FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes

Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu
State University of New York (SUNY) at Binghamton

ABSTRACT

Fast provisioning of serverless functions is salient for serverless platforms. Though lightweight sandboxes (e.g., containers) enclose only necessary files and libraries, a cold launch still requires up to a few seconds to complete. Such slow provisioning prolongs the response time of serverless functions and negatively impacts users' experiences. This paper analyzes the main reasons for such slowdown and introduces an effective containerization framework, FlashCube. Instead of building a container from scratch, FlashCube quickly and efficiently assembles it through a group of pre-created general container parts (e.g., namespaces, cgroups, and language runtimes). In addition, FlashCube's user-space implementation makes it easily applicable to existing commodity serverless platforms. Our preliminary evaluation demonstrates that FlashCube can quickly provision containerized functions in less than 10 ms (vs. ~400 ms using Docker containers).

1 INTRODUCTION

Cloud computing is moving firmly along a long tradition of ever-higher-level abstractions with lower-complexity user-facing interfaces. One current effort lies in *serverless computing*: Upon a simple *Function-as-a-Service* (FaaS) programming model [1, 13, 14, 25, 29], cloud users program their in-cloud applications as a set of serverless functions and provide such functions to cloud providers. The deployment, management, and runtime operations of these serverless functions are taken care of by serverless platforms. Further, as resource management is fully controlled by serverless platforms, it creates a great opportunity for cloud providers to have better cost efficiencies with more *elastic* resource management.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLOS '21, October 25, 2021, Virtual Event, Germany
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8707-1/21/10…$15.00
https://doi.org/10.1145/3477113.3487273

Despite the advances in *productivity, elasticity, and cost savings* that fuel the adoption of this new computing paradigm, serverless platforms face new challenges in light of the unique characteristics of FaaS-based serverless applications: Unlike traditional, monolithic cloud applications, a serverless application is decomposed into a collection of *small, short-lived* functions. In a *shared, multi-tenant* cloud environment, each serverless function must be encapsulated and running in an isolated virtual execution environment (VEE), such as virtual machines (VM) and containers, with restricted resources and accesses for isolation. However, the construction of a VEE is costly, especially for small, short-lived functions.

For instance, while the function code runs only for hundreds of milliseconds, a serverless platform can take several seconds to construct the VEE before the function execution — e.g., even with lightweight containers (detailed in Section 3). This cold-start latency will be further amplified for chained serverless applications, where a serverless service involves a sequence of serverless functions [2, 4, 34, 36]. The resulted slow service response negatively impacts users' experiences [20, 23, 33, 35].

A straightforward way to mitigate such cold-start latency — commonly used by existing serverless platforms — is to reuse an existing VEE instance (i.e., warm-start). It keeps the same VEE instance for minutes or hours in anticipation of future execution. However, a function's high memory demands make such an approach empirically unsustainable [22, 23, 26, 30]. Facing this limitation in warm-start, reducing the cold-start latency remains a major research focus. Recent efforts reduce the start-up overheads to millisecond-scale by simplifying start-up steps. Still, the tradeoff comes as either weakening the isolation guarantees between different functions [40, 41] or modifying existing operating systems (OS) and language runtimes [23, 24], limiting its applicability.

This paper tackles this problem by first performing an empirical study of Linux containers to analyze the cold-start latency and identify critical bottlenecks causing such a long cold-start. We found that the container start-up requires a long, complex construction pipeline and heavily relies on excessive, expensive system calls and I/O operations. More essentially, there is a lack of a single-cohesive abstraction but rather a variety of fragmented isolation and security mechanisms exposed by the OS kernel, making the container

initialization process complex and costly. Our study motivates us to propose a centralized, streamlined containerization framework, FlashCube. Instead of constructing a container from scratch (i.e., via the basic isolation and security primitives), FlashCube constructs a container with a group of pre-created container parts (e.g., root file systems, network namespaces, cgroups, and program runtimes). Since the creation of various container parts can be conducted asynchronously and in advance, their creation latencies are eliminated from the critical path of container provisioning.

We have implemented FlashCube purely in the user space and integrated it into OpenWhish [8]. Our preliminary evaluation demonstrates that FlashCube significantly reduces the provisioning time of containerized serverless functions — less than 10 ms in the best case. We believe our observations, root cause analysis, and design of FlashCube will cast light on optimizing other sandboxing techniques (e.g., VMs, gVisors, and unikernels) to fit in serverless platforms well.

2 BACKGROUND & RELATED WORK

Serverless computing allows cloud tenants to simply upload their application code to serverless computing platforms [1, 13, 14, 25, 29], while the actual application deployment, management, and runtime operation are taken care of by cloud service providers. Serverless brings two significant benefits: (1) It allows developers to only focus on their application logic (thus improving developer velocity) by abstracting away the underlying server management; (2) It creates an opportunity for cloud providers to improve the efficiency of their infrastructure resource usage with *elastic* resource management of serverless applications.

Current serverless techniques unanimously adopt a simple Function-as-a-Service (FaaS) programming model, which allows developers to program their applications as a collection of functions. On the one hand, as each function has narrowly focused business logic and can be developed, tested, and deployed independently, it significantly improves developer velocity. On the other hand, as each function, executing independently in response to specific events, can be automatically scaled in/out on serverless platforms in response to workload change, it enables great service elasticity. Finally, as resources for serverless functions are fully controlled by serverless platforms, more elastic resource management can be achieved. As a result, serverless enables a cost-effective pricing model in which users are charged for the actual amount of resources consumed during the execution.

Isolation serves as the foundation in securing multi-tenancy clouds for *safe* resource sharing between cloud tenants. Serverless platforms also demand isolation for executing serverless functions from different cloud tenants. There is a large body of virtualization techniques providing isolated execution sandboxes, such as VMs [18, 27], containers [5, 38, 43], and many other lightweight virtualization designs [17, 21, 31, 42]. VMs ensure *strong* isolation but with high-performance overhead due to multiple virtualization layers (i.e., VM kernels and hypervisors); such overhead becomes more significant and dominant when the hosting entity becomes a *small* function in serverless. Containers, on the other hand, remove VM kernels and only rely on host kernel-level isolation, hence being lightweight. However, such isolation is too weak to be *directly* adopted in public clouds due to sharing the same host kernel — kernel bugs can be exploited via a large attack surface (e.g., 400+ system calls) [37, 44].

Other efforts [17, 42, 44] seek to address the tension between isolation and performance through *minimization*: AWS's Firecracker [17] tailors a VM's kernel with the minimal components and simplified I/O model for mitigated performance overhead. NEC's LightVM [42] links a hosting application into a tiny unikernel image under a single address space for improved efficiency. Google's gVisor [44] attaches a user-space guest kernel (with a substantial portion of the Linux surface) to a container for VM-like isolation. While these efforts have blurred the isolation boundaries of traditional VMs and containers, seeking to offer both security of VMs and speed of containers, they remain incurring nontrivial start-up latency — e.g., at least 100 ms to boot a minimized Firecracker VM [17]. To further reduce such start-up latency, recent approaches either added new host OS primitives to skip the sandbox initialization [23] or modified language runtimes to reuse the same sandbox for multiple invocations of the same functions (e.g., Java runtime [24]). However, the cost of these invasive approaches lies in the lack of compatibility with many existing commodity systems. In contrast, we propose to preserve the existing isolation boundaries (abstractions) set by the OS for better compatibility and adoptability.

Containers [5, 16, 44], as an alternative to VMs, offer a much lightweight, operating system (OS) level virtualization approach. Containers execute applications directly on the host OS, while isolation between containers is enforced through *kernel-level features* — e.g., namespaces [12] and cgroups [6]. Namespaces give each container its own view of system resources such as the PID space, file systems, networks, etc.; cgroups allow each container to have a fine-grained, precise allocation of resources, such as CPU, memory, disk, and network bandwidth. Processes running inside a container are directly managed by the host OS, but they are treated as a group and share the quota and limit in resource allocation. Unlike VMs, the lack of virtual hardware abstraction and direct execution on host OS allows containers to incur little performance overhead, being lightweight. In addition, containers enable unprecedented portability for applications —

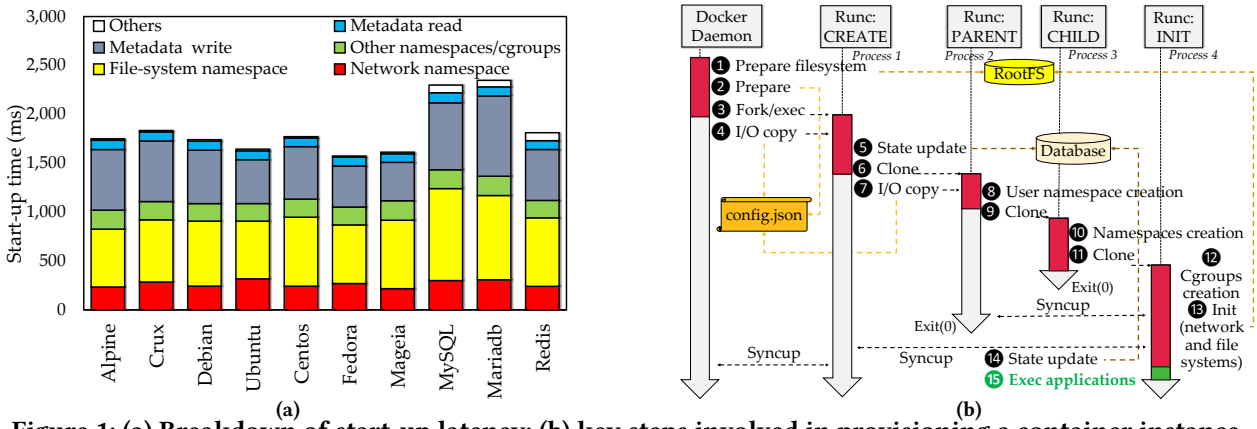

Figure 1: (a) Breakdown of start-up latency; (b) key steps involved in provisioning a container instance.

one can build an application in the form of a container image (i.e., contains minimally what is needed to run a containerized application such as its binary code, libraries, and storage data) and execute it on any host supporting containers. Due to these, containers have been served as the default sandbox in many serverless platforms [8, 25]. As discussed above, containers cannot be directly adopted in public clouds due to their weak isolation. Instead, a secure container runtime can be enabled by running containers with lightweight VMs, which uses hardware virtualization technology as a second layer of defense [16]. Compared to other sandboxing techniques (e.g., VMs or unikernels), containers closely depend on OS abstractions (e.g., namespaces and cgroups) and involve more complex initialization steps. In this paper, we choose containers as the sandbox target to study the start-up problems and explore unrevealed opportunities for fast and efficient sandbox initialization.

3 ANALYSIS OF CONTAINER START-UP

We performed a study to analyze the cold-start latency on Linux containers with a wide range of containerized functions using HelloBench [28].

Experimental setup: All experiments were conducted on the server equipped with one Xeon E5-2630 processor (2.2 GHz and 10 physical cores with hyper-threading disabled), 64 GB memory, and one 1 TB 7,200 RPM HDD. To construct Linux containers, we used a commonly-used open source containerization platform, Docker [5], which exploits various Linux features (e.g., namespaces and cgroups) to allocate storage, create networks, and isolate performance when constructing a Linux container. All experimental results were averaged over five or more runs.

Figure 1a illustrates the cold start-up time from when the Docker daemon receives a provisioning request to when the container environment is completely ready (i.e., with all required namespaces and cgroups provisioned). Note that Figure 1a focuses on container-incurred initialization overhead; thus, the start-up time of function code is not included. In addition, entire container images (e.g., function code and libraries) are cached locally to avoid network latency.

We observed that it took up to **2.4 seconds** to complete a single Linux container initialization before its encapsulated function code executes. Such cold-start latency is relatively consistent in spite of different containerized functions (ten of them are listed in Figure 1a) and container runtimes (e.g., we used both Golang-based runC [11] and C-based Crun) – mostly ranging from 1.6 seconds to 2.4 seconds. To pinpoint the key factors causing the long container cold-start, we performed a thorough code study of Docker:

(1) Serialized initialization pipeline: The construction of a Docker container goes through a *long, serialized* pipeline, as depicted in Figure 1b. It involves multiple (e.g., *four*) interactive processes for provisioning *one* containerized process. Unlike VMs that rely on a core virtualization infrastructure (e.g., with QEMU/KVM [19, 32]) for provisioning, there is *no* single cohesive infrastructure for container construction. Instead, the user-space container tool, e.g., Docker, must invoke various types of mechanisms exposed by the Linux kernel (e.g., namespaces, cgroup, and seccomp) through multiple auxiliary processes. These processes need frequent (and slow) synchronization to coordinate the different initialization stages, such as allocating storage and network resources, isolating allocated resources, and filtering system calls.

(2) Costly namespaces construction: Creating various isolation components for a container instance contributes more than *50%* to the total cold-start latency, as illustrated in Figure 1a (i.e., denoted as namespaces/cgroups). We observe that two dominant operations are the assembly of file-system and network namespaces (longer than 1 second), which is also aligned with prior investigations [40]. Our study further found that the high cost is, again, because there is no single-cohesive infrastructure during container

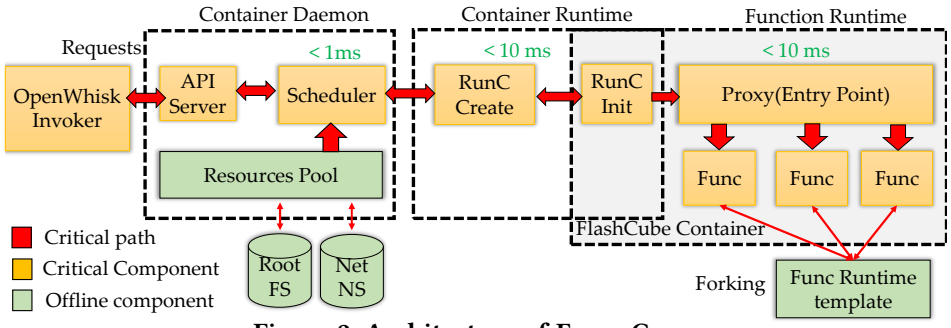

Figure 2: Architecture of FlashCube.

construction. For example, a container instance's virtual file-system starts from a *read-only* container image to which a containerized application/function is confined. Yet, unlike VMs, although the *read-only* container image only contains a portion of the root file-system, such as directories and files with data and codes needed by the execution of containerized applications, a set of *runtime* specific files and folders (e.g., under /etc, /var, and /run) need to be generated and/or prepared during initialization, which is prohibitively slow. Further, to make the virtual file system be the new root of the container instance, it needs to be *mounted* via an overlay file-system approach (e.g., AUFS or Overlay2). We found that the initialization time for an overlay file-system could be highly costly (e.g., hundreds of milliseconds).

Similarly, constructing a network namespace is also time-consuming, as the user-space tool needs to delicately create virtual network devices (e.g., a veth pair), connect them to the external network via a container bridge, configure them with the internal private IP addresses, map them to the external network addresses, and finally place the container instance to the newly constructed network namespace.

(3) Excessive I/Os and long latency: We observed that a lot of I/O activities take place throughout the container initialization, which inevitably slows down the whole start-up process. As shown in Figure 1b, such I/O operations mainly include (1) creating/copying/reading configuration files across multiple auxiliary processes, (2) updating/writing initialization status to a management database, and (3) populating runtime files/folders for the file-system and network namespaces. When we measured the results (shown in Figure 1a) using a typical hard disk drive (HDD) setup, the first two I/O-incurred overheads (e.g., denoted as metadata read/write) contribute around *30%* ∼ *40%* of the total cold-start latency. We further eliminated slow HDD and used main memory as the backing storage (i.e., ramdisk). This time, we observed much lower cold-start latency, around 400 ms, confirming excessive I/O activities during namespaces creation.

(4) Long function runtime initialization: Serverless platforms allow developers to choose programming languages in their preference — e.g., Java, Go, PowerShell, Node.js

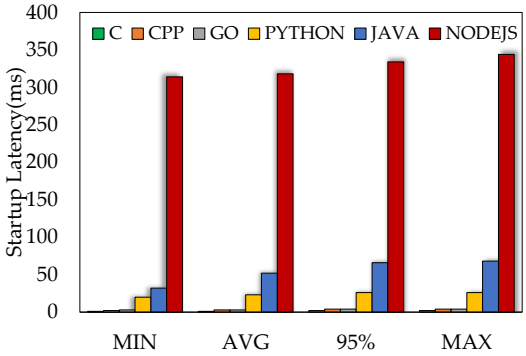

Figure 3: Runtime start-up latency.

JavaScript, C#, Python, and Ruby are all supported languages for AWS Lambda functions. In addition to the container sandbox construction, before running the target function, the function's program runtime needs to be first loaded and initialized. We have measured the initialization latency of the six common program language's runtimes as shown in Figure 3. The three widely used languages, Python, Java, and Node.js, need significant time to load and initialize their runtimes, e.g., around 70 ms for Java and more than 300 ms for Node.js.

To sum up, the existing general-purpose containerization technique (e.g., Docker or LXC) relies on a variety of isolation, security, and provisioning mechanisms exposed by the OS kernel. While the abundant OS primitives and mechanisms enable great flexibility, as individual isolation and security features can be continuously developed and updated, simply piecing them together makes the whole container initialization process complex and costly.

4 DESIGN OF FLASHCUBE

Our evaluation and analysis motivate us to propose a centralized, streamlined containerization framework, FlashCube, specialized for fast provisioning of containerized serverless functions with mitigated cold-start latency. The design of FlashCube is based on the following simple idea. Instead of constructing a container from scratch (i.e., via the basic isolation and security primitives provided by the OS), FlashCube constructs a container with a group of pre-created container parts (e.g., root file systems, network namespaces, cgroups,

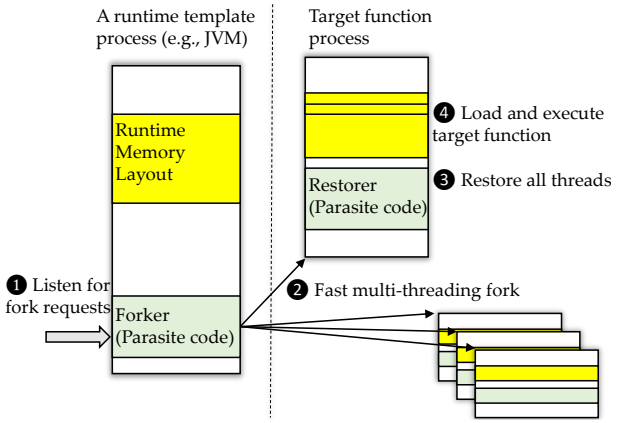

Figure 4: User-space runtime forking.

and program runtimes). Since the creation of various container parts can be conducted *asynchronously and in advance*, their creation latencies are eliminated from the critical path of container provisioning.

As illustrated in Figure 2, we have integrated FlashCube into a serverless platform, Apache OpenWhisk [8]. FlashCube consists of three key components: (1) *Container daemon* responds to each container creation request from platform invokers. Given a specific container creation request, the scheduler of the container daemon retrieves necessary pre-created container parts (e.g., namespaces, cgroups, and program runtimes) from the resource pool and provides these parts to (2) *container runtime*. Container runtime provisions an `init` process from a (3) *program runtime template* (e.g., JVM) via a novel forking mechanism (introduced shortly) and attaches all the container parts to the `init` process. After those stages, a container sandbox sets to be ready with all necessary namespaces, cgroups, and an `init` process running within a function runtime. Finally, the `init` process converts itself to the target function (e.g., by loading binary code), and the function code starts execution.

As we can see, the heavyweight namespace construction and function runtime initialization are not in the critical path of container provisioning. As a result, the container's cold-start latency reduces from ~ 400 ms (i.e., using ramdisk) to ~10 ms with C-written functions and ~20 ms for Java-written functions. Note that FlashCube is a pure user-space approach without any modifications to OS kernels or function runtimes, which preserves the portability property of containers. Note that the effort to shorten container start-up through pre-creation is not new. Mohan et al. [39] pre-creates an empty container that is used to encapsulate future user codes. However, the approach suffers from limited flexibility under its rigid assumption that all user functions need the same amount of resources and configurations. In contrast, our approach pre-creates distinct container parts separately; the parts can be used selectively for future requests.

4.1 Container Parts

By pre-creating various container parts, FlashCube provides higher-level abstractions for container runtime to assemble a container sandbox quickly. For instance, a containerized function's virtual root file system can be pre-allocated/created (offline) while *attaching* to a container instance on the fly during its start-up (via *mount* and *chroot* operations). The "attaching" process (in the critical path) is exceptionally lightweight, requiring less than 1 ms for "mounting" the overlay file system and several microseconds for "chroot".

Pre-creation of container parts does consume extra system resources. For example, to pre-create a virtual root file system, a series of directories and folders need to be provisioned, consuming both storage space and host file system resources (e.g., allocation of inodes). Nevertheless, pre-created container parts can be efficiently shared among functions — concurrent invocations of the same serverless function only need one shared pre-created (read-only) virtual root file system and one individual writable folder for each container instance. Different functions can still share the same base virtual file system (e.g., from the same base file system). Furthermore, since parallel invocations of the same function (i.e., from same cloud tenant) can trust each other, FlashCube allows multiple invocations of the same function to share the same container for reduced resource consumption.

4.2 Container Runtime

FlashCube's container runtime serves as the assembler, which attaches all the needed namespaces and cgroups (i.e., container parts) to an `init` process and configures the proper privileges [1], thus creating the final container sandbox for executing the target function code. Unlike Docker's default container runtime written in the Go programming language, FlashCube's container runtime is written in the C programming language for high efficiency. As shown in Figure 1b, the process of creating/joining namespaces and cgroups, configuring capabilities (i.e., limiting access to files and folders), and filtering system calls (via *seccomp*) could be intricate and error-prone [3]. We adopt the runtime validation tool from Open Container Initiative (OCI) [7] to ensure that FlashCube's runtime passes all runtime tests, thus to initialize a container's isolation environment with correct namespaces/cgroups, capabilities, and seccomp configurations.

4.3 User-space Runtime Forking

FlashCube mitigates the start-up latency of runtimes provisioning via a novel and efficient user-space multi-threading forking mechanism. Unlike existing approaches [23, 24], FlashCube's user-space approach does not need any new OS

[1] Containers start with restricted capabilities defining access privileges to files/folders, allowed system operations, and filtered system calls.

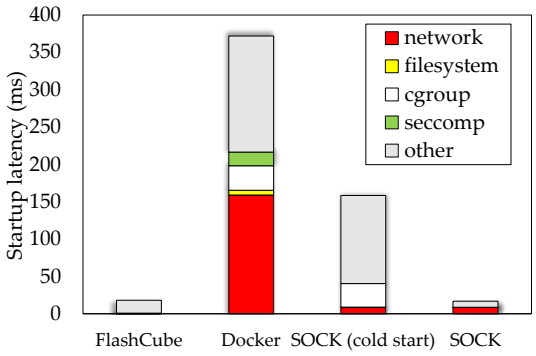

Figure 5: Container start-up latency comparisons between FlashCube, Docker container, and SOCK [40]

primitives and modifications to runtimes, preserving compatibility to existing commodity serverless platforms.

As shown in Figure 4, FlashCube prepares and maintains a template runtime process for each programming language (e.g., Python, Java, and Node.js). The template runtime process pre-loads all common libraries in advance. Then, during the provisioning of a new function, the function's runtime can be *forked* from the template runtime process — the copy-on-write based forking is super fast (e.g., less than 10 ms).

More specifically, to add control logic into an existing process in the user space, FlashCube leverages the Linux `ptrace` mechanism [10]. It pauses the normal execution of a target template runtime process and injects the parasite code (containing control logic) into it. Afterward, the parasite code starts running and serves as a lightweight web server listening for the forking requests — from FlashCube's container runtime for creating the `init` process (❶ in Figure 4). Once receiving a forking request, the parasite code uses the `clone` system call to create a child process of the template runtime process (❷). In the child process, the parasite code starts its restoring steps to leave the parasite code execution and resume the normal execution of the template runtime process (❸). To achieve this, FlashCube leverages the `sigreturn` mechanism [15], which allows the current process to jump to a specified execution point (❹). The execution point starts the logic to connect to the container runtime (in Section 4.2) for receiving the information of the target function, loads the function code, and begins execution. Note that, FlashCube's forking mechanism purely works in the user space and only relies on existing kernel exposed primitives — `ptrace` and `sigreturn`. It can be applied to any language runtimes without any modifications to them.

5 PRELIMINARY EVALUATION

To evaluate FlashCube's efficacy, We compared FlashCube with the default Docker container and the serverless-optimized container approach, SOCK [40]. FlashCube and Docker container were evaluated with OpenWhisk [8]; SOCK was tested under OpenLambda [9] — SOCK is by default integrated into the OpenLambda platform. All other configurations are the same as Section 3, except we used main memory as the backing storage (i.e., ramdisk) for low I/O latency, thus highlighting more container construction overhead.

Container start-up latency: As depicted in Figure 5, to compare the start-up latency for container construction, we ran a C-written "Hello" serverless function with three approaches. The Docker container approach takes 370 ms for the container sandbox initialization. In contrast, FlashCube takes around 18 ms to build the exact same sandbox. The improvement shows that FlashCube can yield substantial container start-up latency reduction because the time-consuming namespace/cgroup constructions are performed asynchronously and in advance with FlashCube. The pre-created container parts are welded together during the container start-up, completed in a few milliseconds. Interestingly, while SOCK takes 160 ms to provision the first container instance of a serverless function, the consecutive container provisioning takes much less time — almost the same as FlashCube. The time difference is due to the "fork" server, which must be provisioned first to serve a container creation. We further observed that SOCK overly simplifies the container start-up steps, weakening isolation guarantees. For example, SOCK does not create a separate "mount" space for each container; thus, a container user can escape its root path simply using "chroot" and "chdir". In addition, SOCK does not support `seccomp` (that could be time-consuming) to filter system calls for individual containers. In contrast, FlashCube completely enables all isolation mechanisms as Docker containers with a full validation through [7].

Function runtime start-up latency: So far, we have enabled FlashCube's multi-threading forking mechanisms upon JVM, which is a complex runtime target with multi-threading. To compare the start-up latency for JVM construction, we ran a Java-written "Hello" function. Both Docker and SOCK take ~70 ms to build up the JVM. In contrast, FlashCube takes only 10 ms to fork a multithreaded JVM from the JVM template runtime process.

6 CONCLUSIONS

We have discussed the start-up latency of serverless functions under Docker containers and identified that excessive, expensive system calls and I/O activities prolong the container start-up process. Yet, we have demonstrated that the proposed user-space containerization framework, FlashCube, can significantly reduce container start-up latency by efficiently assembling pre-created container parts. Though challenging, extensions of FlashCube to support other main sandboxing techniques, such as Google's gVisor and AWS's firecracker, are the subject of our ongoing investigations.

REFERENCES

[1] "AWS Lambda," https://aws.amazon.com/lambda/. [Online]. Available: https://aws.amazon.com/lambda/

[2] Chaining together lambdas: Exploring all the different ways to link serverless functions together. https://www.refinery.io/post/how-to-chain-serverless-functions-call-invoke-a-lambda-from-another-lambda.

[3] CVE-2019-5736 Detail. https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

[4] Deathstarbench. https://github.com/delimitrou/DeathStarBench.

[5] *Docker*, https://www.docker.com/.

[6] "Linux control groups," https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[7] oci-runtime-tool. https://github.com/opencontainers/runtime-tools.

[8] "Open source serverless cloud platform," https://openwhisk.apache.org/.

[9] OpenLambda. https://github.com/open-lambda.

[10] Playing with ptrace, Part I. https://www.linuxjournal.com/article/6100.

[11] runc. https://github.com/opencontainers/runc.

[12] "Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces," https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[13] Serverless. https://cloud.google.com/serverless/.

[14] Serverless computing. https://azure.microsoft.com/en-us/overview/serverless-computing/.

[15] sigreturn(2) — Linux manual page. https://man7.org/linux/man-pages/man2/sigreturn.2.html.

[16] "The speed of containers, the security of VMs," https://katacontainers.io/.

[17] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, 2003.

[19] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[20] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 645–650. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/boucher

[21] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," ser. CLOUDCOM '15. USA: IEEE Computer Society, 2015, p. 250–257.

[22] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3392698

[23] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481.

[24] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. Association for Computing Machinery, 2020.

[25] A. Ellis, "Introducing Functions as a Service (OpenFaaS)." https://blog.alexellis.io/introducing-functions-as-a-service/.

[26] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 386–400. [Online]. Available: https://doi.org/10.1145/3445814.3446757

[27] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, Feb. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1344209.1344217

[28] T. Harter, "Hellobench," 2015. [Online]. Available: https://research.cs.wisc.edu/adsl/Software/hello-bench/

[29] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80.

[30] Z. Jia and E. W. Qureshi, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021.

[31] A. Kantee, "The Design and Implementation of the Anykernel and Rump Kernels, 2nd Edition," 2016, http://book.rumpkernel.org.

[32] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: the linux virtual machine monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[33] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[34] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 805–820. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kotni

[35] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-efficient microservices on smartnic-accelerated servers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 363–378. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/liu-ming

[36] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 285–301. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/mahgoub

[37] 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*, 2017, pp. 218–233.

[38] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," in *Linux Journal*, 2014.

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

[40] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 57–70. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oakes

[41] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[42] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 199–211. [Online]. Available: https://doi.org/10.1145/3267809.3267845

[43] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/young

[44] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.