Comparative Performance Study of Lightweight Hypervisors Used in Container Environment

Guoqing Li[1][a], Keichi Takahashi[1][b], Kohei Ichikawa[1][c], Hajimu Iida[1][d], Pree Thiengburanathum[2][e] and Passakorn Phannachitta[2][f]

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

[2]*Chiang Mai University, Chiang Mai, Thailand*

Keywords: Lightweight Hypervisor, Container, Virtualization, Performance Evaluation.

Abstract: Virtual Machines (VMs) are used extensively in cloud computing. The underlying hypervisor allows hardware resources to be split into multiple virtual units which enhances resource utilization. However, VMs with traditional architecture introduce heavy overhead and reduce application performance. Containers have been introduced to overcome this drawback, yet such a solution raises security concerns due to poor isolation. Lightweight hypervisors have been leveraged to strike a balance between performance and isolation. However, there has been no comprehensive performance comparison among them. To identify the best fit use case, we investigate the performance characteristics of Docker container, Kata containers, gVisor, Firecracker and QEMU/KVM by measuring the performance on disk storage, main memory, CPU, network, system call and startup time. In addition, we evaluate their performance of running Nginx web server and MySQL database management system. We use QEMU/KVM as an example of traditional VM, Docker as the standard container and the rest as the representatives of lightweight hypervisors. We compare and analyze the benchmarking results, discuss the possible implications, explain the trade-off each organization made and elaborate on the pros and cons of each architecture.

1 INTRODUCTION

The traditional VM architecture exemplified by QEMU/KVM offers strong isolation (Matthews et al., 2007) since a thin layer of hypervisor sits in between the hardware and guest OS, which is the only way for the guest VM to communicate with the hardware. However, emulating hardware resources imposes heavy performance overhead (McDougall and Anderson, 2010). In contrast, Docker containers utilize Linux's builtin features such as cgroups and namespaces to manage resources which have significantly less overhead (Li et al., 2017). Although containers excel at performance, their isolation is generally poor and expose a large attack surface (Combe et al., 2016) compared to VMs as they share the same host kernel. In 2017 alone, 454 vulnerabilities were found in the Linux kernel [1], which can be devastating for containerized environments. Public cloud service providers who base their services on containers are highly concerned since they have no control of the kind applications that are running in their cloud. Several organizations responded by developing lightweight hypervisors in order to strike a balance between traditional VMs and containers. As the current trend of container technology is rapidly reshaping the architecture of virtualization platforms, a comprehensive performance evaluation is important in considering the trade-offs of different approaches.

This paper takes a detailed look at lightweight hypervisors and compares the performance benchmarking results. We use the host machine as the baseline and measure different aspects of the computing system. The performance measurement is divided into two parts. Part I - the low level aspect which covers startup time, memory footprint, system call latency, network throughput, Disk I/O and CPU performance. Part II - the high level aspect which cov-

[a] https://orcid.org/0000-0001-9915-252X
[b] https://orcid.org/0000-0002-1607-5694
[c] https://orcid.org/0000-0003-0094-3984
[d] https://orcid.org/0000-0002-2919-6620
[e] https://orcid.org/0000-0001-6983-8336
[f] https://orcid.org/0000-0003-3931-9097

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

Li, G., Takahashi, K., Ichikawa, K., Iida, H., Thiengburanathum, P. and Phannachitta, P.
Comparative Performance Study of Lightweight Hypervisors Used in Container Environment.
DOI: 10.5220/0010440502150223
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 215-223
ISBN: 978-989-758-510-4

ers two real-world applications: Nginx webserver and MySQL database. Our goal is to understand the overhead imposed by KVM/QEMU, gVisor, Kata-qemu and Kata-firecracker. This paper makes the following three contributions:

- We provide an extensive comparative performance analysis of QEMU/KVM VM, Docker container, gVisor container, Kata-qemu and Kata-firecracker containers.
- We identify the best fit use case for practitioners by analyzing the pros and cons of each architecture in detail.
- We elaborate on the limitations of each architecture that affect virtualization performance.

The rest of the paper is organized as follows. Section 2 describes the motivation and architecture of each environment. Section 3 describes the evaluation results of low level and high level aspect across all environments. Section 4 presents the discussion and Section 5 reviews related work. Lastly, Section 6 concludes the paper and suggests possible future work.

2 BACKGROUND

2.1 Motivation

Various lightweight hypervisors are gaining traction. Because the trade-off between isolation and performance is almost inescapable, different organizations choose to optimize their hypervisors according to their requirements. Taking Firecracker as an example, its design is tailored to serverless applications (Baldini et al., 2017). It is crucial to understand (1) what trade-off each organization made and (2) the performance characteristics and limitations of each architecture to make informed decisions. The following subsection presents the overall architecture of each hypervisor.

2.2 Architecture of Each Lightweight Hypervisor

QEMU/KVM (Bellard, 2005; Kivity et al., 2007) represents the traditional VM architecture. QEMU is an emulator that is often used in conjunction with KVM. KVM is a Linux kernel module that acts as a hypervisor and take the advantage of hardware virtualization support such as Intel VT-x (Neiger et al., 2006) to improve performance. In our experiments, we make use of paravirtual drivers (i.e. virtio (Russell, 2008)) to improve disk I/O and network performance.

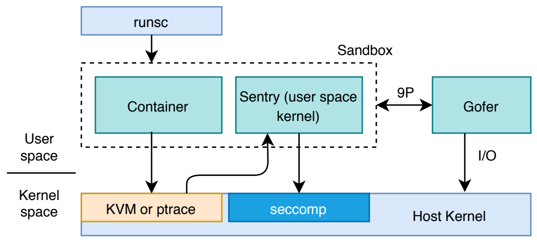

Figure 1: Architecture of gVisor.

Docker (Merkel, 2014) utilizes the builtin features of Linux such as *cgroups*,*namespaces* and Overlay FS[2] to isolate the resources between processes. It introduces little overhead since containers run directly on top of the host kernel.

Google's gVisor[3] is a combined user space kernel and lightweight hypervisor that provides an isolation layer between applications and host kernel. Figure 1 shows the high-level architecture of gVisor. The *Sentry* is a user space kernel that implements Linux system calls by using a limited set of system calls. To intercept the system calls from applications, gVisor provides two platforms[4]: ptrace and KVM. The ptrace platform uses the *ptrace* system call to intercept system calls whereas the KVM platform takes advantage of hardware virtualization support through KVM. System calls invoked from the Sentry are further filtered using *seccomp*[5]. File I/O is handled by a separate process called *Gofer*, which communicates with the Sentry through the 9P protocol (Pike et al., 1995). These layers of isolation imposed by gVisor enhance security but also increase performance overhead.

Amazon's Firecracker (Agache et al., 2020) began as a fork of Google's hypervisor crosvm[6]. Its high level architecture is similar to QEMU/KVM, but devices that are unnecessary in serverless applications (such as USB, video and audio devices) are not implemented thereby reducing memory footprint and attack surface.

Kata containers[7] is a common effort by the Intel Clear container[8] and hyper.sh[9] communities. It

[2]https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
[3]https://gvisor.dev/
[4]https://gvisor.dev/docs/architecture_guide/platforms/
[5]https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html
[6]https://chromium.googlesource.com/chromiumos/platform/crosvm/
[7]https://katacontainers.io/
[8]https://software.intel.com/content/dam/develop/external/us/en/documents/intel-clear-containers-2-using-clear-containers-with-docker-706454.pdf
[9]https://github.com/hyperhq

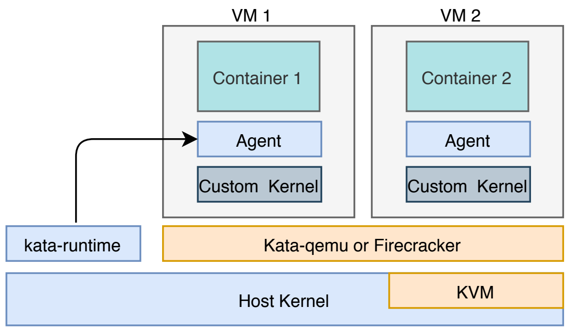

Figure 2: Architecture of Kata Containers.

is a container runtime that builds lightweight VMs which seamlessly integrate with the container ecosystem. Its default hypervisor is Kata-qemu[10], which is based on QEMU/KVM. However, significant optimization has been made. It comes with an highly optimized guest kernel which reduces boot time and memory footprint. Figure 2 shows Kata's high-level architecture. Each Kata container runs inside a dedicated lightweight VM to enforce strong isolation between containers. This VM is created and booted every time a new container is created. In addition, an agent runs on top of the guest kernel to orchestrate the containers. Recent releases of Kata support running VMs on Firecracker as well.

3 EVALUATION

We first evaluate the low level aspects of a computing system: startup time, memory footprint, system call latency, network throughput, disk I/O and CPU throughput. We then benchmark two common applications: Nginx webserver and MySQL database, which serve the purpose of confirming the CPU, Memory and Disk I/O benchmark results. 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 more realistic setting.

All tests were executed on a x86-64 machine with a six-core Intel Core i7-9750H CPU (hyper-threading enabled), 32GB of DDR4 SDRAM and 1TB of NVMe SSD. Table 1 shows the details of each execution environment.

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

Table 1: Execution Environment.

Environment	Software Versions
host	Ubuntu 19.10 (Eoan), Kernel 5.3
qemu_kvm	QEMU 4.4.0, libvirt 5.4.0, Guest Kernel 5.3
kata_qemu	Kata 1.11.0-rc0, Guest Kernel 5.4, QEMU 4.1.0
kata_fc	Kata 1.11.0-rc0, Guest Kernel 5.4, Firecracker v0.21.1
gV_kvm	gVisor release-20200115.0-94, Guest Kernel 4.4.0
gV_ptr	gVisor release-20200115.0-94, Guest Kernel 4.4.0
docker	Docker 1.0.0-rc10, containerd 1.2.13

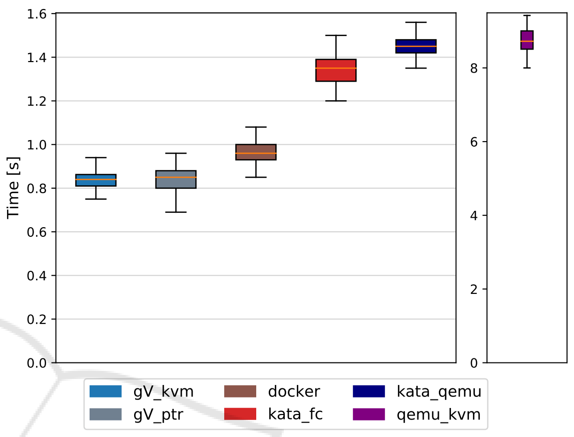

Figure 3: Startup Time.

3.1 Low Level Aspect

3.1.1 Startup Time

Fast startup time is crucial for container/VM provisioning (Mao and Humphrey, 2012). We used the `time` command to measure the time from launching a container/VM to the stage of the network stack being successfully initialized. Ubuntu:Eoan base image and `bash` program runs 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 3 shows the elapsed time of 100 complete container creations. Since the startup time for QEMU/KVM was significantly slower, the inaccuracies resulted from from the virtual clock is negligible.

On average, gVisor-kvm and gVisor-ptrace were 0.13 seconds faster than Docker. Kata-firecracker was 0.50 seconds slower than gVisor and Kata-qemu was only 0.10 seconds slower than Kata-firecracker. Overall, the startup time of all containers were significantly faster than QEMU/KVM. On average, It took 8.76 seconds for QEMU/KVM to finish booting the kernel and initializing the same user space.

Some important factors that affect start-up time are the size of the image, configuration files and etc.

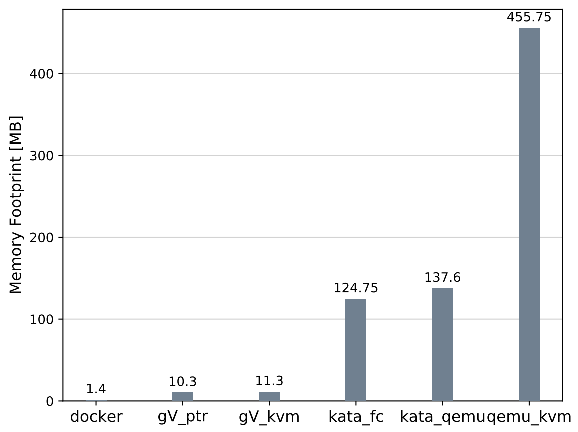

Figure 4: Average Memory Footprint of 100 containers.

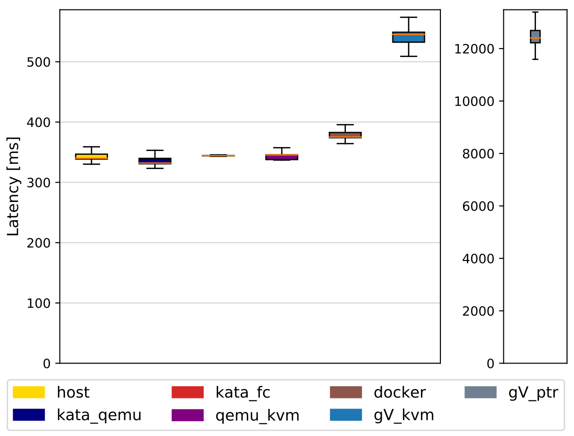

Figure 5: System Call Latency.

The time it takes to load these images and files from disk is directly proportional to its size. Docker and gVisor showed much faster startup time compared to Kata and QEMU due to the significantly smaller image size.

3.1.2 Memory Footprint

Smaller memory footprint allows the user to create higher density of containers. We quantified the memory footprint of each container running the Ubuntu:Eoan 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 infinity. We allocated 700MB 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 2048MB of memory for each VM.

As Fig. 4 indicates, Docker has the smallest footprint. This is because Docker does not run its own kernel and requires only one additional process (container-shim) to launch a container. Note that there is a daemon shared by all containers (dockerd), which uses around 40MB of memory. gVisor-ptrace and gVisor-kvm were comparable to Docker, but still used 7× more memory. Kata-qemu used slightly more memory than Kata-firecracker, but both consumed 88× more memory than Docker and 10× more than gVisor. Kata containers run their own guest kernel and many processes such as virtiofsd, qemu-virtiofs-system-x86, Kata-proxy and Kata-shim, which lead to significant larger memory footprint than both Docker and gVisor. However, none of these containers were comparable with QEMU/KVM. QEMU/KVM needed 325× more memory than Docker. Since the guest kernel used by Kata is highly optimized, it leads to much smaller memory footprint compared to a QEMU/KVM VM running a full Ubuntu:Eoan server distribution.

3.1.3 System Call

System call performance gives insight into the cost of user-kernel context switching. Figure 5 shows the total latency of invoking the `getpid()` system call for 10,000 times. There was a significant overhead with gVisor. In particular, gVisor-ptrace showed 36× larger larger latency than the host. This is a direct result of intercepting every system call using ptrace to examine the system call before passing it down to the host kernel. KVM makes gVisor-kvm faster than gVisor-ptrace because the use of KVM reduces the cost of system call interception. Nevertheless, gVisor-kvm's system call performance was still approximately 1.5× slower than the rest. Kata-qemu, Kata-firecracker and QEMU/KVM were comparable with the host. However, Docker performed slightly slower than the host on average. This is potentially caused by the use of *seccomp*, a kernel facility that restricts the system calls that can be invoked by applications.

3.1.4 Network Throughput

We measured the network throughput from the host to each virtual environment by using iPerf 3.6[11]. A TCP stream the from host to each virtual environment was generated for a total duration of 40 seconds. The graph in Fig. 6 shows the measured average network throughput. gVisor scored the worst on both KVM and Ptrace platforms. This attributes to its user space

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

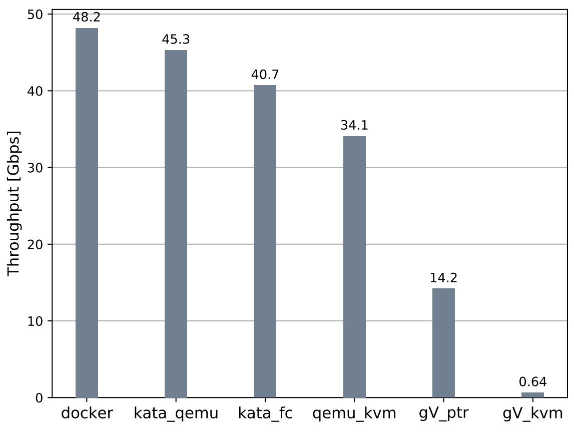

Figure 6: Network Throughput.

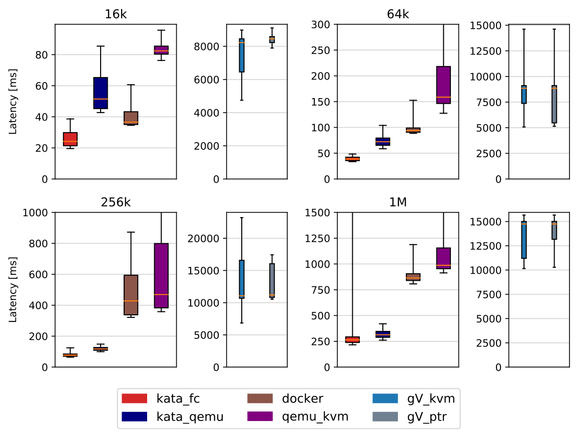

Figure 8: Sequential Write Latency.

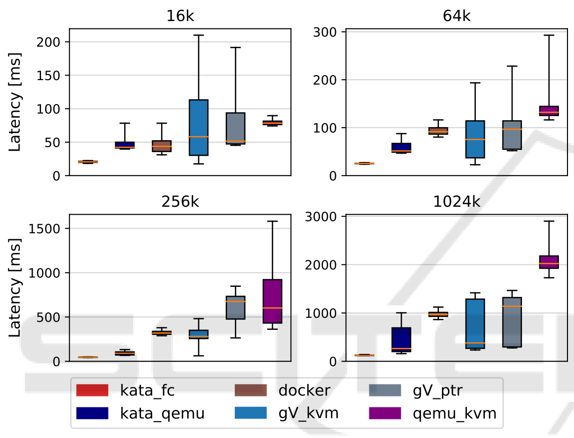

Figure 7: Sequential Read Latency.

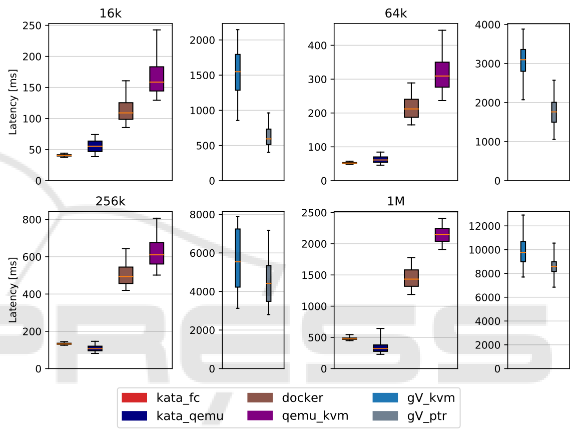

Figure 9: Random Read Latency

network stack *netstack*. Netstack handles all aspects of network stack in the user space to ensure isolation from the host network, but this design also introduces heavy overhead. It should be noted that recent releases of gVisor allow network pass-through so that containers can use host network stack directly at the cost of weaker isolation. gVisor-kvm is still in its experimental stage and the network performance was poor and therefore it should be used with caution. QEMU/KVM shows good network throughput thanks to the paravirtual virtio driver. Docker scored the best network throughput indicating that little overhead is imposed.

3.1.5 Disk I/O

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

Figure 7 demonstrates the sequential read latency for different block sizes. QEMU/KVM had the worst mean performance among all block sizes. In contrast, Kata-firecracker had the best mean performance and least variance compared to all others. In the case of 16KB block size, Kata-firecracker was 3.8× faster than QEMU/KVM, 2.8× faster than gVisor-kvm, 2.5× faster than gVisor-ptrace and almost 2× faster than both Kata-qemu and Docker. There was not much latency increase when the block size changed to 64KB. However, when we used 256KB block size, the latency increased sharply. Kata-firecracker was 13×

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

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

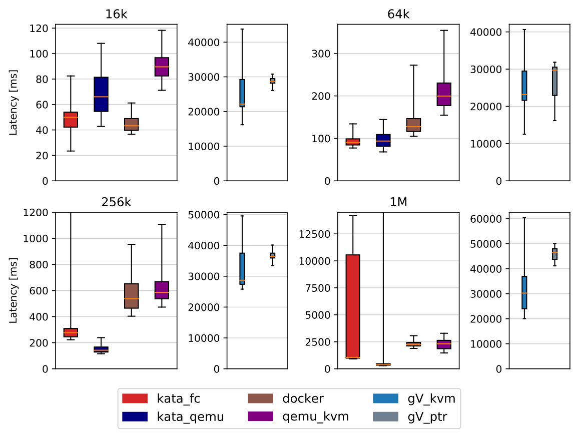

Figure 10: Random Write Latency.

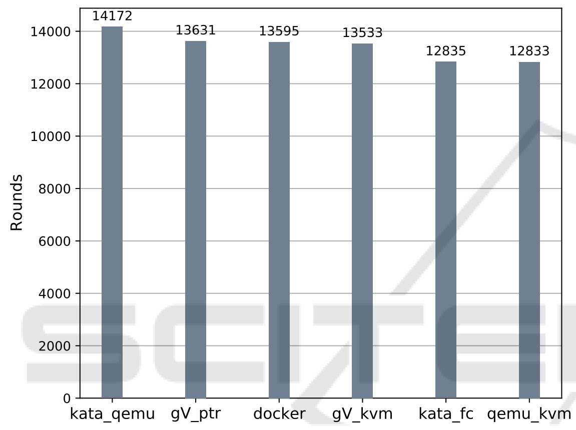

Figure 11: CPU Performance.

faster than QEMU/KVM, 15× faster than gVisor-kvm, 7× faster than Docker, 6.2× faster than gVisor-ptrace and 2× faster than Kata-qemu. Similar trend was observed with 1MB block size.

Both gVisor-kvm and gVisor-ptrace had massive performance overhead when it came to sequential write as indicated in Fig. 8. Kata-firecracker became the dark horse again in all cases, followed by Kata-qemu, Docker and QEMU/KVM. In the case of 16KB block size, both gVisor modes started from 103× slower than QEMU/KVM to 342× slower than Kata-firecracker. There was a similar trend with 64KB block size. The gap became half smaller when the block size is 1MB. QEMU/KVM showed a large variance at the fourth quartile in both 64KB and 256KB block. Kata-firecracker had significant variance at the fourth quartile with 1MB block.

Figure 9 shows the random read latency. It shows the similar trend to sequential read. However, Kata-qemu surpassed the performance of Kata-firecracker when the block size was greater than 64KB. In contrast to sequential read, gVisor-ptrace performed better than gVisor-kvm, but both still ranked the lowest.

As Fig. 10 indicates, Docker has the best performance with 16KB block size in the case of random write. Kata-firecracker and Kata-qemu outperformed Docker when the block size is bigger than 16KB. At the same time, Kata-qemu beats Kata-firecracker again. When the block size is 256KB and bigger. All environments except for gVisor showed similar performance results.

Kata container comes with a special device-mapper storage driver which uses dedicated block devices rather formatted file systems. Instead of using an overlay file system for the container's root file system, a block device can be used directly and map to the top read-write layer. This approach allows Kata container to excel at all aspect of Disk I/O performance.

In contrast, 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 (every file access, such as `open(path)` and `stat(path)`, requires a remote procedure call to the Gofer to access the file). gVisor engineers have started working on rewriting the current virtual file system to address this issue 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.

3.1.6 CPU

We used sysbench[14] 1.0.17 to measure the CPU performance. Figure 11 shows the total number of rounds finished verifying prime numbers between 2 to 200,000 using a single thread for a duration of 600 seconds (higher is better). The difference between hypervisors is small. This is expected since the hypervisor could utilize the CPU built-in hardware acceleration feature Intel VT-x.

3.2 High Level Aspect

3.2.1 MySQL

Sysbench 1.0.17 and MySQL[15] 8.0.20 were used to measure the database performance. We populated a table with 10 million rows of data and benchmarked the throughput and latency using a mixture

[14]https://github.com/akopytov/sysbench
[15]https://www.mysql.com/

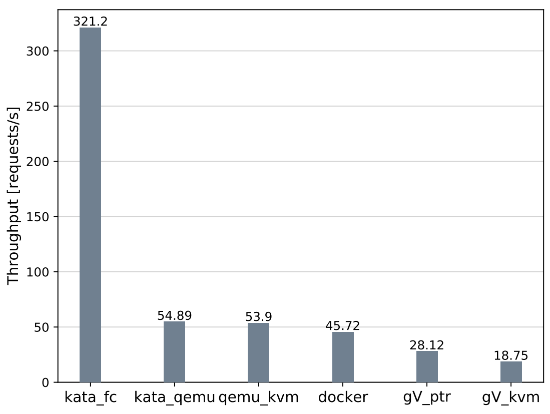

Figure 12: MySQL OLTP Throughput.

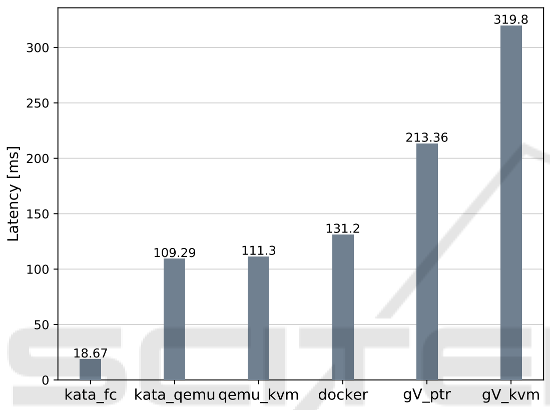

Figure 13: MySQL OLTP Latency.

of queries consisting of: Select (70%), Insert (20%), Update and Delete (10%) queries. Figures 12 and 13 show the throughput and average latency, respectively. gVisor-kvm and gVisor-ptrace 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.

3.2.2 Nginx with Static Web Page

We used wrk[16] 4.10 to measure the throughput of Nginx[17] serving a 4KB static web page. Each environment was configured to allow 1 worker process and maximum of 1024 concurrent connections. We used wrk to simulate HTTP GET requests for a duration of 10 seconds with 1,000 open connections.

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

[17]https://nginx.org

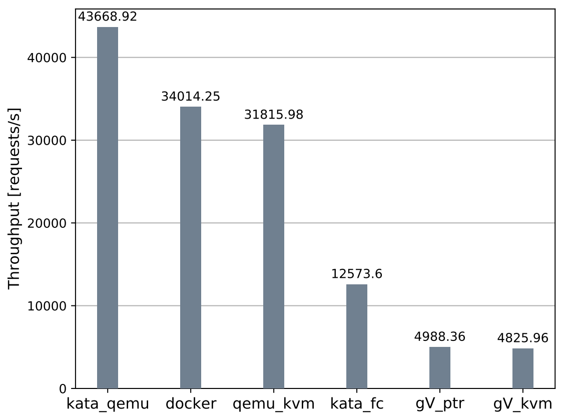

Figure 14: Nginx Webserver Throughput.

Figure 14 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 might attribute to its limitation of handling I/O serially (Agache et al., 2020). Both gVisor-kvm and gVisor-ptrace had almost the same throughput sitting at the bottom of the list.

4 DISCUSSION

As the benchmark results indicate, different system has different performance characteristics because of different motivation and purpose.

Google developed gVisor to mitigate the security risk of untrusted applications running in their public cloud. gVisor provides a secure container sandbox that has been built into Google's infrastructure to provide serverless computing services such as Google App Engine, Cloud Run and Cloud Functions. These services are used to run cloud native applications, which are small and loosely coupled. Apart from its wide deployment on Google's infrastructure, gVisor is seamlessly integrated with Docker and follows the OCI standard[18], which allow users to run Docker containers with better isolation. However, gVisor in both modes exhibits the worst performance in almost every aspect. Having a user space kernel adds an extra layer of isolation, but it also reduces application performance. In principle, gVisor-kvm should have performance advantage over gVisor-ptrace since KVM utilizes CPU's hardware virtualization support. gVisor-kvm is still in its experimental stage, and thus its performance result might change in the future. Databases or applications that require high performance net-

[18]https://opencontainers.org/

working should avoid using gVisor. In addition, gVisor only covers around 73% of the Linux system calls[19]. Therefore, applications that require the unimplemented system calls cannot be executed on gVisor. However, gVisor has much smaller memory footprint compared to Kata containers and QEMU/KVM VMs, which gives economic advantage since the user can launch more instances given the same amount of available memory.

Docker is not necessarily a good option for database applications due to the fact that its image is build up from a series of layers. Storage drivers such as device-mapper might perform poorly. However, Docker and gVisor containers showed much faster startup time and smaller memory footprint compared to Kata containers. Despite these performance advantages, Docker containers share the same kernel of the host. This might not be as secure as gVisor since gVisor provides a dedicated user-space kernel for each container. Docker still has very competitive advantages not only to mention its low memory footprint and fast startup time, but more importantly its ecosystem and methodologies (e.g. Docker Hub) which make deploying, testing and shipping applications much quicker.

QEMU/KVM clearly shows its heavyweight nature. Even with the performance boosts powered by paravirtual drivers, it is still not comparable with other systems, especially in terms of memory footprint and startup time. However, QEMU/KVM and traditional VMs are by far one of the most mature and best supported solutions. They are well tested and have been in the market for decades, and the strong isolation provided by traditional VMs is credible.

Kata-firecracker can be the best option for database applications. This is indicated by both disk I/O and MySQL benchmark results. Firecracker is the backbone of AWS Lambda[20] and Fargate[21] that power Amazon's serverless computing. It is commonly seen as an alternative lightweight hypervisor for Kata-qemu. Amazon implemented its own container runtime interfaces which does not comply with OCI standard. The good news is that Kata containers offers multi-hypervisor support which allow users to run Kata containers in firecracker based VMs.

Kata containers strikes a good balance between isolation and performance. Both Kata-qemu and Kata-firecracker have much less memory footprint comparing to a traditional QEMU/KVM VM and offers competitive performance advantage. Kata-containers can start many times faster than traditional QEMU/KVM VMs, which is mainly because the image we used to boot QEMU/KVM is bulky. If memory footprint is not a concern, Kata containers can easily beat all other options. Currently, Baidu Cloud AI has adopted Kata containers[22]. However, Kata container also has several limitations. At the time of writing, *SELinux* is not supported by Kata's guest kernel and joining an existing VM networks is not possible due to its architectural limitation[23].

[19]https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/

[20]https://aws.amazon.com/lambda/

[21]https://aws.amazon.com/fargate/

5 RELATED WORK

IBM (Felter et al., 2015) studied the CPU, memory, storage and network performance of Docker containers and VMs, and elaborated on the limitations that impact virtualization performance and concluded that containers result in equal or better performance than VMs in all aspects. However, (Li et al., 2017) disagreed with IBM's conclusion and they investigated the performance variability and found out that container's performance variability overhead could reach as high as 500%. (Kozhirbayev and Sinnott, 2017) adopted a similar methodology as IBM; however, they targeted Docker and its rival Flockport (LXC). Their results were similar to IBM's, but they pointed out that Docker allows only one application per container, which reduces utilization, whereas Flockport does not impose such restriction.

On top of all, none of these previous works studied the memory footprint and boot time, which are critical to the cloud. More importantly, container technologies have been marching forward over the years, and a hybrid approach—running containers inside of lightweight VMs is becoming a promising alternative. Yet there has been no existing research providing detailed analysis. Our research focuses on those emerging technologies backed up by Google, Amazon, Baidu and Intel, and gives the community a better picture of the current technological trend.

6 CONCLUSION AND FUTURE WORK

We have conducted a comprehensive performance analysis of various popular lightweight hypervisors

[22]https://katacontainers.io/collateral/ApplicationOfKataContainersInBaiduAICloud.pdf

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

that are backed by Docker, Amazon, Google and Intel. The benchmark results showed that various tradeoffs are made by each organization and a number of bottlenecks that affect virtualization performance are identified. The pros and cons of each system are discussed in detail and some limitations that could be potentially addressed in the future are also pointed out. It is evident that the current architectural trend of lightweight hypervisors tends to march forward a new era and lightweight VMs might have the potential to partially replace the role of traditional VMs. However, these lightweight hypervisors have not reached the point to become a mature alternative, and thus traditional VMs would still be the preferred options for many organizations. Kata is on the right track to earn the title of having the security of a VM and offering the performance of a container.

Future research on reducing the memory footprint of lightweight hypervisor based containers would be practical. A research using optimized Xen hypervisor with customized unikernels (Manco et al., 2017) opens the possibility of creating lighter and safer VMs than containers is worth looking into.

REFERENCES

Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. (2020). Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 419–434.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer.

Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC'05)*, pages 41–46.

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

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. (2007). KVM: The Linux virtual machine monitor. In *the Linux symposium*, volume 1, pages 225–230.

Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175–182.

Li, Z., Kihl, M., Lu, Q., and Andersson, J. A. (2017). Performance overhead comparison between hypervisor and container based virtualization. In *IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962.

Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. (2017). My VM is lighter (and safer) than your container. In *26th Symposium on Operating Systems Principles (SOSP'17)*, pages 218–233.

Mao, M. and Humphrey, M. (2012). A performance study on the VM startup time in the cloud. In *IEEE Fifth International Conference on Cloud Computing (CLOUD 2012)*, pages 423–430.

Matthews, J. N., Hu, W., Hapuarachchi, M., Deshane, T., Dimatos, D., Hamilton, G., McCabe, M., and Owens, J. (2007). Quantifying the performance isolation properties of virtualization systems. In *2007 Workshop on Experimental Computer Science*, pages 6–es.

McDougall, R. and Anderson, J. (2010). Virtualization performance: perspectives and challenges ahead. *ACM SIGOPS Operating Systems Review*, 44(4):40–56.

Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. (2006). Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3).

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from Bell Labs. *Computing systems*, 8(2):221–254.

Russell, R. (2008). Virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103.