Cluster Computing (2022) 25:1497–1513
https://doi.org/10.1007/s10586-021-03517-8

Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes

Xingyu Wang[1] · Junzhao Du[1] · Hui Liu[1]

Received: 8 March 2021 / Revised: 21 October 2021 / Accepted: 5 December 2021 / Published online: 22 January 2022

Abstract
Containers are resource-efficient and most IT industries are adopting container-based infrastructure. However, the security and isolation of the container is rather weak. In this work, we aim to conduct an in-depth quantitative analysis of the performance characteristics of containerization technologies that strengthen container isolation and security, and discuss the applicable scenarios of various containerization technologies. We evaluate multiple cloud resource management dimensions of RunC, gVisor, and Kata Containers runtimes, including performance, system call, startup time, density, and isolation. Experimental results show that RunC and Kata Containers have less performance overhead, while gVisor suffers significant performance degradation in I/O and system call, although its isolation is the best. Our work deepens the understanding of the container performance characteristics and may help cloud computing practitioners in making proper decisions on platform selection, system maintenance and/or design.

Keywords RunC · gVisor · Kata Containers · Performance evaluation · Isolation

1 Introduction

Currently, container-based virtualization has been widely used in cloud computing, but the limitations of this technology are also obvious. The advantages of containers are fast startup and low performance overhead, but the container mechanism of the operating system itself cannot solve the security and isolation problems of containers [1]. Some researchers and developers focused on designing a true sandboxed container that is as isolated as possible from the host OS. Strengthening the isolation and security of containers is usually at the expense of performance. Therefore, it is necessary to comprehensively evaluate the latest sandbox container technologies to understand their trade-offs between performance, isolation, and security.

To strengthen the isolation and security of containers, most solutions design a hybrid architecture that leverages the strong trust boundary from the VM and focuses on the high efficiency of the container. IBM Nabla [2] and LightVM [3] build containers on top of Unikernels. Google gVisor [4] creates a dedicated guest kernel for running containers. Amazon Firecracker [5] is an extremely lightweight hypervisor for sandbox applications. Kata Containers [6] are placed in a dedicated VM optimized for the container orchestration platform. The aforementioned lightweight isolation platforms establish a stronger security boundary for the container than the namespace isolation by adding an extra layer between the container and the host OS, but at the same time, they also introduce additional performance overhead compared to the native container. Therefore, the above solutions need to be fully evaluated to better understand the trends, design trade-offs, and basic limitations. References [7–10] study the CPU, memory, storage, and network performance of RunC, gVisor, Firecracker, and Kata Containers, as well as the execution of the host kernel code by the container. The disadvantage of the above works is that there is no comparison and analysis of system call, startup time, density, and container isolation. Our work conducts a more comprehensive evaluation

✉ Junzhao Du
dujz@xidian.edu.cn

✉ Hui Liu
liuhui@xidian.edu.cn

Xingyu Wang
spaceimagine@163.com

[1] School of Computer Science and Technology, Xidian University, Xi 'an, China

of the above container runtimes, including performance, system call, startup time, density, and isolation.

This paper aims to quantitatively analyze the performance of the lightweight isolation platforms in terms of startup time, density, and isolation, and the performance overhead introduced while strengthening container isolation and security. We choose RunC, gVisor, and Kata Containers for comparison. First, we introduce the sandboxed container technologies based on Unikernel and MicroVM. Second, we use a set of representative benchmarks to evaluate the CPU, memory, storage, and network performance overhead of different container runtimes. Third, we measure system call, startup time, and density metrics of the container runtimes. We also evaluate the isolation of the lightweight isolation platforms. Although the isolation and security of containers are constantly improving, it is difficult to find a solution with the best security and performance at the same time. Our work will provide users with a wider range of choices, not just choosing between the strong security and high performance of the container.

The remainder of this paper is organized as follows. Section 2 provides an overview of the related research. Section 3 introduces the Unikernel-like and MicroVM-based sandboxed container technologies. Section 4 describes the performance test methods and performance metrics. In Section 5, we give the CPU, memory, storage, and network benchmark results, as well as container startup time, density, system call, and isolation. Concluding remarks are presented in Section 7.

2 Related work

In recent years, many researches have evaluated different aspects of hardware-based virtualization and operating system-based virtualization technologies. First we review the research work on the performance comparison of virtual machines and containers. Then we focus on the performance researches related to the latest sandboxed containers, such as Nabla, gVisor, Firecracker, and Kata Containers.

In [11], the authors evaluated the performance of Docker and Flockport [30] (LXC). Experimental results show that Docker and Flockport have incurred some performance overhead in terms of I/O and operating system interaction. In a traditional virtual machine environment, performance isolation is usually measured based on the performance loss rate. Zhao et al. proposed a performance isolation measurement model that combines the performance loss and resource shrinkage of containers. This model is more likely to reflect the trend of container performance isolation [12]. Tesfatsion et al. analyzed the virtualization performance overheads of KVM, Xen, LXC, and Docker. They believed that no system can provide the best results for all the metrics of interest [13]. Mavridis et al. studied the performance and power consumption of running containers (Docker) on virtual machines (KVM, XEN, and Hyper-V). Combining containers with traditional virtual machines not only enhanced the isolation and security of containers, but also incurred additional performance overhead. Therefore, virtual machines must be highly optimized [14]. Chae et al. investigated the performance of web servers by gradually increasing the number of running containers and virtual machines, large file replication, and different concurrency settings. Experimental results showed that Docker is faster than KVM and uses fewer resources [15]. Shih et al. studied the execution time of the same workload in three environments (bare metal, Docker container, and virtual machine) under the big data processing scenario to understand the difference between the characteristics of each environment [38]. The results showed that, as lightweight virtualization technology, containers create less runtime overhead for applications than VMs. Bhatt et al. studied the impact of CPU-intensive load execution in a coexisting virtual machine on the userspace file system running in the virtual machine [37]. The author tested the impact of block size and cache management on the performance of file read and write operations.

Espe et al. evaluated the performance and scalability of containerd and CRI-O runtimes on RunC and gVisor. The evaluation results highlighted CRIO/RunC as the best option almost for any use-case, whereas Containerd/RunC turned out to excel at I/O-heavy workloads [16]. Ethan G et al. conducted a comparative analysis of the runtime performance of RunC and gVisor. The analysis results showed that, compared with the runc, gVisor has larger system calls, memory allocation, and I/O performance overheads [17]. Kumar et al. analyzed the performance of Docker and Kata Containers. Kata Containers improves the security of containers, but due to its architecture, the performance is affected [7]. Williams et al. proposed that when Unikernel runs as a process, an isolation level similar to that of a virtual machine can be achieved, and the performance in terms of throughput, startup time and memory density can be greatly improved [2]. Agache et al. compared the startup time, memory usage, and I/O performance of Firecracker, IntelCloud Hypervisor [34] and QEMU. Firecracker performs better than QEMU in terms of startup time and memory usage, but its I/O performance needs to be improved [18]. Manco et al. achieved lightweight VMs by using unikernels for specialized applications. The study showed that lightweight virtualization achieves good isolation and performance equivalent to or better than that of containers at the same time [3]. Anjali et al. conducted a

system-wide Linux code coverage analysis and benchmark test for LXC, Firecracker and gVisor. Their results of the study revealed that although many functions have been moved out of the kernel, both Firecracker and gVisor execute more kernel codes than native Linux [8]. Debab et al. studied operating system virtualization (Docker [19], Podman [20], Rkt [21]), MicroVM-based containerization (WSL2 [22], Firecracker [5], Kata Containers [6]), system call virtualization (gVisor [4]) and Unikernels (IBM Nabla [23]) performance characteristics of the containerization technology [9]. Viktorsson et al. investigated the deployment time and performance of applications (TeaStore, Redis, Spark) in RunC, gVisor, and Kata Containers, showing that higher security pays a high price [10]. Zhao et al. performed a comprehensive characterization and redundancy analysis on the images and layers stored in the Docker Hub registry [36]. The research results showed that there is great potential for file-level data deduplication for large-scale registries. In addition, this work further studied the I/O performance of image retrieval and storage drivers.

Although the above works have compared the performance of RunC, gVisor, and Kata Containers from different perspectives, they have not analyzed their density and isolation. Our work analyzes the performance, system call, startup time, density, and isolation of RunC, gVisor, and Kata Containers, so as to reflect the advantages and disadvantages of these container technologies more comprehensively.

3 Sandboxed container technologies

The security issues of traditional containers have motivated research on true sandboxed containers. Most solutions adopt a hybrid architecture that combines the strong isolation of virtual machines with the high efficiency of containers. We divide related technologies into the Unikernel-like and MicroVM-based sandbox container technologies.

3.1 Unikernel-like sandbox container technology

In cloud computing, general-purpose operating systems are usually used to build virtual machines. The general-purpose operating systems can run more types of applications, but are not optimized for virtualized environments. Unikernel packages applications and their related kernel functions into an image and can run directly on the hypervisor. The advantages of Unikernel are its improved security, small footprint, high optimization, and fast startup.

Nabla IBM's Nabla container project uses Unikernel dedicated monitors, allowing fewer system calls and significantly improving security. Besides, running Unikernel processes in a dedicated virtual machine monitor (Nabla Tender) can reduce hardware requirements and allow the use of standard process debugging and management tools [23]. In Fig. 1, Tender intercepts the hypercalls sent by Unikernels to the hypervisor and converts them into system calls. The Linux seccomp policy prevents all other system calls that Tender does not need. Nabla also provides the Nabla runtime RunNC that complies with the Open Container Initiative (OCI) standard. Due to the different file systems between Unikernels and traditional containers, Nabla images do not conform to the OCI image specification, so Docker images are not compatible with RunNC.

gVisor Google's gVisor is essentially a combination of the client kernel and VMM. What gVisor and Nabla have in common is that they both strengthen the security of the container by protecting the host OS. They all use less than 10% of Linux system calls to interact with the host kernel. gVisor creates a multi-purpose kernel, and Nabla relies on Unikernels, both running a dedicated client kernel in user space to support sandbox applications. In Fig. 2, gVisor intercepts all system calls made by the application to the host kernel, and uses the gVisor kernel implement Sentry in user space to process them, thereby sandboxing the application. The Gofer process provides the container with access to the host file system [4]. Besides, gVisor includes an Open Container Initiative (OCI) runtime called RunSC.

3.2 Sandboxed container technology based on MicroVM

Although VMs create strong isolation for containers in the public cloud, using generic VMMs and VMs for sandbox applications is not resource-efficient.

Firecracker Firecracker minimizes the possibility of damage from untrusted applications through a security boundary layer. Firecracker VMM provides minimum operating system functions and simulated devices for each guest VM to improve security and performance. Firecracker processes strictly restrict system calls, hardware resources, file systems, and network activities through

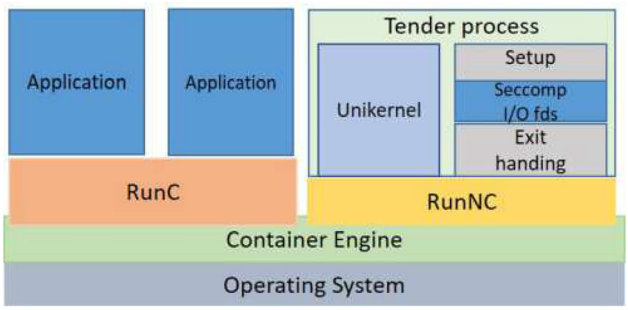

Fig. 1 Nabla architecture

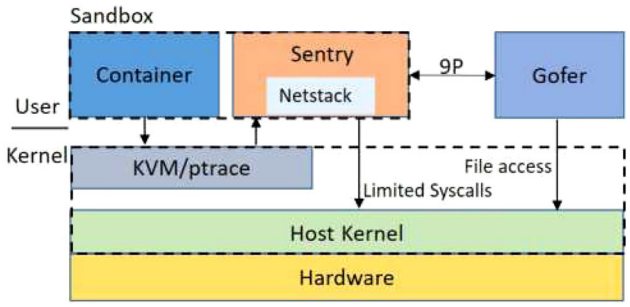

Fig. 2 gVisor architecture

seccomp, cgroup, and namespace. In Fig. 3, each Firecracker process runs the following threads: API, VMM, and vCPU. The VMM thread exposes the machine model, device model, micro virtual machine metadata service, and VirtIO device simulation network and block devices, and has the I/O rate limit function.

Kata Containers Although Kata Containers and Firecracker are both VM-based sandboxing technologies, their implementation methods are completely different. Firecracker is a dedicated VMM that creates a secure virtualized environment for guest OS, while Kata Containers is a lightweight VM highly optimized for running containers. As shown in Fig. 4 [6], Kata Containers run in a virtual machine sandbox, and each virtual machine runs an Agent. Kata-runtime is responsible for processing all commands in the OCI runtime specification and starting the Kata-shim process. The Kata-runtime on the host uses the gRPC protocol to communicate with the Agent and sends instructions to the container in the virtual machine. The hot plugging feature allows the VM to start with minimal resources (such as CPU, memory, virtio block), and add other resources when required later. In addition, Kata runtime complies with OCI and CRI specifications and can be orchestrated and scheduled by Kubernetes.

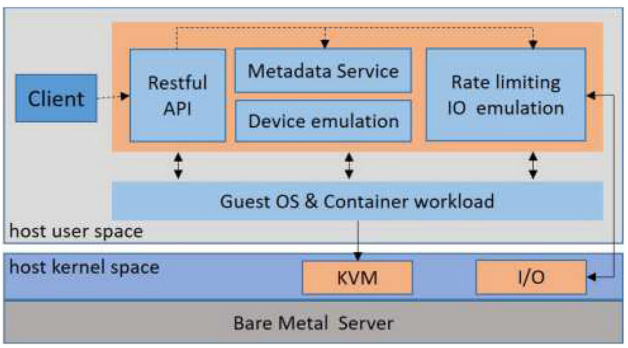

Fig. 3 Firecracker architecture

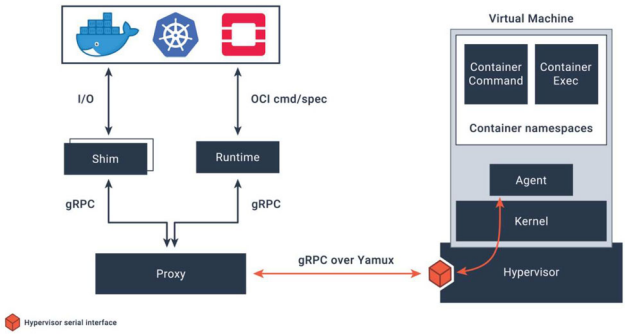

Fig. 4 Kata Containers architecture

4 Metrics and measurement methods

4.1 Virtualization overhead metrics

The virtualization performance overhead is quantified according to the performance loss of the virtualization system relative to the physical environment, and is specifically defined as:

$$perf_{ovh} = \frac{|perf_{virt} - perf_{native}|}{perf_{native}} \quad (1)$$

where $perf_{ovh}$ is the performance overhead of the virtualization system, which is calculated by dividing the performance loss of the virtualization system ($perf_{virt}$) relative to the performance of the physical environment by the performance of the physical environment ($perf_{native}$).

4.2 Measurement methods

We use a set of test scripts to perform the automatic test of all performance metrics. Our test methods are described below according to different test items. Table 1 summarizes the workloads and metrics used to evaluate the different container runtimes.

4.2.1 Benchmarks

Our benchmarks measure the performance of CPU, memory, storage, network, and system calls. Figure 5 shows the automated experiment procedures by using scripts. The following are the detailed steps of the benchmark:

(1) Log in the system under test (SUT) and configure the container environment based on the user inputs. The scripts first check the container's runtime, read its configuration information, build the container image, and install the necessary shell tools.
(2) Before running the test, delete all containers on the host to ensure that subsequent test results are not affected.

Table 1 Benchmarks and performance metrics

Test item	Benchmark	Metric
CPU	Sysbench	Events Per Second
	Dav1d	PFS(Frames Per Second)
Memory	RAMspeed/SMP	Throughput(MB/s)
	Redis	Requests Per Second
Disk	IOzone	Throughput(MB/s)
	Sqlite	Seconds
	Blogbench	Read/Write score
Network	Netperf	Throughput(KB/s)
		Transaction Rate Per Second
	Etrh	Throughput(KB/s)
		Latency(us)
Startup Time	Shell Script	Seconds
Density		KB
System Calls	Unixbench	Number of executions
Isolation	Linpack	GFlops
	Shell Script	CPU utilization(%)

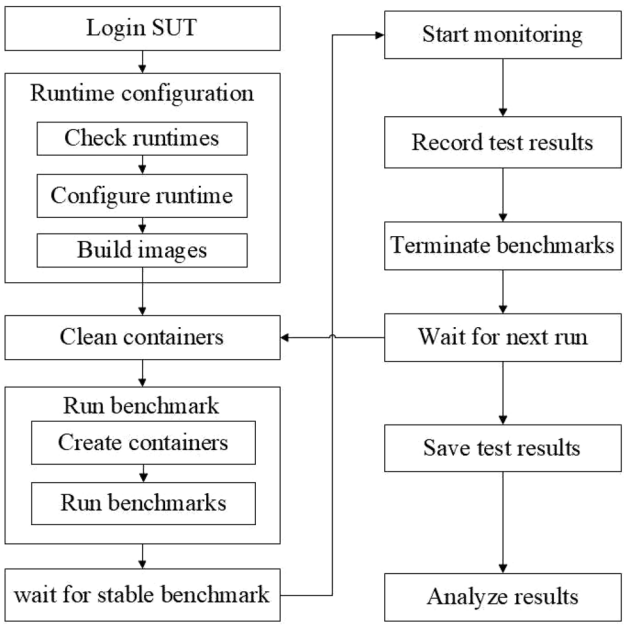

Fig. 5 Automated performance testing process

(3) Create test containers, set container resource limits, benchmark parameters, etc., and execute the script in the containers or on the host (such as recording container startup time or resource utilization, etc.).
(4) After a period of time specified by the user, start recording the test results, such as the output of the benchmark, the resource utilization of the container, and other information.
(5) Upon completing the test and recording resource utilization and performance metrics, trigger the termination of the benchmark and monitoring scripts.
(6) To increase the reliability of the test results, repeat steps (2)-(5) for a number of times (10 times by default in this work) at intervals specified by the user.
(7) Upon completing the specified number of repeated tests, save all the results in a file.
(8) Process the test results and calculate the metrics such as mean, variance, performance loss, and resource utilization.

4.2.2 Startup time and density

We use scripts to measure the startup time metric for different runtime containers. The test script first creates a container based on the configuration information input by the user, such as container runtime, image, and resource limits. Then the script starts the container, which exits after outputting the container start time. The test is completed after repeating the above steps for a user-specified number of times (50 repeats in this work). Finally, the script saves the collected data in a file in json format. In addition, the script can also evaluate the impact of the number of containers running on the host on the container startup time. Specifically, we continue to increase the number of containers running on the host as we measure the container startup time. When the number of containers scales to the number set by the user, the test is terminated.

The test procedures for density and startup time are roughly the same. We use scripts to measure the memory footprint of runtime components of RunC, gVisor, and Kata Containers. We keep increasing the number of containers running on the host (50 by default) and calculate the total memory usage of all containers. The smaller the memory footprint of a container, the more containers can be accommodated on the host at the same time, and the greater its density is.

4.2.3 Isolation

A system is performance-isolated, if for customers working within their quotas the performance is not affected when other customers exceed their quotas [32]. The container system mainly consists of two types of containers: affected containers and overloaded containers. If a container's workload is increasing abnormally and affecting the performance of other containers, we call this container an overloaded container [12]. We evaluate the impact of contention for host resources due to overloaded containers

on the performance and resource utilization of affected containers.

In the isolation test, we first measure the performance and resource utilization of the container before it is interfered. Then, we start the overloaded container and run the Linpack benchmark in it. At the same time, we start the monitoring tools to record the performance and resource utilization of the two containers.

We use the performance loss rate [39, 40]to measure the isolation of the container, p_i and p_j respectively represent the pre-interference performance and post-interference performance of the specific container to be observed. The greater the performance loss rate, the worse the performance isolation of the container.

$$I = \frac{|p_i - p_j|}{p_i} \tag{2}$$

5 Experimentation

All the experiments were performed on physical machines with a 3.8 GHz quad-core Intel Core i5-7500 CPU, 8 GB RAM, and 1T hard disk, supporting nested virtualization. We used the Ubuntu 18.04 LTS (5.4.0) Linux distribution as host. Virtualization was achieved using Docker 20.10.1, KVM 2.0.0, gVisor(20201030.0), and Kata Containers 1.12.0-rc0. We create the containers through Docker. In addition, all containers adopt the default configuration. Unless otherwise stated, we do not impose any restrictions on the resources used by the container. In order to measure the performance overhead of different container technologies, we use the performance of the bare metal as the baseline for performance evaluation.

5.1 CPU

We tested the computational performance of RunC, gVisor, and Kata Container runtimes. It is expected that there should be a small loss in computing performance of the above three container runtimes. We evaluated the CPU performance with the sysbench [25] benchmark. We configured sysbench to calculate 20,000 prime numbers with four threads and report the number of events executed per second as the performance metric. Besides, we used Dav1d [24] to perform the CPU performance test in typical video decoding application scenarios. Dav1d is an open-source AV1 video decoder which contains four test videos: Summer Nature 1080p, Chimera 1080p, Summer Nature 4K, and Chimera 1080p 10-bit. We choose Summer Nature 1080 as the test video.

We can see from Fig. 6 that when RunC, gVisor, and Kata Containers perform CPU-bound workload, their CPU

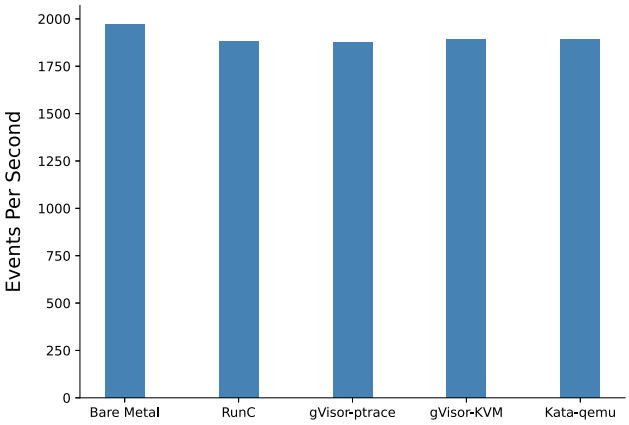

Fig. 6 CPU performance(Sysbench benchmark)

performance is almost the same. Specifically, the performance loss of RunC and Kata QEMU containers are 4.67% and 3.97%, respectively. The CPU performance of the gVisor KVM platform container is slightly better than gVisor ptrace platform. gVisor does not perform emulation or otherwise interfere with the raw execution of CPU instructions by the application. Therefore, there is no runtime cost imposed for CPU operations [4]. Figure 7 shows the Dav1d video decoding performance. Compared with other container runtimes, the gVisor ptrace container exhibits the worst performance (203.5 FPS), with a performance overhead of 13.26%. This may be due to the need to read the video file from the disk in the video decoding task. The video decoding performance of the gVisor KVM container is equivalent to that of the bare metal. The test results in Sect. 5.3 demonstrate that the file operation performance loss of the gVisor ptrace container is higher than that of the gVisor KVM container. There is almost no difference in performance between RunC and Kata Containers, and the performance drops by 4.37% and 3.99%, respectively. In summary, when performing compute-

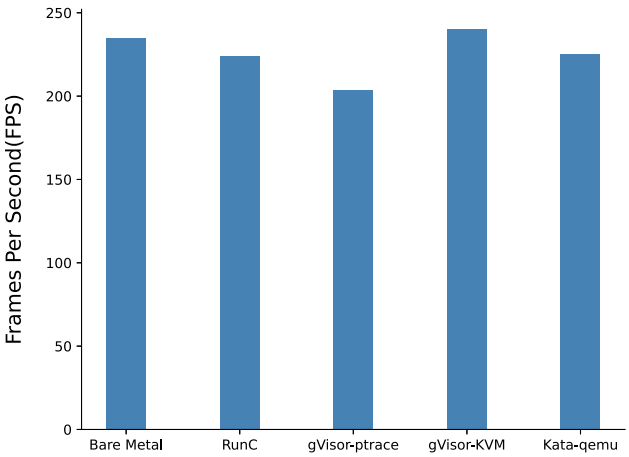

Fig. 7 CPU performance(Dav1d benchmark)

intensive loads and emphasizing the security of the container, the Kata Containers is the best choice.

Not surprisingly, consistent with the test results of [9], the computing performance of RunC, gVisor and Kata Containers are almost the same as that of the bare metal. Although the computational performance loss of the container runtimes mentioned above is very small, for mixed loads, such as in video decoding application scenarios, the performance overhead of the gVisor ptrace container is the largest. It follows from the experimental results in Sect. 5.8 that the advantage of the gVisor container is its strong isolation with the host system and with the coexisting container. Therefore, for applications with special requirements for security, the gVisor container is the first choice. Otherwise, the RunC and Kata Containers are better choices.

5.2 Memory

Due to the support of modern processor hardware virtualization extensions, combined with the design principles and architecture analysis of the sandbox container technology in Sect. 3, we can expect that RunC, gVisor, and Kata Containers should have a little memory performance overhead. In order to have a more in-depth understanding of the memory performance overhead of the container runtime, we separately tested the access and allocation performance of the container memory. We use RAMspeed and Redis benchmarks to evaluate the memory performance of different container runtimes. RAMspeed consists of four sub-tests (Copy, Scale, Add, Triad) and supports multi-threading. The Redis benchmark is used to test the performance of the LPOP, SADD, LPUSH, GET, and SET operations of the Redis database for different container runtimes.

As illustrated in Fig. 8, the memory bandwidth of RunC, gVisor, and Kata Containers is equivalent to that of bare metal. Unlike the memory bandwidth test results, there are apparent differences in the Redis operation performance of different container runtimes. From the Fig. 9, we can see that the performance of RunC and Kata Containers is almost the same, and the worst performance is the gVisor ptrace container. The performance loss of gVisor ptrace and KVM platform is 95.38% and 56.07%, respectively. Since the memory allocation of the RunC container is entirely executed by the host OS and its raw memory access performance is almost the same as that of the physical machine, the Redis performance of the RunC container is the best. gVisor implements the interception of application system calls through ptrace and KVM. The context switch caused by the interception and redirection of system calls will introduce performance overhead. The gVisor KVM platform can take advantage of the virtualization extensions available on modern processors to improve the isolation and performance of address space switching. In addition, although gVisor implements memory management in Sentry, it still relies on the host to implement this function like LXC [8]. Frequent memory allocation will increase the overhead of Sentry's system calls to the host OS, but once the memory allocation is complete and available for use by applications, there is no additional overhead.

The raw memory access of the above container runtimes hardly introduces any additional performance overhead, which is in line with our expectations. However, the performance loss of the Redis benchmark of the gVisor container surprised us. The design of gVisor emphasizes security. While gVisor increases security, it also introduces additional performance overhead (system call overhead). The structural cost brought about by this design choice cannot be eliminated. Therefore, applications that

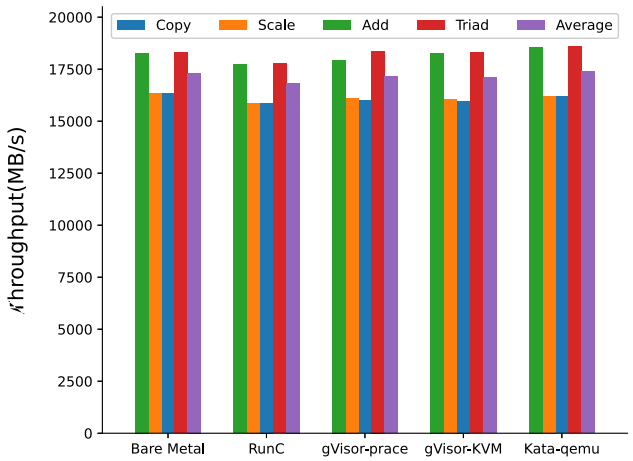

Fig. 8 Memory access performance

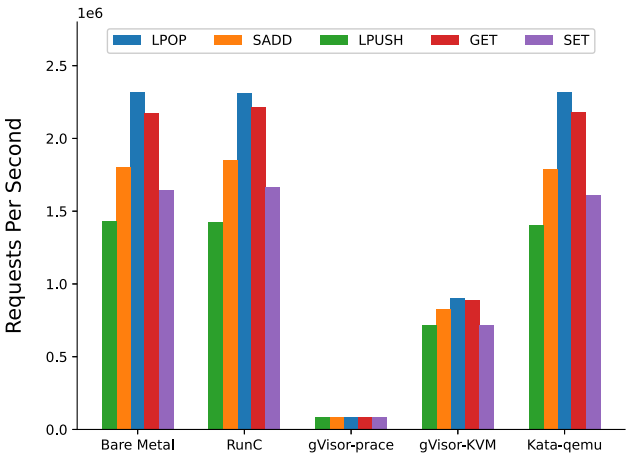

Fig. 9 Redis performance for different container runtimes

frequently allocate and free memory may experience significant performance loss on gVisor. For developers, it is first necessary to improve the memory management efficiency of the gVisor container. In addition, by reducing the degree of participation of the host system in the memory management of the container, the performance overhead caused by context switching is reduced and the isolation of the container is improved.

5.3 Storage

For virtualization based on hardware abstraction, I/O performance has always been its bottleneck. The RunC container can almost directly access the system hardware, so it should have a slight I/O performance loss. gVisor implements container file system access through a separate Gofer process and 9P protocol, so we expect that its disk I/O operations should introduce additional performance overhead. Kata Containers is a virtual machine first, so we expect its I/O performance loss will be more significant. In what follows, we will verify the above discussion through experiments.

We measure I/O throughput with IOzone [29] that performs reads and writes of various sizes on a 16 GB file. Before each measurement, we flush the file to disk and drop the file cache. In addition, we measure the I/O performance of the database using SQLite [31] benchmark. SQLite reports the time to perform a pre-defined number (2500) of insertions on an indexed database.

Figure 10 shows the disk I/O performance test results. In all cases, the throughput achieved using RunC containers is slightly lower than that for bare metal. The performance of gVisor is second only to the performance of RunC. According to the gVisor security model, the file system access of the gVisor container must be routed through Gofer, so gVisor introduces a small fixed overhead for the data that transitions across the sandbox boundary. Besides, we notice that the gVisor KVM platform has better file read and write performance than the ptrace platform. The KVM platform uses the kernel's KVM functionality to allow sentry to act as a guest operating system and VMM at the same time, thus reducing the system call overhead of sentry [4]. The performance loss of Kata Containers is the largest, and the performance overhead of file reading and writing is 12.4% and 16.7%, respectively. Kata Containers uses virtio-9p as the default file sharing mechanism [6]. virtio-9p is based on existing network protocols, which are not optimized for virtualization use cases. Therefore, their performance is not as good as the local file system. In addition, we also measured the I/O performance of the tmpfs file system. As shown in Fig. 11, the test results show that the tmpfs I/O performance of the Kata Containers and RunC is comparable to that of the bare metal. The performance loss of gVisor ptrace and KVM containers are 35.53% and 13.51%, respectively. Since the tmpfs only persisted in the memory, the above test results reflect performance overhead similar to copying data in memory.

Figure 12 shows the performance characteristics of the container runtimes in the database operation scenario. We set SQLite to use a single thread to perform database record insertion operations. As shown in Fig. 10, the performance of RunC database record insertion is almost the same as that of bare metal. The performance overhead of Kata Containers database operations is 16.96%. gVisor shows the worst performance with a performance overhead of 125.17%. Consistent with the conclusion described in [4], when gVisor frequently performs file operations with small pieces of static content, VFS operations exhibit high performance overhead.

To further verify the performance test results of the container runtimes mentioned above in the frequent small file operation scenario, we used Blogbench [35] to test the file operation performance in the concurrent environment.

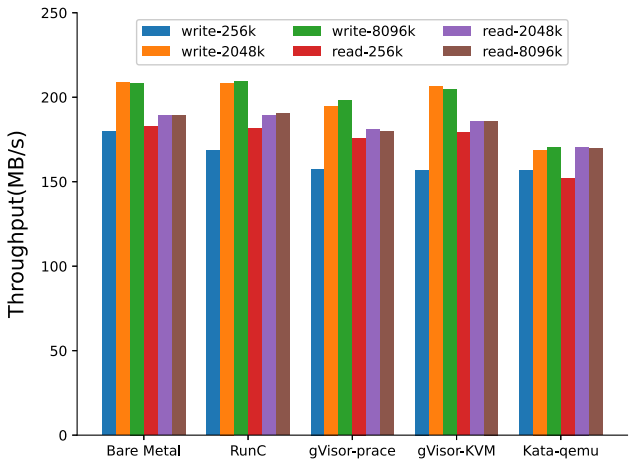

Fig. 10 Disk read and write performance

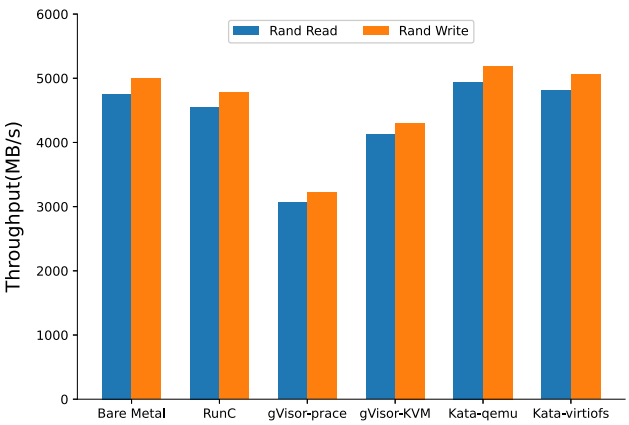

Fig. 11 Disk read and write performance(tmpfs overlay)

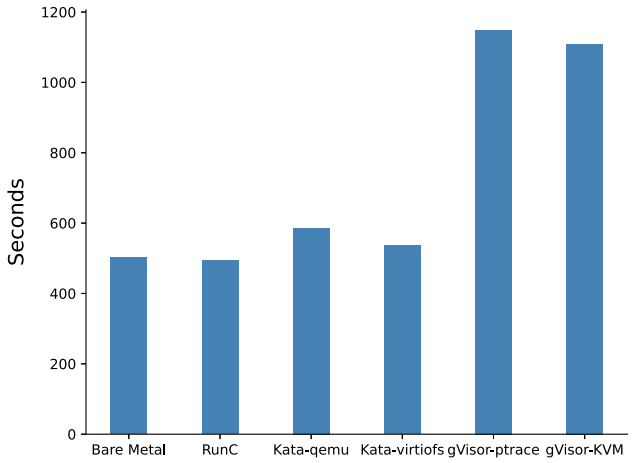

Fig. 12 Database record insertion performance

Blogbench was initially designed to mimic the blog's behavior and stresses the filesystem with multiple threads performing random reads, writes, and rewrites. The read and write buffers for file operations are 64kb and 8kb, respectively. We created three concurrent authors, one rewriter, ten readers, and five commenters. Consistent with the ApacheBench test results in [4], it can be seen from Table 2 that the gVisor shows very poor performance in the small file reads and writes and high concurrent scenarios. The high overhead comes principally from the VFS implementation that needs improvement, with several internal serialization points (since all requests are reading the same file) [4]. The Kata Containers is better in performance than gVisor, but its performance loss is also significant. The performance of the RunC container is comparable to that of the bare metal, and its performance overhead is 7.95%.

In short, of all the tested containers, the RunC container achieved the best performance, which was in line with our expectations. Although in terms of raw disk I/O, gVisor, and Kata Containers do not introduce significant overhead, in high-concurrency disk I/O scenarios, the performance loss of gVisor and Kata Containers is unexpected. Based on the above conclusions, when the application requires high disk I/O performance, the RunC container is the best choice. If users emphasize both security and I/O performance, they can consider gVisor and Kata Containers but avoid high-concurrency I/O application scenarios. In addition, we suggest that researchers and developers further optimize the I/O performance of gVisor and Kata Containers in a high-concurrency environment.

5.4 Network

Containers will typically live in their own, possibly shared, networking namespace. Container engines will usually add one end of a virtual ethernet (veth) pair into the container networking namespace. The other end of the veth pair is added to the host networking namespace. By default, the Kata Containers network uses traffic control to transparently connect the veth interface with the VM network interface. gVisor handles most of the network tasks inside netstack but still performs some checks on the host. From the implementation method of the container network and the path length of data processing, we believe that in terms of network performance, the RunC container should be the best, followed by the Kata Containers. At present, since the implementation of the gVisor network stack is still being optimized, the network performance of the gVisor container should be the worst.

We choose the two metrics of network throughput and latency to evaluate the network performance of different container runtimes. Network bandwidth is measured by running Netperf [26] benchmark on two identical physical machines. We configure gVisor to use the userspace network stack instead of using the host network stack. With Netperf, we measured the performance of TCP_STREAM, TCP_RR, TCP_CRR, and UDP_RR. We also measured the throughput and latency of TCP and HTTP using Ethr [27].

Figure 13 presents the TCP_STREAM performance. RunC and Kata Containers are very close in performance to bare metal, with their performance loss less than 1%. Compared with the performance of other container runtimes, the network throughput of gVisor is the lowest. Figure 14 shows the TCP_RR, TCP_CRR, and UDP_RR performance. The performance of the RunC container is almost the same as that of the bare metal, and its performance loss does not exceed 1%. The performance losses of TCP_RR, TCP_CRR, and UDP_RR of the Kata qemu container are 0.89%, 18.52%, and 17.23%, respectively. As with the TCP_STREAM, the gVisor has the largest performance degradation. This is likely due to its user space network stack, which is not as optimized as the Linux network stack [8]. As stated in [4], gVisor's network performance is mostly bound by implementation costs, and its network stack is improving rapidly.

Table 2 Blobench benchmark score for container runtimes(More is better)

	Bare metal	RunC	gVisor ptrace	gVisor KVM	Kata QEMU	Kata virtiofs
Read	654280	602242	88	343	2493	12673
Write	1932	1874	7	6	347	1106

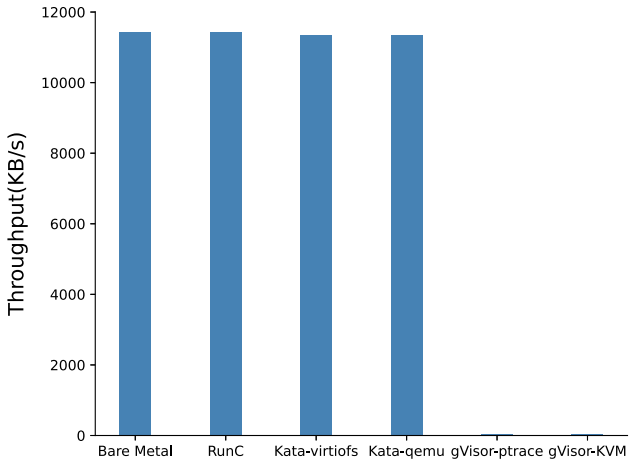

Fig. 13 TCP_STREAM network performance

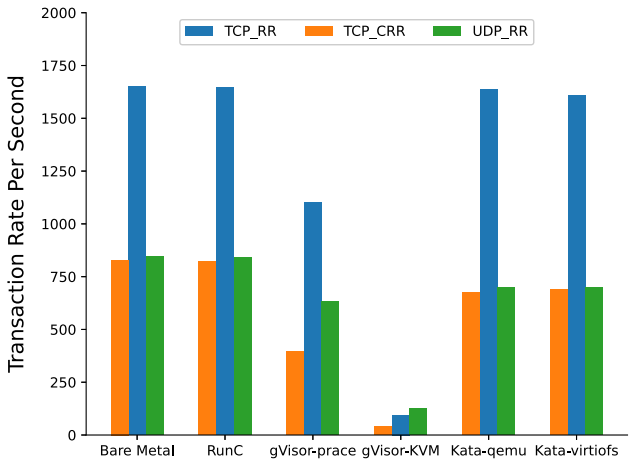

Fig. 14 TCP_RR, TCP_CRR and UDP_RR performance

Table 3 shows the TCP and HTTP throughput and latency performance reported by the Ethr benchmark. Similar to Netperf throughput measurement, RunC has the best performance, followed by Kata Containers, and gVisor has the worst performance. The performance degradation of the TCP and HTTP bandwidth of the Kata QEMU container is 1.08% and 13.12%, respectively. Consistent with the Netperf network performance test results, Ethr benchmark reports that gVisor has the lowest TCP and HTTP throughput. In terms of network latency, TCP's network latency is significantly less than that of HTTP. The RunC container has the smallest network delay under the TCP and HTTP protocols. The TCP and HTTP network latency performance losses of Kata Containers are 18.9% and 38.5%, respectively. gVisor has the largest network latency. gVisor does most of the processing inside netstack, but some data packet parsing still occurs in the host kernel as an integrity check before routing them. The double packets management mechanism at Sentry (netstack) and Kernel levels leads to gVisor's poorest network latency performance.

In short, except for the physical machine, the RunC container has the best network performance, followed by the Kata Containers, and gVisor has the worst performance. Although it is true that the network performance overhead of the gVisor container is relatively large, the performance loss of its user space network stack is unexpected. Therefore, for high-performance network applications, we recommend disabling the user-space network stack and using the host network stack, including loopback. However, using the host network stack will reduce the isolation between the container and the host, which will bring security risks. We suggest minimizing the host kernel's participation in processing network data and further optimizing the performance of the gVisor user-space network stack.

5.5 System calls

It is a well-known fact that the system call performance overhead of the gVisor container is serious. This is because the system calls initiated by the application must be intercepted by ptrace or KVM and redirected to Sentry (Sentry implements most of the Linux system calls) to perform the system calls. The system calls of RunC, and Kata Containers are conducted by the host system and the hypervisor, respectively (System calls involving sensitive resources will trap into the VMM and be executed by the VMM). Through the above analysis, we expect that RunC and Kata Containers should suffer a slight system call overhead.

We run the Unixbench [28] on each virtualization platform and identify the cost of entering and leaving the

Table 3 TCP and HTTP network performance of container runtimes (Ethr benchmark)

	Bare metal	RunC	gVisor-ptrace	gVisor-KVM	Kata-QEMU	Kata-virtiofs
TCP Bandwidth (KB/s)	96662.02	96204.8	655.36	686.08	95621.12	86778.88
HTTP Bandwidth(KB/s)	53196.8	49112.58	262.40	268.80	46215.68	43569.15
TCP Latency(us)	593.71	592.29	938.48	900.09	710.96	700.76
HTTP Latency(us)	730.29	776.61	1382.93	1161.59	1015.23	1010.72

operating system kernel, i.e., the overhead for performing a system call. Unixbench executes system calls in a loop and reports the number of loops executed in a given time. We measure the system call overhead in both single-thread and 4-thread cases.

As illustrated in Fig. 15, the performance overhead caused by the system calls of the gVisor is very serious. The system call performance overhead of Kata Containers is less than 1%. The performance overhead of RunC container system call is 11.62%. The system call performance of the gVisor KVM platform container (94.9%) is significantly better than that of the ptrace platform (48.71%). Under the ptrace platform, Sentry intercepts application system calls and acts as a guest kernel. The cost of this is reduced application compatibility and higher system call overhead. Just as sensitive instructions in virtual machines are trapped and simulated in the VMM, system calls across sandbox boundaries are very expensive. The KVM platform uses the kernel's KVM functionality, so the overhead of system call is significantly reduced. Kata Containers are containers that run on virtual machines. Except for sensitive instructions that need to be trapped in the host kernel and processed by the VMM, most of the system calls are processed by the guest kernel, so the resulting performance overhead is small.

Not surprisingly, gVisor has the most significant system call overhead, while the performance overhead of RunC and Kata Containers is almost negligible. With the system call heavy workload, the ptrace mode should be avoided [17]. Unlike the way that a virtual machine uses a guest operating system and a set of virtualized hardware devices to replace the interaction between the application and the host, the Sentry component of gVisor intercepts the application's system calls and implements most of the host's system APIs. Although this mechanism enhances isolation, it introduces significant performance overhead. We believe that it is necessary to further optimize Sentry's system API implementation. In addition, reducing the number of system calls across sandbox boundaries can further reduce the performance overhead of system call.

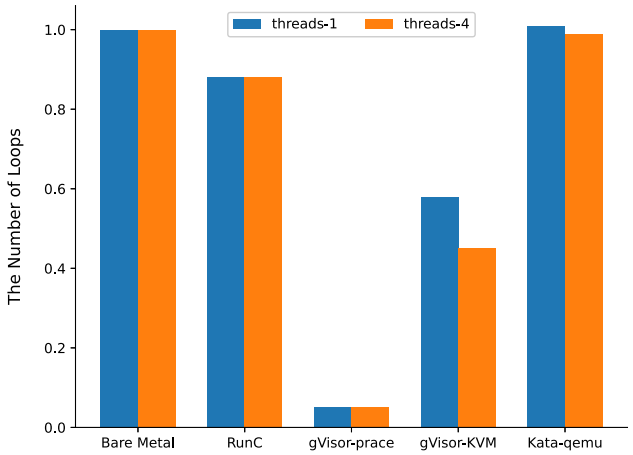

Fig. 15 System call performance

5.6 Startup and destruction time

Kata Containers is a highly optimized VM with a built-in container engine that can run on hypervisors. gVisor can be considered as a combined guest kernel and VMM, including two processes, Gofer and Sentry. Virtual machines require starting an entire operating system before any work can be processed. Containers are started and stopped within a running operating system, which means their lightweight nature enables them to be created or destroyed in seconds. This structure lends itself to dynamic scalability, making it possible to react to workload changes in real-time. We expect that the startup time of RunC and gVisor containers should be relatively close, and that the startup time of the Kata Containers should be the longest.

This section deals with the container startup time and the impact of the number of containers running on the same host on the container startup time. We first measure the startup time of a single container 50 times and calculate the average startup time of the container. In Fig. 17, we scale the number of containers running on the host to 50, and each container has 500M of memory.

As illustrated in Fig. 16, RunC has the shortest startup time, followed by gVisor, and the Kata Containers has the longest startup time. The average startup time of the RunC container is 1.62s. The startup time of the gVisor container is 9.12% longer than that of RunC. The average startup time of Kata Containers is 2.06s, which is 27.53% longer than RunC. The longer startup time of the Kata container is attributed to the architecture, that is, the container runs in a lightweight virtual machine [7]. However, the startup of the Kata Containers is only 0.44s slower than that of the RunC container. This is because the default kernel provided in Kata Containers is highly optimized for the kernel startup

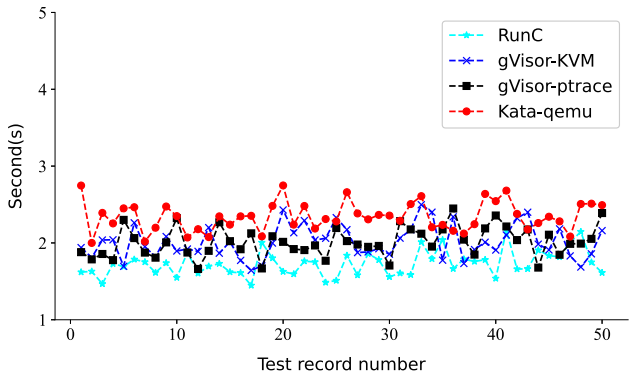

Fig. 16 Container startup time

time and minimum memory footprint and only provides the services required by the container workload. In addition, the Kata Containers supports hotplug capability, which allows virtual machines to start with minimal resources (such as CPU, memory, virtio block) and add additional resources when requested.

Figure 17 shows the impact of the number of containers running on the host on the container startup time. It can be seen that when the number of containers running on the host exceeds 40, the startup time of the Kata Containers fluctuates sharply and rises rapidly. Compared with Fig. 15, the startup time of the RunC, gVisor, and Kata Containers in Fig. 16 has increased by 22.63%, 20.24%, and 101.82%, respectively. From the test results of Sect. 5.7, we find that the Kata Containers has a large memory footprint. When the number of containers running on the host exceeds 40, the startup time of the Kata Containers increases significantly due to insufficient free memory on the host.

We also measured the destruction time of the container. Compared with the startup time of the container, it takes less time to delete the container. The gVisor container has the shortest destruction time, with an average destruction time of 0.99s. The average destruction time of RunC containers is 1.04s. Kata Containers have the longest destruction time, with an average destruction time of 1.26 seconds. When the number of containers running on the host continues to increase, the destruction time of RunC, gVisor, and Kata Containers increases by 21.58%, 21.28%, and 73.53%, respectively. It follows that Kata Containers enhances the security of the container, but due to its architecture, performance is affected.

In short, the RunC container has the fastest startup, followed by the gVisor container, and the Kata Containers has the slowest startup. Due to the gVisor and Kata Containers architecture, the above test results are expected, representing the cost of starting two additional processes (Sentry and Gofer) for the gVisor and starting the VM and hypervisor for the Kata Containers. The startup time of the gVisor container and Kata Containers is only 0.14s and 0.44s longer than that of the RunC container, which is very surprising. Both the gVisor container and the RunC container can rapidly scale the containers up and down as required, but the gVisor container has more advantages in terms of safety. However, if the application compatibility issues, security, and rapid scaling of container instances are considered, the Kata Containers has more advantages. No container runtime is advantageous in all aspects, so the specific needs of users need to be considered when choosing which container runtime. Kata Containers is lightweight virtual machine that is highly optimized for running container, while Firecracker is a specialized VMM that can create a secure virtualized environment for guest operating systems. Perhaps running Kata Containers on Firecracker VMM is a good choice. After all, the optimization of Firecracker VMM is more extensive and in-depth than the optimization of QEMU by the Kata Containers.

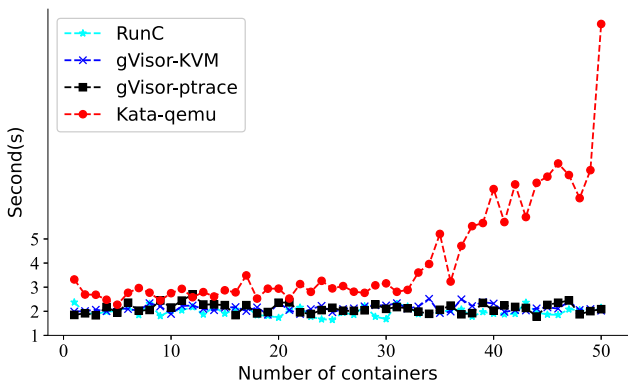

Fig. 17 Container startup time(Scaling the number of containers to 50)

5.7 Density

In this subsection, we examined the memory usage of the container. The Kata Containers is a lightweight virtual machine, and gVisor can be considered as a combination of the client kernel and VMM. Therefore, compared with the RunC container, we believe that the Kata Containers and gVisor should take up more memory. We use scripts to test the memory usage of runtime components of RunC, gVisor, and Kata Containers. This test launches a number of containers sequentially in idle mode and checks the amount of memory used by all containers (using the PSS measurement) after waiting for a configurable time. For Docker containers, we measure the memory usage of the child processes of each containerd-shim instance in the system. We count the memory usage of Sentry and Gofer processes as the memory footprint of the gVisor runtime. For the Kata Containers runtime (QEMU hypervisor), we record the memory usage of proxy, hypervisor, and shim processes. We create containers through the Ubuntu (5.4.32) image and set the memory size of each container to 200M.

In Fig. 18, the three different container technologies all show good scalability. The average memory usage of RunC, gVisor, and Kata Containers is 115.42KB, 8324.6 KB, and 112876 KB, respectively. The hypervisor, shim, and proxy of Kata Containers account for 93.9%, 3.89%, and 2.21% of the container's total memory, respectively. Kata Containers run containers on virtual machines, so the hypervisor process occupies most of the container memory. The Gofer and Sentry processes of gVisor account for 37.83% and 62.17% of the total memory, respectively. Sentry (currently, out of the 348 system calls in Linux, 256 system calls have complete or partial implementations, and

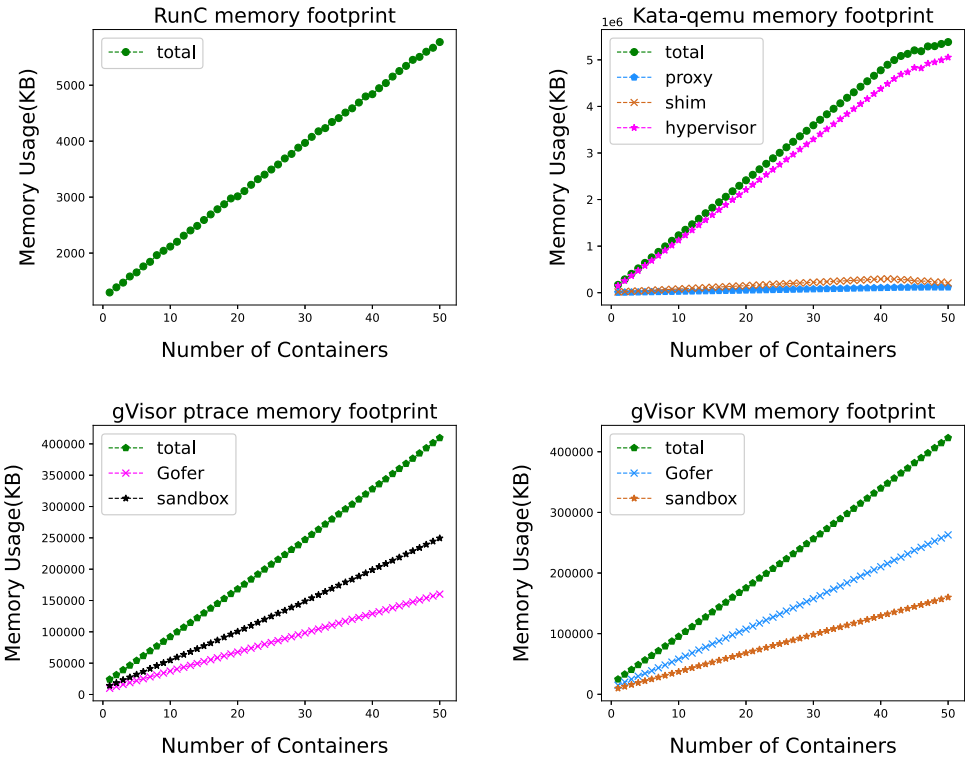

Fig. 18 Container memory usage

92 unsupported system calls) is the largest component of gVisor, and it can be seen as an application kernel. gVisor and Kata Containers strengthen isolation and security by adding an extra layer between the container and the host, at the cost of reduced density.

Consistent with our qualitative analysis results, the RunC container has the smallest memory footprint, followed by gVisor, with the Kata Containers having the most significant memory footprint. If there are no strict requirements for security, RunC containers are very suitable for high-density environments and small and medium-sized deployments that require fewer resources to complete more tasks. Users can decompose applications into fine-grained components and deploy them in different runtime containers according to the characteristics of each component to obtain actual differential scaling and maximize resource density while meeting user-specific needs. In addition, the combination of the test results of the container memory footprint and the application resource demand prediction results can improve the resource elastic scaling efficiency of the cloud computing platform. Most individual VMs deployed in the cloud today are dedicated to a single application, such as a proxy or database. To reduce the memory footprint of the Kata Containers, we propose running a dedicated Unikernel on a highly optimized hypervisor in addition to optimizing the QEMU-KVM hypervisor. The most essential features of Unikernels are improved security, small footprint, high optimization, and fast startup.

5.8 Isolation

The major difference between virtual machines and containers is the level of virtualization. The VM shares the physical hardware of the host, and the container shares the host's hardware and operating system kernel. Generally speaking, virtualized hardware isolation creates a stronger security boundary than namespace isolation does. gVisor enhances the isolation of containers by intercepting and implementing system APIs and minimizing the system APIs accessible by Sentry itself. Based on the above analysis, we expect the RunC container to have the worst isolation, and the gVisor and Kata Containers have relatively strong isolation.

We evaluated the isolation of RunC, gVisor, and Kata Containers using shell scripts. We launch two containers on the host: the affected container and the overloaded container. Each container is bound with two CPU cores and the memory size is set to 4G (host memory is 8G). We use the computing speed(GFlops:Giga Floating-point Operations Per Second) of the container reported by the Linpack benchmark as the performance measurement of the container in the isolation test. The Linpack problem size of the affected container and the overloaded container is set to 5000 and 20000, respectively. We first started the affected

container and recorded the Linpack test results and CPU utilization. Then, we started the overloaded container and its Linpack load and recorded the performance and CPU utilization of the affected container and the overloaded container at the same time. We measure isolation by the performance loss rate of the affected container.

5.8.1 RunC isolation

As shown in Fig. 19, after starting the overloaded container, the performance of the affected container dropped almost linearly. We set the Linpack load to run 20 times in the overloaded container. When the Linpack load in the overloaded container completes one run and starts the next run, the performance of the affected container first rises and then drops. The performance trajectory of the affected container reflects that it is sensitive to resource contention.

Affected by resource contention, the average CPU utilization and performance of the affected containers declined. Because containers are very sensitive to resource competition, the resource utilization and performance of containers fluctuate. The misbehaving containers could easily overwhelm the resource and interfere with the performance of the well-behaved containers [12]. Before and after the overloaded container was started, the variance of the CPU utilization of the affected container was 40.37 and 57.32, respectively. Compared with the CPU utilization, the performance degradation of the affected container is more significant. After starting the overloaded container, the average Linpack test performance of the affected container dropped from 77.19 GFlops to 53.59 GFlops, and the performance loss rate was 0.306. The container shares the hardware and the operating system kernel of the host. The more resources shared between the container and the host, the worse the isolation of the container.

5.8.2 Kata containers isolation

As illustrated in Fig. 20, the performance and CPU utilization trajectories of Kata Containers and RunC containers are similar. Affected by resource contention, the average CPU utilization of affected containers dropped from 197.38% to 182.26%. The average performance of the affected container before and after the interference was 73.02 GFlops and 53.66 GFlops, and the performance loss rate was 0.265. At the moment the Linpack benchmark started running in the overloaded container, the performance of the affected container plummeted. Kata Containers are sensitive to the increased workload of overloaded containers. The most apparent difference between the Kata Containers and the RunC container is the deeper isolation and security level between the containers. In Kata Containers, each containers runs its own kernel instead of using namespace to share the host system's kernel with the host and other containers. Each container can also obtain its own I/O, memory access, and other low-level resources without sharing them.

5.8.3 gVisor isolation

In the isolation test of gVisor containers, as shown in Figs. 21 and 22, the performance and resource utilization of ptrace and KVM platform containers show similar behavior. In terms of resource utilization, different from the RunC container, when the Linpack load is running, the CPU utilization of the gVisor container fluctuates greatly, and this fluctuation shows a certain pattern. Affected by resource competition, the average CPU utilization of gVisor ptrace and KVM platform containers dropped by 16.05% and 32.9%, respectively. Compared with the RunC container, the performance of the gVisor container fluctuates very little and is hardly affected by resource contention. Specifically, the performance fluctuation of gVisor

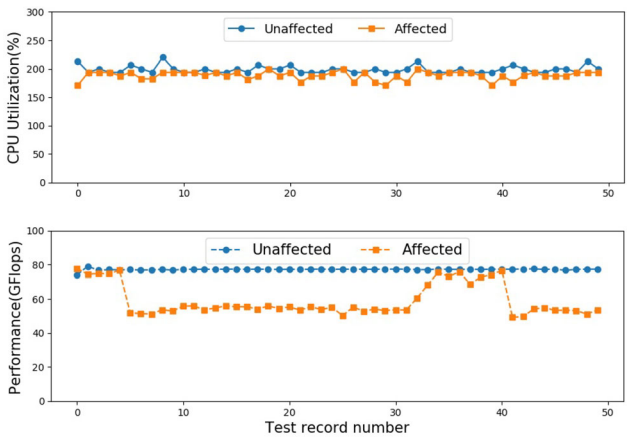

Fig. 19 RunC isolation

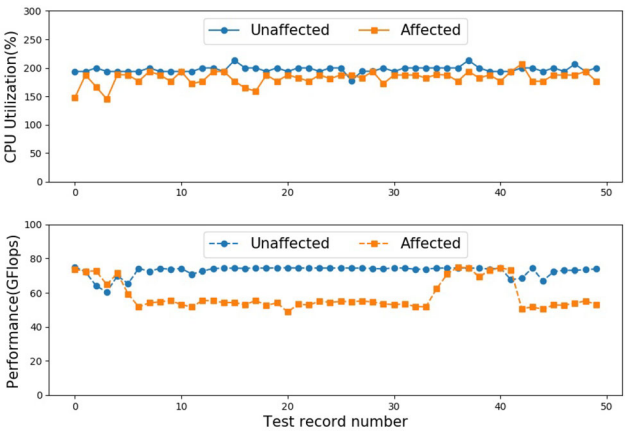

Fig. 20 Kata Containers isolation

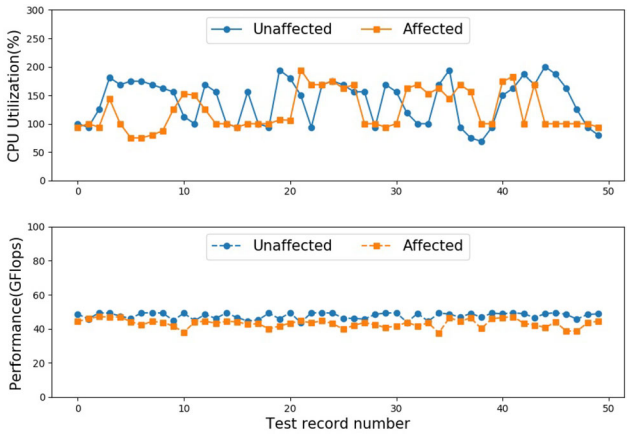

Fig. 21 gVisor ptrace isolation

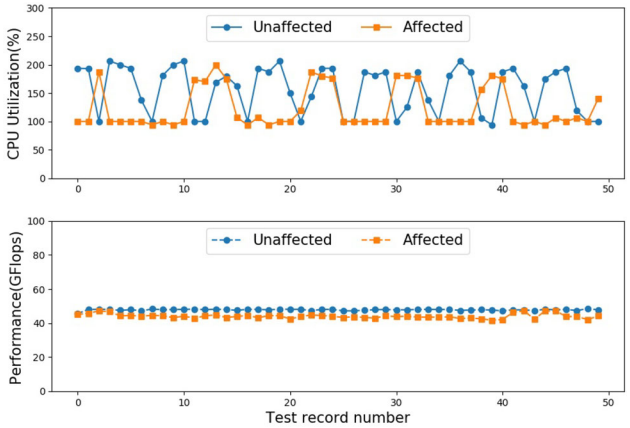

Fig. 22 gVisor KVM isolation

KVM platform containers is smaller than that of ptrace platform containers. After the Linpack load in the overloaded container was started, the average performance of the gVisor ptrace container dropped from 47.68 GFlops to 43.2 GFlops, and the performance loss rate was 0.094. The performance of the gVisor KVM container before and after interference by the overloaded container is 47.84 GFlops and 44.11 GFlops, respectively, and the performance loss rate is 0.078. In addition, unlike RunC containers, gVisor is not sensitive to resource competition introduced by overloaded containers. Throughout the measurement, the performance of the gVisor container was very stable and showed good isolation.

gVisor combines two ways to strengthen the isolation of containers. One is to reduce the sharing between the container and the host system, and the other is to achieve highly dynamic management of container resources. gVisor achieves strong isolation between the container and the host system through the Gofer and Sentry processes. In addition, gVisor's resource model adopts a dynamic resource provision strategy. That is, no fixed number of vCPUs or physical memory are allocated to the containers. When possible, gVisor delegates the dynamic management of the underlying physical resources to the host system to optimize the provision of container resources globally. In many cases, it may be more efficient to reserve container resources and assign them to containers when they become overloaded. However, immediately releasing the resources to the host allows the host to reuse them more efficiently and apply effective global policies to improve the isolation between containers.

5.8.4 Combining performance loss and resource shrinkage to measure isolation

In container environments, the performance loss of containers may also be caused by resource shrinkage [12]. At the same degree of resource shrinkage, the greater the performance loss, the worse the performance isolation is. However, if the resource shrinkage is more significant in the case of the same performance loss, the performance isolation is better. The performance loss rates of gVisor KVM and ptrace platform containers were 0.078 and 0.094, respectively, and their CPU utilization dropped by 21% and 11%. Therefore, the isolation of gVisor KVM platform containers is better. Similarly, compared with the RunC container, the Kata Containers has less performance degradation, but its resource reduction is 51% higher than that of the RunC container. Therefore, if the performance isolation is measured in terms of both performance loss and resource shrinkage, compared with the test results of the performance degradation rate, the Kata Containers exhibits stronger isolation than the RunC container.

Consistent with our expectations, the RunC container has the worst isolation, but its performance loss is only 14.8% larger than that of the Kata Containers. The gVisor container shows the strongest isolation. The gap in isolation between the gVisor container and the Kata Containers is surprising. Specifically, the performance degradation of the Kata Containers is 2.72 times that of the gVisor container, but the resource shrinkage of the gVisor container is 1.43 times that of the Kata Containers. When users have more stringent requirements for SLA and security, the gVisor container is undoubtedly the best choice. In [33], the authors proposed to combine different types of loads on the containers to reduce interaction. Their results suggest the combination of CPU intensive and I/O intensive loads to alleviate the performance impacts, rather than a combination of disk-intensive and memory-intensive loads. A simple way to optimize performance isolation is to effectively manage the resources of the container. In designing isolation optimization methods, it is necessary to ensure the elastic supply of container resources. Just like the resource allocation strategy adopted by gVisor's resource model, a

more dynamic and fine-grained resource provisioning method can facilitate the host's global optimization of container resource provision, thereby improving the isolation between coexisting containers.

6 Conclusion

Because RunC containers share more resources from the host, their utilization of resources is much more efficient than gVisor and Kata Containers. However, it is precisely because the RunC container shares more resources with the host that the isolation between the containers is weaker. In addition, of all the container runtimes evaluated, the RunC container has the shortest startup time and the smallest memory footprint. The performance overhead of gVisor containers in terms of memory allocation, network, and system call is relatively large. The density and startup time of the gVisor is second only to the RunC container. In addition, gVisor has the strongest isolation. Of all the container runtimes measured, the disk I/O performance of the Kata Containers degraded the most, and its performance in other aspects is comparable to that of the RunC container. Since the Kata Containers is a VM-based sandbox technology, the startup of the Kata Containers is slower and the runtime components have a larger memory footprint. In terms of isolation, the isolation of the Kata Containers is stronger than that of the RunC container, but it is weaker than that of gVisor. For security-sensitive applications, gVisor and Kata Containers are good solutions, while for applications with higher performance requirements, RunC and Kata Containers may be the most appropriate choices.

The findings of this study will serve as the basis for future work. We believe that focusing on performance, startup time, density, and isolation issues is the first step in building a container system that is both resource-efficient and secure. In future work, we will evaluate more technologies to enhance container security, including the Unikernel-based sandbox technology.

Supplementary Information The online version of this article contains supplementary material available https://doi.org/10.1007/s10586-021-03517-8.

Acknowledgements This work is partially supported by a grant from the National Natural Science Foundation of China (No.62032017), the National Key R&D Program of China (2018YFB1003605), the Key Industrial Innovation Chain Project in Industrial Domain of Shaanxi Province (Nos. 2021ZDLGY03-09, 2021ZDLGY07-02), and the Youth Innovation Team of Shaanxi Universities.

Author contributions All authors contributed equally to this work.

Data availability Data available on request from the authors.

Declarations

Conflict of interest The authors declare that they have no conflicts of interest.

Informed consent Informed consent was obtained from all individual participants involved in the study.

Research involving human participants or animals This paper does not contain any studies involving human participants or animals performed by any of the authors. Xingyu Wang carried out the experiment and wrote the manuscript.

References

1. Bachiega, N.G., Souza, P.S., Bruschi, S.M., De Souza, S.D.R.: Container-based performance evaluation: a survey and challenges. In: 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 398–403 (2018)
2. Williams, D., Koller, R., Lucina, M., Prakash, N.: Unikernels as processes. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC '18), Association for Computing Machinery, New York, NY, USA, pp. 199–211 (2018)
3. Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., et al.: My VM is lighter (and safer) than your container. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), pp. 218–233 (2017)
4. https://gvisor.dev/docs/user_guide/. Accessed 20 Dec 2020
5. https://github.com/firecracker-microvm/firecracker/. Accessed 20 Dec 2020
6. https://katacontainers.io/. Accessed 15 Dec 2020
7. Kumar, R., Thangaraju, B.: Performance analysis between RunC and kata container runtime. In: 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), Bangalore, India, pp. 1–4 (2020)
8. Caraza-Harter, T., Swift, M.M.: 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 (VEE '20), Association for Computing Machinery, New York, NY, USA, pp. 101–113 (2020)
9. Debab, R., Hidouci, W.K.: Containers runtimes war: a comparative study. In: Proceedings of the Future Technologies Conference, Springer, pp. 135–161 (2020)
10. Viktorsson, W., Klein, C., Tordsson, J.: Security-performance trade-offs of kubernetes container runtimes. In: 2020 Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, November 17–19, Nice, France pp. 1–4 (2020)
11. Kozhirbayev, Z., Sinnott, R.O.: A performance comparison of containerbased technologies for the cloud. Future Gener. Comput. Syst. **68**, 175–182 (2017)
12. Zhao, C., Wu, Y., Ren, Z., Shi, W., Ren, Y., Wan, J.: Quantifying the isolation characteristics in container environments. In: IFIP International Conference on Network and Parallel Computing, Springer, pp. 145–149 (2017)
13. Tesfatsion, S. K., Klein, C., Tordsson, J.: Virtualization techniques compared: performance, resource, and power usage overheads in clouds. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, pp. 145–156 (2018)
14. Mavridis, I., Karatza, H.: Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. Future Gener. Comput. Syst. **94**, 674–696 (2019)

15. Chae, M., Lee, H., Lee, K.: A performance comparison of linux containers and virtual machines using Docker and KVM. Clust. Comput. **22**, 1765–1775 (2019)
16. Espe, L., Jindal, A., Podolskiy, V., Gerndt, M.: Performance evaluation of container runtimes. In: CLOSER, pp. 273–281 (2020)
17. Young, E.G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: The true cost of containing: a gVisor case study. In: 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19) (2019)
18. 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), pp. 419–434 (2020)
19. https://docs.docker.com/. Accessed 10 Dec 2020
20. https://podman.readthedocs.io/en/latest/index.html. Accessed 10 Dec 2020
21. https://coreos.com/rkt/docs/latest/. Accessed 10 Dec 2020
22. https://docs.microsoft.com/en-us/windows/wsl/wsl2-index. Accessed 10 Dec 2020
23. Frazelle, J.: Research for practice: security for the modern age. Commun. ACM **62**(1), 43–45 (2018)
24. http://www.jbkempf.com/blog/post/2018/Introducing-dav1d. Accessed 18 Dec 2020
25. https://github.com/akopytov/sysbench. Accessed 18 Dec 2020
26. https://hewlettpackard.github.io/netperf/doc/netperf.html. Accessed 20 Dec 2020
27. https://github.com/microsoft/ethr. Accessed 20 Dec 2020
28. https://github.com/kdlucas/byte-unixbench. Accessed 20 Dec 2020
29. https://www.iozone.org. Accessed 10 Nov 2020
30. www.flockport.com/. Accessed 5 Nov 2020
31. https://openbenchmarking.org/test/pts/sqlite. Accessed 12 Nov 2020
32. Krebs, R., Momm, C., Kounev, S.: Metrics and techniques for quantifying performance isolation in cloud environments. Sci. Comput. Programm. **PT.B**(2), 116–134 (2014)
33. Xavier, M.G., De Oliveira, I.C., Rossi, F.D., Dos Passos, R.D., Matteussi, K.J., De Rose, C.A.: A performance isolation analysis of disk-intensive workloads on container-based clouds. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, pp. 253–260 (2015)
34. https://github.com/cloud-hypervisor/cloud-hypervisor. Accessed 21 Nov 2020
35. https://github.com/jedisct1/Blogbench. Accessed 7 June 2021
36. Zhao, N., Tarasov, V., Albahar, H., Anwar, A., Rupprecht, L., Skourtis, D., et al.: Large-scale analysis of docker images and performance implications for container storage systems. IEEE Trans. Parallel Distrib. Syst. **32**(4), 918–930 (2021)
37. Bhatt, G., Bhavsar, M.: Performance consequence of user space file systems due to extensive CPU sharing in virtual environment. Clust. Comput. **23**(4), 3119–3137 (2020)
38. Shih, W.C., Yang, C.T., Ranjan, R., Chiang, C.I.: Implementation and evaluation of a container management platform on Docker: Hadoop deployment as an example. Clust. Comput. **24**, 3421–3430 (2021)
39. Tang, X., Zhang, Z., Wang, M., Wang, Y., Feng, Q., Han, J.: Performance evaluation of light-weighted virtualization for paas in clouds. In: International Conference on Algorithms and Architectures for Parallel Processing, Springer, pp. 415–428 (2014)
40. Walraven, S., Monheim, T., Truyen, E., Joosen, W.: Towards performance isolation in multi-tenant saas applications. In: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing., pp. 1–6 (2012)

Xingyu Wang is a PhD candidate at Xidian University.His main research interests are performance evaluation of virtualization and cloud computing.

Junzhao Du PhD supervisor, is a professor at the School of Computer Science and Technology, Xidian University.His main research interests include edge computing, cloud computing, cloud measurement, etc.

Hui Liu PhD supervisor, is an associate professor at the School of Computer Science and Technology, Xidian University.His main research interests are cloud computing, cloud evaluation and recommendation algorithms.