Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot

Wonseok Shin[*]
SK Telecom

Wook-Hee Kim
Konkuk University

Changwoo Min
Virginia tech

Abstract

Serverless computing is a new paradigm that is rapidly gaining popularity in Cloud computing. One unique property in serverless computing is that the unit of deployment and execution is a serverless function, which is much smaller than a typical server program. Serverless computing introduces a new pay-as-you-go billing model and provides a high economic benefit from highly elastic resource provisioning. However, serverless computing also brings new challenges such as (1) *long start-up times* compared to relatively short function execution times, (2) *security risks* from a highly consolidated environment, and (3) *memory efficiency* problems from unpredictable function invocations. These problems not only degrade performance but also lower the economic benefits of Cloud providers.

To address these challenges without any compromises, we propose a novel ***VM-level post-JIT snapshot*** approach and develop a new serverless framework, ***Fireworks***. Our key idea is to synergistically leverage a virtual machine (VM)-level snapshot with a language runtime-level just-in-time (JIT) compilation in tandem. Fireworks leverages JITted serverless function code to reduce both start-up time and execution time of functions and improves memory efficiency by sharing the JITted code. Also, Fireworks can provide a high level of isolation by using a VM as a sandbox to execute a serverless function. Our evaluation results show that Fireworks outperforms state-of-art serverless platforms by 20.6× and provides higher memory efficiency of up to 7.3×.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Keywords: serverless computing, start-up time, snapshot, just-in-time compilation

[*]This work was conducted while Wonseok Shin was at Virginia Tech.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

EuroSys '22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519581

ACM Reference Format:
Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519581

1 Introduction

Serverless computing is a new paradigm in Cloud computing that is becoming increasingly popular [31]. It eliminates the burden of administrating infrastructure and provides extreme elasticity for resource provisioning without requiring developer efforts. Moreover, it introduces a new pay-as-you-go billing model [8] compared to the traditional Cloud computing billing model, which is based on server uptime regardless of actual usage. Because of these advantages, most Cloud providers already offer their own serverless computing services, including Amazon Lambda [14], Microsoft Azure [42], Google [23], and IBM Cloud Functions [28].

A serverless application consists of chains of *serverless functions*, which are a unit of deployment and execution in serverless computing. Each function is executed on a sandbox (*e.g.*, virtual machine, container) provided by a Cloud provider. Most serverless applications are written in high-level languages such as Node.js and Python so that serverless platforms provide language runtimes to execute serverless functions on top of a sandbox. For instance, in AWS lambda [44], Node.js accounts for 53% and Python accounts for 36%, so these two languages account for 89% of the total serverless functions. Basically, a sandbox and runtime are launched upon a user's request (*e.g.*, function call, triggering event). The resources allocated to the application are determined by the number of functions corresponding to the application and the load of each function.

Serverless computing has unique characteristics in contrast to traditional Cloud computing. First of all, the execution time of a serverless function is much shorter than in traditional server programs. For example, in Microsoft Azure Cloud, 50% of functions take less than 1 second on average and function execution times are roughly at the same order of magnitude as the cold start times [48]. Also, due to the inherent nature of serverless functions having short execution times, Cloud providers aim to consolidate a large number (*e.g.*, thousands) of serverless functions in a single machine to utilize hardware resources more efficiently.

These unique characteristics introduce the unique challenges in serverless computing [27]. The relatively short execution time of a function implies that any overhead besides function execution will impose a large overhead. Moreover, such short execution times make efficient execution in language runtime difficult because modern language runtimes (*e.g.*, Node.js, Python, and Java) leverage the runtime profile of code to decide which functions should be just-in-time (JIT) compiled. In particular, a serverless function's *start-up time* is one of the biggest overheads [19, 48]. It includes the time needed to boot the VM, OS, and container, as well as the time to load the language runtime and the application code itself. Compared to a function's execution time, which often takes less than a second, the start-up time imposes significant overhead. The loading time of the language runtime and application itself impose significant overhead in serverless applications, as most functions are written in an interpreter language such as Node.js or Python. Besides ensuring performance for serverless users, reducing start-up time is important to Cloud providers for higher profitability because the start-up time is not charged to users.

There have been recent research efforts to shorten long start-up times in serverless computing. For example, Cloudflare Workers [33] uses *lighter sandboxes* – *e.g.*, sharing a language runtime – to reduce the start-up time. However, lighter sandboxes fundamentally provide a weaker isolation guarantee, increasing the security risk, especially in a highly consolidated serverless environment [41].

Another approach is the *warm pool-based approach*, which pre-launches a sandbox before a function is invoked [24, 58]. Pre-launched sandboxes can efficiently hide the start-up time without compromising the isolation level. However, pre-launched sandboxes take hardware resources and hinder high consolidation on the host. This approach is especially not efficient for non-popular functions. For example, a previous study [48] reports that only 18.6% of popular functions are called more than once a minute. In other words, the warm pool based approach is not effective for the other 81.4% of functions.

Besides the challenge in shortening a functions start-up time, the execution time penalty is also problematic, especially for interpreter languages, such as Node.js and Python. Although just-in-time (JIT) compilation has been very successful in improving performance, notably in long-running server applications, it is known that JIT is not very beneficial in serverless computing [16] for two main reasons. First, the JIT compilation cost cannot be offset due to the functions short execution time. Second, most serverless sandboxes are configured to use a single CPU. This configuration forces the JIT compilation to compete with the application's actual execution for CPU time. After all, JIT compilation not only introduces performance unpredictability [39] but also often incurs a significant slow down onto application execution. For example, some data center applications spend up to 33% of their time performing JIT compilation [38].

In this paper, we propose Fireworks, a new serverless platform that offers extremely short start-up time and high performance without compromising the security isolation level or wasting hardware resources. We focus on interpreter languages, especially Node.js and Python, in which the majority of serverless applications are written.

We propose a novel *VM-level post-JIT snapshot.* We found that, if properly used, JIT has the potential to significantly improve the performance of serverless applications. When installing a serverless function, Fireworks creates a snapshot of a VM *after* a language runtime loads the serverless function and finishes JIT compilation for the serverless function. Upon invoking the function, Fireworks resumes the VM snapshot and executes the function with new arguments. Thereby, the function is already loaded and JITted in the VM snapshot, so we do not need to repeatedly pay the cost of: booting a VM & OS, launching the language runtime, performing application loading, or JIT compilation during function execution. Moreover, Fireworks shares the snapshots of previously JITted functions among concurrent function invocations in a copy-on-write (CoW) manner so Fireworks can reduce memory consumption. The VM-level post-JIT snapshot meets our goals in three aspects: (1) the highest level of isolation by using a VM as a sandbox, (2) instant start-up and fast execution by resuming a JITted snapshot, and (3) less memory consumption by sharing the snapshot.

We make the following contributions in this paper:

- **VM-level post-JIT Snapshot.** We propose a novel VM-level post-JIT snapshot for serverless functions written in interpreter languages. Fireworks creates a VM-level snapshot after loading and JITting a serverless function upon installation. Then upon invoking the function, it simply resumes the snapshot with new arguments.
- **Serverless Framework.** We develop a new serverless framework named *Fireworks* based on an open-source serverless platform, OpenWhisk [13], and a lightweight hypervisor, Firecracker [5]. Fireworks analyzes a serverless function written in Node.js or Python and annotates it to trigger JIT compilation during installation. It then creates a VM-level snapshot after finishing loading and JITting the function. Finally, Fireworks resumes the snapshot when the function invocation takes new arguments from the OpenWhisk framework.
- **Evaluation.** We evaluate Fireworks using two representative serverless benchmarks: FaaSdom [40] for microbenchmarking and ServerlessBench [61] for real-world applications. Our evaluation shows that Fireworks outperforms state-of-the-art serverless computing frameworks by up to 20.6× in performance and shows higher memory efficiency up to 7.3×.

The rest of the paper is organized as follows. §2 introduces the background and motivations of this work. §3 describes

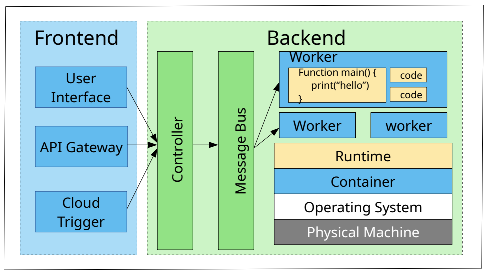


Figure 1. Overall architecture of a serverless platform.

the design of FIREWORKS in detail. §4 explains the implementation of FIREWORKS. §5 shows our evaluation results with microbenchmarking based on FaaSdom and real-world workloads based on serverlessBench. §6 discusses limitations, §7 provides related work to FIREWORKS, and §8 concludes.

2 Background and Motivation

2.1 How a Serverless Platform Works

Figure 1 illustrates the typical architecture of a serverless computing platform. The frontend consists of three components: (1) user interface, (2) API gateway, and (3) Cloud trigger. A user builds and deploys a serverless application through a web-based user interface. When a user sends a request to the serverless platform, the API gateway relays the request to one of the backend servers, which execute the requested function. When a registered event occurs – *e.g.*, inserting a record into the DBMS or changing a file in Cloud storage, the Cloud trigger invokes a function to execute. After the controller processes the requests, the corresponding messages are put in the message bus, which is the communication backbone of a serverless platform, passing a request with arguments. The messages initiate a new runtime to be created to execute the functions or deliver a request to one of the existing runtimes, often called a *worker* running inside a sandbox.

2.2 Characteristics of a Serverless Workload

Serverless applications show unique workload characteristics, which sharply differentiate them from traditional server-oriented Cloud workloads.

Short function execution time. The most prominent difference in serverless workloads is their short execution time. The execution time of a function is remarkably shorter than the lifetime of a worker. The average execution time of a function is about a second [48]. Such a short function execution time makes their start-up time – the time from when a function is invoked to when the function actually starts executing – dominant in the end-to-end latency. When a function is invoked, a serverless platform first launches a sandbox (*e.g.*, VM, container) and a language runtime (*e.g.*, Node.js, Python) before loading the function. The start-up time in this case – a so-called *cold start* – is often longer than the function execution time. The current practice by Cloud providers to mitigate this cold start is to defer termination of the worker sandbox for a certain period, hoping another request for the same function will be delivered. If that happens, the start-up time in this case – a so-called *warm start* – will be much faster than a cold start. However, for many applications, it is hard to predict when the next function invocation will be delivered [48]. That is because in these applications most functions are not frequently called. If a new request is not delivered within a certain duration, the serverless platform will terminate the sandbox as it wastes hardware resources, especially memory.

High consolidation in a server. Such a short function execution time allows a Cloud provider to consolidate a large number of serverless functions in a server with low hardware costs. However, such a highly consolidated environment is vulnerable to security risks. In particular, serverless platforms often use a sandbox with a lower isolation level (*e.g.*, container over VM) to reduce the start-up penalty [13, 25, 33, 37]. Since sandboxes with a lower isolation level share more resources between sandboxes, they are exposed to high security risks. On the other hand, sandboxes with a high isolation level (*e.g.*, a microVM in Firecracker hypervisor [5]) incur more overhead, such as longer cold start-up times.

Memory efficiency. Memory is the most scarce hardware resource in serverless computing because memory is needed to maintain sandboxes waiting for a request, and it is the main limiting factor for consolidation [59].

2.3 Optimizing Serverless Platforms

We compare state-of-the-art serverless platforms in three crucial aspects – (1) isolation, (2) performance, and (3) memory efficiency – as summarized in Table 1. Then, we introduce each approach in more detail.

Firecracker [5] is a lightweight virtual machine monitor (VMM) developed by Amazon and optimized for serverless applications. It provides a high isolation level by executing a serverless function in a VM. It also provides a VM-level snapshot so that the VM-level snapshot can be resumed and shared by multiple VM instances.

OpenWhisk [13] is a open-source container-based serverless platform deployed in IBM Cloud. While a Linux container is more lightweight than a VM, it provides a lower isolation guarantee because containers share kernel resources in the same host [10, 37].

gVisor [25] is a container-based sandbox providing a strong security guarantee. A previous study reports that containers allow the most system calls (*e.g.*, 306 out of 350 Linux system calls) [10], such that exposed system calls can be exploited for security attacks. gVisor addresses this vulnerability by intercepting a container's access to the Linux kernel and limits some system calls to improve security [30].

Serverless Platform	Isolation	Performance	Memory Efficiency
Firecracker (Amazon) [5]	High (VM)	Medium (snapshot)	High (snapshot)
OpenWhisk (IBM) [13]	Medium (container)	Low (no optimization)	Low (pre-launching)
gVisor (Google) [25]	Medium (container)	Medium (snapshot)	High (snapshot)
Cloudflare Workers [33]	Low (runtime)	High (pre-launching)	High (process sharing)
Catalyzer [19]	Med (container)	High (pre-launching)	High (process sharing)
Fireworks	High (VM)	Extreme (snapshot+JIT)	Extreme (snapshot+JIT)

Table 1. Design comparison of serverless platforms.

Specifically, Sentry and Gofer in gVisor intercept system calls and I/O requests from the container and filter them before allowing them to be processed by the host OS [30].

Cloudflare Workers [33] is a lightweight runtime-based sandbox for Node.js. Unlike other serverless computing platforms, Cloudflare Workers does not use containers or VMs for isolation. Instead, it utilizes `V8:Isolate`, a lightweight context in the V8 javascript engine. Cloudflare Workers exhibits high performance because a single process can perform hundreds of `V8:Isolate`. However, multiple functions are executed in a single V8 process, so its isolation level is weaker than others [17].

Catalyzer [19] is a gVisor-based serverless platform, specially designed to reduce start-up time. It provides a sandbox fork approach that restores a function from a checkpoint image. By restarting from a checkpointed image, it can reduce start-up time significantly. In addition, it proposes a new OS primitive, `sfork`, to further reduce initialization costs for a warm start. The `sfork` primitive uses a clean state sandbox template for the user's application. Catalyzer guarantees the same isolation level with gVisor because it is based on gVisor.

Summary. Current serverless computing platforms have limitations in performance, resource efficiency, and safety. To the best of our knowledge, Fireworks is the first serverless computing platform, which is highly performant, memory-efficient, and guarantees high VM-level isolation. Fireworks fulfills these qualities by leveraging the combination of Just-In-Time Compilation (JIT) and VM-level snapshots.

3 Fireworks Design

In this section, we present the design of Fireworks, a novel serverless computing framework providing (1) high performance both in start-up and execution, (2) a high isolation guarantee to provide a high security level, and (3) high memory efficiency for saving hardware resources. We first overview the Fireworks design (§3.1) and then describe the detailed design of each component, which perform each step from installing to launching the function (§3.2–§3.6).

3.1 Design Overview

The overall flow of Fireworks is divided into two phases, (1) the installation phase and (2) the invocation phase, as shown in Figure 2. In the installation phase, Fireworks creates a post-JIT VM snapshot of an installed serverless function.

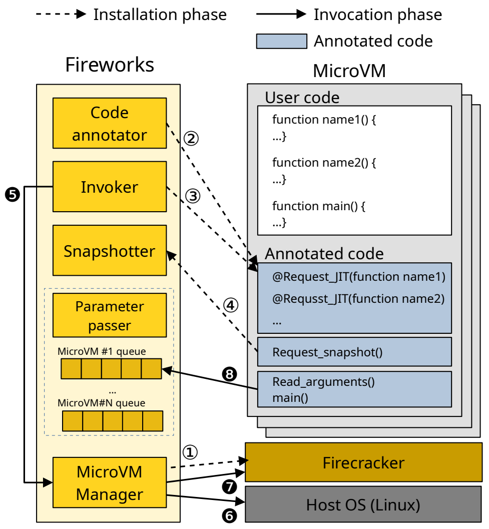


Figure 2. Fireworks overall architecture.

Then in the invocation phase, Fireworks restores the post-JIT VM snapshot of the serverless function and resumes execution with new arguments.

Fireworks components. Fireworks consists of five main components: (1) microVM manager, (2) code annotator, (3) invoker, (4) snapshotter, and (5) parameter passer, as shown in Figure 2. The microVM manager controls creating, snapshotting, resuming a microVM on the Firecracker hypervisor. The code annotator is responsible for transforming a user-provided serverless function's source code, such that the serverless function follows the Fireworks procedure of installation and invocation. The added code (marked blue in Figure 2) by the code annotator includes: enforcing JITting of the function code, requesting VM-level snapshot creation, and taking function parameters. The invoker receives the user's request from the serverless frontend and launches the function. The snapshotter creates a VM-level snapshot according to the annotated function request. The parameter passer helps the function get parameters when the snapshot resumes at the regular function entry. We will explain each phase of Fireworks's operation in detail in the rest of this section.

Installation phase. Fireworks adds annotations to a user-provided serverless function code and creates a VM-level snapshot after finishing the JIT compilation of the serverless function code. The installation procedure is as follows; Fireworks first sends a request to create a microVM that is ready for a runtime to the Firecracker hypervisor (① in Figure 2). Then Fireworks transforms the source code of the serverless function (written in either Node.js or Python) to perform JIT and create a snapshot (②). Fireworks invokes the annotated serverless function (③), and the annotated serverless function performs JITting itself and creates a snapshot right before the original serverless function entry point (④). Note that Fireworks's use of JIT is conceptually similar to Ahead-Of-Time compilation (AOT) provided by some language runtimes (*e.g.*, C#), but it additionally leverages VM-level snapshot creation for instant start-up and higher isolation guarantees.

Invocation phase. In the invocation phase, Fireworks restarts the snapshot created in the installation phase. When the serverless function is invoked, Fireworks first sets up a network for the microVM (❺❻ in Figure 2) and restarts the VM snapshot for the serverless function (❼). The annotated serverless function is resumed right after the snapshot point. It then reads the function parameters from Fireworks and executes the regular serverless function entry (❽).

How Fireworks meets the requirements. Fireworks meets our design goals as follows:

- **High isolation level**: Fireworks runs each invoked serverless function in a separate VM (*i.e.*, microVM in Firecracker), providing a high isolation level compared to approaches using a container or a runtime as a sandbox.
- **High performance**: Fireworks achieves high performance with an instant start-up and JITted function execution. Essentially, Fireworks uses a microVM snapshot that is created after a function is loaded and JITted. Hence, there is no initialization time – such as booting the VM and OS, launching a runtime, or loading the serverless function to the runtime – so Fireworks achieves an instant start-up of a serverless function. Moreover, the annotated serverless function is already JIT-compiled within the VM snapshot so that the serverless function can be directly executed during runtime. Since JIT-compiled code is faster than an interpreter and we do not need to pay the cost of JIT compilation during the runtime, the serverless function execution time is significantly reduced.
- **Memory efficiency**: Multiple instances of a serverless function can share the VM-level memory snapshot in a copy-on-write (CoW) manner. Fireworks uses private mapping (*i.e.*, `MAP_PRIVATE`) for the snapshot, so it shares the guest physical pages if there is no change; otherwise, it relies on copy-on-write. Thus, the snapshot shares the states of the microVM, OS, library, runtime, and even the JITted code. Only argument-specific execution state, which is updated during serverless function execution, will not be shared.

```
1  # A simplified serverless function printing "hello world".
2  @jit(cache=True)
3  def main(params):
4    print("hello world ",  params)
5
6  # Trigger JIT compilation of all user functions, "main".
7  def __fireworks_jit():
8    main(default_params)
9
10 # Send an HTTP request to host creating a VM snapshot.
11 def __fireworks_snapshot():
12   ploads = {'snapshot':'y', 'name':SERVERLESS_FUNCTION_NAME,
13             'srcfcID':srcfcID}
14   r = requests.get('http://172.17.0.1', params=ploads)
15
16 # This is where the program execution starts first time.
17 def function __fireworks_main(stdout):
18   # First it performs JIT compilation.
19   __fireworks_jit()
20   # Then it creates a VM snapshot.
21   __fireworks_snapshot()
22   # Upon invocation, it resumes here and first gets parameters.
23   users_params = subprocess.check_output(
24     'kafkacat -C -b 172.17.0.1:9092 -t topic'+
25     str(fcID)+' -o -1 -c 1',
26     shell=True).decode("utf-8")
27   # Then it start the entry point of a serverless function
28   # with given parameters.
29   main(user_params)
```

Figure 3. A simplified example of Fireworks source code annotation written in Python.

3.2 Automatic Source Code Annotation

The code annotator automatically adds additional instrumentation to a user's serverless function. Figure 3 shows an example of an annotated version of user code written in Python. We omit its implementation details for clarity. Added instrumentation consists of three parts: (1) performing JIT compilation (Lines 7–8), (2) creating a VM snapshot (Lines 11–14), and (3) starting the serverless main function with new parameters upon invocation (Lines 23–29).

To perform JIT compilation with annotation, Fireworks leverages Python Numba [9]. Python Numba performs JIT compilation of functions annotated with `@jit(cache=True)` (Line 2). Fireworks adds this JIT annotation for all methods in a serverless function. In Figure 3, Fireworks adds `__fireworks_jit()`, which triggers JIT compilation of application code. Similarly, the `V8` javascript engine used in the Node.js runtime offers comparable annotation opportunities, which performs JIT compilation of specified methods.

A VM snapshot should be created after finishing JIT compilation and before starting at the serverless function entry point. To precisely control the snapshot point, Fireworks adds snapshot creation code (`__fireworks_snapshot()` at Line 21) to the source code. The snapshot is directed to be created right after finishing JIT compilation (Line 19) and before resuming execution (Line 29).

The last part is the main function (`__fireworks_main()` at Line 17), where the language runtime starts program execution. After JIT compilation and snapshot creation, it gets parameters from a message queue associated with the function (Line 23). To distinguish the function instance (`fcID`, a

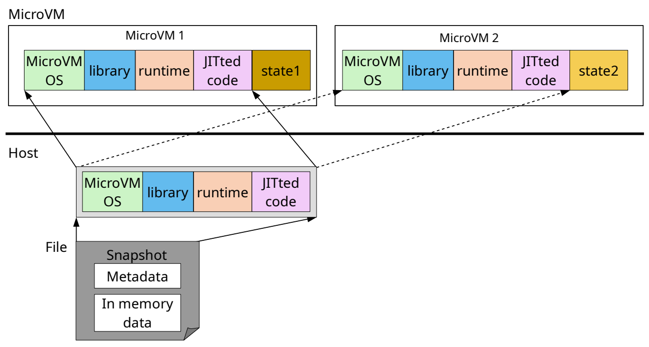


Figure 4. Sharing VM-level memory snapshot among multiple microVMs (*i.e.*, sandboxes for a serverless function) in Fireworks.

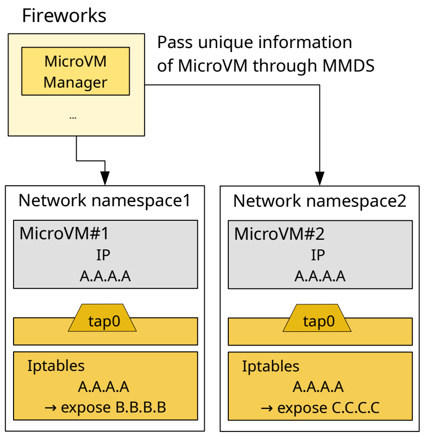


Figure 5. Network connectivity in Fireworks.

microVM ID), Fireworks uses Firecracker's microVM Metadata Service (MMDS) [20].

3.3 Creating a post-JIT VM Snapshot

`__fireworks_jit()` is called in the annotated program (Line 19 in Figure 3, ③ in Figure 2). Python performs JIT compilation for the methods annotated with `@jit(cache=True)`. Similarly, for Node.js, the runtime first interprets the source code into bytecode, and the Node.js JIT compiler engine, `turbofan`, generates machine code from the Abstract Syntax Tree (AST) for the methods with JIT annotation [2]. After JIT compilation, Fireworks creates a snapshot to store the current memory status. Fireworks creates a VM-level memory snapshot, which stores all guest physical memory containing the microVM, OS, libraries, language runtime, and even JITted machine code, as illustrated in Figure 4. Fireworks sends an HTTP request to the host to create a snapshot using the Firecracker API (`__fireworks_snapshot()` at Line 21 in Figure 3). Firecracker then creates a memory snapshot of an entire microVM and stores the snapshot in a file.

This completes the Fireworks installation phase of a serverless function (①–④ in Figure 2).

3.4 Invoking a Serverless Function

After a serverless function is installed, Fireworks can invoke the serverless function from a request by resuming the VM snapshot (❺–❽ in Figure 2). When a user sends a request to the serverless function, Fireworks puts the requested information into the parameter passer queue and requests to resume the microVM to the Firecracker hypervisor. A microVM is restarted with the corresponding snapshot image. Hence, invoking the serverless function is nothing but loading the snapshot as a file into memory. Fireworks resumes execution right after the post-JITted VM snapshot image is created. Before resuming the snapshot, Fireworks configures the network and prepares parameter passing so the resumed microVM can access the network and take the parameters of the requested function. We will describe these two parts in detail in the following sections.

3.5 Enabling Network Connectivity

While the VM-level snapshot provides excellent performance, it causes a problem in using the network inside a microVM. Suppose that multiple microVMs based on the same snapshot are launched. In this case, the microVM will have network resource conflicts – *i.e.*, each microVM resumed from the same snapshot will have the same IP address and MAC address.

To solve this network connectivity problem, Fireworks uses the network namespace [21] and Network Address Translation (NAT). Figure 5 illustrates how the network namespace works in Fireworks when two microVMs share the same snapshot image. When microVM#1 receives a network packet, the packet comes to the IP address `B.B.B.B`, the external exposed IP of microVM#1. Since the IP address `B.B.B.B` belongs to the network namespace 1, the destination IP of the packet is changed to `A.A.A.A` through the NAT table in iptables of network namespace 1. When the packet arrives at microVM#1 through tap device 0 (`tap0`), the tap device name is associated with the snapshot. Although both microVM#1 and microVM#2 have the same tap device name, `tap0`, the network namespaces of each microVM are different. Thus, there is no conflict. The reply packet goes out through the `tap0` device again and goes outside through the NAT after translating the source IP (`A.A.A.A`) to its external IP (`B.B.B.B` or `C.C.C.C`) associated with its network namespace.

Note that the information for network connectivity of microVMs that are created from the same snapshot image are identical. So Fireworks needs additional information to distinguish the microVMs. We leverage Firecracker's MMDS [20] to track the metadata (*e.g.*, microVM ID) of the microVMs.

3.6 Taking Serverless Function Arguments

Typically, a serverless platform receives a user request from the serverless frontend. Then it internally creates an appropriate sandbox and executes the code on it. When creating the environment for the request, it passes arguments in a request to the serverless function. However, in Fireworks's snapshot/restart-based approach, a serverless function cannot receive arguments from its memory because the memory states are exactly the same after resuming.

To solve this problem, Fireworks adds code to fetch parameters from outside of the microVM as soon as it loads the snapshot. Fireworks exploits a Kafka queue [32] – a software message bus – to pass arguments to a serverless function. Before resuming the microVM, Fireworks first puts the function arguments into a designated Kafka queue. When the microVM resumes, it first fetches the parameters from a queue (Line 23 in Figure 3). Since the snapshot image is identical, the serverless function needs additional information to find the corresponding Kafka queue. The identification information is inserted by Fireworks using MMDS when resuming the snapshot. When the snapshot image resumes successfully, the serverless function checks its identification number (microVM ID) and reads arguments from the appropriate queue.

4 Implementation

We implement Fireworks based on Firecracker v0.24.0. Fireworks leverages Firecracker's microVM features for guaranteeing a high level of isolation with the snapshotting function. We develop an end-to-end serverless infrastructure including an invoker, code annotator, snapshotter, and a parameter passer as described in §3. The parameter passer is implemented based on Kafka [32], a software message bus. The source code of Fireworks collectively consists of 3,500 lines of Bash, Node.js, Python, and C++ code.

5 Evaluation

We evaluate Fireworks by answering the following questions:

- How much can Fireworks reduce the end-to-end latency of various serverless functions, including compute-intensive and I/O-intensive ones, by reducing the start-up time and making function execution time faster? (§5.2)
- How effective is Fireworks's approach in improving the performance of real-world serverless applications? (§5.3)
- How much memory can Fireworks save by sharing memory snapshots across microVMs? (§5.4)
- How effective are Fireworks's design choices (VM-level snapshot, post-JIT snapshot) in improving performance and saving memory usage? (§5.5)

5.1 Evaluation Methodology

Evaluation environment. We perform our evaluations on a Intel Xeon Platinum 8180 CPU (64 physical cores) server with 128 GB memory and 2 TB SSD. For all evaluations, we configured a microVM similar to typical configurations in serverless computing: one vCPU, 512 MB memory, and 2 GB of disk space [48].[1] Besides these specifications, Fireworks uses the default settings of Firecracker [5], such as disk emulation.

[1]Average memory size in a serverless sandbox is 170MB [48].

We compare Fireworks against the state-of-the-art serverless frameworks and sandboxes designed for serverless computing: OpenWhisk [12], gVisor [25], and Firecracker [5]. We use OpenWhisk v20.11 with Kubernetes and gVisor v1.0.2 with Docker. Regarding language runtimes, we use the latest versions as of this publication: Node.js v12.18.3 and Python v3.8.5. We do not include Catalyzer [19] because its source code is not publicly available.

Note that we follow the standard practice in serverless computing to evaluate our new serverless framework, Fireworks. We follow the evaluation settings of previous studies [19, 54, 61] on a single machine and measure the start-up time, execution time, and memory footprint of the serverless functions.

Workloads. We used two serverless benchmarks: FaaSdom [40] and ServerlessBench [61] as shown in Table 2.

For micro-benchmarking, we chose the FaaSdom benchmark, which consists of two compute-intensive benchmarks (`faas-fact` and `faas-matrix-mult`) and two I/O-intensive benchmarks (`faas-netlatency` and `faas-diskio`) written in Node.js and Python. `faas-fact` and `faas-matrix-mult` perform integer factorization and multiplication of large matrices multiple times. The `faas-netlatency` measures only network latency by immediately sending a small-sized HTTP response as soon as the benchmark function is invoked. The `faas-diskio` is a benchmark to measure disk I/O performance, including I/O operations to disk.

In addition, we chose two real-world serverless applications from ServerlessBench [61]: Alexa Skills and data analysis, both of which are written in Node.js. Alexa Skills is a collection of applications performed through the Amazon Alexa AI speaker device. The Amazon Alexa AI speaker receives a user's voice commands and performs the corresponding Alexa Skills via serverless functions in the Cloud based on its analysis. It provides the functionality of a scheduler and smart home management. Another application is the data analysis application. It calculates bonuses and taxes with employees' roles and makes statistics. The wages of multiple employees are continuously entered when a user wants. Note that these two real-world applications consist of a chain of serverless functions, as illustrated in Figure 8. For the real-world applications, we only compare against OpenWhisk [13]. This is because sandbox managers, such as Firecracker [5] and gVisor [25], cannot process a chain of serverless functions, and only OpenWhisk and Fireworks are able to process a chain of serverless functions.

Evaluation configuration. We measured and reported cold start performance (denoted with `c`) as well as warm start performance (denoted with `w`) in Figure 6 and Figure 7. We follow the same configuration for warm start evaluation as done in previous studies [19, 61]. Specifically, for gVisor

Application Name	Description	Language
FaaSdom: faas-fact	Integer factorization	Node.js, Python
FaaSdom: faas-matrix-mult	Multiplicaiton of large matrices	Node.js, Python
FaaSdom: faas-diskio	Disk I/O performance measurement	Node.js, Python
FaaSdom: faas-netlatency	Network latency test that immediately responds upon invocation	Node.js, Python
ServerlessBench: Alexa skills	Apps run through Alexa AI device	Node.js
ServerlessBench: data analysis	Store and analyze the statistics employees' wage	Node.js

Table 2. Tested serverless applications.

and Firecracker, we first launched a sandbox, installed an application on it, and then paused the sandbox to keep the sandbox in memory. For OpenWhisk, we first registered and invoked a function. When the function is invoked, a container is lauched and the function is executed in the container. OpenWhisk keeps the container alive in memory for a while. We do not distinguish cold or warm starts in Fireworks because Fireworks always resumes from a VM snapshot.

Post-JIT snapshot creation time. We measure the post-JIT snapshot creation time in Fireworks's installation phase. The whole process installs the necessary packages, runs the application for JITing, and creates a snapshot. For the FaaSdom benchmark written in Python, the post-JIT snapshot creation time depends on the complexity of the application due to JIT compilation of the application code. For the FaaSdom benchmark written in Node.js, the npm package installation process dominates installation time. However, making a snapshot takes 0.36-0.47 seconds for the FaaSdom benchmark in Nodejs, and takes 0.38-0.44 seconds in Python. More complex ServerlessBench applications depend on how many functions they contain.

5.2 FaaSdom Microbenchmark

We first discuss the evaluation results of the Node.js version of the FaaSdom benchmark (§5.2.1) and then discuss the Python version (§5.2.2) because we found that these two language runtimes have different performance characteristics.

5.2.1 Node.js Benchmarks. Figure 6 shows the performance comparison and latency breakdown of the Node.js version of the FaaSdom benchmark. We measured the latency from invocation of the serverless function to the serverless function's termination. We breakdown latency into start-up time (*start-up*), function execution time (*exec*), and all other times besides these two (*others*), which include network latency, disk I/O, etc, depending on benchmark behavior. For OpenWhisk, gVisor, and Firecracker, we measured the latency for both cold start (`c`) and warm start (`w`). However, Fireworks, which relies on snapshot restarting, does not have such a distinction (`both`).

(1) Compute-intensive benchmarks. Figure 6(a) is the evaluation result for the CPU-intensive integer factorization benchmark (`faas-fact`). Fireworks shows an extremely fast start-up time, which is comparable to or even faster than warm start-up in other serverless platforms. Specifically, it shows up to 133× faster cold start-up and up to 3.8× faster warm start-up due to its snapshot-based approach.

Firecracker shows the slowest cold start-up time because it is a VM-based approach, which requires booting of the VM, guest OS, and runtime before loading the serverless function.

The container-based OpenWhisk shows a relatively shorter cold start-up time than Firecracker, but it has a pretty high overhead to initialize a container (*e.g.*, authentication and message queue initialization) in the case of a cold start.

gVisor shows slower cold start-up time and execution time as it enforces additional security checks on the container to improve security of the runtime environment.

The gap in execution time between Fireworks and others is not as significant as the start-up time. That is because Node.js runtime quickly performs JIT compilation for hot methods. However, Fireworks still shows up to 38% faster performance in cold start cases and 25% faster performance in warm start cases.

The evaluation results of (`faas-matrix-mult`), another compute-intensive benchmark, show similar performance trends as with the integer factorization benchmark, discussed earlier.

(2) Disk-intensive benchmark. The disk-intensive benchmark, `faas-diskio`, performs 10KB-sized file read and write operations 100 times. Overall, Fireworks shows significantly shorter latency than other serverless platforms. Specifically, as shown in Figure 6(c), Fireworks shows up to 68× faster cold start-up and up to 3.8× faster warm start-up times. Moreover, it shows up to 9.2× faster execution time than other serverless frameworks. Also, Fireworks shows 25% performance improvement against Firecracker using a snapshot. Interestingly, we found that the execution time in I/O-intensive workloads is mostly determined by the I/O efficiency of the sandbox mechanism used, while the performance impact of JIT compilation is marginal.

gVisor shows the slowest I/O performance because system call monitoring uses two additional components: Sentry and Gofer. Sentry monitors system calls using a seccomp filter [29] and sends file I/O requests to Gofer I/O to check all access requests for host resources [53]. While the cost of system call monitoring using Sentry and Gofer is much higher than the I/O overhead in Fireworks, the security level of

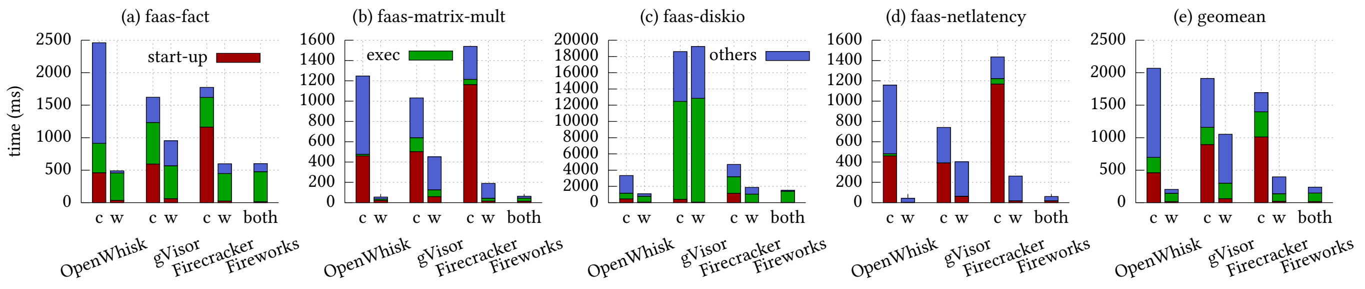


Figure 6. Latency comparison of the Node.js version of FaaSdom benchmark. "c" is short for "cold start" and "w" is short for "warm start".

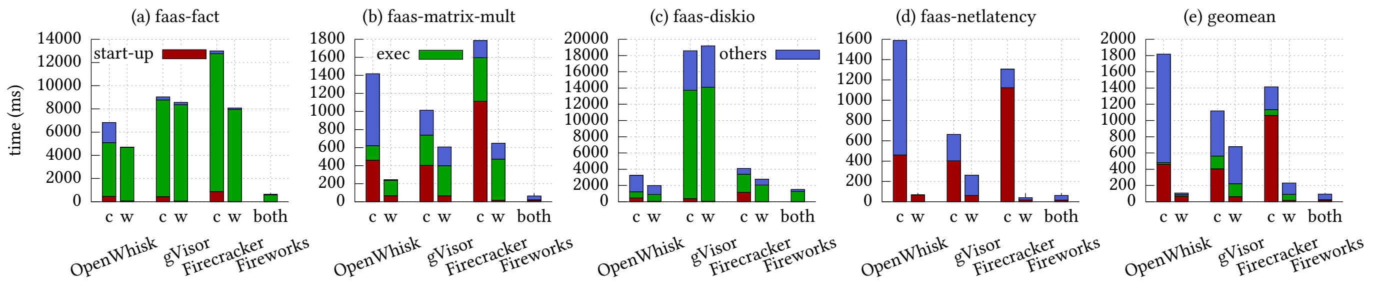


Figure 7. Latency comparison of the Python version of FaaSdom benchmark. "c" is short for "cold start" and "w" is short for "warm start".

gVisor is still lower than VM-based FIREWORKS because the container-based approach shares resources on the same host.

Firecracker shows higher performance than gVisor because it eliminates heavy QEMU for I/O emulation. Also, Firecracker removes unnecessary devices and uses a lightweight 9p filesystem in crosvm [5].

Container-based OpenWhisk uses OverlayFS [18] and chroot [1] while it interacts directly with the host filesystem. Therefore, I/O speeds are faster than for a microVM, which has its own VM file systems.

(3) Network-intensive benchmark. The network benchmark, `faas-netlatency`, sends a small-sized HTTP response (79-byte body, 500-byte header) without any operation and reports the latency. FIREWORKS shows up to a 25× faster cold start-up time and up to a 3.6× faster warm start-up time. FIREWORKS is up to 22× faster in cold start cases, up to 6.5× faster in warm start cases, and up to 4.2× faster than Firecracker using a snapshot. FIREWORKS has the additional overhead of NAT and tap device, but this additional network overhead is negligible in the end-to-end latency. FIREWORKS shows high performance mainly because the benchmark code starts immediately after loading the snapshot.

(4) Summary. Figure 6(e) shows the geometric mean of four FaaSdom benchmarks. Overall, FIREWORKS shows up to a 8.6× shorter latency compared to other serverless frameworks.

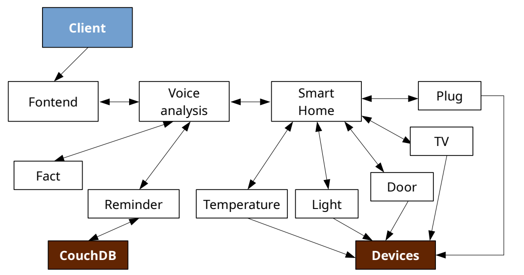
5.2.2 Python Benchmarks.

Figure 7 shows the evaluation results of the Python version of FaaSdom benchmarks. The evaluation results reveal that Python is, in general, slower than Node.js.

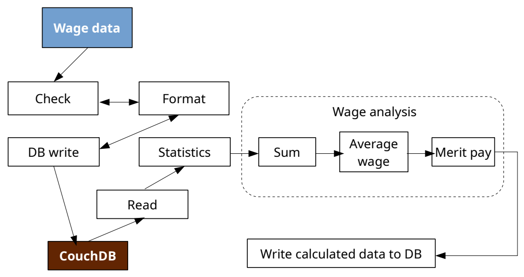
(1) Compute-intensive benchmarks. FIREWORKS shows significantly higher performance in compute-intensive benchmarks compared to other serverless frameworks. Interestingly, FIREWORKS dramatically reduces the execution time for Python by leveraging post-JIT machine code in the snapshot. For integer factorization shown in Figure 7(a), FIREWORKS shows a 59.8× faster cold start-up time, a 4.4× faster warm start-up time, and is 12.3× faster than Firecracker by using a snapshot. Also, it achieves 20× faster execution time in cold start cases and 14.6× faster execution time in warm start cases. For matrix multiplication shown in Figure 7(b), FIREWORKS shows similar performance trends as in integer factorization. FIREWORKS shows up to a 74.2× faster cold start-up time and a 4.4× faster warm start-up time. Also, it achieves up to 80× faster execution time for cold start cases and up to 75× faster execution time for warm start cases.

(2) I/O-intensive benchmarks. Both disk-intensive and network-intensive benchmarks in Figure 7(c) and (d) show similar performance trends to their Node.js counterparts in Figure 6(c) and (d) because I/O latency is the dominant factor in performance.

(3) Summary. Figure 7(e) shows the geometric mean of the four benchmarks. Overall performance improvement of FIREWORKS is up to 19×, which is 2.2× higher than Node.js. In particular, we observe that the overall execution time considerably decreases, showing that the effect of post-JIT is significant in Python. Another interesting point is that I/O performance is similar between Python and Node.js. This means that I/O performance mostly depends on the I/O efficiency of the sandbox mechanisms rather than the language or runtime.



(a) Alexa Skills (nested)



(b) Data Analysis (sequence + nested)

Figure 8. Real-World serverless applications.

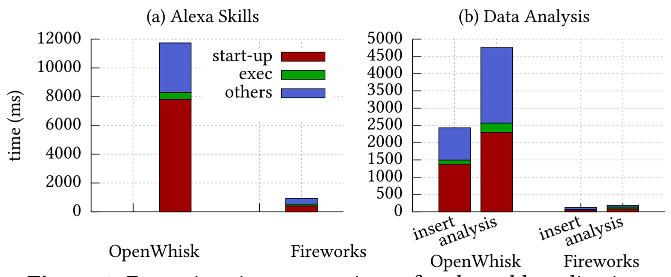


Figure 9. Execution time comparison of real-world applications.

5.3 Real-World Serverless Applications

We chose two real-world applications in ServerelessBench [61] because these two are the only applications written in Node.js (or Python). These two applications are composed of a chain of serverless functions, which interact with each other to deliver the processed data in the form of pipes. Thus, we could not evaluate gVisor or Firecracker for these real-world applications because they cannot execute a chain of functions. Hence, we evaluate FIREWORKS against OpenWhisk. Figure 8 shows the flow of real-world applications, where a node represents a serverless function running on a microVM, and an edge represents a function invocation.

(1) Alexa Skills. Alexa skills [7] supports three skills: fact, reminder, and smart home. It performs voice analysis using input text provided by the user and identifies the request. The fact skill answers simple common sense, and the reminder skill searches or enters a schedule into CouchDB [11], a NoSQL database. The data inserted to CouchDB includes item, place, and related URL fields. The smart home skill notifies the on/off status of each device (*e.g.*, light, door, and TV) to the user. Note that the input parameters of Alexa are provided by a JSON file, and the scenario of Alexa is quite complicated because it simulates the real Alexa's workload such as door password, schedule details, etc. Also, the types of input parameters for a function could be different from what was JITted. Hence our experiments include the worst-case in JIT compilation, de-optimizing JITted code [2].

Figure 9(a) shows the performance comparison between FIREWORKS and OpenWhisk as well as its latency breakdown. For the evaluation, we sent requests to Alexa Skills asking for a simple fact, then checking the schedule through reminder and checking the on/of the status of home appliances through smart home. FIREWORKS shows 12.5× faster start-up time and 2.4× faster execution time compared to OpenWhisk.

(2) Data Analysis. The data analysis application receives personal wage data from the user and analyzes the wage. When a user inserts personal wage data, it checks the format to ensure that it is valid data and then changes the format for insertion to CouchDB. The data entered in CouchDB consists of a name, ID, role, and base payment. The analysis function chain in the dashed box of Figure 8(b) is triggered when a database is updated. The analysis results are inserted into the CouchDB. The evaluation result for the data analysis application shows similar performance trends to Alexa Skills. For the data insertion step, FIREWORKS shows 25.6× shorter start-up time and 11.8× faster execution time. The data analysis step shows 27× faster start-up time and 4.9× faster execution time.

5.4 Memory Usage

FIREWORKS shares memory snapshots across multiple microVMs as discussed in §3.3. In order to understand the memory saving effect in FIREWORKS, we compare the amount of memory usage of FIREWORKS and Firecracker, both of which provide VM-level isolation. We used the FaaSdom integer factorization benchmark (`faas-fact`), which is a computation-intensive workload used in §5.2.1. We measured the Proportional Set Size (PSS) in Linux using `smem` [36]. PSS is a physical memory consumption considering memory sharing; for the memory shared by `N` processes, PSS accounts `1/N` to each process. Figure 10 shows the memory usage of Firecracker and FIREWORKS as the number of microVMs increases. We set the kernel parameter – `vm.swappiness` – to 60, meaning that swapping starts to happen when 60% of physical memory is consumed. We ran the test until swapping happened to see the maximum possible amount of consolidation. FIREWORKS could launch 565 microVMs, but Firecracker could only launch 337 microVMs before triggering swapping. This means that FIREWORKS allows for consolidating 167% more sandboxes than Firecracker as the amount of memory shared by JIT and the OS snapshots increases.

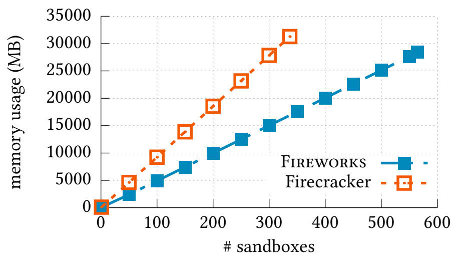


Figure 10. Memory usage comparison between Fireworks and Firecracker.

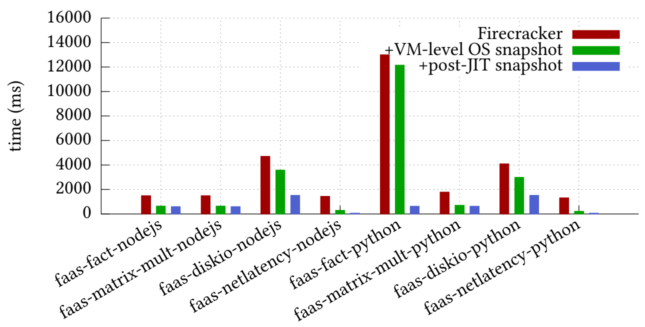


Figure 11. Performance impact of Fireworks optimizations.

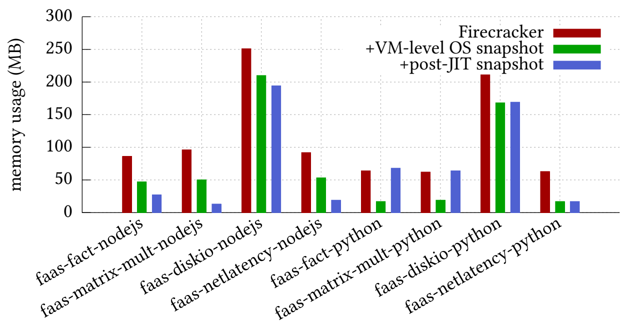


Figure 12. Memory impact of Fireworks optimizations.

5.5 Impact of VM-level Snapshot and post-JIT

We analyze how our design choices affect performance (§5.5.1) and memory consumption (§5.5.2). We separately investigate the impact of using a VM-level OS snapshot (creating a VM snapshot after a guest OS finishes booting) and post-JIT snapshot (creating a VM snapshot after loading a serverless function and finishing JIT compilation of the function). We start from the original version of Firecracker as a baseline, which does not use a snapshot.

5.5.1 Performance. We conduct the factor analysis to measure the performance gains from a VM-level OS snapshot and our post-JIT snapshot, as shown in Figure 11.

+ VM-level OS snapshot. Adding a VM-level OS snapshot to Firecracker improves the performance of serverless functions by eliminating the OS boot time. For compute-intensive workloads written in Node.js, adding this snapshot improves performance by 2.3×. For network-intensive workload written in both Node.js and Python, this snapshot shows up to 6.1× faster performance.

+ Post-JIT snapshot. Adding a post-JIT snapshot improves the performance significantly, as shown in Figure 11. Since the interpreters can trigger JIT compilation during program execution even when the VM-level OS snapshot is used, the actual performance improvement of the post-JIT snapshot depends on when the JIT compilation is triggered in the execution of the VM-level OS snapshot. If JIT compilation is triggered at an early stage of function execution, the performance difference between the VM-level OS snapshot and the post-JIT snapshot is relatively small. We checked when JIT compilation is triggered in the VM-level OS snapshot by using `GetOptimizationStatus()` in V8 and by manually instrumenting the Python interpreter. We found that JIT compilation was triggered near the end of function execution in `faas-diskio-nodejs` and `faas-netlatency-nodejs`. Hence the Node.js code was executed in an interpreter mode (no JIT) for most of the benchmark time, so the performance improvement of the post-JIT snapshot over the VM-level OS snapshot is significant. Also, the performance benefit of the post-JIT snapshot is substantial in Python benchmarks because the Python interpreter in our experiments did not perform JIT compilation.

5.5.2 Memory Usage. We also conduct factor analysis to analyze the memory impact of a VM-level OS snapshot and our post-JIT snapshot. To see the memory impact of sharing memory snapshots (shown in Figure 4), we ran 10 microVMs concurrently, running the same benchmark, and reported the memory usage of one microVM as in §5.4.

+ VM-level OS snapshot. All applications written in both Node.js and Python show memory efficiency improvements. OS snapshot shows up to a 73% improvement in memory utilization by sharing resources.

+ post-JIT snapshot. In the case of applications written in Node.js, adding post-JITting reduces memory usage up to 74%. However, there is no significant improvement in memory usage for Python applications. The memory usage of the post-JIT snapshot mainly depends on the memory efficiency of the JITted code. Specifically, Node.js V8 has been optmized to lazily allocate memory for storing execution state [55]. On the other hand, Python Numba duplicates a JITted Python function to many modules due to the restriction of LLVM MCJIT so increasing the memory usage [35]. Even though Fireworks consumes a similar amount of memory, Fireworks gains dramatic performance improvements.

6 Discussion and Limitation

De-optimization of JITted code. JITted code has a unique challenge called de-optimization, which reverts a previous JIT compilation. When code is JITted, the compiler applies specialized optimizations for the runtime to the code. Thus, the code needs to perform an undo operation when the runtime profile is changed (*e.g.*, type in dynamic typed languages) or when further optimization of the JITted code is necessary so that the performance may decrease temporarily. However, the undo operation executes bytecode, which is already generated, so it does not incur high overhead. Furthermore, each language runtime tries to minimize performance degradation from de-optimization [2, 4]. Note that our evaluations use various arguments, which can trigger de-optimization. For instance, the Alexa workload uses various arguments such as the door password and schedule information. Nonetheless, our evaluation results always show a performance improvement.

Disk space overhead for function snapshots. In serverless computing, thousands of serverless functions are running in the same host machine simultaneously. When serverless functions trigger to create their snapshot, disk space overhead could be high. To resolve this disk space overhead, previous works using a snapshot-based approach [54] leverage remote storage. We could also limit disk space and use a replacement policy, which keeps frequently accessed functions' snapshots.

Security implications in snapshot-based approach. Similar to other snapshot-based approaches [54], launching multiple microVMs based on the same memory snapshot in Fireworks has security implications. The multiple instances of microVMs will have the same random number generator and the same memory layout, both of which reduce the system's entropy. For the random number generator, we can leverage Linux random number generation (RNG) facilities, providing unique entropy from the kernel to userspace. RNG provides enough entropy when the CPU used is IvyBridge or newer Intel processors. For address space layout randomization (ASLR), we can mitigate this issue by periodically re-generating the VM snapshot similar to REAP [54].

7 Related Work

In this section, we discuss previous research efforts regarding serverless computing.

Characterizing FaaS workloads and benchmarks. Serverless computing is a new paradigm, so there is not yet enough information to understand the characteristics of serverless workloads. Thus, there are several previous studies that conducted reverse engineering to understand commercial serverless computing platforms. Wang *et al.* [59] analyze the characteristics of serverless computing platforms of well-known Cloud providers, including AWS Lambda, Google Cloud, and MS Azure, by conducting many experimental measurements for serverless functions. Shahrad *et al.* [48] show characteristics of serverless workloads on MS Azure serverless computing platforms. They also propose a resource management policy to leverage serverless computing platforms. FaaSProfiler [47] is a profiling platform for serverless services. FaaSProfiles shows the architectural impact of serverless computing platforms.

Optimizing serverless platform. There are two main categories of research to optimize serverless computing platforms. One direction is to reuse resources, including network, sandboxes, and runtime environments. Reuse mechanisms can cache these resources or use checkpoint/restore-based snapshots. Replayable Execution [57] proposed to use checkpoint and restore on the OS kernel to reduce start-up time and needed resources for serverless functions. The key differences between Replayable Execution and Fireworks are as follows: (1) Fireworks relies on JIT annotation of dynamic languages; (2) Fireworks leverages a VM-level snapshot, so it ensures a higher level of security; (3) our Fireworks approach is not limited to specific languages.

HotTub [38] eliminates JVM overheads in a parallel processing environment by keeping the warm-up pool of JVM runtime. It is applicable to serverless environments. Mohan *et al.* [43] reduce cold start-up times by pre-creating the network and attaching the network to new serverless function containers. SOCK [45] is a streamlined container system optimized for serverless computing. SOCK extends Zygote ideas (*i.e.*, pre-launching sandboxes) to serverless computing platforms. It generalizes the provisioning of Zygote and proposes package-aware caching to minimize start-up time. SAND [6] uses application-level sandboxing. Checkpoint/restore for instantiation is also one of the main directions to optimize start-up time. REAP [54] mitigates performance bottlenecks from page faults when it loads the serverless function's snapshot from disk to memory. It proposes proactively configuring the memory's working set before loading a snapshot to reduce page faults. While both REAP and Fireworks use snapshots, their approaches are fundamentally different. REAP focuses on reducing page fault overhead by prefetching snapshot images. Meanwhile, Fireworks reduces start-up time by post-JITting function code. Hence, Fireworks can also employ REAP's prefetching to further reduce the overhead for reading snapshots from disk.

Ignite [16] is a new runtime platform to reduce the start-up time by leveraging JIT and sharing code and runtime profiles. Ignite shares the motivation to leverage JIT to reduce the start-time with Fireworks. Still, Fireworks also considers the security problems present in consolidated serverless functions, which is also a critical problem in serverless computing frameworks, by using a VM-level snapshot.

AWS supports JIT only for pre-provisioned instances written in C#/.NET [3]. However, the JIT of .NET does not allow sharing of code or resources.

Another direction to optimize serverless platforms is to optimize the sandbox. SEUSS [15] leverages unikernel snapshot stacks for the execution of serverless functions. It deploys page-level sharing across the snapshot. LightVM [41] proposes a lightweight VM through use of a unikernel. Faasm [50] is a serverless runtime that leverages WebAssembly [26] for efficient isolation of serverless functions.

Another approach is to place sandboxes close to where data is stored. Shredder [62] implements a sandbox near the data storage with `V8:Isolate`.

The main difference between Fireworks and previous works is that we post-JIT the source code. The JITted code can efficiently reduce not only the cold start-up time but also the execution time for serverless functions.

Extending serverless computing. Since most serverless platforms support stateless serverless functions only, it is hard to extend applications that are disk I/O intensive into serverless applications. So, a few previous works [46, 52] extend serverless applications to support stateful serverless functions instead. Cloudburst [52] presents a stateful FaaS that can guarantee mutable state and communication using Anna Key-Value store [60]. Anna provides auto-scaling and logical disaggregation of compute nodes and storage nodes to support stateful function efficiently. Starling [46] is a query engine for leveraging serverless computing platforms. Starling shows the feasibility of query engines for serverless computing platforms with several optimizations to mitigate the high latency of serverless storage access.

In addition, there are some research efforts to extend scientific applications to serverless applications. Numpywren [49] is an application that runs linear algebra operations as serverless functions on top of disaggregated storage. Excamera [22] is an application that leverages cloud functions for video processing. Excamera shows efficient video processing using the flexible use of thousands of functions. Pocket [34] designs a serverless platform to satisfy scalable demands. It automatically scales to support the applications at hand and determines the storage tier needed to provide high-performance with low costs. Infinicache [56] is an in-memory object caching system built on top of stateless serverless functions. It shows the feasibility of the use of serverless computing for in-memory object caching systems. AFT [51] introduces an atomic fault tolerance shim between serverless functions and storage engines. It leverages transaction concepts for serverless functions to provide fault tolerance. Shredder[62] suggests the storage function that allows computation to be performed in a storage node with low latency.

8 Conclusion

In this paper, we present Fireworks, a fast, efficient, and safe serverless computing platform. To the best of our knowledge, Fireworks is the first serverless computing platform that fulfills the three aspects for an ideal serverless platform: (1) high isolation level, (2) high performance, and (3) high memory efficiency. We show how our VM-level post-JIT snapshot efficiently reduces memory usage without compromising the isolation level or increasing security risks. We evaluate Fireworks against state-of-art serverless platforms, including gVisor, OpenWhisk, and Firecracker, using two representative benchmark suites, FaaSdom and ServerlessBench. In our evaluation, Fireworks significantly outperforms these state-of-art serverless platforms, and it shows high memory efficiency with the guarantee of a high level of isolation.

Acknowledgement

We thank our shepherd Abhilash Jindal and the anonymous reviewers for their helpful comments to improve the paper. This work was supported in part by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No. 2014-3-00035) and National Research Foundation of Korea (NRF) (No. NRF-2021R1A6A3A03046359).

References

[1] chroot(2) — Linux manual page. https://man7.org/linux/man-pages/man2/chroot.2.html, visited 2021-05-18.
[2] An introduction to speculative optimization in v8, 2017. https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8, visited 2021-09-26.
[3] Managing lambda provisioned concurrency, 2021. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html, visited 2021-09-26.
[4] Turbofan v8, 2022. https://v8.dev/docs/turbofan, visited 2022-03-03.
[5] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 419–434, 2020.
[6] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 923–935, 2018.
[7] Amazon. Build An Alexa Fact Skill, 2020. https://github.com/alexa/skill-sample-nodejs-fact, visited 2021-05-18.
[8] Amazon. AWS Lambda Pricing, 2021. https://aws.amazon.com/lambda/pricing, visited 2021-05-18.
[9] Anaconda. Numba: A High Performance Python Compiler, 2018. http://numba.pydata.org/, visited 2021-05-18.
[10] FNU Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: a study of firecracker and gvisor. *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.
[11] Apache. CouchDB, 2021. https://couchdb.apache.org/, visited 2021-05-18.
[12] Apache OpenWhisk. Apache OpenWhisk Documentation, 2016. https://openwhisk.apache.org/documentation.html#deploy_kubernetes, visited 2021-05-18.
[13] Apache OpenWhisk. Open Source Serverless Cloud Platform, 2016. https://openwhisk.apache.org/, visited 2021-05-18.
[14] AWS. AWS Lambda, 2021. https://aws.amazon.com/lambda/, visited 2021-05-18.
[15] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys, 2020.

[16] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: Leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 58–64, Ann Arbor, Michigan, June 2021.

[17] Cloudflare. Cloudflare Docs - How Workers works, 2021. https://developers.cloudflare.com/workers/learning/how-workers-works, visited 2021-05-18.

[18] docker. docker docs - Use the OverlayFS storage driver, 2021. https://docs.docker.com/storage/storagedriver/overlayfs-driver, visited 2021-05-18.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[20] Firecracker. microVM Metadata Service, 2020. https://github.com/firecracker-microvm/firecracker/blob/master/docs/mmds/mmds-design.md, visited 2021-05-18.

[21] Firecracker. Network Connectivity for Clones, 2020. https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/network-for-clones.md, visited 2021-05-18.

[22] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 363–376, 2017.

[23] Google. Cloud Functions, 2021. https://cloud.google.com/functions, visited 2021-05-18.

[24] Google Cloud. Configuring Warmup Requests to Improve Performance, 2020. https://cloud.google.com/appengine/docs/standard/python/configuring-warmup-requests, visited 2021-05-18.

[25] gVisor Authors. gVisor User Guide, 2021. https://gvisor.dev/docs/user_guide/install/, visited 2021-05-18.

[26] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. June 2017.

[27] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[28] IBM Cloud. IBM Cloud Functions, 2021. https://cloud.ibm.com/functions/, visited 2021-05-18.

[29] Jake Edge. A seccomp overview, 2015. https://lwn.net/Articles/656307/, visited 2021-05-18.

[30] Jeremiah Spradlin and Zach Koopmans. gVisor Security Basics - Part 1, 2019. https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1, visited 2021-05-18.

[31] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[32] Apache Kafra. Apache kafka quickstart, 2017. https://kafka.apache.org/quickstart, visited 2021-05-18.

[33] Kenton Varda. Fine-grained sandboxing with v8 isolates, 2019. https://www.infoq.com/presentations/cloudflare-v8/, visited 2021-05-18.

[34] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 427–444, 2018.

[35] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 7:1–7:6. ACM, 2015.

[36] Linux. smem - Linux man page, 2009. https://linux.die.net/man/8/smem, visited 2021-05-18.

[37] Linux. Introduction to Linux Containers, 2021. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers, visited 2021-05-18.

[38] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 383–400, 2016.

[39] Martin Maas, Krste Asanovic, and John Kubiatowicz. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal, editors, *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS*, pages 138–143, 2017.

[40] Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. Faasdom: a benchmark suite for serverless computing. In *DEBS : The 14th ACM International Conference on Distributed and Event-based Systems*, pages 73–84, 2020.

[41] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[42] Microsoft. Microsoft Azure, 2021. https://azure.microsoft.com/, visited 2021-05-18.

[43] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud*, 2019.

[44] New Relic One. For the Love of Serverless: 2020 AWS Lambda Benchmark Report for Developers, DevOps, and Decision Makers, 2020. https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020, visited 2021-05-18.

[45] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 57–70, 2018.

[46] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, page 131–141. Association for Computing Machinery, 2020.

[47] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, page 1063–1075, 2019.

[48] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Ada Gavrilovska and Erez Zadok, editors, *USENIX Annual Technical Conference, USENIX ATC*, pages 205–218, 2020.

[49] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC, page 281–295, 2020.

[50] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2020.

[51] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys, 2020.

[52] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, 2020.

[53] The gVisor Authors. gVisor Performance Guide, 2021. https://gvisor.dev/docs/architecture_guide/performance/#file-system, visited 2021-05-18.

[54] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery Berger, and Christos Kozyrakis, editors, *ASPLOS : 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[55] V8 Team. A lighter V8, 2019. https://v8.dev/blog/v8-lite.

[56] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST)*, pages 267–281, 2020.

[57] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference*, EuroSys, 2019.

[58] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ATC, page 133–145, 2018.

[59] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. pages 133–146. USENIX ATC, 2018.

[60] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. In *IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018.

[61] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC, page 30–44, 2020.

[62] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC, page 1–12, 2019.