaces, CGroups and Rootfs) calls. The **containerd** specification is managed by the Cloud Native Computing Foundation (CNCF).

- Docker Images [2–4, 6, 7]: A Docker image is a RO bunch of layers. Each layer corresponds to a file system containing files. Using the COW feature, multiple containers can share the same image with just RW layers for each of them (Fig. 2). Docker Images can be pulled from Docker Hub or from a local registry. Most of OCI runtimes can use Docker Images because they are also OCI compliant. Dockerfile is file with certain syntax that is used to generate images. It is also possible to commit a running Docker container to generate an image.
- Docker Swarm [31]: Docker Swarm is inhouse orchestrator of Docker containers in a cluster of Docker nodes. It is much simple to implement and maintain. Swarm allows creation of services and stacks by ensuring advanced services like service discovery, overlay and ingress networks.
- Docker Security [20]: As any solution, Docker presents some vulnerabilities. It is often criticized for its daemon that runs in root mode. By default, Docker containers use native kernel namespaces and capabilities for their isolation. However, a container can share some namespaces with the host which gives the full control on the host resources. The–**privileged** mode gives the container full access on host devices and resources. The CGroups are not limited by default and the limits have to be explicitly defined. Sharing the same bridge network allows ARP poisoning attacks between containers for example. The Docker images can contain some vulnerabilities. A continual check is a must by checking the current CVEs. For this reason, Docker EE is equipped with a workflow that checks the image before it is used or copied to the on prem registries. In a misconfigured environment, it is possible to trust insecure registries by disabling or ignoring certificates when accessing to registries. Thus, we can define three classes of threats and security vulnerabilities [32, 33]:
  - Container escaping: that is due to containers misconfigurations, shared namespaces and excessive capabilities.
  - Inner-container attacks: that result from malicious code and vulnerabilities in images
  - Cross-container attacks: that result from network defaults, CGroups defaults and excessive capabilities.

  In order to overcome the Docker Vulnerabilities some solutions are proposed:

- Using light kernels tailored for containers like RancherOS [20].
- Auditing Docker platforms using the CIS Docker Security Benchmark [51] to determine factors that affect the security.
- Hardening the host's kernel using security modules like SELinux, Seccomp, Apparmor and TOMOYO by defining custom profiles. **docker-default** Apparmor profile for example permits full access to the host file system [34].
- Applying Least Privilege Principle when running docker operations instead of using the root account.
- Customizing default namespaces and CGroups for better isolation.
- Avoid giving privileges to containers and sharing resources between containers and with the host.
- Continuous Images scan for eventual vulnerabilities and trusting just protected registries.

- Using the Hardware Virtualization to enforce the isolation like Kata Containers [21], Hyper-V Containers [22], WSL2 [24] and Firecracker [23].

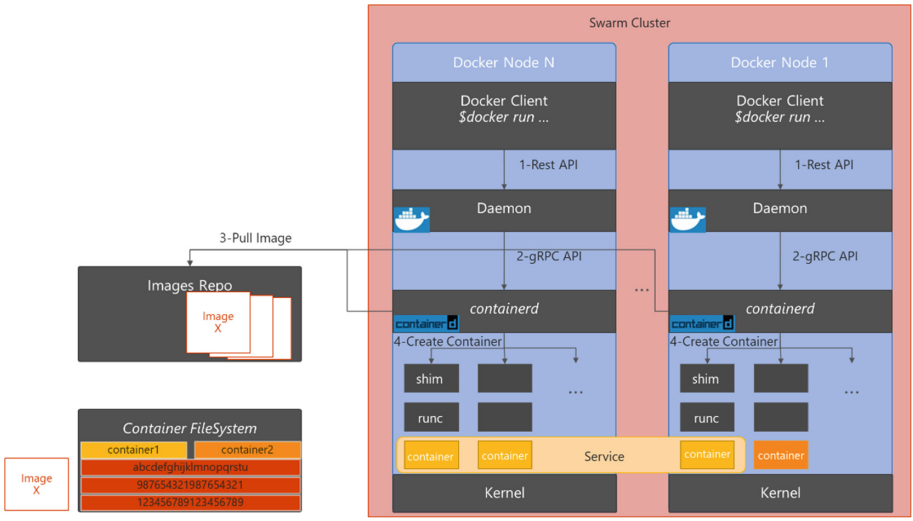

Fig. 2. Docker ecosystem

3.3 Micro VM Based Solutions

Docker is an awesome technology that is more suitable for Dev or test environments where the security is not a big concern. In production and to mitigate the weak isolation of Docker, some solutions based on Hardware Virtualization are proposed:

3.3.1 WSL2

Windows Subsystem for Linux in its second version (WSL2) (Fig. 3) is based on lightweight VMs that run on Hyper-V in order to cope with the weak containers' isolation. WSL2 is faster than WSL1 (syscalls translation), it allows native Linux binaries execution and integrates Linux Kernel with Windows using the 9P protocol [35]. The WSL.exe process launches lightweight Linux VM using the Host Compute Service (1–1, 1–2, 1–3). The Linux VM mounts the Windows Filesystem using the 9p protocol with the 9p File server in the VM worker process (1–5). The windows explorer can visualize the Linux file system following the same protocol (2–1). Docker is currently supporting the WSL2 [36] where the Windows Docker client can communicate with the Linux Docker Engine to leverage all containers operations.

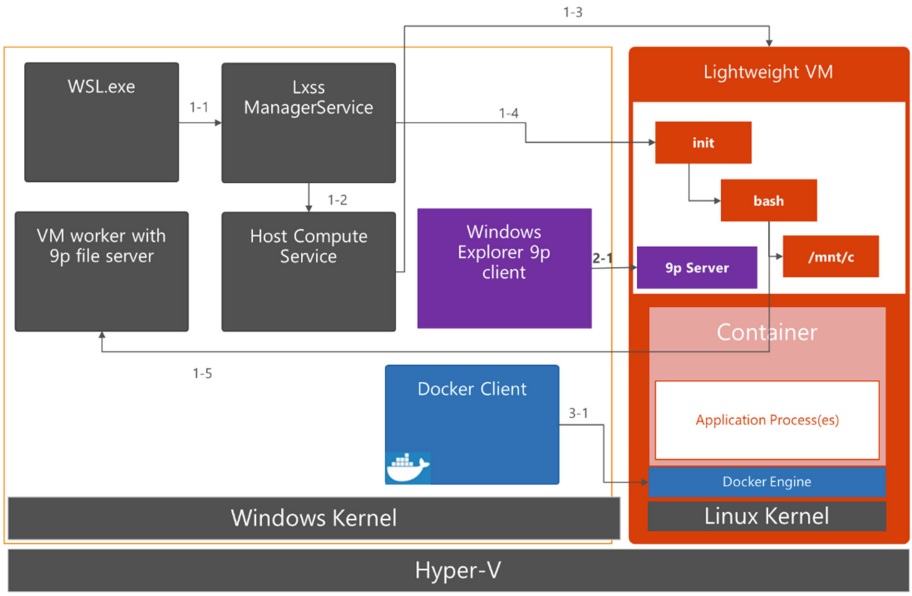

Fig. 3. WSL2 with Docker architecture

3.3.2 Firecracker

Amazon Firecracker is a novel technology that aims to harden the containers isolation by executing workload inside micro-VMs [23]. Today AWS Lambda and AWS Fargate services are both running on Firecracker [37, 38]. Firecracker is basically a VMM based on KVM/QEMU hypervisor to provide a small subset of devices and features to the Guest OS which makes the micro-VMs so lightweight with small memory and disk footprints and boot up more quickly (<=125 ms) [39] than traditional VMs in addition to the low surface attack (Fig. 4). The Firecracker workloads performance are so near to the native ones.

In terms of security, Firecracker VMM is shielded as it allows just 36 syscalls to the kernel host using a custom Seccomp profile. Each micro-VM is a process that runs in user space with attached VirtIO/bloc and VirtIO/net emulated devices. The Firecracker process is initiated by a **jailer process** that defines restrictions (namespaces, CGroups, etc.) and is launched after that in unprivileged mode. Thus, Firecracker offers a double barrier solution: Jailer barrier and Hardware Virtualization barrier.

Firecracker can be considered as a very promising technology because it supports executing kata containers, linux guests, OSv guests and integrating **containerd** via the **firecracker-containerd** engine. This latter allows integration with Docker and can run classical Docker containers via the runc runtime. Thus, Firecracker is considered as an OCI compatible engine.

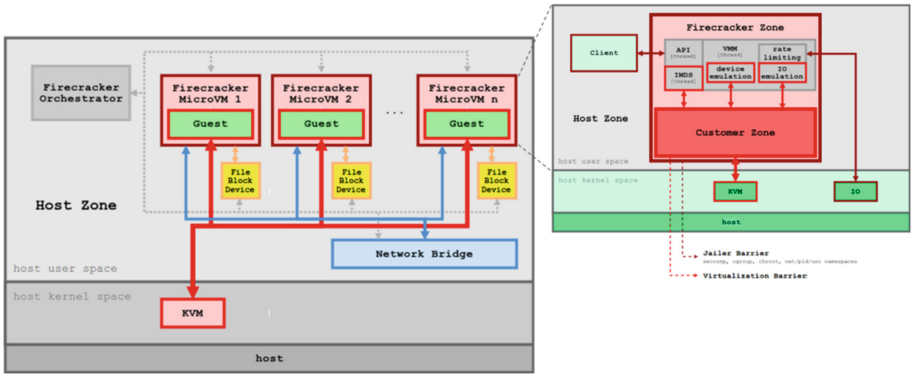

Fig. 4. Firecracker architecture (from [38])

3.3.3 Kata Containers

Kata Containers [21] is another attractive technology based on micro-VMs principle. It is originated from the Clear Containers project of Intel launched in 2015. Clear Containers were based on Intel VT hardware virtualization technology and a customized KVM/QEMU hypervisor. Two years after, Hyper Runv, another OCI hypervisor comes to replace the custom KVM/QEMU in Clear Containers to initiate the new Kata Containers project managed by the Openstack Foundation (OSF).

Kata Containers solution offers the following advantages [40]:

- Security: Runs in a dedicated kernel, providing isolation of network, I/O and memory and can utilize hardware-enforced isolation with virtualization VT extensions like Direct Device Assignment, SRIOV, etc.
- Compatibility: Supports industry standards including OCI container format, Kubernetes CRI interface, as well as legacy virtualization technologies including (KVM/QEMU, Hyper Runv, NEMU, etc.)
- Simplicity [41]: The containers are executed inside lightweight VMs using the kata-runtime That is OCI and CRI compliant. The kata-runtime launches foreach container or pod (Kubernetes) instance a kata-shim that communicates with the kata-agent in the micro-VM using a VIRTIO serial or VSOCK interface which QEMU exposes as a socket file on the host. The kata-agent executes all operations related to containers and pods lifecycles. With kata-shimv2 it is possible to avoid the kata-proxy that plays the role of multiplexor between the kata-agent and the multiple kata-shims that correspond to the containers of a single pod (Fig. 5).
- Performance: Kata Containers solution offers excellent performance thanks to the micro-Vms.

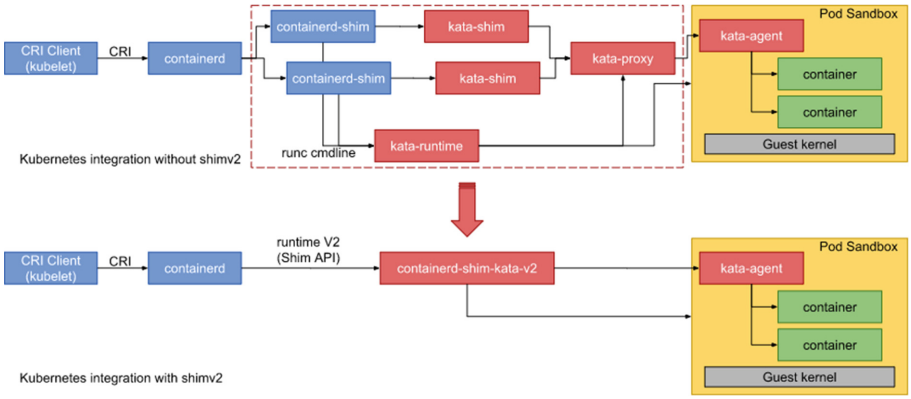

Fig. 5. Kata containers architecture (from [41])

3.4 Syscalls Virtualization

The syscalls Virtualization is a technique that translates the syscalls from a sandboxed environment based on a different kernel to those of the host like WSL1 [25] or intercepts syscalls from the application and executes them by a sandboxed Kernel in User Space like gVisor [23].

WSL1 aimed to translate Linux syscalls into Windows ones using a custom interpreter Kernel Module. While, everyone thought that this is a great idea as Linux and Windows are both POSIX OSs, we discovered that some Linux syscalls don't have equivalent implementations in Windows. The project is nowadays deprecated and replaced by WSL2 [24].

In this section we will just present gVisor as it is more mature and OCI compliant.

3.4.1 gVisor

gVisor [23] from Google is the sandbox technology that is used to host Google

Computing Platform's (GPC) App Engine, Cloud Functions, and CloudML. It is based on two main components (Fig. 6):

- Sentry kernel: it is a User Space Kernel that is written in Go (which is more secure due to type safety and memory management features in Golang). of 347 syscalls, 251 syscalls have a full or partial implementation. There are currently 96 unsupported syscalls [42] with just 53 host syscalls [23]. Sentry intercepts forwarded syscalls from ptrace mode or kvm mode and executes them in a container with a Seccomp profile limiting generating syscalls to the host. Netstack which is a User Space network stack is implemented inside Sentry to handle network communications.
- Gopher: it is a File server implementing 9p protocol to handle filesystem operations from containers. The Gopher process is instantiated for each gVisor container.

gVisor is OCI compliant as it relies on **runsc** runtime that is similar to runc in its implementation. Thus, Docker can be configured to run gVisor via **runsc**.

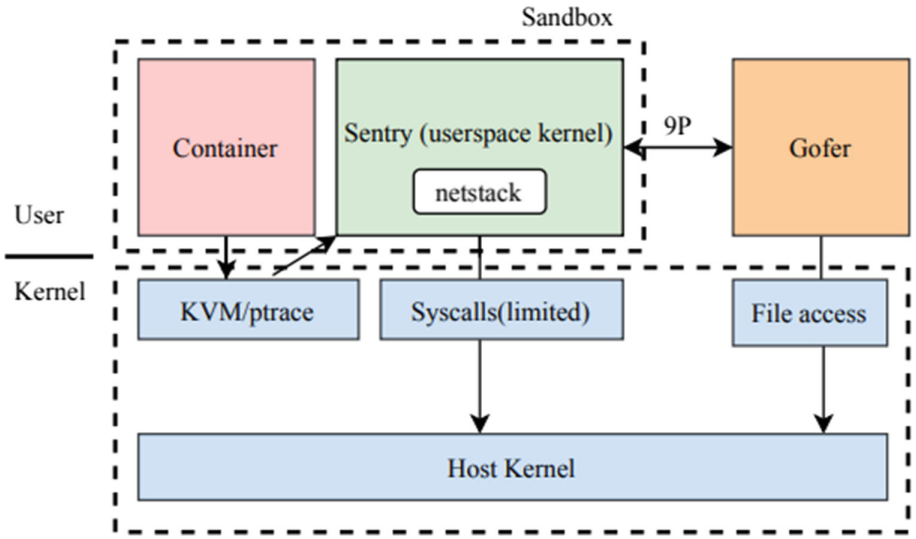

Fig. 6. gVisor architecture (from [23])

3.5 Unikernels Based Solutions

In this part we will present another different approach based on Unikernels [42] solution which aims to package the application and the application-dependent kernel functions into a single image. This will avoid making many syscalls to the kernel, limit the surface attack, and reduce the resources footprints compared to traditional VMs.

IBM Nabla [26] is a novel technology based on Unikernels architecture.

3.5.1 IBM Nabla

IBM Nabla is based on a custom VMM named Nabla Tender to manage lightweight VMs executing Unikernels. The Nabla Tender intercepts hypercalls (storage, network) from Unikernels VMs and translate them into syscalls to the host. The Seccomp mode avoid executing additional syscalls from the Nabla Tender.

Three main components define the Nabla architecture [43] (Fig. 7):

- Rumpkernel: It is the first Unikernel used by Nabla. Rumprun is a Unikernel Framework that facilitates developing LibOS with custom device drivers and hypercalls. Rumprun exposes a POSIX interface to integrate with POSIX apps during compiling operation.
- Solo5: It aims to facilitate the port of libOS/unikernel frameworks on various hardware platforms, i.e. a unikernel that runs on top of Solo5, runs on top of all the hardware targets.
- Runnc: It is an OCI runtime that aims to make Nabla pluggable in any OCI engine like Docker. However, Nabla does not support OCI compliant images yet.

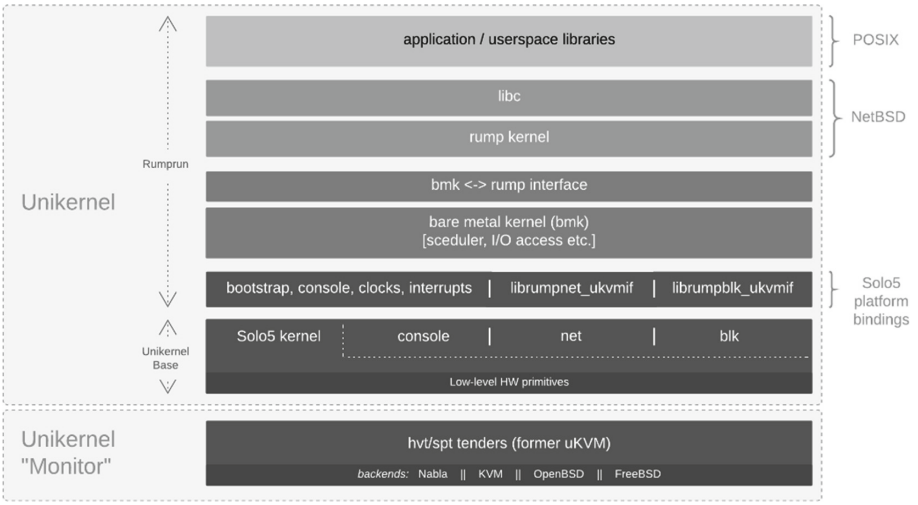

Fig. 7. Nabla architecture (from [43])

3.6 Related Work

In the literature, we can find some studies interested in comparing containers runtimes and engines performance.

In [13], the authors provide a deep evaluation with several benchmarking applications on KVM, LXC, Docker, and OSv.

In [44], performance tests using different benchmarks (CPU, RAM, Disk, Network IOs) were performed on Docker, LXC and native environments.

In [45], Y-cruncher and Bonnie ++ benchmarks were applied to measure CPU and Disk performance respectively on Docker and Rkt.

In [23], the authors looked at the code coverage and performance of CPU, networking, memory management, and file access of both gVisor and Firecracker.

To our knowledge, there is no recent and complete comparative study covering contemporary solutions.

The purpose of this paper is to deliver a more complete comparative study that covers main containers runtimes.

4 Methodology

In this paper, we are interested in OCI compliant runtimes and engines. The selected engines are as follows: Docker, Podman, Rkt, WSL2, Firecracker and gVisor. Podman is a like Docker tool and executes containers in daemonless way. Podman [47] from Redhat aims to replace Docker by offering the same experience and commands with additional features like checkpoint/restore.

Rkt is discontinued today [46] but we keep it for tests because it was a real concurrent to Docker in terms of performance. Rkt is not OCI compliant but can execute Docker images. Rkt source code is open on github and could be used in the future as a basis for a new engine.

IBM Nabla was not included in the tests because of not supporting OCI images; we have to build custom images to execute Nabla containers. However, we confirm the **runnc** compliance as we could configure it as a default runtime of Docker.

In order to measure the overhead of the strong isolation with the hardware virtualization, we have applied different benchmarks to calculate performance data related to CPU, RAM, Disk IOs and Network IOs.

All tests were performed on Azure VMs from the same template and located in the same datacenter. The configuration of the VMs is as follows (Table 4):

Table 4. Azure VM Config

Setting	Value
Size	D2_v3 (nested virtualization supported for microVMs based solutions)
Region	France
vCPU	2
RAM	8Go
Disk	Standard HDD, Max Throughput: 500MiB/s, Max IOPS : 2000

The containers engines and runtimes versions are listed in the following in Table 5:

Table 5. Azure VM Config

Engine	Version	OS
Podman	1.6.2	Ubuntu 18.04
Rkt	1.29.0	Ubuntu 18.04
Docker18	19.03.8 containerd 1.3.3 runc 1.0.0-rc10	Ubuntu 18.04
Docker20	19.03.8 containerd 1.3.3 runc 1.0.0-rc10	Ubuntu 20.04
WSL2		Windows 10/Ubuntu 20.04
gVisor		
FireCracker	0.18.1 containerd 1.3.3 runc 1.0.0-rc10	Ubuntu 18.04
Kata Containers	Kata-runtime 1.11.0-rc0 containerd 1.3.3	Ubuntu 18.04

5 Results

In this section we present the results of our analysis. As explained previously, the selected benchmarks measure CPU, Memory, Disk IOs and Network IOs performance.

5.1 CPU Performance

Y-cruncher [48] is generally used to calculate the PI value and is considered as a good tool to stress the CPU and test its performance. Y-cruncher has the advantage to be multithreaded and execute the generated threads on the cores.

Due to RAM limit, we calculated PI value with 500.000 digits. Foreach engine, the benchmark was launched over 10 times. The average computation and total times values are shown in the following table (Table 6) and figure (Fig. 8):

Table 6. Y-Cruncher average values

Container Engine	Average of computation time (min)	Average of total time (min)
Podman	4.6532	4.8111
Docker20	4.6833	4.8433
Rkt	4.82	4.9835
Kata Containers	5.0001	5.3054
gVisor	5.0237	5.1892
Docker18	5.074	5.246
Firecracker	5.1726	5.4014
WSL2	5.5852	6.0235

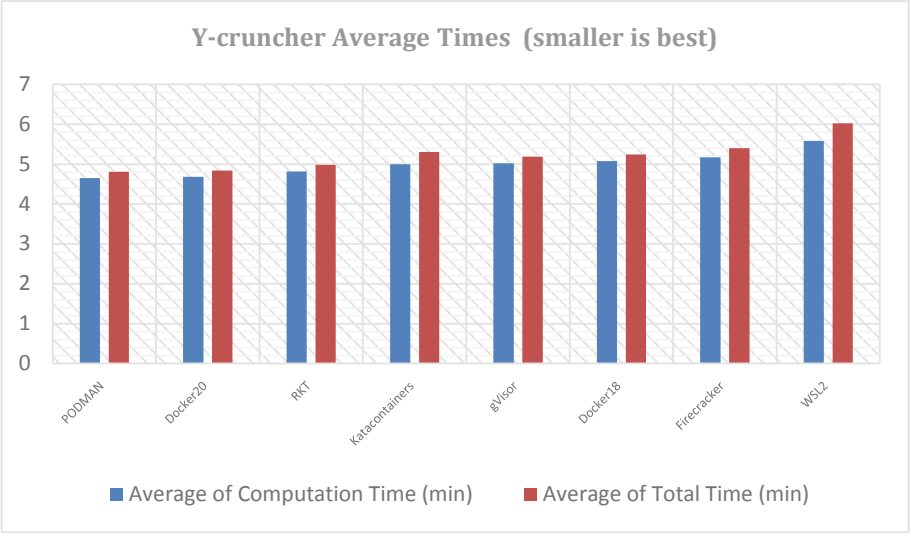

Fig. 8. Y-cruncher results

With no surprise, we can notice that the native engines (Podman, Docker and Rkt) perform better with a good score for Podman. Kata Containers, gVisor and FireCracker display performance almost like the native engines. The overhead is due to Hardware Virtualization and the nested virtualization as KVM/QEMU are nested in VMs. WSL2 needs more improvements because the WSL2 micro-VMs are heavier than other concurrent micro-VMs.

The Multi Core and CPU efficiency results are presented in Table 7. Podman and Rkt perform better than other solutions. gVisor was unable to obtain CPU information. Perhaps this is due to syscalls limitations.

Table 7. Y-Crunch multi core efficiency

Container engine	Average CPU utilization (%)	Average mutlti-core efficiency (%)
Podman	198.087	99.043
Rkt	198.035	99.018
Kata Containers	197.757	98.878
Docker 20	197.705	108.739
FireCracker	196.189	98.094
WSL2	195.938	97.969
Docker 18	192.916	96.458
gVisor	??	??

5.2 Disk I/O Performance

Bonnie ++ [49] is a small utility with the purpose of benchmarking file system IO performance. It performs different operations on files: Sequential outputs (Per Chars, Blocks), Inputs (Per Chars, Blocks), Random Seeks, Sequential Creates, Sequential Reads. Foreach metric, the tool measures throughputs, CPU usage and latencies.

Table 8. Bonnie sequential operations results

Engine	Sequential output block throughput (MB/s)	Sequential output char throughput (MB/s)	Sequential input block throughput (MB/s)	Sequential input char throughput (MB/s)
Docker 18	24	447	3913	1303
Docker 20	24	446	3612	1231
WSL2	25	385	2670	1043
Podman	24	418	2898	1055
Rkt	24	425	3598	1158
gVisor	368	74	545	78
Kata Containers	21	19	97	23
Firecracker	73.5	1.2	347.5	1.7

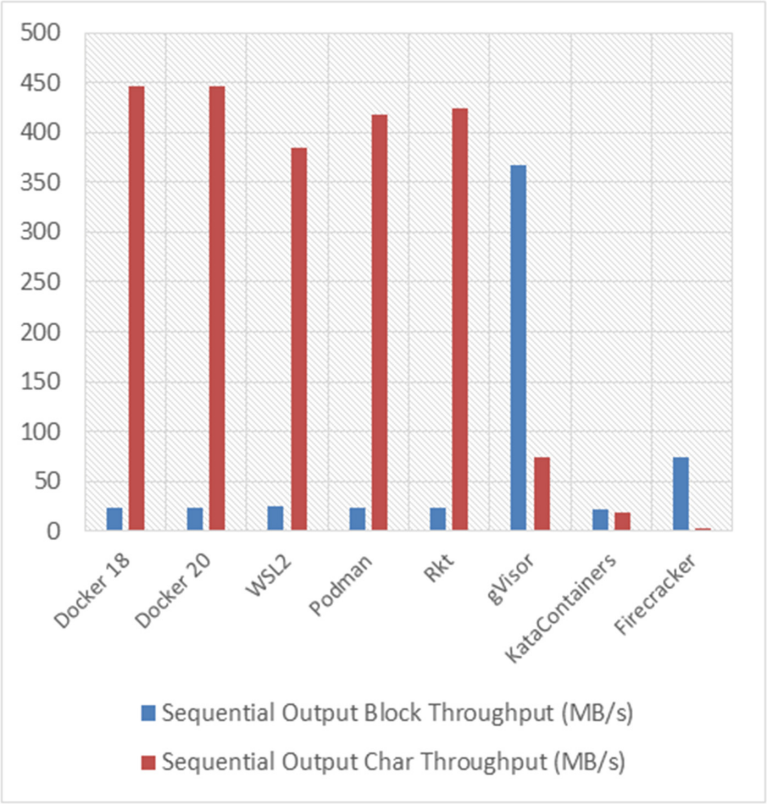

Fig. 9. Bonnie sequential output

We have performed the same tests on all VMs using these parameters: s (2048 MB) for the file size to handle and r (1024 MB) for the used RAM.

Table 8 shows the results related to Sequential Read/Write operations using two methods: block and per character. The same results are summarized in Fig. 9 and Fig. 10.

When analyzing the results, we can be surprised by gVisor that outperformed all of the concurrent engines in the case of Sequential Output Block. We have applied different parameters like disabling the system cache and using a small amount of memory to handle big files. The results remain the same. This is because, perhaps, of a buffer system handled by the Sentry Kernel and the Gopher file system.

Firecracker outperformed other engines in the Sequential Output Block scenario (except of gVisor) because it does on rely on union file systems; all files operations are performed by an emulated device that is backed directly by a file on the host.

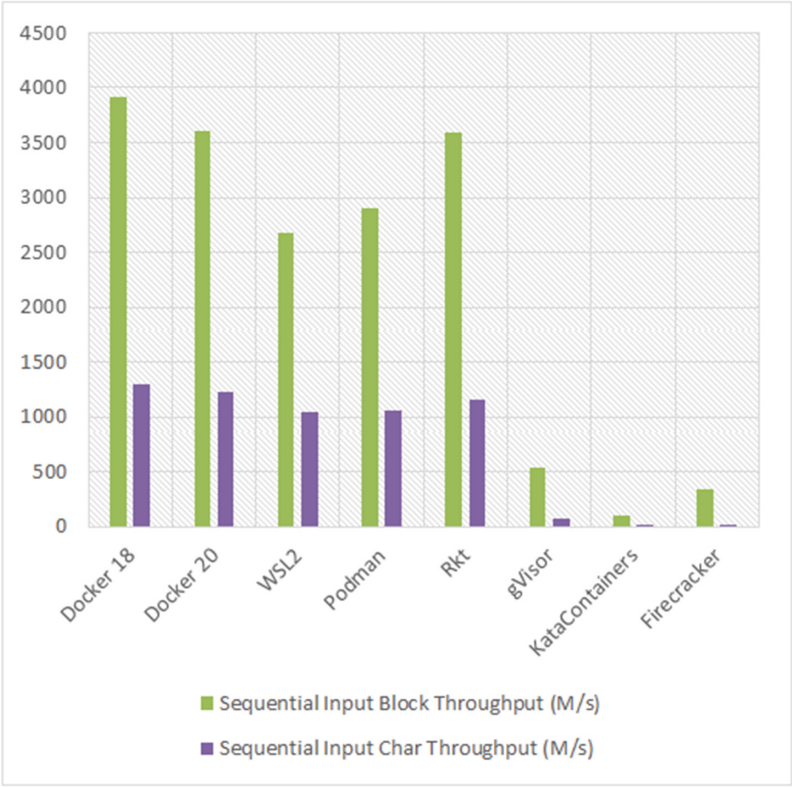

Fig. 10. Bonnie sequential input

Another test was performed using Y-cruncher to analyze I/O performance by applying the settings in Table 9:

Table 9. Y-Cruncher I/O Parameters

Setting	Value
Memory	2 GB
Swap file Size	16 GB
Multithreading	Enabled with randomization
Min I/O size	256 KB

During the tests, sequential reads/writes, strided reads/writes and streaming files from and to disk are performed. Table 10 and Fig. 11 to 14 depict results related to sequential IO operations throughput, strided IO operations throughputs and streaming disk IO with ratio (Disk/CPU). Throughputs are all in MB/s:

Table 10. Bonnie Sequential Operations Results

Engine	Sequential write	Sequential read	Threshold strided write	Threshold strided read	Streaming computaion	Streaming disk I/O	Ratio
Docker 20	25.8	51.6	25.8	51.6	618	38.7	0.062625
Docker 18	25.8	51.5	25.7	51.5	665	38.6	0.0581172
gVisor	17	51.6	14	234	403	50	0.124293
Podman	25.8	51.6	25.8	51.6	496	38.7	0.0779373
Rkt	25.8	51.6	25.8	51.6	653	38.7	0.059275
Kata Containers	15.3	20.2	22.5	19.3	385	52.5	0.136159
firecracker	17.8	36.1	19.9	34.3	503	26.9	0.0534311
WSL2	12.7	51.6	10.1	51.3	532	71.4	0.134208

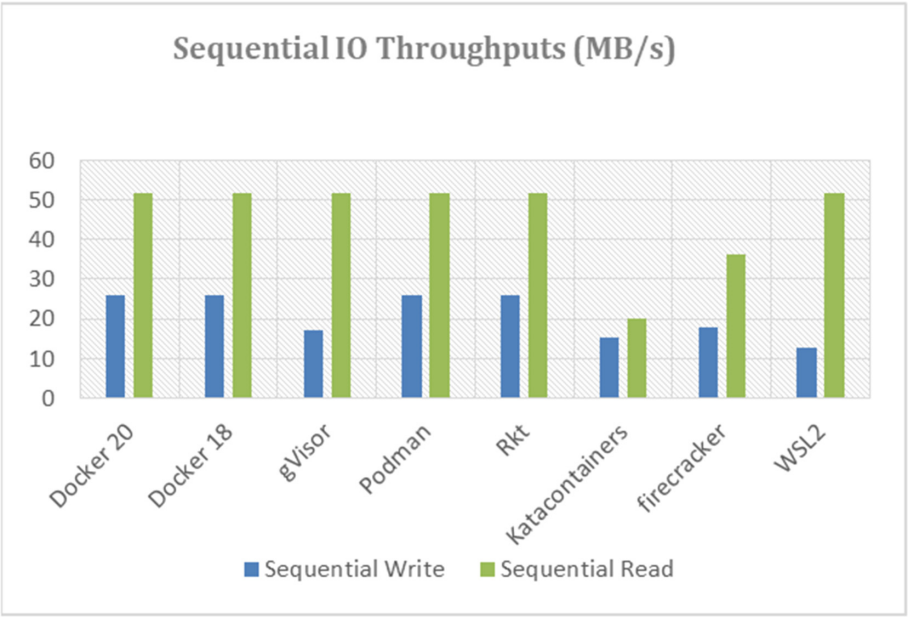

Fig. 11. y-cruncher sequential IOs

According to these results, we can see easily that native engines (docker, podman and rkt) perform better than Kata Containers, WSL2 and Firecracker. A small exception for gVisor which outperforms in the strided read IOs scenario. gVisor, in general, has good results while handling syscalls in user space. The small overhead is related to the syscalls forwards from the kernel.

WSL2 records good results in Read operations with streaming IOs while Kata Containers is performing timid scores.

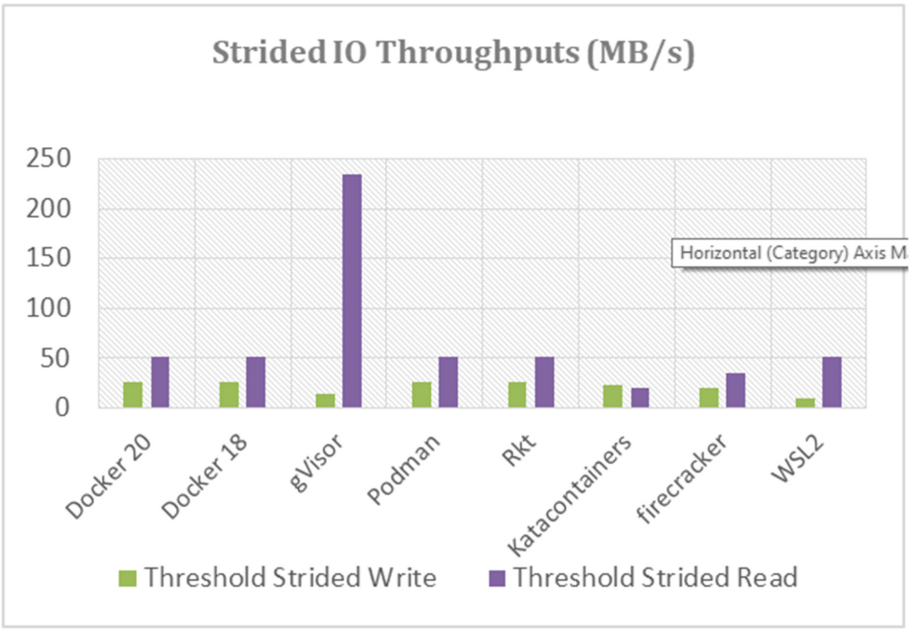

Fig. 12. y-cruncher strided IOs

5.3 Memory Performance

As suggested by [13], we use STREAM [50] benchmark to test the memory performance. STREAM software measures memory performance using vector kernels operations. It produces results for four different operations: Copy, Scale, Add, and Triad.

Table 11, Fig. 15 and Fig. 16 depict results related to the throughputs of the different operations and corresponding average times.

When analyzing these results, we can see that almost engines perform similarly. Podman and gVisor record the slowest times. However, Kata Containers were performing the best throughputs and the shortest times.

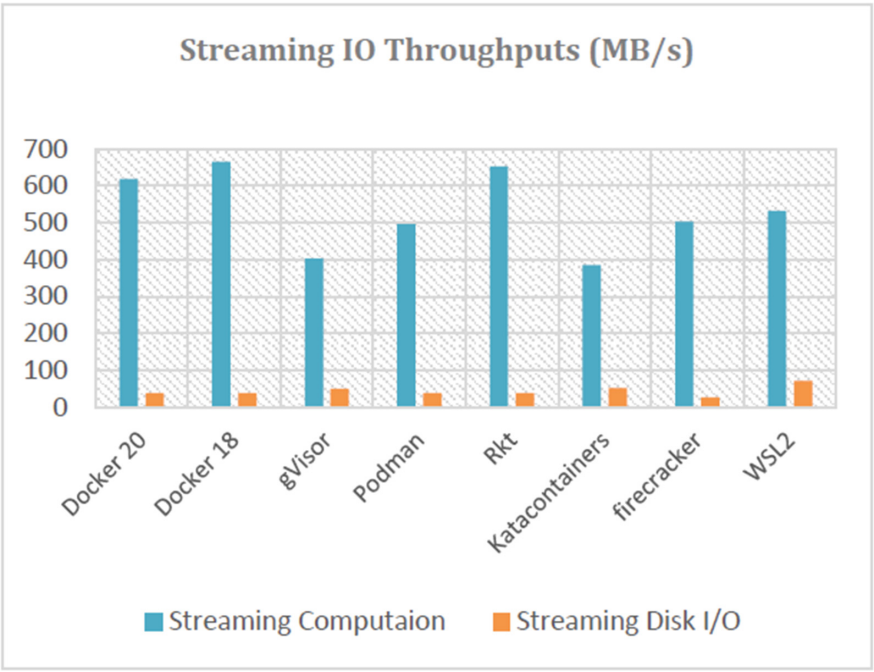

Fig. 13. y-cruncher streaming IOs

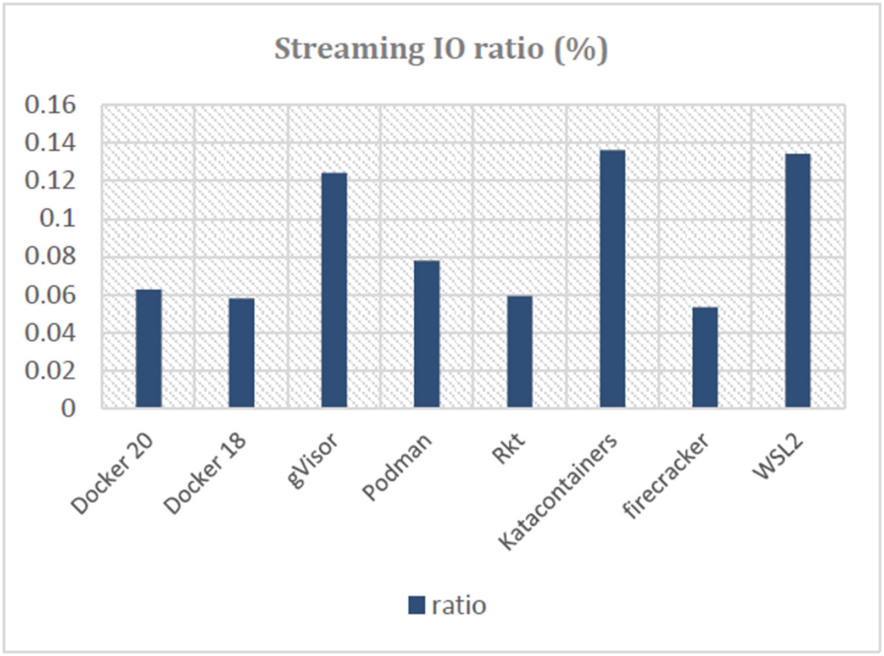

Fig. 14. y-cruncher streaming ratio

Table 11. STREAM operations results

Engine	Copy best rate (MB/s)	Copy Avg time (s)	Scale best rate (MB/s)	Scale Avg time (s)	Add best rate (MB/s)	Add Avg time (s)	Triad best rate (MB/s)	Triad Avg time (s)
Docker 20	8921.6	0.445063	11400.8	0.347658	12095.2	0.4913836	12116.8	0.489696
Docker 18	9172.4	0.403607	11272.4	0.327293	12270.4	0.455376	12206.3	0.458414
gVisor	14304.3	0.285993	8125.2	0.506526	8760.4	0.701291	8890.7	0.693913
Podman	13286.1	0.326353	7806.2	0.558741	8395.2	0.778691	8332.1	0.768194
Rkt	9117.8	0.442803	11193.2	0.358651	12108	0.499062	12114.7	0.50304
Kata Containers	9453.1	0.138571	12403.9	0.106128	12796.3	0.153434	12701.9	0.154853
firecracker	9098.1	0.420361	10332.9	0.377862	11719.8	0.500446	11707.4	0.490449
WSL2	8392.5	0.464674	10705.3	0.371615	11127.4	0.501817	11405	0.538316

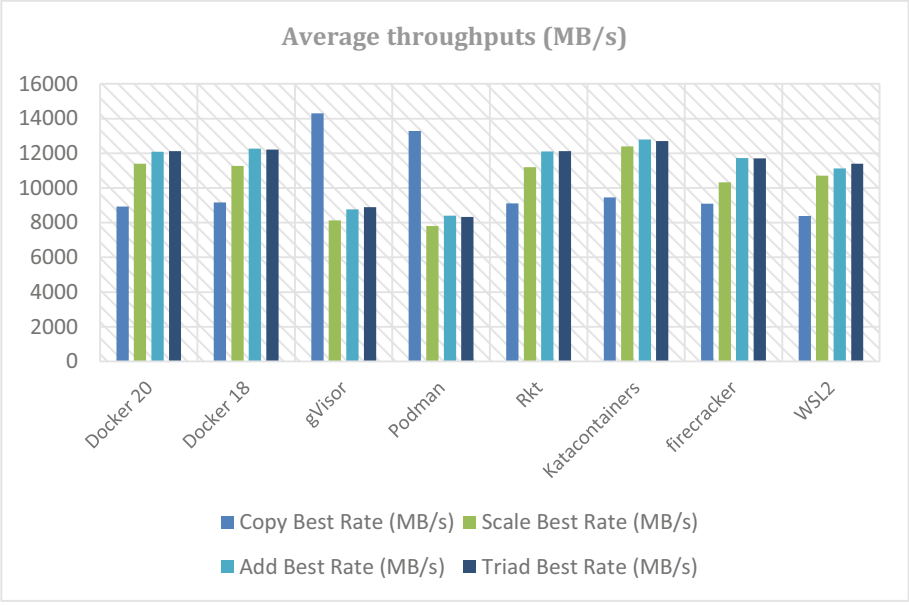

Fig. 15. Average stream operations throughputs

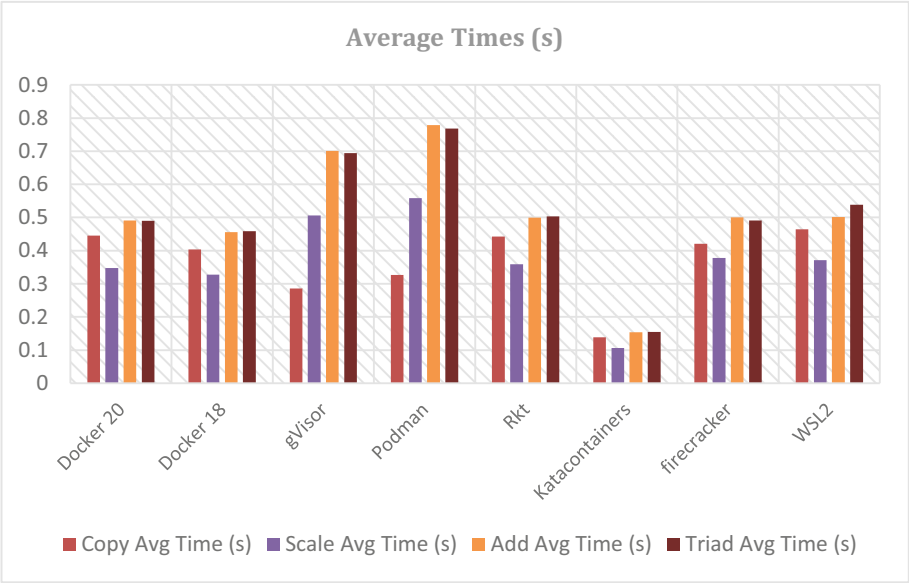

Fig. 16. Average stream operations times

5.4 Network Performance

As suggested by [44], we use NetPerf tool [52] to test the Network I/Os performance.

The tests were performed on two containers (NetPerf client and server) on the same host to simulate intra containers communication and files transfer in a Micro Services scenario. For each scenario, we run 15 iterations of 10 s each. Using NetPerf tool, we measured throughputs and latencies for these tests: TCP_Streaming, UDP_Streaming, TCP_RR and UDP_RR.

Table 12, Fig. 17 and Fig. 18 show the obtained results:

Table 12. NetPerf Benchmark Results

Engine	TCP_Stream Throughput (MB/s)	TCP_Stream latency(ms)	UDP_Stream Throughput (MB/s)	UDP_Stream latency(ms)	TCP_RR throughput (MB/s)	TCP_RR latency (ms)	UDP_RR Throughput (MB/s)	UDP_RR latency (ms)
Docker 18	12675.93	15.76	11778.28	19.55	12878.11	15.92	13151.05	19.21
WSL 2	13015.39	77.14	11065.14	50.96	13469.26	72.14	11186.48	40.34
Docker 20	10837.26	40.45	11051.91	33.87	11048.06	40.11	11109.7	39.39
Podman	15751.87	8.84	15425.41	8.67	15500.61	10.53	15638.8	9.65
Kata Containers	2254.81	144.89	2189.7	139.13	2104.88	138.8	2277.82	146.16
gVisor	3405.69	1168.45	3534.93	1095.63	3514.15	1130.2	3557.79	1084.6
Firecracker	2179.23	171.96	2044.78	162.56	2119.73	163.95	2066.8	169.19
RKTx	9557.65	21.38	9413.41	19.71	9618	20.04	9376.97	19.09

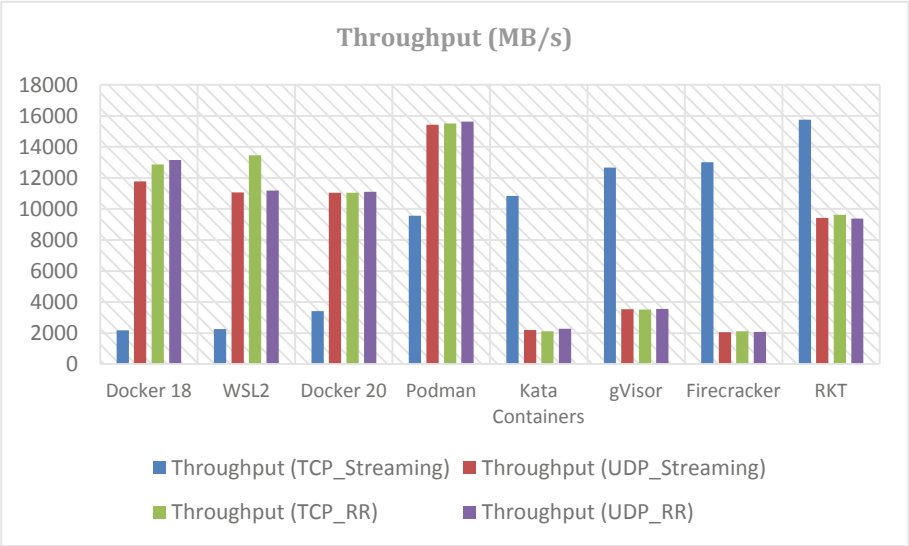

Fig. 17. Average NetPerf Tests throughputs

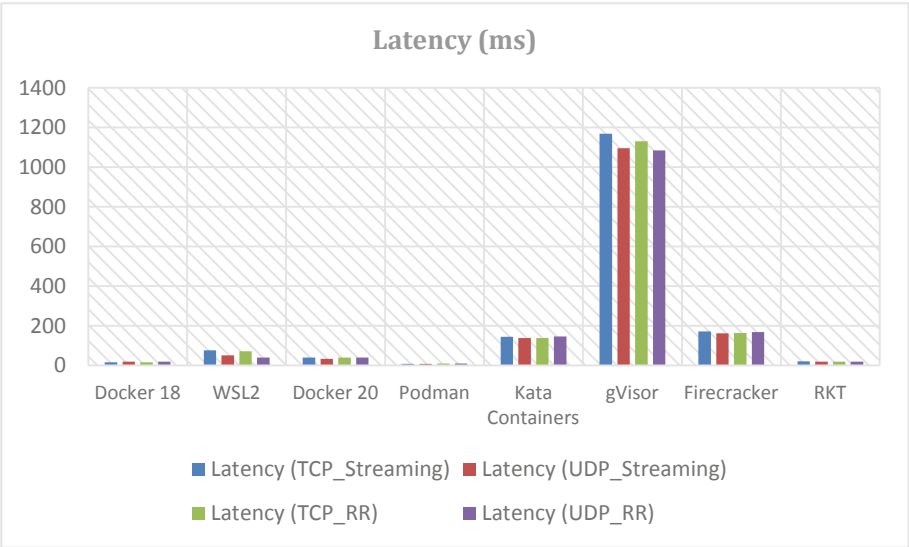

Fig. 18. Average NetPerf tests latencies

For the TCP_Streaming test, Rkt and Podman were by far better than Docker in the native category. Kata Containers, gVisor and Firecracker recorded good results and outperform Docker despite of the Virtualization layer. This is because of the good TCP window and socket buffers management at the lightweight VM guest level and the fast network bridges implemented in memory unlike the Docker bridge. For the other tests, Podman was by far the best followed by Docker, Rkt and the microVMs solutions.

Concerning the latencies, gVisor recorded the worst results due to the translation overhead of syscalls in user space and the double packets management at Sentry (netStack) and Kernel levels [23]. With no surprise, native solutions recorded short latencies.

6 Conclusion and Future Work

We have performed different tests to measure some metrics related to CPU, Memory and Disk IOs using the same Docker Images. All the tested engines were OCI images compliant.

For the CPU performance part, native engines were better like Podman, Docker and Rkt. In many cases, Podman and Rkt performed better than Docker because of their daemonless architecture. The Docker Daemon adds some overhead as it is the parent of all descendant containers. The engines based on microVMs except WSL2 recorded interesting results. This is due to the light Hardware Virtualization layer. WSL2 needs perhaps more improvements but for the moment this solution is just dedicated for developers that desire to work on two different environments on the same machine and transparently. For production cases, WSL2 must be integrated with optimized Linux kernels.

For the Disk IOs performance part, we have applied two benchmarks and we got mismatches between bonniee ++ and Y-cruncher IOs and between our obtained results and those obtained in the literature [23]. Generally, gVisor during our tests was performing well and we can say that the User Space kernel has proven its efficiency. Y-cruncher benchmark results seem to be more reliable and the native engines performed nearly the same results. gVisor and Kata Containers are so promising in this field compared to other microVMs solutions.

For the Memory performance part, we noticed that almost engines perform similarly. Podman and gVisor record the slowest times. However, Kata Containers were performing the best throughputs and the shortest times.

For the Network IOs performance part, we have applied the NetPerf benchmark and generally the native engines outperform other solutions. This is due to the Virtualization overhead for the microVms based engines in addition to the syscalls translation overhead for gVisor. Docker recorded low performance results in the TCP_Streaming scenario. Generally, Podman is the best choice for network applications.

Before writing this paper, we were not expecting such promising results related to the microVMs which perform nearly performance as the native engines.

Some studies need to be accomplished in the future by including the Unikernel based solutions like Nabla.

This study has concluded on performance differences between the different engines. We will try in a future work to propose an architecture of an orchestrator that can manage heterogenous engines at the same time. This orchestrator, based on a desirability function, could assign the best engine to a new workload. Using checkpoint/restore techniques it should be possible to migrate workloads between different engines. This approach is so feasible when engines are OCI compliant.

References

1. Senthil Kumaran, S.: Practical LXC and LXD: Linux Containers for Virtualization and Orchestration. Apress, New York (2017)
2. Goasguen, S.: Docker Cookbook: Solutions and Examples for Building Distributed Applications. O'Reilly Media, Inc, Newton (2015)
3. Cochrane, K., Chelladhurai, J.S., Khare, N.K.: Docker Cookbook: Over 100 Practical and Insightful Recipes to Build Distributed Applications with Docker. Packt Publishing Ltd, Birmingham (2018)
4. Turnbull, J.: The Docker Book: Containerization is the New Virtualization. James Turnbull, Melbourne (2014)
5. Ruan, B., Huang, H., Wu, S., Jin, H.: A performance study of containers in cloud environment. In: Asia-Pacific Services Computing Conference (2016)
6. Miell, I., Sayers, A.H.: Docker in Practice. Manning Publications Co, Shelter Island (2016)
7. Nickoloff, J.: Docker in Action. Manning Publications Co, Shelter Island (2016)
8. Stubbs, J., Moreira, W., Dooley, R.: Distributed systems of microservices using docker and serfnode. In: 2015 7th International Workshop on Science Gateways. IEEE, pp. 34–39 (2015)
9. Jaramillo, D, Nguyen, DV., Smart, R. Leveraging microservices architecture by using Docker technology. In: SoutheastCon 2016, p. 1–5. IEEE (2016)
10. Pérez, A., Moltó, G., Caballer, M., et al.: Serverless computing for container-based architectures. Future Gen. Comput. Syst. **83**, 50–59 (2018)
11. Lloyd, W., Ramesh, S., Chinthalapati, S., et al.: Serverless computing: an investigation of factors influencing microservice performance. In: 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 159–169. IEEE (2018)
12. Thönes, J.: Microservices. IEEE Softw. **32**(1), 116 (2015)
13. Morabito, R., Kjällman, J., Komu, M.: Hypervisors vs. lightweight virtualization: a performance comparison. In: 2015 IEEE International Conference on Cloud Engineering, pp. 386–393. IEEE (2015)
14. VMware Open Source Program Office. https://vmware.github.io/vic-product/#documentation. Accessed 12 May 2020
15. Kang, H., LE, M., Tao, S.: Container and microservice driven design for cloud infrastructure devops. In: 2016 IEEE International Conference on Cloud Engineering (IC2E), pp. 202–211. IEEE 2016
16. Brikman, Y.: Terraform: Up & Running: Writing Infrastructure as Code. O'Reilly Media, Newton (2019)
17. OCI Homepage. https://www.opencontainers.org/. Accessed 12 May 2020
18. CNCF Homepage. https://www.cncf.io/. Accessed 12 May 2020
19. Kubernetes CRI Homepage. https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/. Accessed 12 May 2020