SN Computer Science (2023) 4:387
https://doi.org/10.1007/s42979-023-01827-9

ORIGINAL RESEARCH

The Convergence of Container and Traditional Virtualization: Strengths and Limitations

Guoqing Li[1] · Keichi Takahashi[2] · Kohei Ichikawa[1] · Hajimu Iida[1] · Chawanat Nakasan[3] · Pattara Leelaprute[4] · Pree Thiengburanathum[5] · Passakorn Phannachitta[5]

Received: 30 September 2021 / Accepted: 12 April 2023 / Published online: 11 May 2023

Abstract

Virtual machines (VMs) are used extensively in the cloud. The underlying hypervisors allow hardware resources to be split into multiple virtual units which enables server consolidation, fault containment, and resource management. However, VMs with traditional architecture introduce heavy overhead and reduce application performance. Containers are becoming popular options for running applications, yet such a solution raises security concerns due to weaker isolation than VMs. We are at the point of container and traditional virtualization convergence where lightweight hypervisors are implemented and integrated into the container ecosystem to maximize the benefits of VM isolation and container performance. However, there has been no comprehensive comparison among different convergence architectures. To identify limitations and best-fit use cases, we investigate the characteristics of Docker, Kata, gVisor, Firecracker, and QEMU/KVM by measuring the performance of disk storage, main memory, CPU, network, system call, and startup time. In addition, we evaluate their performance of running the Nginx web server and the MySQL database management system. We use QEMU/KVM as an example of running traditional VMs, Docker as the standard runc container, and the rest as the representatives of lightweight hypervisors. We compare and analyze the benchmark results, discuss the possible implications, explain the trade-off each organization made, and elaborate on the pros and cons of each architecture.

Keywords Virtualization performance · runC container · Lightweight hypervisor · gVisor · Kata · Firecracker

Introduction

The traditional VM architecture exemplified by QEMU/KVM offers strong isolation [14] since each guest VM has its own file system, authentication mechanism and a layer of hypervisor sits in between the host, and guest OS, which is the only way for the guest VM to communicate

This article is part of the topical collection "Cloud Computing and Services Science" guest edited by Donald Ferguson, Markus Helfert and Claus Pahl.

✉ Guoqing Li
guoqing_li@pm.me

Keichi Takahashi
keichi@tohoku.ac.jp

Kohei Ichikawa
ichikawa@is.naist.jp

Hajimu Iida
iida@itc.naist.jp

Chawanat Nakasan
chawanat.n@ku.th

Pattara Leelaprute
pattara.l@ku.ac.th

Pree Thiengburanathum
pree.t@cmu.ac.th

Passakorn Phannachitta
passakorn.p@cmu.ac.th

[1] Nara Institute of Science and Technology, Nara, Japan

[2] Tohoku University, Sendai, Japan

[3] Kanazawa University, Kanazawa, Japan

[4] Kasetsart University, Bangkok, Thailand

[5] Chiang Mai University, Chiang Mai, Thailand

with the physical hardware. However, virtualizing hardware resources imposes heavy performance overhead [15]. In contrast, Docker containers utilize Linux's built-in features such as cgroups to manage resources and namespaces to isolate running processes which comes with much less performance overhead [11]. Cloud providers have the economic incentive to run as many containers as possible on the same host, and although containers excel at performance, their isolation is generally weaker and they expose a larger attack surface [6] compared to VMs. There is a possibility of exploiting existing kernel bug to break out of a container onto the host and misuse of privilege containers have lead to many security incidents. In 2017 alone, 454 vulnerabilities were found in the Linux kernel,[1] which can be a major risk for containerized environments. A recent Kernel vulnerability—*CVE-2020-14386* found in 5.7 kernel release could even allow container to escape and gain root privileges. There are legitimate concerns from public cloud service providers who base their services on containers, since they have no control over what kind of applications are running in their cloud.

Several organizations address these security concerns by implementing lightweight hypervisors which, when used in conjunction with containers, strike a good balance between performance and security isolation. Rapid innovation in this area has resulted in several different architectural approaches. A comprehensive performance evaluation as presented in this paper is important when considering the trade-offs of different approaches. We take a detailed look at lightweight hypervisors used in containerized environments and compares various performance metrics. Our goal is to understand the overhead imposed by QEMU/KVM, Docker, gVisor, Kata-qemu, and Kata-Firecracker.

This paper makes the following four contributions:

- We provide an extensive comparative performance analysis of QEMU/KVM VM, Docker, gVisor, Kata-qemu, and Kata-Firecracker.
- We identify the best-fit use case for practitioners by analyzing the pros and cons of each architecture in detail.
- We elaborate on the non-obvious limitations of each architecture that affect virtualization performance.
- We provide up-to-date reviews on existing container and traditional virtualization convergence technologies.

This paper extends our previous work [10] by adding experiments to evaluate the CPU compute performance (NAS Parallel Benchmark Suite), memory bandwidth (STREAM benchmark), and read/write system call performance. We also reconfigured the Kata container environment with the newest runtime and redid the experiments. Both the Architecture and Evaluation sections are rewritten ton include more details and in-depth analyses. The Discussion section is restructured to provide better insights.

The structure of the paper is organized as follows: Sect. "Background" describes the motivation and architecture of each environment. Section "Evaluation" presents our evaluation results in two parts:

- Part I—the low-level aspects which cover startup time, memory footprint, system call latency, network throughput, Disk I/O, and CPU performance.
- Part II—the high-level aspects which cover two real-world applications: Nginx webserver and MySQL database.

Section "Discussion" describes the interpretation of our benchmark results, and the pros and cons of each virtualization platform. Section "Related Work" reviews related work. Section "Conclusion and Future Work" concludes the paper and suggests possible future work.

Background

Motivation

With the emergence and adoption of container technology and lightweight hypervisors, there is inevitably a trade-off between virtualization performance and security isolation. For example, gVisor implemented the network stack in user-space and using Gofer proxy to redirect I/O calls to provide better isolation, but imposes heavy I/O performance overhead. Practitioners face the challenge of making this kind of trade-off. Different organizations choose to implement their hypervisors according to their specific needs which limits generality. Taking Firecracker as an example, it is designed to run containers on a stripped down Linux kernel [3]. Kata-qemu runs a minimal Clear Linux guest OS on a QEMU VM, and containers are launched inside of the guest via an agent. In both cases, this leaves no options for running guests with different kernels or Linux distributions. To make good technology choices, it is crucial to understand (1) what trade-offs each architecture made and (2) the performance characteristics and limitations of each architecture. The following subsection presents the overall architectural components of each virtualization system and how they are connected.

[1] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

Architecture

QEMU/KVM

QEMU/KVM [4, 8] is an established approach for running traditional VMs in the cloud. There are several other open source hypervisors available which have their own design considerations. For example, KVM is merged into the Linux mainline kernel which utilize kernel's existing scheduler and memory manager, whereas Xen builds their own dom0 kernel and included several CPU schedulers. In our paper, we will focus on QEMU/KVM only since Kata-qemu, Firecracker, and gVisor are all the fork of KVM project.

The early generation of x86 hypervisors used the technique of trap and emulate [18], relying on the mechanism of CPU exceptions, such as memory faults. The privileged instruction exceptions will be trapped and the control will be passed back to the hypervisor. Privileged instructions are then emulated by the underlying hypervisor against the guest VM state. The performance overhead of this technique is costly, since it requires the CPU to run many more cycles to execute the trapping and emulation instructions. This performance overhead can be mitigated by the use of binary translation [19], which works by translating certain sensitive instructions, so they can run directly without causing traps. Further improvement was made by the introduction of CPU virtualization extensions in the hardware. Such as Intel VT-x and AMD-V [16] which allow the classical trap and emulate technique to run more efficiently. KVM is a Linux kernel module that interfaces with the hardware virtualization extension and reuses the Linux kernel's existing CPU scheduler and memory manager to provide a uniform API.

Intel's Extended Page Table is used to enable Memory Management Unit (MMU) virtualization to avoid the overhead caused by software managed shadow page tables. MMU virtualization is an important milestone for hardware assisted virtualization, because the VT-x extensions alone could not offer better performance compared to binary translation [1]. The KVM-userspace side is handled by QEMU to serve requests that KVM cannot handle directly, such as device emulations (block devices, network card, display, etc.) The release of virtio drivers took the Disk and Network I/O performance to the next level. Nowadays, virtio drivers have become the de facto standard for storage and network virtualization. PCI devices with Single Root I/O Virtualization (SR-IOV) feature also push the boundaries of network virtualization performance.

Docker

Containers utilize Linux built-in features: *cgroups* which allow processes to be organized into hierarchical groups, so that various types of resource usage (CPU and memory etc.) can be controlled and monitored, and *namespaces* which wrap a particular global system resource in an abstraction that makes the processes running inside of this namespace appear to own the whole global system resource, changes of processes running in different namespaces are invisible to other processes. Some of the important Linux namespaces used by containers include: *PID*, *NET*, *IPC*, *MNT*, *UTS*, *MOUNT*, and *CGroup*. RunC is a low-level command line interface (CLI) tool to create and run containers. It creates the cgroups and namespaces and bind the processes. Containerd is a daemon for managing the life-cycle of containers. Both runC and containerd are used by all of the virtualization systems discussed in this paper except for QEMU/KVM.

Docker is built on top of runC and containerd. It provides a rich set of CLI commands to manage containers, and a common storage format for container images based on the Overlay2[2] which is a union mount file system for Linux to persist data at the writable layer. Btrfs and zfs are also supported to enable advanced features such as creating snapshots. Logical volume management can also be set up using device mapper storage driver.

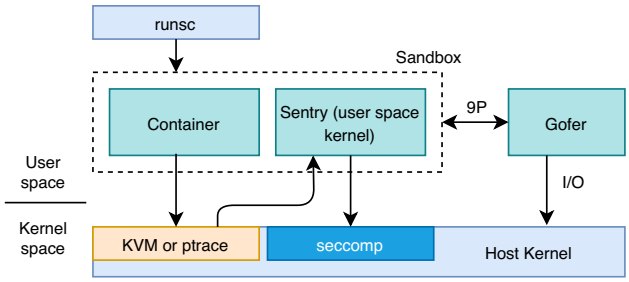

Fig. 1 Architecture of gVisor

gVisor

Google's gVisor[3] provides an isolation layer between containerized applications and the host kernel, which is a feature not offered by containers alone. It creates a sandbox to intercept and redirect system calls to a secure userspace kernel. Figure 1 shows the high-level architecture of gVisor.

gVisor provides two platforms[4] for intercepting system calls: ptrace and KVM. The ptrace platform uses the *ptrace* debugger mechanism built into the host kernel to trap system calls, whereas the KVM platform uses the Linux KVM kernel module to take advantage of hardware virtualization

[2] https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[3] https://gvisor.dev/.

[4] https://gvisor.dev/docs/architecture_guide/platforms/.

support for better performance. gVisor ptrace is important for running containers in virtual environments where CPU virtualization extensions are not available, such as a public cloud environment where nested virtualization is disabled.

The *Sentry* is the userspace kernel that implements all the kernel functionalities required by applications, such as system calls, memory management, and page faulting logic. The CPU scheduling is, however, handled by the Go-routine scheduler. At the time of writing, there are 224 out of 291 system calls are implemented by Sentry which is sufficient for most applications. Effectively, Sentry is playing both the role of guest OS and hypervisor. Host system calls invoked from Sentry are further filtered using *seccomp*,[5] a kernel facility that restricts the system calls that can be invoked by applications with a configurable security policy.

File I/O is handled by a separate file proxy process called *Gofer*, which communicates with the Sentry through the 9P file system protocol [17]. This allows users to set up a writable temporary file system overlay on top of the entire file system, so that the container sandbox is isolated from the host file system. Users can also enable read only file system sharing between different containers. This allows some degree of file space and cache performance optimization.

gVisor implements its own networking stack—netstack, which is a component of Sentry. Netstack includes: TCP connection state, control messages, and packet assembly which are isolated from the host networking stack. Host networking pass-through feature is provided to applications that require high networking performance. This model of security in depth, with its many layers of isolation, also imposes some performance overhead especially in the case of ptrace due to the significant cost of context switching.

gVisor has been built into Google's cloud infrastructure to provide serverless computing services, such as Google App Engine, Cloud Run, and Cloud Functions.

Kata QEMU

Kata containers[6] are a collaboration between the Intel Clear Linux project[7] and hyper.sh[8] projects. It is an example of a container and virtualization convergence technology that allows users to run containers within lightweight VMs. It has been seamlessly integrated into the containerd system using a shim, named containered-shim-kata-v2.

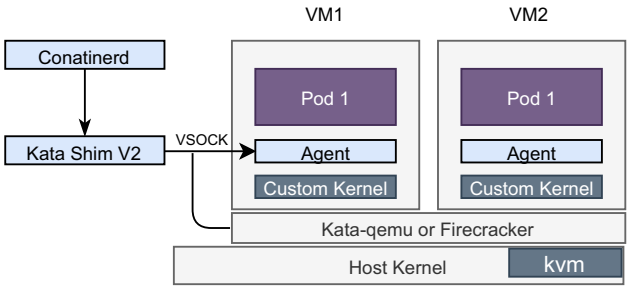

Fig. 2 Architecture of Kata Containers

The virtualization part is based on QEMU/KVM to enable hardware assisted virtualization, and a highly optimized guest kernel includes the functionalities to run only container workloads. The customized kernel is optimized to reduce boot time and memory footprint, and a minimal root file system based on Clear Linux reduces attack surface significantly by removing many of the binaries commonly found in general purpose Linux distributions. The only two processes running inside of the VM at startup are Systemd and a kata agent. The containerd-shim-kata and kata agent communicate across a VSOCK socket.

By default, Kata uses the QEMU/KVM hypervisor.[9] However, Kata allows users to run on different hypervisors: Cloud Hypervisor, Firecracker, etc. Figure 2 shows Kata's high-level architecture. Kata allows running multiple containers in a *pod*, where a pod is a group of related containers which share the same network namespace. Each pod of containers runs in a separate lightweight VM to provide strong isolation.

Kata Firecracker

Amazon's Firecracker [2] began as a fork of crosvm[10] hypervisor written in Rust which runs VMs through the KVM interface with a sandbox around virtual devices to enhance security. Firecracker is an alternative to QEMU that is lighter weight with minimal attack surface for security and only the following six of essential device emulations provided: a serial console, a minimal keyboard controller, and four virtio devices (virtio-net, virtio-balloon, virtio-block, and virtio-vsock) to handle network, memory, disk I/O, and host/guest communication, respectively. Amazon aims at integrating Firecracker into the container ecosystem which is again another example of container and traditional virtualization convergence technology. Firecracker uses an

[5] https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

[6] https://katacontainers.io/.

[7] https://software.intel.com/content/dam/develop/external/us/en/documents/intel-clear-containers-2-using-clear-containers-with-docker-706454.pdf.

[8] https://github.com/hyperhq.

[9] https://github.com/kata-containers/qemu.

[10] https://chromium.googlesource.com/chromiumos/platform/crosvm/.

emulated block device that can be mounted into the guest via Device Mapper.

Our specific focus is to benchmark the performance of running Kata containers inside of the Firecracker microVM, which requires the configuration of the devmapper snapshotter. In essence, the container root file system is a device mapper snapshot mounted into Firecracker as an emulated virtio-block device. Firecracker also uses seccomp filters to limit the system calls it can use, to enhance the security isolation. The Firecracker process needs to be started by a jailer process, which configures the required system resources and permissions and then executes the Firecracker binary as a non-privileged process. Firecracker also provides a cpuset subsystem based on Cgroups which allows users to set the CPU affinity of microVMs, preventing the host scheduler from migrating between vCPUs which may cause resource contention.

Evaluation

We first evaluate the low-level aspects of a computing system: startup time, memory bandwidth, memory footprint, system call latency, network throughput, disk I/O, and CPU performance. We then benchmark two common applications, Nginx webserver and MySQL database, which serve as realistic example workloads. Low-level benchmark metrics serve as the fundamental indicator of performance characteristics for each system and the high-level benchmark metrics are used develop a better understanding of how different hypervisors perform in a practical setting environment.

All systems are set up on an x86-64 server with Intel Xeon Silver 4208 CPUs. There are two sockets, and each socket contains 8 physical cores with hyper-threading enabled which provides a total of 32 logical cores. There are 32 GB of DDR4 SDRAM spanning across two NUMA nodes and the ext4 file system is installed on a 548 GB hard disk drive.

Table 1 shows the details of each execution environment. We upgraded QEMU to match the version that Kata runtime uses. Kata 2.2.0 is installed for both Kata-qemu and Kata-Firecracker Containers, which is significantly different from 1.x. Kata 2.x drops the support for Docker. We had to use *crictl* utility to launch pods and containers.

Table 1 Execution environment

Environment	Software versions
host OS	Ubuntu 20.04.3 LTS, Kernel 5.4.0-84
QEMU 5.2.0	libvirt 6.6.0, Guest Kernel 5.4.0-84
kata_qemu	Kata 2.2.0, Guest Kernel 5.10.25-85, QEMU 5.2.0
kata_fc	Kata 2.2.0, Guest Kernel 5.10.25-85, Firecracker v0.23.1
gVisor	gVisor release−20210720.0, Guest Kernel 4.4.0
Docker	Docker 20.10.8, containerd v1.5.3, runc, 1.0.0
crictl	crictl version 1.20.0-24-g53ad8bb7

Low-Level Aspects

CPU Performance

NAS Parallel Benchmark tool is used to stress the CPU with the "C class" matrix computation workload. We modified the host CPU configuration to avoid thermal variation using *frequency-set -g performance* to maximize and fix the CPU clock speed. Four vCPUs are allocated for each container and VM. In the case of Docker and gVisor, *–cpuset-cpus* is used to restrict the container process to use only vCPUs 0,2,4,6 which are physically separate cores on the host. Similarly, we use *cpuset* to pin QEMU/KVM VM to those same cores. This setup will prevent the host scheduler from migrating processes between vCPUs, thus maximizing cache utilization, potentially achieving better performance than the default configurations.

Figure 3 shows the average floating point compute performance for calculating different problem sets. Most of the platforms show similar benchmark results. One reason is that KVM utilizes the hardware extensions, such as VT-x for nearly native performance which explains why there is little difference between Docker and KVM-based platforms. The other reason is because by default all systems use Linux kernel's Complete Fair Scheduler in conjunction with Cgroups to control CPU resources. gVisor is an exception, because the CPU scheduling relies on the Go runtime scheduler which is a userspace scheduler runs on top of OS kernel's scheduler, and it may improve the throughput of application workload that is highly concurrent, such as a web server. In the case of the *cg* and *bt* problem sets, gVisor even surpassed the performance of Docker and Kata-qemu, suggesting that gVisor could be a good option for CPU-bound applications.

Memory Bandwidth

We use the STREAM benchmark tool to measure memory bandwidth in different systems. Figure 4 shows the memory bandwidth performance of vector operations on: COPY, SCALE, ADD, and TRIAD. The result is similar to the CPU benchmark. All KVM-based platforms have similar memory bandwidth, while docker has slightly better performance compared to others.

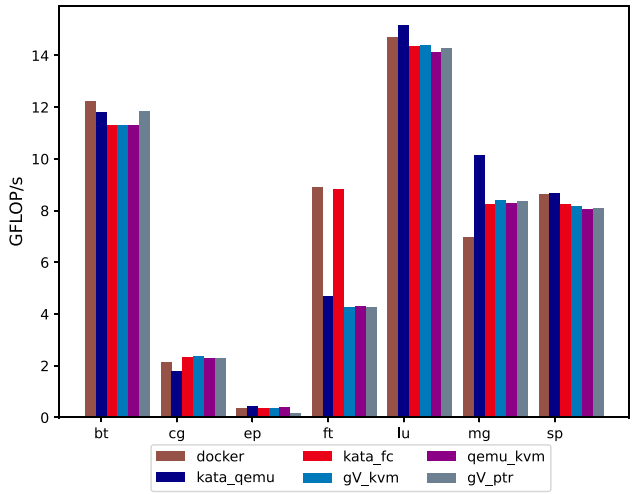

Fig. 3 CPU throughput (NPB-OMP)

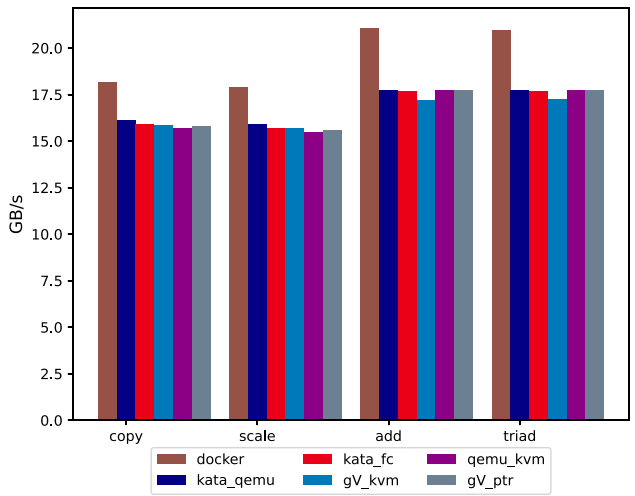

Fig. 4 Memory bandwidth (STREAM)

Startup Time

Fast startup time is crucial for container/VM provisioning [13], it is extremely important in the case of live migrations. We use the `time` command to measure the time from launching a container/VM to the stage of the network stack being successfully initialized. We run an Ubuntu:focal base image and a `bash` program inside of the container. In the case of QEMU/KVM VM, we placed the `systemd-analyze time` command in the startup script to collect the startup time for 100 rounds. Figure 5 shows the average elapsed time of 100 complete container creations. Since the startup time for QEMU/KVM was significantly slower, the inaccuracies resulting from the virtual clock are negligible.

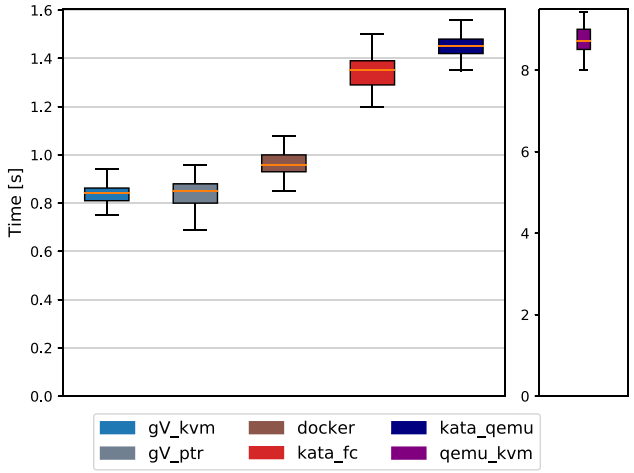

Fig. 5 Startup time

The startup times varied between approximately 0.7 and 1.6 s in all container based environments. On average, gVisor-kvm and gVisor-ptrace were the fastest, Docker was 0.13 s slower, Kata-firecracker was 0.37 s slower than that, and Kata-qemu was only 0.10 s slower than Kata-firecracker. Overall, the startup time of all containers were significantly faster than full QEMU/KVM VM which took 8.76 s to finish booting.

Some important factors that affect start-up time are the size of the hypervisor executable, boot sequence, and configuration complexity etc. The time it takes to load these binaries and files from disk is directly proportional to its size. Docker and gVisor showed much faster startup time compared to Kata and QEMU. This is because the former two are essentially running two lightweight processes with very small images. Kata runs Clear Linux which has a highly optimized boot path, it runs only a systemd init daemon and the kata agent, then the agent manages the creation of containers, which takes slightly longer start up times compared to Docker and gVisor. It is not surprising that QEMU/KVM takes the longest time to start up since the guest kernel image is bulky and many other processes need to be initialized.

Memory Footprint

Smaller memory footprint puts less pressure on both RAM and the CPU cache, allowing users to create a higher density of containers, and making more efficient use of system resources. We estimate the memory footprint of each container running the Ubuntu:focal base image by measuring the size of *private dirty* pages of related processes. We launched 100 containers in total and calculated the average size of private dirty pages. We chose private page size, because it is the closest approximation of the actual memory usage for each process as the number of containers scale to

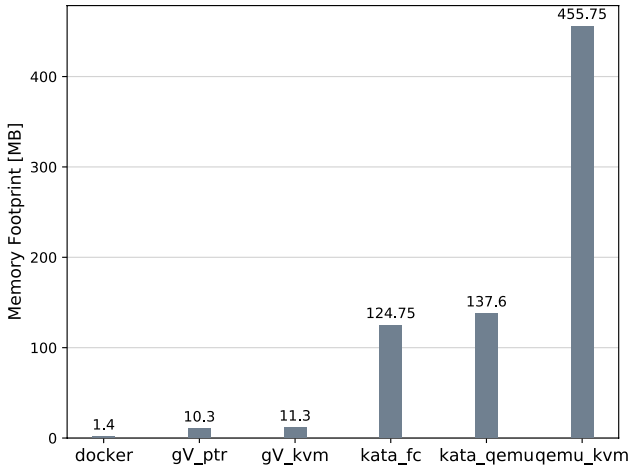

Fig. 6 Average memory footprint of 100 guests

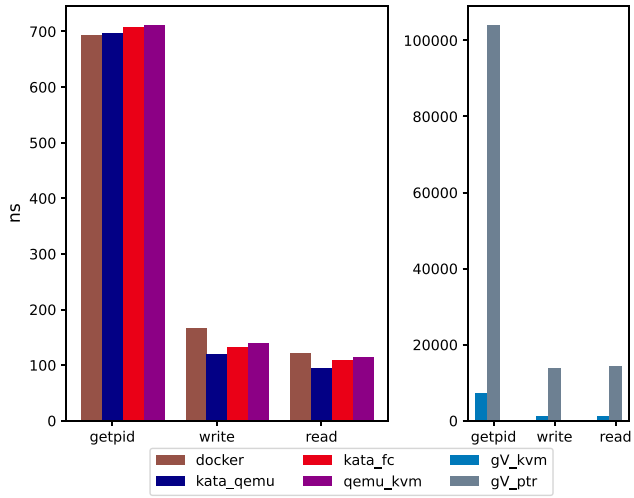

Fig. 7 System call latency

infinity. We allocated 2 GB of memory for each QEMU/KVM VM, which is the minimal size the guest can boot a full ubuntu image successfully. In case of Kata-qemu and Kata-Firecracker, we used the default configuration which allocates 2 GB of memory for each VM.

As Fig. 6 indicates, Docker has the smallest footprint, which is to be expected, because there is no additional hypervisor. A Docker container is just an application process on the host with namespaces and cgroup limits configured. There are some additional processes running alongside each container. For instance, the containerd-shim running in the background sits as the parent process of each container, this allows dockerd or containerd to detach from the container process without stopping the container. Another process containerd.sock is also running to facilitate the communication between containerd and the containerd-shim. Both processes have quite a small memory footprint. The docker daemon uses around 40 MB of memory, but they are shared by all containers, so, as the number of containers scale, the size of the daemon becomes a trivial contributor to the memory footprint.

The two variants of gVisor (ptrace and KVM) use nearly the same amount of memory per instance. They both consume 7× more memory compared to Docker. Both gVisor platforms run an additional process, runsc-sandbox, to wrap around containers. In addition to that, a runsc-gofer process for each container is also running in the background to handle I/O requests.

Kata-qemu consumes 88× more memory than Docker and 10× more than gVisor. The main reason is that a Kata container is not just a container, but a container running within a lightweight VM with a dedicated guest kernel. Many processes are running in the background, such as virtiofsd, qemu-virtiofs-system-x86, Kata-proxy, and Kata-shim, which lead to a significantly larger memory footprint compared to both Docker and gVisor. Kata creates one pod per VM which means, for each pod, the user will have to run additional kernels and all other related processes. Multiple containers can be run in a single pod, but they will be sharing the same namespaces and resources, so the memory footprint will be large if a single container pod is desired. Kata-firecracker uses slightly less memory compared to Kata-qemu, which is what we expect, since Firecracker is an optimized version of QEMU. Unnecessary device emulation drivers have not been included making it lightweight.

However, none of these container-based systems are comparable with QEMU/KVM. QEMU/KVM needed 325× more memory than Docker. Since we installed a general purpose Linux distribution that runs many services. Apart from that, the guest kernel used by Kata is highly optimized, which leads to much smaller memory footprint compared to a full Ubuntu:focal server distribution.

System Call

System call performance gives insight about the cost of user-kernel context switching. Figure 7 shows the average wall-clock time for invoking `getpid()`, `read()`, and `write()` system calls of ten iterations. For the read system call, the program reads from the `/dev/zero` device file, and for the write case, it writes to the `/dev/null` device file, so that the system calls will not spend any time waiting for disk I/O. There was a significant overhead with both gVisor-ptrace and gVisor-KVM. In particular, gVisor-ptrace `getpid()` latency is over 100× larger than Docker, Kata, and QEMU/KVM, which is a result of the inefficient ptrace mechanism, which requires additional context switches for each system call. With gVisor-KVM, the system call latency is about 10 × less than gVisor-ptrace, but still quite

slow compared to the others. Docker, Kata-qemu, Kata-firecracker, and QEMU/KVM are comparable. However, Docker `getpid()` latency is slightly slower than QEMU/KVM based systems which is potentially caused by the use of seccomp. At the time of writing, Kata does not yet support seccomp.

Network Throughput

We measured the network throughput from the host to each virtual environment using iPerf 3.6.[11] A TCP stream from the host to each virtual environment was generated for a total duration of 40 s. The graph in Fig. 8 shows the average network throughput. gVisor scored the worst on both KVM and ptrace platforms. This is due to its immature user-space network stack, *netstack*, which handles all aspects of network activities in the userspace to provide an additional layer of isolation at the cost of higher overhead. It should be noted that recent releases of gVisor optionally allow network pass-through, so that containers can use the host network stack directly, which is faster but weaker isolation. gVisor-kvm has by far the worst performance, as it is still in an experimental stage and has not yet been optimized. QEMU/KVM shows good network throughput thanks to the virtio network driver. Docker scored the best network throughput, indicating that little overhead is imposed.

Disk I/O

Disk I/O performance is important to applications that perform frequent I/O operations. Fio−3.12[12] was used to measure the Disk I/O performance. Each environment was allocated 2 GB of RAM and four virtual CPUs. Docker was configured to use the device mapper storage driver[13] in *direct-lvm* mode. QEMU/KVM was configured with virtio drivers. We created a dedicated LVM logical volume formatted ext4 file system to bypass the host file system cache. `O_DIRECT` flag is set to enable non-buffered I/O, and a 10 GB file (5× of the allocated RAM) is used to minimize the effect of memory caching.

Figure 9 demonstrates the sequential read latency for different block sizes. QEMU/KVM has the worst mean performance among all block sizes. In contrast, Kata-firecracker has the best mean performance and least variance compared to all the others.

Table 2 summarizes the latencies relative to Kata-firecracker. Most notably, the sequential write performance is two orders of magnitude slower than the others as indicated in Fig. 10. The reason for this is that an I/O request from the container sandbox needs to route through ptrace or KVM then the Sentry, then it is passed over the 9P protocol to a Gofer proxy, and then finally to the host file system. In the case of 16 KB block size, Kata-firecracker is at least twice as fast as any of the others. There was not much latency increase when the block size changed to 64 KB. However,

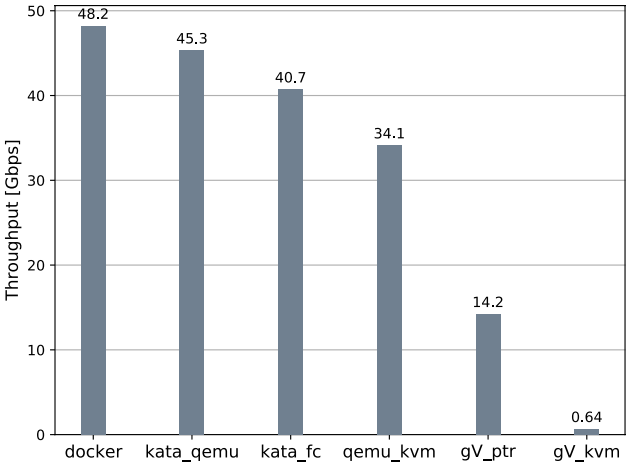

Fig. 8 Network throughput

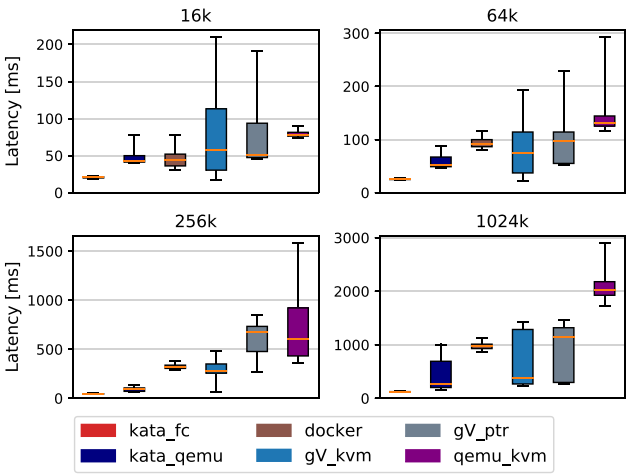

Fig. 9 Sequential read latency

Table 2 Relative latency compared to Firecracker

Environment	16K read	256K read	16K write	256K write
QEMU/KVM	3.8	13.0	2.0	5.0
kata_qemu	2.0	2.0	3.9	1.2
kata_fc	1.0	1.0	1.0	1.0
gV_ptr	2.0	6.2	328.0	151.0
gV_kvm	2.5	15.0	334.0	150.0
Docker	2.0	7.0	1.7	4.0

[11] https://github.com/esnet/iperf.

[12] https://github.com/axboe/fio/.

[13] https://docs.docker.com/storage/storagedriver/device-mapper-driver.

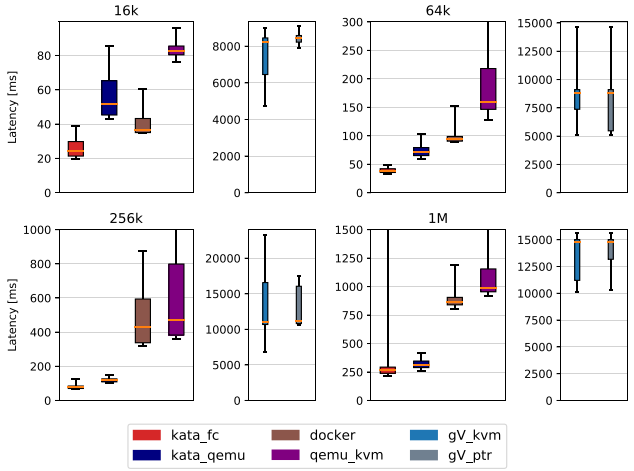

Fig. 10 Sequential write latency

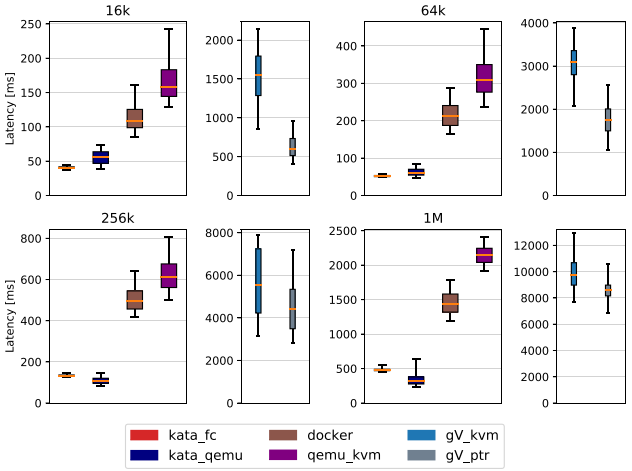

Fig. 11 Random read latency

when a 256 KB block size is used, the latency increased sharply which is possibly evidence that smaller block sizes fit within caches at the hardware layer.

Kata-firecracker is clearly ahead in all cases, followed by Kata-qemu, Docker, and QEMU/KVM. There was a similar trend with 64 KB block size. The performance gap between Kata and the next best doubled at a block size of 1 MB. QEMU/KVM showed a large variance at the upper end in both 64 KB and 256 KB block.

Figure 11 shows the random read latency. All platforms showed a similar trend. However, Kata-qemu surpassed the performance of Kata-firecracker when the block size was greater than 64 KB. In contrast to sequential read, gVisor-ptrace performed better than gVisor-kvm, when doing random reads, but both still ranked the worst performance.

Figure 12 shows the random write latency. Docker has the best performance with 16 KB block size. Kata-firecracker and Kata-qemu outperformed Docker when the block size is bigger than 16 KB.

Kata containers come with a special device-mapper storage driver which uses virtualized block devices rather than formatted file systems. Instead of using an overlay file system for the container's root file system, a block device is used directly. This approach allows Kata containers to outperform the others at all aspects of Disk I/O performance.

gVisor introduces overhead in several places: communication between components needs to cross the sandbox boundary and I/O operations must be routed through the Gofer process to enforce the gVisor security model. More importantly, the internal virtual file system implementation in Sentry has serious performance issues due to the fact that I/O operations need to perform path walks on every file access, such as `open(path)` and `stat(path)`. Each I/O operation requires a remote procedure call over 9P to Gofer to access the file. gVisor contributors have started working on rewriting the current virtual file system to address this

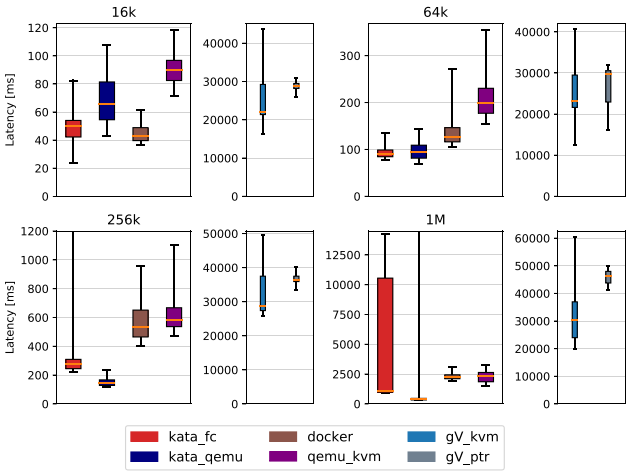

Fig. 12 Random write latency

performance bottleneck by delegating the path resolution to the file system. This new virtual file system implementation is being tested at Google internally at the time of writing.

Real-World Workload

MySQL

Sysbench 1.0.17 and MySQL[14] 8.0.20 were used to measure the performance of a typical database workload. We populated a table with 10 million rows of data and benchmarked the throughput and latency using a mixture of queries consisting of: Select (70%), Insert (20%), and Update and Delete (10%) queries. Figures 13 and 14 show the throughput and average latency, respectively. gVisor-kvm and gVisor-ptrace

[14] https://www.mysql.com/.

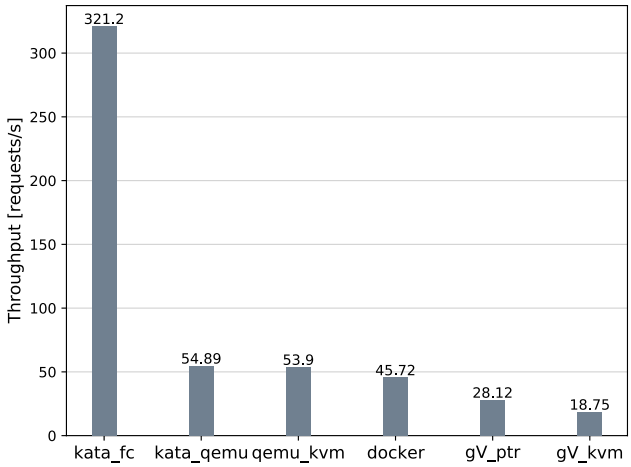

Fig. 13 MySQL OLTP throughput

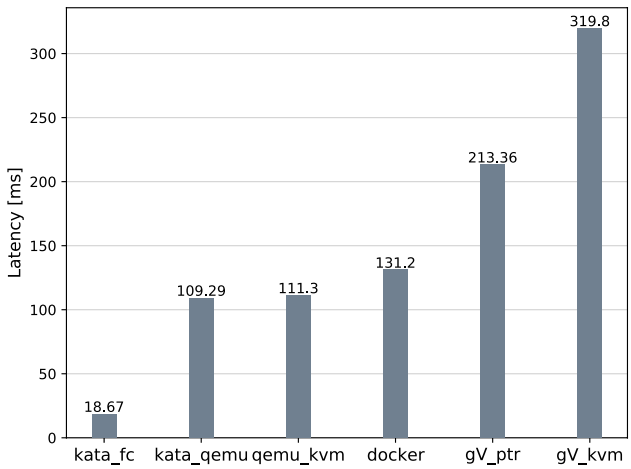

Fig. 14 MySQL OLTP latency

had the worst performance which achieves only 6% and 9% of what Kata-firecracker achieved respectively. Docker, QEMU/KVM, and Kata-qemu were comparable, but still ranged from 5.68 to 6.82× less throughput than Kata-firecracker. The average latency was negatively correlated with the throughput. The larger the latency, the less throughput was achieved. Both throughput and latency results were consistent with the disk I/O benchmark.

Nginx Serving a Static Web Page

We used wrk[15] 4.10 to measure the throughput of Nginx[16] serving a 4KB static web page. Each environment was configured to allow one worker process and a maximum

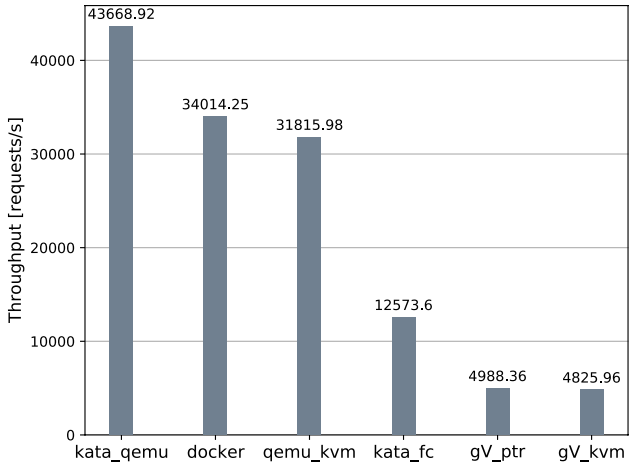

Fig. 15 Nginx webserver throughput

of 1024 concurrent connections. We used wrk to simulate HTTP GET requests for a duration of 10 s with 1,000 open connections.

Figure 15 shows the throughput of HTTP GET request for each environment. Kata-qemu performed the best which was slightly better than qemu-kvm and Docker. However, its throughput was 3.47× higher than Kata-firecracker. Low throughput from Kata-firecracker can be attributed to its limitation of handling I/O serially at the rate no greater than 13,000 IOPS [2]. Both gVisor-kvm and gVisor-ptrace were the worst with almost the same throughput.

Discussion

As the benchmark results and analysis indicate, different virtualization platforms have different performance characteristics because of their different architectures, aims, and purposes.

Security Isolation

Security is never guaranteed, but isolating application processes from the host kernel reduces the attack surface, and so reduces risk. Docker containers have full access to the host kernel, and so have the weakest isolation of the systems discussed in this paper.

QEMU/KVM offers strong isolation from the host provided by KVM, and a small attack surface in the QEMU hypervisor which is fairly mature and passed the test of time. Kata-qemu runs containers on QEMU/KVM VM, but it uses stripped down version of QEMU that has been compiled with many unnecessary features disabled resulting in a QEMU binary less than half the size of the standard QEMU build. Disabling features reduces the attack surface.

[15] https://github.com/wg/wrk.

[16] https://nginx.org.

Kata-firecracker uses the Firecracker hypervisor which was specifically designed to be minimal, and therefore minimal attack surface. It was written in Rust which is a memory safe programming language.

gVisor has a particular good isolation due to its modular architecture split into application container, Sentry, Gofer, and using seccomp to provide a sandbox.

Startup Time

Fast startup time is important for applications that require automatic scaling, dynamic load balancing, or latency-sensitive event-driven services. We might expect Docker to startup fastest, because it is not loading a hypervisor, but it is creating a multi-layer OverlayFS file system. This causes Docker to startup slightly slower than gVisor.

gVisor is actually the fastest, despite loading a Sentry and Gofer for each container, because its OverlayFS write directory tmpfs is in RAM. QEMU/KVM is the slowest, because it loads a bulky QEMU binary and a general purpose guest kernel. Kata-qemu is faster, because it uses a stripped down version of QEMU and a minimal guest kernel, and the same is true for Kata-firecracker.

Memory Overhead

RAM is the most precious resource, because it is the limiting factor on the number of guest instances that can be running at the same time. Smaller memory footprint is better to optimize the overall utilization of the host hardware. Docker has negligible memory overhead, because it is simply running the application process in a container, and the container is nothing more than a small data structure in the host kernel. QEMU/KVM has the largest memory overhead consisting of RAM allocated to the guest operating system and the RAM consumed by the hypervisor process itself. Kata-qemu also allocates RAM to the guest operating system, but the hypervisor and guest kernel are both smaller; on average, the memory overhead is less than a third of a standard QEMU/KVM VM. Kata-firecracker has only slightly less memory overhead, because, although Firecracker is smaller than the stripped down QEMU, they are still running the same guest kernel. gVisor's memory overhead is much smaller than Kata, because it is only running the lightweight Sentry and Gofer processes.

Compute Performance

Faster compute performance results in shorter runtimes, more responsive applications, and higher throughput. Compute performance is a function of CPU clock speed, memory bandwidth, and cache hit rate. The underlying hardware is the same, so the benchmarks of memory bandwidth and CPU throughput are basically the same for each virtualization system except for the discrete 3D Fast Fourier Transform benchmark (ft) where Docker and Kata-firecracker outperform the others. We do not yet have an explanation for this discrepancy.

System Call Latency

System calls are used frequently during I/O operations, such as file and network transfers. Applications that perform a lot of I/O benefit from low latency system calls. Given that QEMU/KVM, Kata-qemu, and Kata-firecracker all trap system calls using KVM, and all have fairly optimized system call handlers, it is unsurprising that the performance of these three is similar and essentially as good as Docker, which directly calls the host kernel.

Notably, gVisor's system call latency is very much slower, especially when using the ptrace platform. The difference between ptrace and KVM is that KVM traps system calls in hardware, whereas ptrace traps them entirely in userspace. Having a userspace kernel adds an extra layer of isolation, but it also reduces application performance. It is worth noting that gVisor only implements about half the Linux system calls, so some applications may experience compatibility issues. In addition, Sentry is still under active development and has not yet been well optimized. At the time of writing, we cannot yet recommend running I/O intensive applications on gVisor.

Network

Network performance is important for many common use cases such as web application servers. gVisor-kvm network throughput is extremely poor even compared to gVisor-ptrace which is counter intuitive given that KVM is hardware accelerated, whereas ptrace is done in software. We suspect that there is a bug in the implementation, and so cannot recommend gVisor-kvm for network applications at the time of writing. However, using host network pass-through mitigates this issue. All the other virtualization platforms achieved in excess of 30 Gbps throughput which is more than sufficient for most applications.

Disk I/O

Good disk I/O performance is important for applications such as databases, file servers, static web servers, and video streaming servers.

gVisor has by far the worst disk I/O performance by an order of magnitude. This is due to the security architecture that requires I/O requests to go through many layers. Also, the path walk limitation of the virtual file system is highly inefficient. The project is still early in its development, we

would expect this and the other optimizations to improve over the next few years.

Docker is not much better than QEMU/VM and significantly slower than Kata-qemu and Kata-firecracker, we believe that this is mainly due to the overhead of OverlayFS. The clear performance winner here is Kata-firecracker closely followed by kata-qemu.

Other Points of Note

There are other aspects to consider when choosing a virtualization platform which have an indirect impact on performance in practice. For example, virtualization systems that comply with the Open Container Initiative (OCI) specification[17] can be seamlessly integrated into the container ecosystem, including higher level management systems, such as Kubernetes, which provides orchestration, scheduling, self-healing, and load balancing. OCI compatible systems include Docker, gVisor, and Kata.

The size of the developer and user communities, and the maturity of the supporting infrastructure is an important factor to consider when choosing a virtualization solution. For example, Docker has wide support and DockerHub provides a quick and simple deployment method. Similarly, QEMU/KVM is a mature and well-established virtualization solution. In contrast, gVisor and Kata are still quite immature and not widely adopted.

Another consideration is live migration, which is a mature feature of QEMU/KVM, but not available in any of the other systems discussed in this paper.

Kata has some of the best performance characteristics but also has several limitations. At the time of writing, *SELinux* is not supported by Kata's guest kernel and joining the host or an existing VM network is not possible due to the guest–host isolation.[18]

Related Work

[7] studied the CPU, memory, storage, and network performance of Docker containers and VMs, and elaborated on the limitations that impact virtualization performance. They concluded that containers result in equal or better performance than VMs in all aspects. [5] studied the isolation characteristics of both gVisor and Firecracker from the perspective of Linux Kernel code footprint. Linux containers exercised a significantly larger kernel code footprint compared to Firecracker. Firecracker is very effective at reducing the frequency of system calls to the host kernel, whereas gVisor executes even more kernel code compared to native Linux, since its design leads to a lot of duplicated functionality. [9] adopted a similar methodology as [7]; however, they targeted Docker and its rival Flockport (LXC). Their results were similar to [7], but they pointed out that Docker allows only one application per container, which reduces utilization, whereas Flockport does not impose such restrictions. [20] did a similar performance study of gVisor, Runc, and Kata containers. Their result is similar to what we have discovered.

None of those aforementioned studies investigated the characteristics of memory footprint and startup time, which are critical to many container applications. Apart from that, container and virtualization convergence solutions have emerged to be a promising alternative that strikes a good balance between performance and isolation, yet there has been no existing research providing detailed analysis on their limitations and benefits. Our paper focuses on those emerging technologies backed up by Google, Amazon, Baidu, and Intel, and provides practitioners a comprehensive analysis and review on the most up-to-date solutions.

Conclusion and Future Work

We have conducted a comprehensive performance analysis of various innovative lightweight hypervisors used in conjunction with containers. The benchmark results showed various trade-offs made by each solution and a number of bottlenecks that affect virtualization performance are identified. The pros and cons of each system are discussed at length, and some limitations that could be potentially addressed in the future are pointed out.

It is evident that the current architectural trend of virtualization platforms is to converge. Container and hypervisor hybrid solutions have the potential to supplant traditional VMs as the leading virtualization architecture. However, these solutions have not yet become a mature alternative. Traditional VMs would still be the preferred option for many use cases. Kata is on the right track to earn the title of having the security of a VM and the performance of a container, but it also has quite a few limitations. gVisor solved some practical problems in today's cloud environment; however, their I/O architecture is not yet full optimized.

Future research on reducing the memory footprint of lightweight hypervisor-based containers would be desirable. Research using Xen hypervisor with customized unikernels [12] opens the possibility of creating lighter and safer VMs than containers. KubeVirt, a Kubernetes addon that runs VMs alongside containers at scale, is an important virtualization convergence technology that solves the issues that some legacy applications cannot be easily containerized.

[17] https://opencontainers.org/.

[18] https://github.com/kata-containers/documentation/blob/master/Limitations.md.

Acknowledgements This work is partly supported by JSPS KAKENHI under Grant Nos. JP18K11326, JP20K19808, and JP21K11913. We would like to thank Mr. James for many detailed discussions and suggestions, and his tremendous help on the proof reading.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest and this research is not sponsored by any organizations discussed in this paper.

References

1. Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. ACM Sigplan Not. 2006;41(11):2–13.
2. Agache A, Brooker M, Iordache A, Liguori A, Neugebauer R, Piwonka P, Popa D-M. Firecracker: Lightweight virtualization for serverless applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20), 2020:419–434.
3. Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A. et al. Serverless computing: Current trends and open problems. In: Research Advances in Cloud Computing, pp 1–20. Springer; 2017.
4. Bellard F. QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference (ATC'05), 2005:41–46.
5. Caraza-Harter T, Swift MM. Blending containers and virtual machines: a study of firecracker and gvisor. In: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2020:101–113.
6. Combe T, Martin A, Di Pietro R. To Docker or not to Docker: a security perspective. IEEE Cloud Comput. 2016;3(5):54–62.
7. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015:171–172.
8. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. KVM: The Linux virtual machine monitor. In: The Linux symposium. 2007;1:225–30.
9. Kozhirbayev Z, Sinnott RO. A performance comparison of container-based technologies for the cloud. Future Gener Comput Syst. 2017;68:175–82.
10. Li G, Takahashi K, Ichikawa K, Iida H, Thiengburanathum P, Phannachitta P. Comparative performance study of lightweight hypervisors used in container environment. In: Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER, pp 215–223. INSTICC, SciTePress; 2021.
11. Li Z, Kihl M, Lu Q, Andersson JA. Performance overhead comparison between hypervisor and container based virtualization. In: IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), 2017:955–962.
12. Manco F, Lupu C, Schmidt F, Mendes J, Kuenzer S, Sati S, Yasukata K, Raiciu C, Huici F. My VM is lighter (and safer) than your container. In: 26th Symposium on Operating Systems Principles (SOSP'17), 2017:218–233.
13. Mao M, Humphrey M. A performance study on the VM startup time in the cloud. In: IEEE Fifth International Conference on Cloud Computing (CLOUD 2012), 2012:423–430.
14. Matthews JN, Hu W, Hapuarachchi M, Deshane T, Dimatos D, Hamilton G, McCabe M, Owens J. Quantifying the performance isolation properties of virtualization systems. In: 2007 Workshop on Experimental Computer Science, pp. 6–es; 2007.
15. McDougall R, Anderson J. Virtualization performance: perspectives and challenges ahead. ACM SIGOPS Oper Syst Rev. 2010;44(4):40–56.
16. Neiger G, Santoni A, Leung F, Rodgers D, Uhlig R. Intel virtualization technology: Hardware support for efficient processor virtualization. Intel Technol J, 2006;10(3).
17. Pike R, Presotto D, Dorward S, Flandrena B, Thompson K, Trickey H, Winterbottom P. Plan 9 from Bell Labs. Comput Syst. 1995;8(2):221–54.
18. Popek GJ, Goldberg RP. Formal requirements for virtualizable third generation architectures. Commun ACM. 1974;17(7):412–21.
19. Sites RL, Chernoff A, Kirk MB, Marks MP, Robinson SG. Binary translation. Commun ACM. 1993;36(2):69–81.
20. Wang X, Du J, Liu H. Performance and isolation analysis of runc, gvisor and kata containers runtimes. Clust Comput. 2022;25(2):1497–513.