Lightweight and Holistic-scalable Serverless Secure Container Runtime for High-density Deployment and High-concurrency Startup

Zijun Li, Chenyang Wu, Chuhao Xu, Quan Chen, Shuo Quan, Bin Zha, Qiang Wang, Weidong Han, Jie Wu, Minyi Guo

***Abstract*—The secure container that hosts a single container in a micro virtual machine (VM) is now used in serverless computing, as the containers are isolated through the microVMs. There are high demands on the high-density container deployment and high-concurrency container startup to improve both the resource utilization and user experience, as user functions are fine-grained in serverless platforms. Our investigation shows that the entire software stacks, containing the cgroups in the host operating system, the guest operating system, and the container *rootfs* for the function workload, together result in low deployment density and slow startup performance at high-concurrency.**

We propose a lightweight and holistic-scalable secure container runtime, named RunD-V, to resolve above problems in serverless computing. RunD-V proposes a guest-to-host runtime template for microVM scaling-out, and CR-bind feature in guest kernel for microVM scaling-up. Using guest-to-host runtime template, over 200 secure containers can be launched within $1s$ on a node equipped with 104 vCPUs. It also enables more than 2,500 secure containers to be deployed on a node with 384GB of memory. The vertical scaling mechanism CR-bind further enhances both startup concurrency and deployment density.

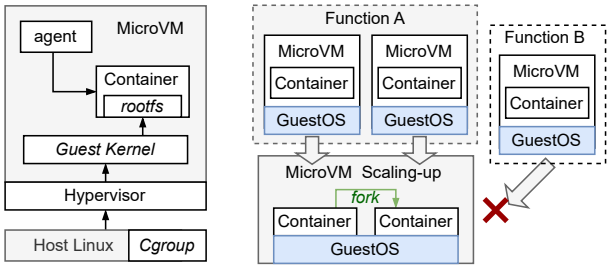
***Index Terms*—Serverless, MicroVM, Container, Template**

I. Introduction

With serverless computing (Function-as-a-Service), tenants submit functions directly to the Cloud without renting virtual machines, and the cloud provider uses containers to host invocations on-demand [1]–[5]. Most cloud providers publish the serverless computing services with the pay-for-use pricing model, such as Amazon Lambda, Microsoft Azure Functions, Google Cloud Function and Alibaba Function Compute.

When hosting function invocations, traditional containers (e.g., Docker, LXC) only provide process level isolation [6], [7]. They cannot prevent privilege escalation, information disclosure side channels, and covert channel communication [8]. To this end, secure containers that achieve the same isolation with the traditional virtual machines are often preferred, which often creates a normal container within the microVM. As depicted in Figure 1(a), microVM is for isolation, and the container is for abstraction [9]. Kata Containers [10] provides practical experience in implementing such secure container.

Zijun Li, Chenyang Wu, Chuhao Xu, Quan Chen and Minyi Guo are with the School of Computer Science, Shanghai Jiao Tong University, Shanghai, China, 200240. Shuo Quan are with the China Telecom Cloud Computing Research Institute, Beijing, China, 100053. Jie Wu is with the China Telecom Cloud Computing Research Institute, Beijing, 100088, China and with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA Bin Zha, Qiang Wang and Weidong Han are with the Alibaba Group, Hangzhou, China, 310052.



(a) Secure container (b) Multi-Replica paradigm

Fig. 1: The secure container architecture and the opportunity of exploiting vertical scaling-up for microVMs.

The lightweight and short-term nature of functions emphasizes the necessity for high-density deployment and high-concurrency startup of secure containers in serverless computing. For example, 47% of AWS Lambda functions operate with the minimum memory specification of 128MB, and approximately 90% of applications on Microsoft Azure never exceed a memory usage of 400MB [11]. Given that physical nodes often have substantial memory capacities (e.g., 384GB), they should theoretically accommodate many functions. Moreover, a large number of function invocations may arrive in a short time. However, the additional hierarchy of guestOS in secure containers will significantly impact the deployment density and the concurrency of container startups.

This paper re-examines the security model of secure containers in serverless and proposes a Multi-Replica paradigm that preserves guestOS-level isolation between functions, while enabling multiple container replicas of the same function to coexist within a single microVM. As shown in Figure 1(b), this paradigm significantly amortizes the memory footprint of the guestOS and accelerates container startup by leveraging fork. Comparing to the conventional Single-Replica paradigm where each microVM hosts only one container replica, the Multi-Replica paradigm effectively enhances deployment density and startup concurrency for serverless functions by integrating both horizontal and vertical scaling mechanisms.

To achieve higher system-level startup concurrency and deployment density, it is fundamental to identify and mitigate bottlenecks in existing scale-out mechanisms. Our investigations reveal that guest-side *rootfs* mounting introduces redundant CPU/memory overhead, while host-side cgroup operations are serialized by mutex locks. These inefficien-

cies hinder the high-density deployment and high-concurrency startup capabilities of secure containers. To address these issues, we propose a guest-to-host runtime template comprising: (1) a hybrid *rootfs* template for serverless containers, and (2) lightweight cgroup templates that utilize rename-based pool management. These adaptations enhance system-level microVM startup concurrency and deployment density.

When hosting multiple container replicas within a microVM via vertical scaling, it is essential to tailor the guestOS for improved sharing efficiency and to provide stable, low-overhead resource hot-(un)plugging support, particularly for memory. In our tailored guest kernel, the built-in feature CR-bind is patched to the runtime template. CR-bind enables on-demand memory hot-(un)plug operations and designates a specialized *memory region* where a memory device can be exclusively allocated to a container. Above efforts effectively enhance function-level startup concurrency and deployment density.

Based on the guest-to-host runtime template and CR-bind, we propose RunD-V, a lightweight and holistic-scalable serverless secure container runtime. Both scaling-out and scaling-up are supported within RunD-V, with a dedicated holistic scaling strategy ensuring that they can cooperate in a harmonious and efficient manner. To the best of our knowledge, RunD-V is the first exploration to integrate both scaling-out and scaling-up mechanisms to achieve high-density deployment and high-concurrency startup for secure containers.

The main contributions of this paper are as follows.

1) **The Multi-Replica paradigm re-examines the security model and offers insights into scaling mechanisms.** We identify bottlenecks of horizontal scaling-out and explore the opportunities afforded by vertical scaling-up.
2) **RunD-V builds a guest-to-host runtime template for horizontal scaling-out.** The container rootfs template based on hybrid read/write device mounting, and a lightweight cgroup template with pool management effectively resolve the identified bottlenecks.
3) **RunD-V proposes CR-bind for stable and efficient memory hot-(un)plugging.** We also design a holistic scaling strategy that achieves an overlap and balance between horizontal scaling-out and vertical scaling-up.

According to our evaluation, RunD-V boots to application code in 88 ms using the guest-to-host runtime template. On a node with 104 vCPUs, RunD-V can launch over 200 secure containers per second. Similarly, on a node with 384GB of memory, RunD-V is capable of deploying more than 2,500 secure containers. Furthermore, by enabling vertical scaling-up with CR-bind, RunD-V can enhance function-level concurrency by up to 2x while reducing the memory usage for a single function to 1/8 of that required by the original setup.

II. Related Work

The emerging serverless computing has attracted a lot of valuable studies. They also recognize the importance of deployment density and concurrency capabilities of serverless containers in production scenarios.

Higher-density deployment. Regarding serverless computing, in the space of higher function deployment density of Secure Containers and microVMs, the key is designing a more lightweight container runtime both in guest and host. Unikernel [12]–[14] runs as a built-in GuestOS without necessary add-ons, demonstrating great potential for deploying containers with less overhead. Kuo [15] Explores lightweight guest kernel configurations for use in Unikernel environments, which has similarity to the approach towards reducing guest kernel size. SAND [16] and Microless [17] colocate all functions of a application to amortize the memory footprint of sandboxing. However, they fail to handle memory fragmentations in a real-system with high-density deployment. Gsight [18] observes that fine-grained function-level profiling can expose more predictability system-level features in interference. With accurate interference predicting [19], the function density can get improved with QoS guaranteed.

The above studies make sense in improving the effective density with less interference for serverless. They are orthogonal to our work, because RunD-V is motivated to improve the maximum deployment density on a signe node.

Higher-concurrency startup. In the space of higher function startup concurrency, recent approaches leverage the container prewarm pool [11], [20]–[22]. The state-of-the-art on container prewarming, SOCK [22], uses a benefit-to-cost model to select packages pre-installed in zygotes, and builds a tree cache to ensure a fast zygote container forking without unspecified packages. The C/R (Checkpoint/Restore) [5], [23] supporting the microVM snapshotting [24]–[26] captures the state of a running instance as a checkpoint, and then restores it once cold startup. Observing that most functions only access a small fraction of the files and memory loaded in the initialization stage, Catalyzer [26] and Replayable Execution [27] extend the C/R mechanism to achieve a faster on-demand recovery and paging when start containers. REAP [28] proactively prefetches and load the pages that identified critical of the startup from a microVM snapshot into the guest memory for faster concurrency startup. Lambda [29] builds a container image caching system that supports on-demand, deduplicated loading, thereby enabling over 15,000 concurrent startups per second across the cluster.

Above studies are orthogonal to RunD-V, as they concentrate on reducing startup and recovery latencies by leveraging snapshots or images. RunD-V tries to address bottlenecks through guest-to-host optimizations of the secure container architecture, including container rootfs, guestOS and cgroup resource management.

III. Background and Motivation

This section first introduces the secure container and analyzes its security model, and how these considerations motivate our Multi-Replica paradigm and holistic scaling mechanism.

A. Secure Container and In-production Requirements

When hosting function invocations, traditional containers (e.g., Docker, LXC) only provide process-level isolation, which can lead to security vulnerabilities [6], [7]. Consequently, there is a growing trend toward adopting more secure runtime environments. This shift has spurred the development

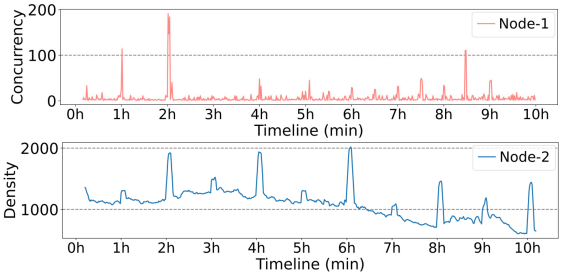
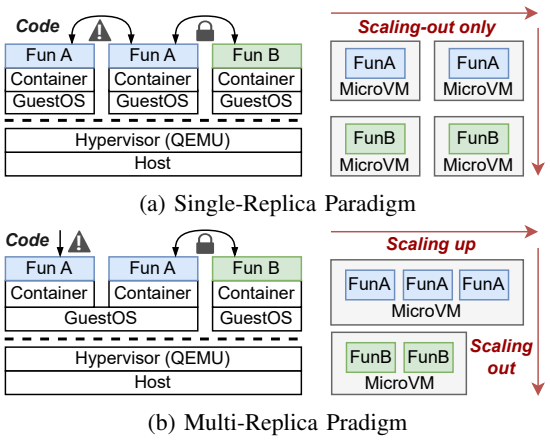


Fig. 2: The startup concurrency (Node-1) and sandbox density trace (Node-2) from our Alibaba serverless platform. Observations of high-concurrency and high-density are typically occur in different nodes due to load balancing.



(a) Single-Replica Paradigm

(b) Multi-Replica Pradigm

Fig. 3: The security model and execution logic of conventional Single-Replica paradigm and our Multi-Replica paradigm. The dashed line separates the trusted and untrusted environments.

of secure container architectures that incorporate a lightweight microVM to provide an extra isolation layer while maintaining compatibility with the existing container ecosystem (e.g., OCI and CRI). For instance, Kata Containers [10] is the most representative secure container implementation, where user functions run inside a runc container that is hosted by own dedicated microVM operating a streamlined guestOS. For cloud vendors, two key requirements for using this secure container architecture in production are achieving high container startup concurrency and high deployment density [8].

High-concurreny startup. In serverless computing, concurrent startup can be broadly classified into two scenarios. One scenario involves the parallel execution of multiple instances of the same function, such as video transcoding tasks employing a map-reduce paradigm. The other scenario arises when different function events are triggered simultaneously; for example, different user functions might set the same timer or invoke a microservice graph. Consequently, as illustrated in the upper part of Figure 2, Alibaba's serverless platform can experience over 100 sandbox-launch requests on a single node. High startup concurrency is crucial in serverless systems as it ensures that containers can be initialized rapidly to handle sudden spikes in function invocations effectively.

High-density deployment. In a traditional Infrastructure-as-a-Service (IaaS) scenario, the overhead impact of a virtual machine is mitigated by the large memory specifications typically allocated to each VM. In contrast, serverless computing typically employs lightweight containers with much smaller memory footprints, making the high memory overhead of a guestOS a critical limiting factor for the number of deployable sandboxes on a single node. For example, in an ideal scenario without overhead, a node with 384GB of memory could theoretically host $8 \times 384 = 3072$ containers, each with a memory footprint of 128MB. However, without further optimizations, Alibaba's platform supports fewer than 2,000 sandboxes per node, as illustrated in the lower part of Figure 2. Achieving a denser deployment of sandboxes is therefore essential for maximizing resource utilization on a single physical machine.

B. Security Model and Execution Paradigm

For serverless platforms, the primary responsibility is to ensure the security of the cloud infrastructure, including mitigating security risks posed by malicious user codes. To this end, secure containers leverage the security model of hardware virtualization and hypervisor, explicitly treating the guest kernel as untrusted by performing syscall inspections.

Existing serverless execution paradigms often adopt a Single-Replica fashion, as illustrated in Figure 3(a). In this paradigm, separate microVMs isolate different functions, and each container replica of the same function is also deployed with its own microVM (referred to as a "sandbox"). Consequently, serverless sandboxes rely exclusively on horizontal scaling (aka. scaling-out) for resource management.

While this paradigm seems to provide strong isolation for each individual sandbox, the fact that all microVMs of the same function running identical code means that any attack from malicious code in one container replica could compromise all microVMs simultaneously. Based on the above analysis, we can find that the Single-Replica paradigm additionally allocates a guestOS layer to each replica, but it cannot reduce the spread of threats to function replicas. This leads us to rethink about the existing Single-Replica execution paradigm, which may not be the most reasonable for the scenario of high concurrency and high-density deployment for serverless.

If we allow a microVM hosts the container replicas of the same function, the container replicas in the same microVM can share the guestOS of the microVM, as illustrated in Figure 3(b). This adaption not only accelerates the startup process for replicas of the same function but also reduces the memory overhead associated with the guestOS. This paradigm, referred to as the Multi-Replica paradigm, is particularly well-suited for serverless systems that demand high startup concurrency and dense deployment. Importantly, the security guarantees remain consistent with conventional Single-Replica paradigm—microVMs continue to isolate different functions rather than individual replicas of the same function.

Under the Multi-Replica paradigm, the computing resource quota for a microVM should be dynamically determined by the real-time aggregate of the resource quotas allocated to the containers it hosts. Failure to align the microVM's resource allocation with its containers' demands may lead to

either waste or insufficient provisioning of critical resources. Multi-Replica also align with the core principle of serverless computing, which is reflected in that the computing resources allocated to a function microVM can scale up and down elastically in response to workload changing.

C. Multi-Replica Paradigm with Holistic Scaling Mechanism

The Multi-Replica paradigm aligns with our production scalability demands by requiring a holistic scaling strategy: inter-function concurrency necessitates horizontal scaling-out, while intra-function concurrency prioritizes vertical scaling-up. Sharing a guestOS across replicas of the same function reduces memory overhead in high-density deployments.

However, it remains unclear where the primary bottleneck lies in scaling out under high concurrency and high-density deployment scenarios, and whether current resource hotplugging technologies can accommodate the frequent scaling up and down required for adjusting microVM resources. Based on our analysis, we derive three key insights for the Multi-Replica paradigm:

- The isolation of different functions can rely on horizontal microVM scaling out. **By identifying and mitigating bottlenecks in existing secure container architectures, it is possible to achieve higher system-level startup concurrency and deployment density.**
- Multiple replicas of the same function can be hosted within a microVM by vertically microVM scaling up. **MicroVMs should support a stable, low-overhead resource hot-(un)plugging, thereby enhancing function-level startup concurrency and deployment density.**
- Both scaling-out and scaling-up can be supported within the serverless secure container runtime, with dedicated strategies ensuring that they can cooperate in a harmonious and efficient manner.

IV. Rationale of High-density Deployment and High-concurrency Startup

It is essential to examine the underlying technologies to identify the bottlenecks and opportunities associated with scaling-out and scaling-up. This section provides the roadmap of scaling-out bottlenecks identification and opportunities of resource hot-(un)plug for scaling-up.

A. Bottlenecks of Horizontal Scaling-out

*1) **Guest-side Container Rootfs**:* According to the Berkeley's view [30], serverless applications are primarily web applications, characterized by numerous I/O operations. Therefore, containers root filesystem (*rootfs*) should provide robust I/O read and write capabilities, as well as the ability to withstand performance interference. In general, *rootfs* can be exposed into the microVMs by filesystem (e.g., *9pfs* [31], *virtio-fs* [32]) or block device (e.g., *virtio-blk* [33]).

Figure 4 illustrates the IOPS (IO-Per-Second) and bandwidth for random/sequential read/write operations using *9pfs*, *virtio-fs*, and *virtio-blk* in Kata Containers. For comparison, the performance metrics of traditional containers using *ext4* and *overlayfs* on the host node are also measured [34], [35].

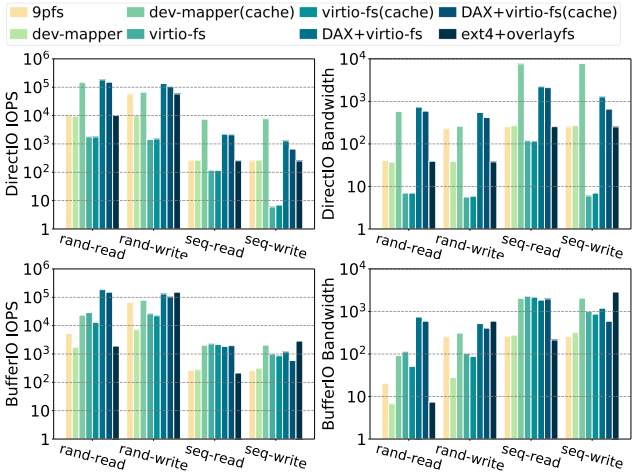


Fig. 4: The IOPS/bandwidth performance of rand/seq directIO/BufferIO read/write when using different *rootfs* mapping in Kata-runtime (dev-mapper represents that *virtio-blk* is used, *ext4+overlayfs* represents the default runc-container *rootfs*).

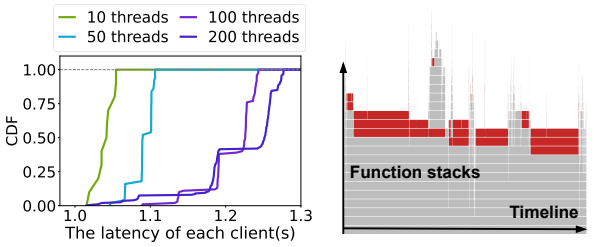
While *virtio-blk* offers the best write performance, its high memory overhead and concurrency limitations are significant drawbacks. According to our measurement, it takes as high as 10 seconds to prepare *rootfs* when 200 microVMs are started concurrently, while it only takes about 30 milliseconds for a single microVM startup. Moreover, it does not support page cache sharing between the host and guest operating systems. When the *virtio-blk* backend reads *rootfs* files into the host page cache, the same content is duplicated in the guest page cache, leading to a high memory footprint.

On the other hand, virtio-fs [32] with DAX (Direct Access) mode enables the guest to directly access the host's page cache for a file without replicating it in guest, thereby mitigating the duplication of page caches. However, this approach comes at the expense of write performance. Besides, the per-microVM *virtiofsd* client daemon takes charge of container I/O operations, resulting in excessive CPU usage on the host with clients increase. This issue is exacerbated under high I/O stress or during intensive metadata operations, which are common in serverless web applications.

The above investigation shows that either virtio-fs or virtio-blk can compromise either deployment density or startup concurrency of microVMs scaling-out. An exploratory alternative would be: using virtio-fs to support the read only part of rootfs for sharing page cache between host and guests,and using virtio-blk to support the writeable part of rootfs for high I/O performance. A solution is also required to further reduce the duplicated writable part for rootfs.

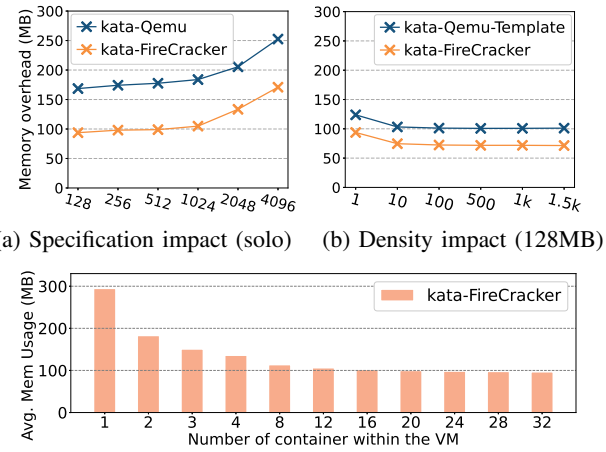
*2) **Host-side Cgroups Efficiency**:* The sudden workload spikes in our trace result in the rapid creation of a large number of microVMs, quickly consuming the host's memory space, as the Azure serverless platform trace [11] also show the same pattern. The rapid creation and deletion of a large number of microVMs in a short period results in delayed exits from critical sections of cgroups in multi-threaded scenarios.

We measure the performance of cgroup operations when creating $2,000$ microVMs concurrently, using different num-



(a) Latency distribution (b) The flame graph of Perf

Fig. 5: The performance of cgroup operations when creating $2,000$ container cgroups concurrently. The stacked blocks in the flame graph show the call chain, with the red segments highlighting cgroup operations and experience long-term spinning due to mutex locks.



(a) Specification impact (solo) (b) Density impact (128MB)

(c) The container memory overhead within a microVM

Fig. 6: The memory overhead of a secure container when kata uses *qemu* or *FireCracker* as the hypervisor. The template and *mmap* helps microVM reduce the memory footprint but still significant when 1000 sandboxes deployed. Multi-replica paradigm can further amortize such overhead.

bers of threads to perform cgroup operations. Figure 5(a) shows the cumulative distribution of container creating latencies. The reason behind the above fact is that the Linux kernel introduces several global locks to serialize cgroup operations. The flame graph of Figure 5(b) indicates that the red part experiences the optimistic spinning if cgroups fail to acquire the "mutex locks" during the timeline. Figure 5(b) highlights the hot functions related to cgroup operations, marked in red. The y-axis indicates that these functions are positioned at the lower and critical levels of the call chain, while the x-axis reveals that $2,000$ cgroup operations performed by 10 clients experience optimistic spinning when they fail to acquire the corresponding "mutex locks".

On the other hand, when the deployment density exceeds 1000 microVMs, a significant increase in CPU overhead on the node is observed. Massive instances attached to cgroups are maintained within the system by the Completely Fair Scheduler (CFS). In a serverless worker node, there can be more than 10,000 cgroups with thousands of sandboxes. The Per-Entity Load Tracking (PELT) mechanism used for load balancing in the CFS must iterate over all cgroups and processes. Therefore, frequent context switching and hotspot functions that involve high-precision calculations in the scheduler become bottlenecks. According to our measurements, they account for 7.6% of CPU cycles on the physical node.

The host-side overhead of cgroups prohibits the high density deployment and high-concurrency startup when scaling out microVMs. Simplifying the cgroup design,and reducing the critical section introduced by the mutex locks, are fundamental solutions to eliminate the high host-side overhead.

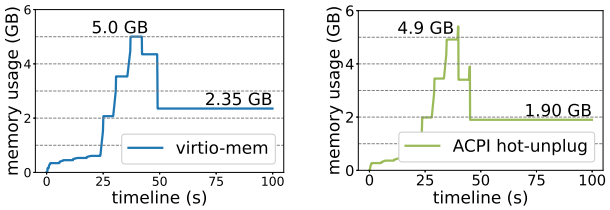
B. Opportunities of Vertical Scaling-up

1) ***GuestOS Overhead Amortization & Marginal Benefit****:* As discussed before, the additional layer of isolation enhances the overall security of the serverless environment. However, this comes at the cost of introducing a full guest environment, which results in additional memory overhead and can significantly reduce the deployment density per node. Taking the Alpine Linux as a lightweight guest environment example, we monitored the overall memory footprint of Kata microVM using QEMU and Firecracker [8] as hypervisors respectively, as illustrated in the Figure 6.

Figure 6(a) first presents the memory footprint and variation of the microVM guest as its memory specification becomes larger. The memory overhead for a 128MB container is 94MB with Kata-FireCracker and 168MB with Kata-qemu. The use of struct page in the Linux kernel introduces memory allocation overhead proportional to the microVM's memory size, with approximately 16MB of metadata required per 1GB of allocated memory. However, compared to the overhead of deploying a separate guestOS for each container replica, this cost is negligible because it is amortized across multiple container replicas within a single microVM.

Since the memory overhead of a microVM is amortized by container replicas within it, there are two primary strategies to increase deployment density: reducing the memory overhead of the base guestOS and increasing the number of container replicas hosted by a microVM. Templating [26], [28] is a popular method to reduce the per-microVM memory overhead, and using *mmap* of kernel also helps share the text/rodata segment among multiple microVMs. Above methods enable new microVMs created in CoW (Copy-on-Write) manner. However, when 1,000 microVMs are deployed on a single node using templating and kernel *mmap*, the overhead remains significant for serverless containers with a 128MB memory specification. For example, the per-microVM overhead decreases to 145MB for Kata-qemu using templating and 71MB for Kata-Firecracker using *mmap*, respectively.

Though increasing the number of container replicas within a microVM can amortize the overhead of guestOS to a greater extent, a larger microVM size also leads to greater complexity introduced by load balancing. An experiment was conducted to determine the optimal number of containers a virtual machine should host, with the results shown in Figure 6(c). The application used in the experiment is a regularly sized matrix multiplication (matmul) with a memory specification of 256MB. The results indicate that for 2 to 8 containers, the



(a) Virtio-mem (b) ACPI Hot-unplug

Fig. 7: The memory usage and that after memory hot-unplug.

benefits of sharing are significant. For 8 containers and above, the amortized cost of the kernel becomes negligible, making benefits of sharing marginal. Based on this observation, we suggest that 8 container replicas per microVM is an ideal limitation for general purposes.

Due to the marginal benefits of memory footprint amortization and the constraints imposed by load balancing, the number of container replicas that a microVM can host and its memory specification are inherently restricted. These limitations highlight the need to condense the base guestOS and optimize the microVM templating efficiency.

*2) **Memory Hot-(un)plug Verification**:* The vertical scaling of MicroVMs requires memory hot-plugging tools. Currently, there are two mainstream implementations: virtio-mem and ACPI hotplug. Virtio-mem is a semi-virtualized memory device based on VIRTIO, while ACPI hotplug simulates memory hot-plugging based on DIMM. Both memory hot-plugging methods effectively insert memory into the VM. However, during our tests of memory hot-unplugging, we discovered that the existing technologies each have their own shortcomings.

Figure 7 illustrates the memory changes within a microVM using two different memory downsizing methods. In our experiment, we creat three 2GB-sized containers within the microVM and executed matrix multiplication within these containers to simulate memory allocation. About 40 seconds later, we trigger the sequential reclamation and memory hot-unplugging of two internal containers. Our expected result is that the memory usage atfer unplugging would equal the memory allocated by the remaining container (approximately 2GB). However, as shown in Figure 7(a), virtio-mem method returned 14.5% less memory than expected. It is because that virtio devices may mark pages that have never been used by the guestOS rather than previously used but currently unallocated pages as offline first, when removing memory pages from the guestOS memory manager. Therefore, virtio-mem fails to guarante memory consistency between host and guest after multiple unplug operations.

On the contrary, DIMM-based ACPI can correctly return the to-be-unplugged pages (struct page overhead excluded). It benefits from the memory device model that it must return all host physical pages that were occupied by the corresponding physical memory of the guest. However, as the spike during each unplugging shows in Figure 7(b), page migration issue arises because each container does not exclusively allocate memory from a single memory device.Consequently, the memory pages allocated by a single container are dispersed across all memory devices in guestOS. This fragmented page distribution triggers page migration which degrades the memory bandwidth during device hot-unplugging.

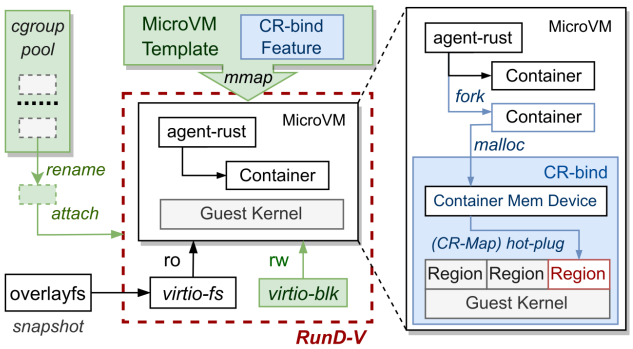


Fig. 8: The lightweight serverless runtime *RunD-V*. Green and blue blocks represent the Guest-to-Host template and CR-bind supporting microVM scaling-out and scaling-up, respectively.

Considering that the Multi-Replica paradigm necessitates frequent scaling up/down of microVMs, ensuring the stability and consistency of memory hot-(un)plug operations is fundamental. However, virtio-mem is not recommended because of its inherent limitations. While ACPI can ensure consistent memory reclamation, it does not provide sufficient stability during hot-unplugging, and additional mechanisms is required to prevent page migration issues.

V. Design Overview of RunD-V

RunD-V is therefore designed to maximize secure container deployment density and startup concurrency by leveraging the Multi-Replica paradigm, which extends the conventional horizontal scaling-out approach with vertical scaling-up. The RunD-V runtime architecture answers three critical questions inherent to the Multi-Replica paradigm:

- How to provide specialized guest-side container rootfs and host-side cgroup architecture that are tailored to the requirements of serverless environments, thereby enhancing the efficiency of horizontal scaling-out? (§Section VI)
- How can we improve the efficiency of microVM templating, which also provides a stable and consistent memory hot-(un)plug support to mitigate page migration issues during vertical scaling-up? (§Section VII)
- How can we design an holistic scaling strategy that optimizes load balancing and maximizes the scaling efficiency of microVMs? (§Section VIII)

Figure 8 shows the RunD-V design and summarizes our methodologies. The green blocks represents the Guest-to-Host runtime template for microVM scaling-out, and blue blocks represents a memory scaling technology call CR-bind to support microVM scaling-up by forking inner containers.

Guest-to-Host Runtime Template for MicroVM. As shown in Figure 8, RunD-V leverages the features of read-only data/runtime and non-persistent storage in serverless environments, proposing guest-to-host solutions. Specifically, the RunD-V runtime performs a read/write device split by providing the read-only layer to *virtio-fs* and using the built-in storage to create a volatile writable layer for *virtio-blk*.

These layers are mounted together as the final container *rootfs* using *overlayfs*. When a secure container is created, RunD-V renames and attaches a lightweight cgroup from the cgroup pool for efficient management.

Container and Memory Region Binding (CR-bind).

As shown in the right part of Figure 8, the scaling-up is facilitated by CR-bind, a feature that designed and built into the guest kernel. CR-bind exploits the granularity characteristic of memory allocation in serverless environments, typically specified in increments of 128MB, to facilitate the hot-(un)plug of memory regions within a Serverless microVM. the principle of RunD-V for memory hot-(un)plug within a Serverless microVM is to manage *Memory Regions* based on container specifications. Each memory region is registered and binded with a specific container within the microVM, ensuring its exclusive usage to avoid page migration issue.

Holistic Scaling Strategy. It is imperative to note that RunD-V enforces sequential memory hot-plugging within a single microVM, due to the mutex lock maintained by the Linux kernel's memory management subsystem. To incorporates both vertical and horizontal scaling, the priority of scaling strategies within RunD-V is designed as follows. (1) Upon concurrent requests of a particular function arrive, the serverless scheduler locates all microVMs of this function and initiates a vertical scaling request to them. (2) MicroVMs that do not have CR-bind active will accept the scaling-up request by invoking CR-bind. It will dynamically allocate a memory region, followed by the creation and binding of a new container to this region. (3) MicroVMs that are either currently running CR-bind or have reached their maximum container replica capacity will reject vertical scaling-up requests. (4) Unattended and rejected requests are then addressed through horizontal scaling-out using RunD-V template.

VI. Runtime Template for Horizontal Scaling-out

In this section, we introduce the runtime template, a Guest-to-Host solution that resolves the problem of duplicated data across containers, and high host-side cgroup overhead.

A. Container Rootfs Template and Hybrid Mounting

We investigate the data in a sandbox in the serverless computing scenario, and find that user-provided code/data is read-only for the operating system, and the system-provided runtime files are also read-only for user functions. Serverless architectures primarily depend on external storage (e.g., S3 [36]) for data persistence, but ephemeral writable storage remains crucial for scenarios in which sensitive data must be transferred from a serverless instance to a secure private environment, user-defined operations require ephemeral file handling, or direct inter-function communication is necessary [37]. Therefore, the inner runc-container may only require a volatile writable device that shares the lifetime of its hosting microVM, without any need for data persistence.

Figure 9 shows the way to mount *rootfs* template by a read-only device and a volatile writable device. According to the investigation in Section IV-A1, *virtio-fs* is used to handle the read-only layer, and *virtio-blk* is used to handle the volatile

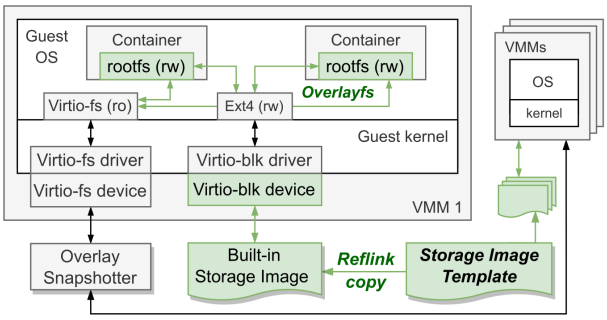


Fig. 9: For container *rootfs*, *Virtio-fs* and *virtio-blk* are used to handle the read-only and volatile writable layer, respectively.

writable layer for better performance. The read-only layer is stored in the host and can be prepared in negligible time when using the overlay snapshotter provided by the container runtime. When multiple container replicas are deployed within a microVM, they share a common virtio-fs layer while each mounts an ext4 filesystem at a isolated path.

We propose the volatile block device as the volatile writable layer, which will not automaticly persist temporary data from user functions. Specifically, the host pre-creates a *storage image template* for all sandboxes, and the *build-in storage image* of a microVM uses symbolic link (e.g., *reflink* [38]) to link the storage image on the host to be block device in the microVM. Once the hypervisor opens this block device, the symbolic link is destroyed. By such means, the *reflink* copy of the writable file template is imperceptible by guests, and only the memory space is occupied in a Copy-on-Write fashion. This volatile writeable layer design relieves the duplicated page cache of container rootfs template for serverless.

According to our evaluation with a 200-concurrency setup, the disk usage is considerably decreased from 60% (4500 IOPS, 100MB/s) to 20% (1500 IOPS, 8MB/s) using rootfs template. The average latency of a rootfs provision is reduced from 207ms to 0.2ms (a reduction of 4 orders of magnitude).

B. Lightweight Cgroup Template and Cgroup Pool

The multi-level design of cgroups leads to chaotic process management, making it less suited as a custom-built solution for serverless environments with large-scale sandboxes. Specifically, cgroups associate a set of tasks with a set of parameters for one or more per-resource-type subsystems. The complex implementation of cgroups involves more than 10 types of resource subsystems (aka. cgroup subsys), making it difficult for multiple controllers to coordinate effectively.

Simplifying the cgroup subsystems and reducing the complexity of cgroup operations is crucial to support high-density deployment and high-concurrency startup in serverless scenarios. An intuitive method is to reduce the scope of the critical section in cgroup mutex, or find a fast path to perform cgroup operations when secure containers are created and deleted.

Our further investigations reveal the optimization opportunity that pre-creating and maintaining cgroups in a pool can effectively reduce the creation overhead, since afterward only the cgroup `rename` is used. The cgroup `rename`, as a special

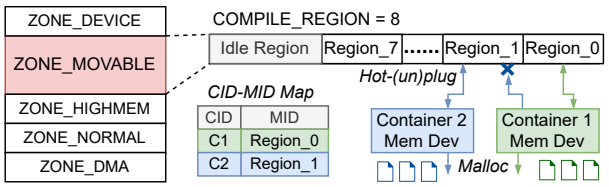


Fig. 10: The tailored guest kernel incorporates a patch that partitions the ZONE_MOVABLE into several distinct regions. Each container's memory device is exclusively and independently assigned to one of these regions for page isolation.

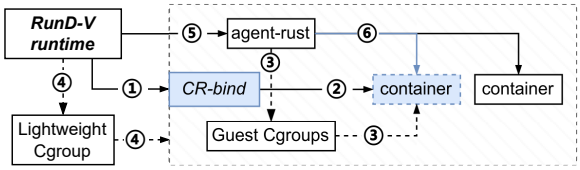


Fig. 11: The vertical scaling-up and resource management workflow of RunD-V. The solid line indicates synchronization and the dashed line indicates asynchronization.

case, is a lightweight operation without acquiring any global lock. To this end, we propose a lightweight cgroup template and the cgroup template pool management.

Rather than creating the cgroup for each subsystem, a joint cgroup controller aggregates necessary cgroup subsystems (aka the *cpu*, *cpuacct*, *cpuset*, *memory*, and *blkio*) into one dedicated lightweight cgroup template. RunD-V pre-creates corresponding lightweight cgroup templates and maintains them in a cgroup pool based on the pre-defined node capacity. These cgroup templates are marked idle when initialized, and are protected in a linked list. For each created container, RunD-V simply allocates an idle cgroup template, updates the state to busy, performs the cgroup `rename`, and then attaches the container to this renamed cgroup template when a container is started. If a container triggers deleting, RunD-V will take the cgroup template back to the pool, kill the corresponding instance process, and then update the returned cgroup template state to idle for subsequent allocating and renaming.

Adopting the above optimizations in kernel mode, we replay the evaluation in Section IV-A2. The cgroups creation only consumes $0.09s$ (1 thread), $0.1s$ (50 threads), and $0.14s$ (200 threads), respectively. Compared with the default mechanism, the lightweight cgroup and the rename-based cgroup pool reduce 94% of the cgroups creation time.

VII. CR-bind for Vertical Scaling-up

In this section, we present our optimizations for tailoring the guest kernel and introduce the CR-bind feature, which enables stable and efficient memory hot-(un)plug operations.

A. Guest Kernel Tailoring

Because the guest environment management for serverless containers is offloaded to the cloud provider, RunD-V can leverage the opportunity to optimize microVM templating efficiency by eliminating redundant features that are unnecessary in a serverless setting. Specifically, we tailor the Linux-based guest operating system kernel by disabling extraneous features: loop device (2.2MB), *acpi* (2MB), *ftrace* (6MB), *graphics*-related items (2MB), *i2c* and *ceph* (3MB).

Validating all features at compile-time case by case, RunD-V effectively reduces the memory footprint of a CentOS 4.19 kernel by about 16MB and tailored the image by about 4MB.

Based on the tailored kernel, we further propose CR-bind, a guest kernel feature that enables binding a container to a designated *Memory Region*. In Linux, ZONE_MOVABLE is a dedicated memory zone intended for tasks such as memory compression and migration. CR-bind subdivides ZONE_MOVABLE into different regions that are arranged in a linear list ordered from high to low addresses, as illustrated in Figure 10. These regions are pre-compiled into the guest kernel via kernel patch. The principle of CR-bind for memory hot-(un)plug operations is to designate a specialized memory region that can be exclusively allocated to a container. In this way, CR-bind avoids fragmented page distribution, effectively addressing potential memory fragmentation over time as containers are frequently created and deleted.

Specifically, upon completion of a container replica fork, the target RegionID (RID) is stored in the newly created container's *task_struct*, and its corresponding ContainerID (CID) is recorded in a CID-MID map. Subsequently, the guestOS activates the memory device, which is then hot-plugged into the region specified by the CID-MID map. When the container initiates a page allocation request whether via CoW or a direct allocation method such as mmap, CR-bind modifies the allocation request's maximum region index to ensure that the page request is directed to the targeted region.

Similarly, when a container terminates, CR-bind initiates region refreshing, and the binded memory device becomes ready for hot-unplug and offline deactivation. The anonymous pages and page cache of this container are released, and the hypervisor deletes the virtual memory device backend on the host, thereby releasing memory back to the host.

B. Workflow of CR-bind Resource Management

When multiple container replicas of a function coexist within the same RunD-V sandbox using CR-bind, managing resource contention among them becomes increasingly complex. In addition to memory, other resources (e.g., CPU and network) must also be dynamically adjusted and isolated as required to maintain optimal performance and security.

It becomes essential to not only establish resource limits on the host's lightweight cgroup but also to utilize guest cgroups to manage all resources for multiple container processes within the microVM. For illustration, we take the CPU resource scaling as an example, and other resources (e.g., Net TAP devices) following a similar logic. The specific steps for CPU vertical scaling in RunD-V are depicted in Figure 11.

First, the RunD-V runtime routes the request to the function agent. Upon detecting that all containers within the microVM are busy, RunD-V triggers vertical scaling by invoking CR-bind to hot-plug a memory region for the container (**Step** ①). Second, CR-bind will fork the initial container and then

the memory region is then bound to the container, while the container remains paused in the background (**Step** ②). Third, the agent calls guest cgroups to limit CPU and memory usage and attaches these limits to the container process (**Step** ③). The separation ensures that resource management within the guest can be handled independently without imposing additional burdens on host cgroups. Concurrently, RunD-V locates the lightweight cgroup attached to the microVM and increases the resource limits according to the container's specifications (**Step** ④). Finally, the RunD-V runtime notifies the agent within the microVM to add the new container to its scheduling group, completing the vertical scaling (**Steps** ⑤ **and** ⑥).

VIII. Methodology of Holistic-scaling

Supporting scaling-out and scaling-up within RunD-V, a dedicated strategy is required to optimize load balancing and maximizes the scaling efficiency of microVMs. A straightforward method is the vertical-first strategy, where a single microVM is filled to capacity before horizontally scaling-out the next microVM. When the fluctuation in function requests does not require a high frequency of container scaling, this method can fully utilize the low memory overhead characteristic of vertical scaling to maximize deployment density. However, when high concurrency requests necessitate substantial scaling of function containers, this approach is not optimal.

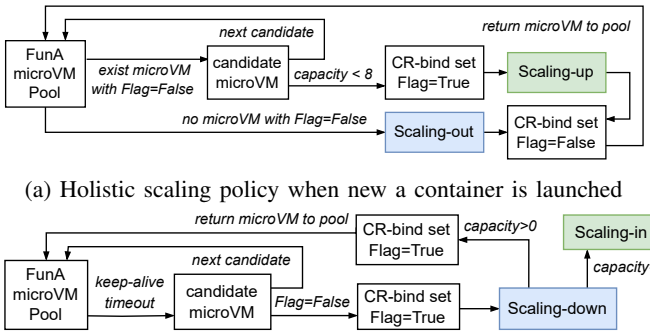
Firstly, in Linux memory management, the use of mutex locks causes concurrent memory hot-plugging within a microVM to degrade significantly. Secondly, the container replicas of vertical scaling that a single microVM can support is limited due to load balancing consideration (e.g., 8 discussed in Section IV-B1). When the concurrency demand exceeds this limit, the system will revert to a higher-overhead horizontal scaling fashion. Therefore, there is a trade-off to be considered: leveraging the higher density provided by vertical scaling while sacrificing concurrency capability.

Based on above investigations, the principle of scaling for RunD-V should follow two guidelines:

- When the container scaling frequency is low, vertical scaling should be the primary approach to maximize the deployment density of functions.
- When the container scaling frequency is high, it indicates to leverage horizontal scaling-out as well, while performing vertical scaling-up within multiple microVMs.

To this end, we propose a holistic scaling strategy that combines scaling-out and scaling-up mechanisms by maintaining a CR-bind $Flag$. When a microVM is performing vertical scaling via CR-bind, this $Flag$ is set to True. Suppose there is a function $FunA$ that needs to create a container, as illustrated in Figure 12(a). RunD-V first checks whether a candidate microVM exists in FunA's microVM pool that is capable of vertical scaling-up. If such a candidate exists and its number of inner container replicas is fewer than limit (e.g., 8), the microVM is locked by setting $Flag = True$ and then released after the vertical scaling-up operation completes. Otherwise, if no suitable candidate is found, the system directly triggers scaling-out to create a new microVM to host the new container.

Figure 12(b) illustrates the logic for recycling container replicas. When a container remains idle beyond the keep-alive timeout, RunD-V identifies the microVM that hosts this container. If that microVM is currently performing memory hot-unplugging, it is skipped in favor of the next candidate. Otherwise, RunD-V performs a scaling-down operation to reclaim this container within the microVM, and once the microVM is empty, a scaling-in operation is triggered to recycle this microVM.



(a) Holistic scaling policy when new a container is launched

(b) Holistic scaling policy when recycling a timeout container

Fig. 12: The holistic scaling strategy of RunD-V that handles creating and recycling a container in horizontal/vertical way.

By this method, we achieve an overlap of horizontal and vertical scaling, striking a balance between high-density deployment and high-concurrency startup capabilities.

IX. Evaluation

In this section, we evaluate the performance of our proposed guest-to-host runtime template and CR-bind in supporting high-concurrency startup and high-density deployment.

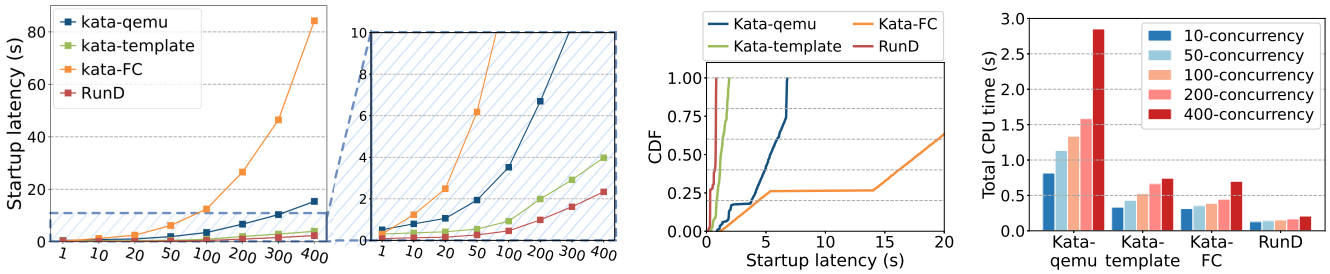
A. Evaluation Setup

Since the holistic scalable RunD-V incorporates both horizontal and vertical optimizations for serverless runtime, we evaluate these two dimensions separately.

Scaling-out Baselines: In experiments B,C,D, we disable CR-bind, at which point RunD-V degress to the Single-Replica paradigm. This variant cannot support verticall scalng-up, therefore we refer to it as RunD. We compare RunD with the state-of-the-art secure container, Kata Containers [10] to evaluate the efficiency of our guest-to-host template for microVM sacling-out. Specifically, we use three popular configurations of Kata containers: *Kata-qemu*, *Kata-template*, and *Kata-FC*. *Kata-qemu* uses QEMU [39] as the hypervisor, *Kata-template* uses QEMU while integrating container template, *Kata-FC* uses lightweight FireCracker [8] as the hypervisor.

Scaling-up Baseline: In experiments E,F,G, we compare the performance of the variant RunD (only scaling-out) and RunD-V that supports both guest-to-host runtime template and CR-bind (both scaling-out and scaling-up). Besides efficiency of scaling-up, we compare CR-bind with ACPI hot-plug to assess the performance of memory hot-unplug operations. They are all evaluated based on the integration within RunD-V. Table I shows the detailed setups.

Testbed: we run the experiments on a node with 104 virtual cores, 384GB memory, and two SSD drives of 100GB



(a) End-to-end startup latency with different concurrency (b) Latency distribution (c) CPU time

Fig. 13: The startup metrics with different runtime and concurrency: (a) e2e latencies of concurrent startups. The right figure shows $y \in [0, 10]$. (b) CDF of startup latencies (200-way concurrent launch). (c) CPU usage of concurrent startups.

TABLE I: Experiment setup in our evaluation.

	Configuration	
Hardware	CPU: 104 vCPUs (Intel Xeon Platinum 8269CY) Memory: 384GB, two SSD drives: 100GB, 500GB	
Software	OS: CentOS7, kernel: Linux kernel 4.19.91	
Runtime	kata-qemu	containerd 1.3.10, kata 1.12.1
	kata-FC	containerd 1.5.8, kata 2.2.3
	kata-template	containerd 1.3.10, kata 1.12.1
	RunD-V (RunD)	containerd 1.3.10

and 500GB. Such specification is widely-used in production clouds. The 100GB drive is used as the root filesystem of the host operating system, and the 500GB drive is used by the secure containers. We use Alpine Linux as the guest operating systems in the microVM for a low memory footprint.

Measurement: The actual memory usage of a container is collected using the `smem` command. In evaluations B, C, and D, we create pod sandboxes using the `crictl` command without including function containers. Since the variant RunD aims to maximize container startup concurrency and deployment density only using guest-to-host runtime template, we initiate empty secure containers without user codes or data. This approach reflects a common practice in serverless environments, where empty containers are often started concurrently for prewarming purposes.

In evaluations E, F, and G, we assess memory hot-plug efficiency. Since CR-bind focuses on optimizing memory usage across containers rather than on CPU utilization, we use the matmul workload from FunctionBench [40] to evaluate CR-bind's memory efficiency. This workload allows us to thoroughly test CR-bind's capability to manage memory effectively. Memory addition or removal, whether for a single container or through parallel hot-(un)plug operations, is executed synchronously for both CR-bind and ACPI hot-(un)plug. Additionally, we measure end-to-end latency based on the atomic operations of container creation and deletion.

B. Concurrent Scaling-out Measurement

In this experiment, we focus on three critical metrics related to user experience: *(1) the time needed to start a large number of sandboxes concurrently*, *(2) the startup latency distribution of the sandboxes*, and *(3) the CPU overhead on the host*. The first metric reveals the throughput of starting sandboxes, and the second metric reveals the experience of every user.

As for the first metric, Figure 13(a) shows the time needed to start sandboxes concurrently. In the figure, the x-axis shows the number of sandboxes to be started concurrently, the y-axis shows the overall time needed to startup all the sandboxes.

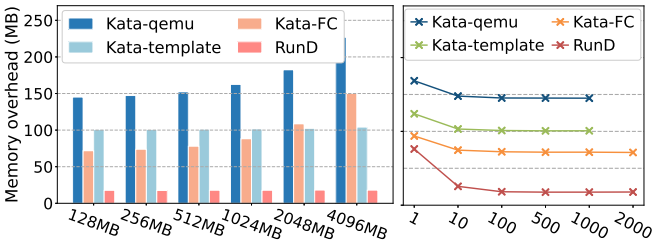
As shown in the figure, RunD uses the shortest time to start a large number of sandboxes for all concurrency levels. When 200 containers are created concurrently (we already observe such high-concurrency in Alibaba serverless platform), Kata-FC, kata-qemu, kata-template, and RunD needs $47.6s$, $6.85s$ and $2.98s$ and $1s$ to create them. Kata-FC requires a much longer time to startup the sandboxes when the concurrency is high. This is because Kata-FC uses *virtio-blk* to create *rootfs*, and the performance is poor at high-concurrency, as we measured in Section IV-A1. There is no such bottleneck in Kata-template and Kata-qemu. Kata-template simply uses template to reduce the overhead of guest kernel and *rootfs* loading, but the inefficient *rootfs* mapping, untailored guest kernel and high host-side overhead of the cgroup operations still exists. As a result, it performs worse than RunD at high startup concurrency. The overall optimizations suggest that RunD provides the performance improvement of about 40% over its nearest baseline, Kata-template, at high-concurrency (e.g., 400-way) startup.

As for the second metric, Figure 13(b) shows the latency distribution of starting each sandbox, when 200 sandboxes are started concurrently. RunD and Kata-template are able to start sandboxes in a stable short time, but the latencies of starting sandboxes with others are out of expected. Users can have identical good experiences with RunD.

As for the CPU overhead, Figure 13(c) shows the CPU time needed on the host to startup sandboxes. When the concurrency is high, RunD greatly reduces the CPU overhead. For instance, when 200 sandboxes are started concurrently, RunD reduces 89.3%, 74.5% and 62.1% CPU overhead compared with Kata-qemu, Kata-template, and Kata-FC, respectively. In addition, the CPU overhead of RunD only increases slightly, when the concurrency increases. This is due to the container rootfs template and the reduction of compute-intensive operations in cgroups. Therefore, RunD is scalable in starting more sandboxes concurrently.

C. Deployment Density of Guest-to-Host Runtime Template

In this experiment, we evaluate the effectiveness of RunD in increasing the sandbox deployment density. In general, the



(a) Specification (b) Amortization

Fig. 14: The memory overhead of specification and the amortizated memory. The missing point around 2,000 in figure(b) indicates the over-subscription for physical memory space.

memory used by each container determines the deployment density, while the CPU time needed by each function invocation is minor in the serverless platform. Figure 14 shows the memory overhead when 100 sandboxes are deployed on the experimental node. In the figure, the x-axis shows the memory specification of each sandbox.

As observed in Figure 14(a), RunD has the least memory overhead among four runtimes, and does not increase with the memory specification. The memory overhead is less than $20MB$ per sandbox with RunD. Compared to kata-qemu, kata-template and kata-FC, the overhead of RunD is reduced by 88.2%, 82.9%, and 76.1%, respectively, even when the memory specification is 128MB. The memory overhead does not increase, because the microVM template technique uses the on-demand memory loading for the containers. Therefore, the page table required for memory management is determined by the actually used memory space. On the contrary, the memory overheads introduced by Kata-qemu and Kata-FC increase with larger memory specifications, as the page table is built for all available memory.

Figure 14(b) shows the average memory overhead of sandboxes when more sandboxes are deployed on a node. The x-axis shows the deployment density. As observed, the average memory overhead reduces with the deployment density, as the sandboxes share the mapped code/data segments. RunD reduces the memory overhead by 87.7%, 82.4%, and 75.1% when 1,000 sandboxes are deployed, respectively, compared with kata-qemu, kata-template, and kata-FC.

D. Impact of Deployment Density on Scaling-out

When some sandboxes are already deployed on a node, the performance of starting sandboxes concurrently is affected. Figure 15 shows the time needed to boot 10 and 200 sandboxes, when some sandboxes are already deployed on the node. The x-axis shows the number of already deployed sandboxes. The y-axis is in the log10 scale.

When 1,000 sandboxes are already deployed, the time needed to startup 10 containers increases by $1.69s$, $0.41s$, $10.8s$, and $0.22s$ compared with the cases in Figure 13(a) with Kata-qemu, Kata-template, Kata-FC, and RunD. In addition, the time needed already increases with the number of already deployed sandboxes. For RunD, the time needed only increases slightly beacuse of reduced cgroup operations.

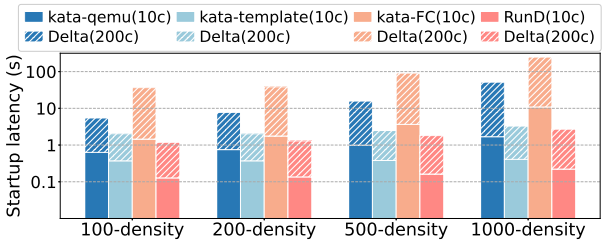
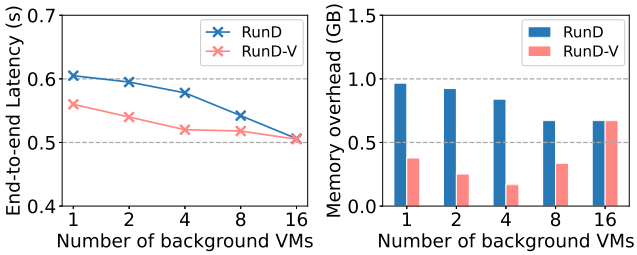


Fig. 15: The end-to-end startup latency at different deployment densities. (10c/200c means a 10/200-way concurrent startup, and the Delta means the overhead increment compared with a 10-way concurrent statrup).



(a) End-to-end latency (b) Total memory overhead

Fig. 16: The end-to-end latency and total memory overhead of burst load (QPS=24) when function have different background secure container. Each microVM creates 8 containers at most.

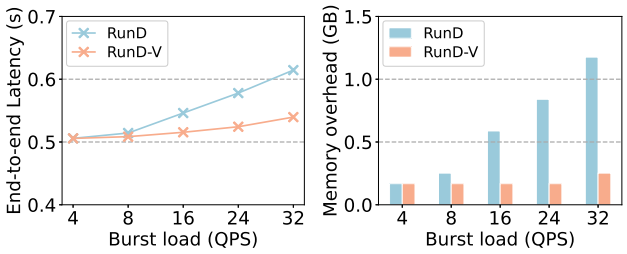
We can also observe that, the time needed to start 200 sandboxes is at least 10 times as much as that needed to start 10 sandboxes at a 1,000-density deployment in all the tests. The significant increase originates from a large number of cgroups in the host operating system. Scheduling and managing containers with these cgroups consume more CPU cycles, thus resulting in CPU bottlenecks appearing earlier than a low-density deployment. The increased time is the smallest with RunD, because it already eliminates many time-consuming cgroup operations.

E. Popularity Impact for Scaling-up Efficiency

In evaluating the efficiency of vertical scaling in terms of concurrency capacity and memory usage, we identified two primary influencing factors. One is the number of microVMs already created in the background for the current function, and the other is the number of concurrent requests.

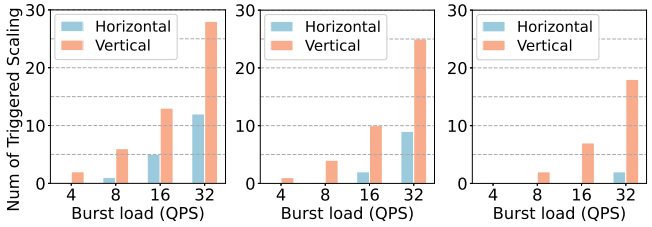
In this experiment, we fixed the number of concurrent QPS (Queries Per Second) at 24 and studied the impact of the number of background microVMs on vertical scaling 128MB-sized containers. It is important to note that the number of background microVMs reflects the current load popularity to some extent; the more microVMs there are, the higher the invocation frequency of the current function.

Figure 16(a) shows that the e2e invocation latencies decrease with more microVMs (or as popularity increases), regardless of whether CR-bind is enabled. When the number of microVMs reaches 16, the current QPS can be fully managed through container reuse, so RunD-V will not further trigger vertical scaling. However, when the number of microVMs is



(a) End-to-end latency (b) Total memory overhead

Fig. 17: The end-to-end latency and total memory overhead when 4 background VMs serve different burst loads.



(a) 1 MicroVM (b) 2 MicroVMs (c) 4 MicroVMs

Fig. 18: The number of triggered horizontal and vertical scaling when different number of background microVMs exist.

lower, the CR-bind feature effectively reduces the frequency of scaling-out, decreasing request queuing and container creation by 57.9% compared to the variant RunD without CR-bind.

Similarly, Figure 16(b) shows that enabling vertical scaling in RunD-V results in a more efficient amortization of the guestOS overhead by the containers, leading to a lower overall memory footprint. Notably, when the number of background microVMs is fewer than 4, the overlap mechanism of vertical and horizontal scaling primarily triggers container creation within microVMs, resulting in lower memory overhead—at least 40% lower than RunD without CR-bind enabled. However, when the number of microVMs exceeds 4, background microVMs are sufficiently numerous, reducing the frequency of vertical scaling, which in turn increases memory usage.

F. Burst Load Impact for Scaling-up Efficiency

In contrast to the previous experiment, we fixed the number of background microVMs at 4 and continuously increased the concurrent QPS to observe its impact on request end-to-end latency and overall memory overhead.

As shown in Figure 17(a), with an increasing QPS, RunD-V's holistic scaling strategy reduces the latency that queuing for sandboxes creation. For low-popularity functions, the request processing efficiency at QPS=8 and QPS=16 of RunD is roughly equivalent to RunD-V's concurrent processing capability at QPS=24 and QPS=32.

Similarly, as shown in Figure 17(b), memory usage increases at QPS=24 due to the triggering of horizontal microVM scaling-out. The reduction in memory usage is proportional to the maximum number of containers that a single microVM can support. Since we recommend setting this number to 8, the memory overhead for the same function is approximately reduced to $1/8$.

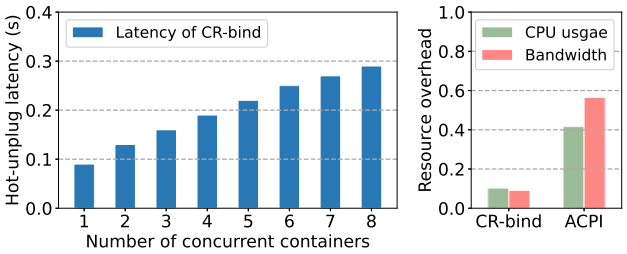


Fig. 19: The memory hot-unplug comparison when different number of containers release memory. The overhead is normalized to the resource occupation of a idle container.

Figure 18 illustrates the benefits of vertical scaling-up, complementing the data shown in Figure 17 and 16. It depicts the number of horizontal and vertical scaling events initiated by RunD-V when handling burst loads at various QPS levels with different number of background microVM. Under high QPS conditions, RunD-V initially scales out a little number of microVMs due to memory lock constraints; subsequent requests are then handled through vertical scaling and reuse of existing container replicas, reducing end-to-end latency and average memory consumption.

G. MicroVM Scaling-down and Interference

Although scaling up microVMs requires low latency, the serverless scheduler do not impose strict latency constraints for resource recycling, such as deleting containers and memory devices. As illustrated in Figure 19, the latency for deleting containers and hot-unplugging memory devices remains low, even under 8 concurrent requests, thus can meet the real-time resource monitoring requirements of the scheduler. Our main concern is whether hot-unplugging memory incurs significant overhead on other containers within the same microVM.

We manually triggered a container reclamation within the microVM and monitored the CPU overhead and memory bandwidth consumption of the thread responsible for hot-unplugging devices within the guest. The right part of Figure 19 presents the collected data. Since CR-bind is implemented on top of ACPI hot unplug, it binds the container process and memory region to avoid the page migration issues that typically occur during memory device hot-unplugging. Compared to the resource usage in an idle state, the CPU and memory bandwidth consumption were reduced from 41.6% and 56.5% to 10.3% and 9.1%, respectively.

H. Discussion and Future Work

RunD-V is primarily focused on the architecture of secure containers. However, emerging processors include various built-in accelerators (e.g., QAT, IAA and DSA), presenting opportunities to offload the high CPU-overhead and memory-sensitive operations during secure container launching. The adaption of these features on hostOS is a promising solution to futher improve the concurrency/denisty of secure containers.

When discussing the security model of the Multi-Replica paradigm, containers are not treated as a shareable layer. Although a container can host multiple user code processes,

such sharing needs intrusive runtime adaptation which risks compromising the isolation and correctness of function execution. Enhanced isolation mechanisms that leverage language runtimes for managing multiple function processes present a promising research direction to enable such container sharing.

X. Conclusion

In serverless computing, the lightweight and short-term nature of functions necessitates high-density container deployment and high-concurrency container startup. This work addresses bottlenecks across the entire software stack and proposes RunD-V, a lightweight and holistic-scalable serverless secure container runtime. RunD-V achieves its goals through a guest-to-host runtime template for scaling-out and CR-bind for scaling-up. The holistic scaling mechanism enables the overlap of these two scaling methods, further enhancing high-density deployment and high-concurrency startup capabilities.

Acknowledgments

This work is partially sponsored by National Natural Science Foundation of China (62232011, 62302302, 61832006). Quan Chen is the corresponding author. Jie Wu's work was done while he was on leave working as a scientist in China Telecom. We acknowledge the Alibaba Cloud for their technical contributions to horizontal scaling implementations in our prior work RunD [41], published at USENIX ATC'2022.

We further acknowledge the China Telecom Cloud Computing Research Institute for their research collaboration and deployment support in advancing the vertical and holistic scaling methodologies of RunD-V in this extended study.

References

[1] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: enable efficient workflow execution for function-as-a-service," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2022, pp. 782–796.

[2] R. Buyya, S. N. Srirama, G. Casale, R. N. Calheiros, and et al., "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 105:1–105:38, 2019.

[3] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019*. ACM, 2019, pp. 1063–1075.

[4] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13*. USENIX Association, 2018, pp. 133–146.

[5] S. Hendrickson, S. Sturdevant, E. Oakes, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *login Usenix Mag.*, vol. 41, no. 4, p. 33–39, 2016.

[6] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and A. Shulman-Peleg, "Secure yet usable: Protecting servers and linux containers," *IBM J. Res. Dev.*, vol. 60, no. 4, p. 12, 2016.

[7] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of linux containers," in *2015 IEEE Conference on Communications and Network Security, CNS 2015*. IEEE, 2015, pp. 559–567.

[8] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27*. USENIX Association, 2020, pp. 419–434.

[9] D. R. Engler, M. F. Kaashoek, and J. W. O. Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995*, M. B. Jones, Ed. ACM, 1995, pp. 251–266.

[10] "Kata containers." https://katacontainers.io/, 2024.

[11] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, and et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, Virtual, July 15-17, 2020*. USENIX Association, 2020, pp. 205–218.

[12] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019*. ACM, 2019, pp. 59–73.

[13] F. Schmidt, "uniprof: A unikernel stack profiler," in *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. ACM, 2017, pp. 31–33.

[14] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 218–233.

[15] H. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *EuroSys '20: Fifteenth EuroSys Conference 2020*. ACM, 2020, pp. 11:1–11:15.

[16] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13*. USENIX Association, 2018, pp. 923–935.

[17] J. Cheng, Y. Zhao, Z. Li, Q. Chen, W. Cui, and M. Guo, "Microless: Cost-efficient hybrid deployment of microservices on iaas vms and serverless," in *29th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2023, Ocean Flower Island, China, December 17-21, 2023*. IEEE, 2023, pp. 2303–2310.

[18] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021, pp. 22:1–22:15.

[19] Q. Chen, S. Xue, S. Zhao, S. Chen, Y. Wu, Y. Xu, Z. Song, T. Ma, Y. Yang, and M. Guo, "Alita: comprehensive performance isolation through bias resource management for public clouds," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*. IEEE/ACM, 2020, p. 32.

[20] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019*. IEEE, 2019, pp. 9–16.

[21] A. Mohan, H. Sane, K. Doshi, and S. Edupuganti, "Agile cold starts for scalable serverless," in *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'19. USENIX Association, 2019, p. 21.

[22] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 57–70.

[23] M. G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017*. IEEE Computer Society, 2017, pp. 405–410.

[24] "Firecracker snapshotting," https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md, 2024.

[25] M. Nelson, B. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the 2005 USENIX Annual Technical Conference, Anaheim, CA, USA, April 10-15*. USENIX, 2005, pp. 391–394.

[26] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*. ACM, 2020, pp. 467–481.

[27] K. A. Wang, R. Ho, and P. Wu, "Replayable execution optimized for page sharing for a managed runtime environment," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, pp. 39:1–39:16.

[28] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *ASPLOS '21: 26th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, 2021. ACM, 2021, pp. 559–572.
[29] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand container loading in AWS lambda," in *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 2023, pp. 315–328.
[30] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, and et al., "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019.
[31] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from bell labs," *Comput. Syst.*, vol. 8, no. 2, pp. 221–254, 1995.
[32] "virtio-fs." https://virtio-fs.gitlab.io, 2024.
[33] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.
[34] V. Tarasov, L. Rupprecht, D. Skourtis, and et al., "In search of the ideal storage configuration for docker containers," in *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, 2017*. IEEE Computer Society, 2017, pp. 199–206.
[35] V. Tarasov, L. Rupprecht, D. Skourtis, W. Li, R. Rangaswami, and M. Zhao, "Evaluating docker storage performance: from workloads to graph drivers," *Clust. Comput.*, vol. 22, no. 4, pp. 1159–1172, 2019.
[36] "Amazon s3-cloud object storage," https://aws.amazon.com/cn/s3/, 2024.
[37] Z. Li, C. Xu, Q. Chen, J. Zhao, C. Chen, and M. Guo, "Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 2023, pp. 57–72.
[38] Y. Zhan, A. Conway, Y. Jiao, N. Mukherjee, I. Groombridge, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, D. E. Porter, and J. Yuan, "How to copy files," in *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27*. USENIX Association, pp. 75–89.
[39] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, Anaheim, CA, USA, April 10-15*. USENIX, 2005, pp. 41–46.
[40] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *12th IEEE International Conference on Cloud Computing, CLOUD 2019*. IEEE, 2019, pp. 502–504.
[41] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *2022 USENIX Annual Technical Conference*. USENIX Association, 2022, pp. 53–68.

XI. Biography Section

Zijun Li is currently a postdoc research fellow working with Prof. Xueyan Tang and Prof. Minyi Guo, jointly supported by the Nanyang Technological University, Singapore, and Shanghai Jiao Tong University, China. He received his Ph.D. degree from Shanghai Jiao Tong University, China. His research interests include cloud-native system and general-purpose serverless computing.

Chenyang Wu is currently a MSc student in the field of computer science supervised by Prof. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. He received his BSc degree from Shanghai Jiao Tong University, China. His research interests include serverless computing, distributed and parallel computing.

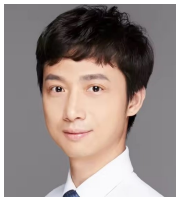
Chuhao Xu is currently a Ph.D student in the field of computer science supervised by Prof. Quan Chen in Department of Computer Engineering Faculty of Shanghai Jiao Tong University, China. He received his B.Sc. degree from Shanghai Jiao Tong University, China. His research interests include serverless computing and datacenter resource management.

Quan Chen is a professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include High performance computing, task scheduling and resource management in various architectures, runtime system and operating system. He got his Ph.D. degree at June 2014 from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.

Shuo Quan receivedthe B.S. and M.S. degrees from the School of Software, Beijing University of Posts and Telecommunications, Beijing, China, in 2013 and 2016, respectively. He is currently an Engineer with the China Telecom Research Institute, Beijing. His current research interests include deep learning, big data, and cloud computing.

Bin Zha is an Engineer with Alibaba Cloud. His primary focus includes working on operating systems, virtualization, serverless computing, and secure containers. Currently, he is responsible for constructing the AI serverless infrastructure, ensuring its robustness, scalability, and security.

Qiang Wang is currently a Research and Development Expert at Alibaba Cloud, with primary interests in the technical research and development of security containers, confidential containers, Al Serverless infrastructure, and the landing of enterprise production applications.



Weidong Han is a Technical Director in the Elastic Computing department, Alibaba Cloud. His research interests include virtualization, operating system and serverless computing. Now he is currently working on system software technology research and development.

Jie Wu (Fellow, IEEE, AAAS and CCF) is the director of the Center for Networked Computing and Laura H. Carnell professor with Temple University. His research interests include mobile computing wireless networks, routing protocols, network trust and security, distributed algorithms, applied machine learning, and cloud computing. He is now on the editorial board of IEEE/ACM Transactions on Networking.

Minyi Guo (Fellow, IEEE and CCF) is currently Zhiyuan Chair professor and head of the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His present research interests include parallel/distributed computing, embedded systems, big data and cloud computing. He is now on the editorial board of IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Cloud Computing and JPDC.