FIREWORKS: A Fast, Efficient and Safe Serverless Framework

Wonseok Shin

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Changwoo Min, Chair
Cameron D. Patterson
JoAnn M. Paul

May 3, 2021
Blacksburg, Virginia

Keywords: Serverless Computing, Cloud, Snapshot, Firecracker

FIREWORKS: A Fast, Efficient and Safe Serverless Framework

Wonseok Shin

(ABSTRACT)

Serverless computing is a new paradigm, and it is becoming rapidly popular in Cloud computing. Serverless computing has interesting, unique properties that the unit of deployment and execution is a serverless function. Moreover, it introduces the new economic model pay-as-you-go billing model. It provides a high economic benefit from highly elastic resource provisioning to the application.

However, it also accompanies the new challenges for serverless computing: (1) *start-up time* latency problem from relatively short function execution time, (2) *high-security risk* from highly consolidated environment, and (3) *memory efficiency* problem from unpredictable function invocations. These problems not only degrade performance but also lowers the economic benefits of Cloud providers.

In this work, we propose ***VM-level pre-JIT snapshot*** and develop **Fireworks** to solve the three main challenges without any compromises. The key idea behind the VM-level pre-JIT snapshot is to leverage pre-JITted serverless function codes to reduce both start-up time and execution time of the function and improve memory efficiency by sharing the pre-JITted codes. Also, Fireworks can provide high-level isolation by storing the pre-JITted codes to the snapshot of microVM's snapshot. Our evaluation shows that Fireworks outperforms the state-of-art serverless platforms by 20.6× and memory efficiency up to 7.3×.

FIREWORKS: A Fast, Efficient and Safe Serverless Framework

Wonseok Shin

(GENERAL AUDIENCE ABSTRACT)

Serverless computing is the most popular in cloud computing. Contrary to its name, developers write and run their code on servers managed by cloud providers. The number of servers, required CPU, memory are automatically adjusted in proportion to the incoming traffic. Also, the users only pay for what they use and the pay-as-you-go attracts attention as new infrastructure. Serverless computing continues to evolve and it is being done as research from business to academic. There are many efforts to reduce cold start, which is the delay in creating the necessary resources when a serverless program runs first. The serverless platforms prepare resources in advance or provide lighter cloud resources. However, this can waste resources or increase a security threat. In this work, we propose a fast, efficient, and safe serverless framework. We use Just-In-Time (JIT) compilation, which can improve the performance of the interpreter languages which are widely used in the serverless. We keep the JIT-generated machine code in the snapshot for reuse. Besides, the security is guaranteed by the VM-level snapshot. In addition, the snapshot can be shared, increasing memory efficiency. Through our implementation and evaluation, we have shown that Fireworks improve up to 20 times in terms of cold start performance and more than 7 times in memory efficiency than state-of-the-art serverless platforms. We believe our research has made a new way to use the JIT and the snapshot in the serverless computing.

Dedication

To my wife and son.

For their endless support and love.

Acknowledgments

First and foremost, I am sincerely thankful to my research advisor, Prof. Min Changwoo, in ECE at Virginia Tech, to lead and guide me on my MS thesis. With his research spirit and insightful guidance, I could keep learning and going forward during the master period. His thinking, knowledge, and endeavor helped me throughout my research.

I thank Prof. Cameron D. Patterson, Prof. JoAnn M. Paul for serving as my thesis committee, and also, thank you for the good comments and feedback.

I am also grateful to my colleagues who worked with me at COSMOSS Lab. We did not have much time together due to the COVID19, but we had many opportunities to share valuable knowledge.

Lastly, I thank my wife and son, who always encourage and cheer me. This has always been a great support. Also, I always thank my parents and extended family members for supporting me from my hometown. Thank you all again for being able to do this with the help of everyone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Serverless computing is a new paradigm in Cloud computing that is becoming increasingly popular recently [24]. It eliminates the burden to administrate the infrastructure and provides extreme elasticity for the resource provisioning without the developer's efforts. Moreover, it introduces a pay-as-you-go billing model [4] compared to the pricing model of the traditional Cloud computing model based on server uptime regardless of actual usage. Due to these advantages, most of Cloud providers already offer their serverless computing models such as Amazon Lambda [10], Microsoft Azure [35], Google [18], IBM Cloud Functions [22].

A serverless application consists of chains of *serverless functions*, the unit of deployment and execution in serverless computing. Each function is executed on a lightweight sandbox (*e.g.*, virtual machine, container) provided by a Cloud provider. The resources allocated to the application is determined by the number of functions corresponding to the application and load of each function. Most of the serverless applications are written in high-level languages such as Node.js and Python so that the serverless platforms provide a language runtime for executing the user's code over the sandbox. Basically, the sandbox and runtime are launched upon a user's request (*e.g.*, function call, triggering event) to execute the function.

Serverless computing is used in various industries. The first use case is the Coca-Cola company's freestyle bending machine with serverless computing. In the COVID19 environment, Coca-Cola bending machines comply with safe cleaning regulations, but they want to provide a safer experience with touchless bending machines. Hovering the camera over the QR code

instantly connects to the cloud and provides a screen to the user's mobile phone interface. It provides web apps through serverless, which provides low latency. As of the end of 2020, 52,000 Coca-Cola freestyle bending machines are providing services in serverless computing.

In addition, Netflix, the world's leading Internet TV, delivers 12,500 hours of media content daily. Netflix is using more than 100,000 serverless computing instances for data analysis, personal media recommendation, and device-specific media transcoding.

Serverless computing has unique characteristics in contrast to the traditional Cloud computing. First of all, the execution time of a serverless function is very shorter than traditional server programs because a serverless application is composed of multiple serverless functions. For example, in the Microsoft Azure Cloud, about 50% of functions take less than 1 second of execution time, and 40% of functions take less than 3 seconds of execution times [42]. Also, due to the short function execution time, Cloud providers aim to consolidate a large number (*e.g.*, thousands) of serverless functions in a single machine to utilize the hardware resources more efficiently.

The unique characteristics accompany the unique challenges in serverless computing [21]. The relatively short execution time of a function implies that any overhead besides the function execution will impose a large overhead. In particular, the *start-up time* is one of the largest overheads [14, 42]. It includes booting time of VM, OS, and container as well as loading time of language runtime and applications. Compared to the function execution time, which often takes less than a second, the start-up time imposes significant overhead. In addition, the loading time of language runtime and application also imposes significant overhead in the serverless applications where the most functions are written in an interpreter language such as Node.js and Python. Besides the performance for the serverless users, reducing the start-up time is important to Cloud providers for higher profitability because the start-up time is not charged to the user.

There have been research efforts recently to shorten the long start-up time in serverless computing. For example, SOCK [39] and Cloudflare [26] use *lighter sandboxes* – *e.g.*, sharing a language runtime [26] – to reduce the start-up time. However, the lighter sandboxes fundamentally provide a lower isolation level, increasing the security risk, especially in highly consolidated serverless environment [34, 37].

Another approach is *warm pool based approach*, which pre-launches sandboxes before a function is invoked [19, 50]. The pre-launched sandbox can efficiently hide the start-up latency without compromising the isolation level. However, the pre-launched sandboxes take hardware resources and hinder high consolidation on the host. Especially, it is not effective for non-popular functions. For example, a previous study [42] reports that 18.6% of popular functions are called more than once a minute. In other words, the warm pool based approach is not effective for 81.4% non-popular functions.

Besides the start-up time latency, the execution time penalty is also problematic, especially for interpreter languages, such as Node.js and Python, which take the majority in serverless applications. Although Just-in-time (JIT) compilation has been very successful to improve performance of interpreter languages especially in a long running server applications, turning off the JIT is the known best practice in serverless computing. That is because of two reasons. First, the JIT compilation cost does not pay off because of the short function execution time. Moreover, the most serverless sandbox is configured to use a single CPU so that the JIT compilation competes the CPU time with the application execution. After all, the JIT compilation not only introduces performance unpredictability [31], but also often incurs a significant slow down in application execution. For example, some data center applications spend up to 33% of their time in JIT warm-up [30].

In this thesis, we propose Fireworks, a new serverless platform that offers extremely short start-up latency and high performance without compromising security isolation level nor

wasting hardware resources. We focus on interpreter languages, especially Node.js and Python, in which the majority of serverless applications are written.[1]

We propose a *VM-level pre-JIT snapshot* approach in Fireworks. We found that JIT has a potential to significantly improve the performance of serverless applications if properly used. When installing a serverless function, Fireworks creates a snapshot of a VM *after* a language runtime loads the serverless function and finishes the JIT compiling of the serverless function. Upon invoking the function, Fireworks resumes the VM snapshot and executes the function with new arguments. The program is already loaded and JITted in the VM snapshot, so we do not need to pay the cost of start-up time, application loading, and JIT compilation during the function execution. Moreover, Fireworks shares the snapshot with the JITted functions among concurrent function invocations in a copy-on-write (CoW) manner so Fireworks can reduce the memory consumption. The VM-level pre-JIT snapshot meets our goals in three aspects: (1) the highest level of isolation using a VM as a sandbox, (2) instant start-up and fast execution performance resuming JITted snapshot, and (3) less memory consumption sharing the snapshot through CoW. In summary, we made the following contributions in this thesis:

- **VM-level-JIT Snapshot.** We propose a novel VM-level pre-JIT snapshot for serverless functions written in an interpreter language. 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 for the function.
- **Serverless Framework.** We develop a new serverless framework named Fireworks. We use an open source serverless platform, OpenWhisk [9], and a lightweight hypervisor, Firecracker [1], as a sandbox. Fireworks analyzes a serverless function written in Node.js and Python and annotates to trigger JIT compilations. Then it creates a microVM snapshot of

[1] In AWS lambda [38], Node.js accounts for 53% and Python accounts for 36% so two languages account for 89% of the total serverless functions.

Firecracker after finishing loading and JITting the function. Finally, Fireworks resumes the snapshot upon the function invocation taking new arguments from OpenWhick framework.

- **Evaluation.** Our evaluation shows that Fireworks outperforms the state-of-the-art serverless computing framework by 20.6× in performance and shows memory efficiency up to 7.3×. We ran two representative serverless benchmarks: FaaSdom [32] for microbenchmark and ServerlessBench [53] for real-world applications.

The rest of the thesis is organized as follows. §2 provides the backgrounds and motivations of this work. Following §3 describes the design of Fireworks in detail. §4 explains the implementation of Fireworks. §5 shows our evaluation results with microbenchmark based on FaaSdom and real-world workload based on the serverlessBench. §6 discusses security issues of a snapshot-based approach. §7 introduces related work of the thesis, and §8 concludes this thesis.

Chapter 2

Background and Motivation

2.1 How Serverless Platform Works

Figure 2.1 illustrates the overall architecture of a serverless computing platform. The front end consists of three components: 1) user interface, 2) API gateway, and 3) Cloud trigger. First, a serverless user builds and deploys a serverless application through a web-based user interface. Next, when a user sends a request to the serverless platform, the request will be processed in the API gateway. The gateway will relay the request to one of the backend servers, which executing the requested function.

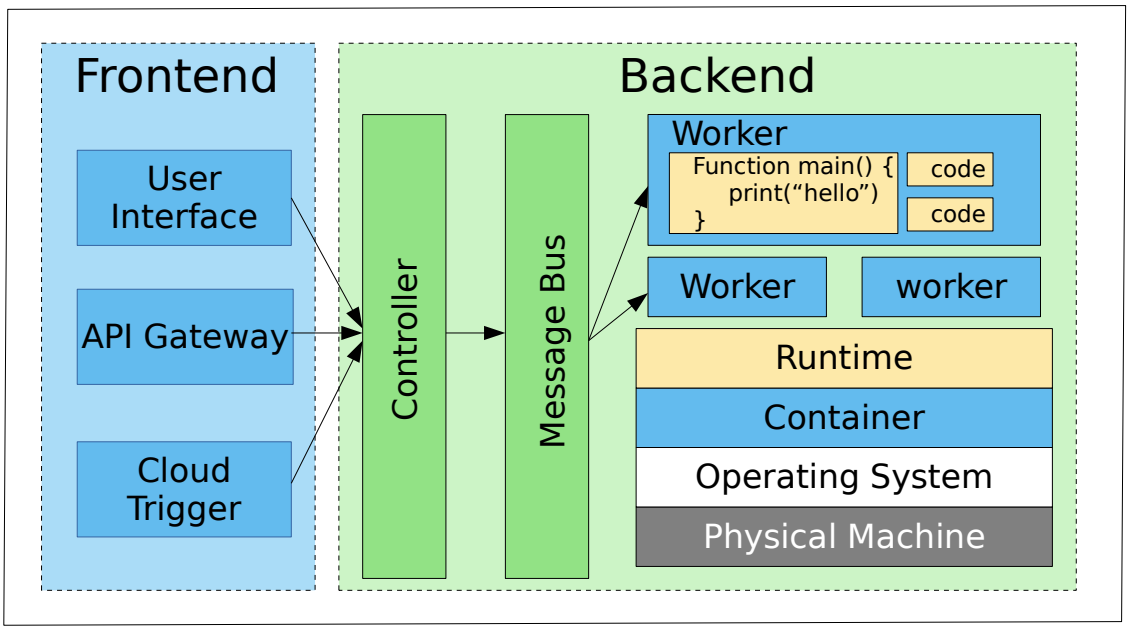


Figure 2.1: Overall architecture of a serverless platform.

Serverless Platform	Isolation	Performance	Memory Efficiency
Firecracker (Amazon) [1]	High (VM)	Medium (snapshot)	High (snapshot)
OpenWhisk (IBM) [9]	Medium (container)	Low (no optimization)	Low (pre-launching)
gVisor (Google) [20]	Medium (container)	Medium (snapshot)	High (snapshot)
Cloudflare [26]	Low (runtime)	High (pre-launching)	High (pre-launching)
Catalyzer [14]	Med (container)	High (pre-launching)	High (pre-launching)
Fireworks	High (VM)	Extreme (snapshot+JIT)	Extreme (snapshot+JIT)

Table 2.1: Design comparison of the serverless platforms.

Lastly, the Cloud trigger is another way to invoke a function, especially upon a registered event occurs – *e.g.*, inserting a record in DBMS or changing a file in a Cloud storage. When the event occurs, the triggering request is delivered to the controller to trigger a registered function to execute. After the controller processes the requests, the corresponding messages are put in the message bus, which is the communication backbone of the serverless platform, passing a request with arguments. The messages are used to create a new runtime for executing the function or deliver the request to the existing runtime. The runtimes are called *worker*, which is running inside a sandbox.

2.2 Characteristics of Serverless Workload

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

Short function execution time. The most prominent difference in serverless workload is short execution time. The execution time of a function is remarkably shorter than the lifetime of a worker. The average execution time of the function is about a second [42]. Also, most functions are not frequently called. In the previous study, it is reported that each function is called once in 30 minutes on average [42]. Such a short function execution time makes start-up time – a 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 before loading the function. This so called *cold start* takes long – ofter longer than the function execution time. The current practice to mitigate the cold start latency is that Cloud providers defer the termination of the worker sandbox for a certain period hoping another request for the same function is delivered. If it happens, the start time in this case – so called *warm start* – will be much faster. However, it is hard to predict when the next function invocation is delivered [42]. If a new request is not delivered for a certain duration, the serverless platform will terminate the sandbox as wasting the hardware resources.

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 cost. 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 vs. VM) to reduce the start-up time latency [9, 20, 26, 29]. However, as the sandboxes with a lower isolation level share more resources between sandboxes, they are exposed to high security risks. On the other hand, a sandbox with high isolation level (*e.g.*, a microVM in Firecracker hypervisor [1]) incurs more overhead, such as cold start time.

Memory efficiency. The most scarce hardware resource is memory because memory is needed to maintain sandboxes waiting for a request, and it is the main limiting factor for consolidation [51]. However, it is hard to predict the invocation of a function, so serverless platforms end up wasting a lot of memory.

2.3 Optimizing Serverless Platforms

We compare the state-of-the-art serverless platforms in three crucial aspects – 1) isolation, 2) performance, and 3) memory efficiency – as summarized in Table 2.1. As the start-up time is the first, the most critical performance bottleneck in the serverless platform, much research effort has been made to reduce the start-up time latency. In the rest, we introduce each approach in Table 2.1 more detail.

Firecracker [1] is a lightweight virtual machine monitor (VMM) developed by Amazon and optimized for serverless applications. It supports a high isolation level by providing VM-level isolation. Also, it provides the VM-level snapshot so that the VM-level snapshot can be resumed and shared by multiple VM instances.

OpenWhisk [9] is a container-based open source serverless platform deployed in IBM Cloud. While a Linux container is more lightweight than VM, it supports a lower isolation level because containers share the kernel resources in the same host [6, 29].

gVisor [20] extends the security level of a container based sandbox. Previous study reports that a container allows the most system calls (*e.g.*, 306 out of 350 Linux system calls) [6], meaning that such exposed system calls can be exploited for a security attack. To improve the security of a container, gVisor intercepts the container's access to the Linux Kernel and limits some systems calls to improve the security level [23]. Specifically, Sentry and Gofer in gVisor intercept the system calls and I/O requests requested by the container and filter them before serving to host OS [23].

Cloudflare [26] is a lightweight runtime based sandbox for Node.js. Unlike other serverless computing platforms, Cloudflare does not use containers or VMs for isolation. Instead, it utilizes V8:Isolate, a lightweight context in V8 javascript engine. Cloudflare shows high performance because a single process can perform hundreds of V8:Isolate. However, multiple

functions are executed in a single process, so its isolation level is weaker than others [12].

Catalyzer [14] is a gVisor-based serverless platform, which is specially designed to reduce start-up latency. It provides a sandbox fork approach that restores a function from the checkpoint. By restarting from the checkpointed image, it can reduce the start-up time significantly. Furthermore, it proposes a new OS primitive sfork to eliminate initialization cost for warm start. The sfork directly uses a clean state of a sandbox template for the user's application. Catalyzer guarantees the same isolation level to gVisor because it is based on gVisior.

Summary. Current serverless computing platforms have limitations in performance, resource efficiency, and safety. Most of the previous studies compromise at least one of these three aspects.

Chapter 3

Fireworks Design

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

3.1 Design Overview

The overall flow of Fireworks is divided into two phases: 1) installation phase and 2) invocation phase, as shown in Figure 3.1. In the installation phase, Fireworks creates a pre-JITted VM snapshot of the installed function. Then in the invocation phase, Fireworks restores the pre-JITted VM snapshot and resumes the function 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 3.1. MicroVM manager controls creating, snapshotting, resuming a microVM on Firecracker hypervisor. Code annotator is responsible for transforming a user-provided function source code, so the function follows Fireworks procedure of installation and invocation. The added code (marked blue in Figure 3.1) by the code annotator includes enforcing JITting

the function code, requesting VM-level snapshot creation, and taking function parameters. Invoker receives the user's request from the serverless frontend and launches the function. Snapshotter creates a VM-level snapshot according to the annotated function request. Parameter passer helps the function get the parameters when the snapshot resumes at the regular function entry. We will explain each phase of Fireworks operation in detail in the rest of the chapter.

Installation phase. Fireworks automatically adds annotations to the user-provided function code and creates a VM-level snapshot after finishing JITting the function code. The installation procedure is as follows: Fireworks first requests to create microVM to Firecracker, and Firecracker creates and makes the microVM ready for a runtime (① in Figure 3.1). 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 function, (③) and the annotated function performs JITting itself and then creates a snapshot right before the original function entry point (④).

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, which will run the requestion function (❺❻ in Figure 3.1). It then restarts the VM snapshot for the serverless function (❼). The annotated serverless function is resumed right after the snapshot point. It reads the function parameters from Fireworks and executes the regular serverless function entry (❽).

How Fireworks meets the requirement. 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), so it provides a high isolation level comparing to the approaches using a container or runtime as a sandbox.

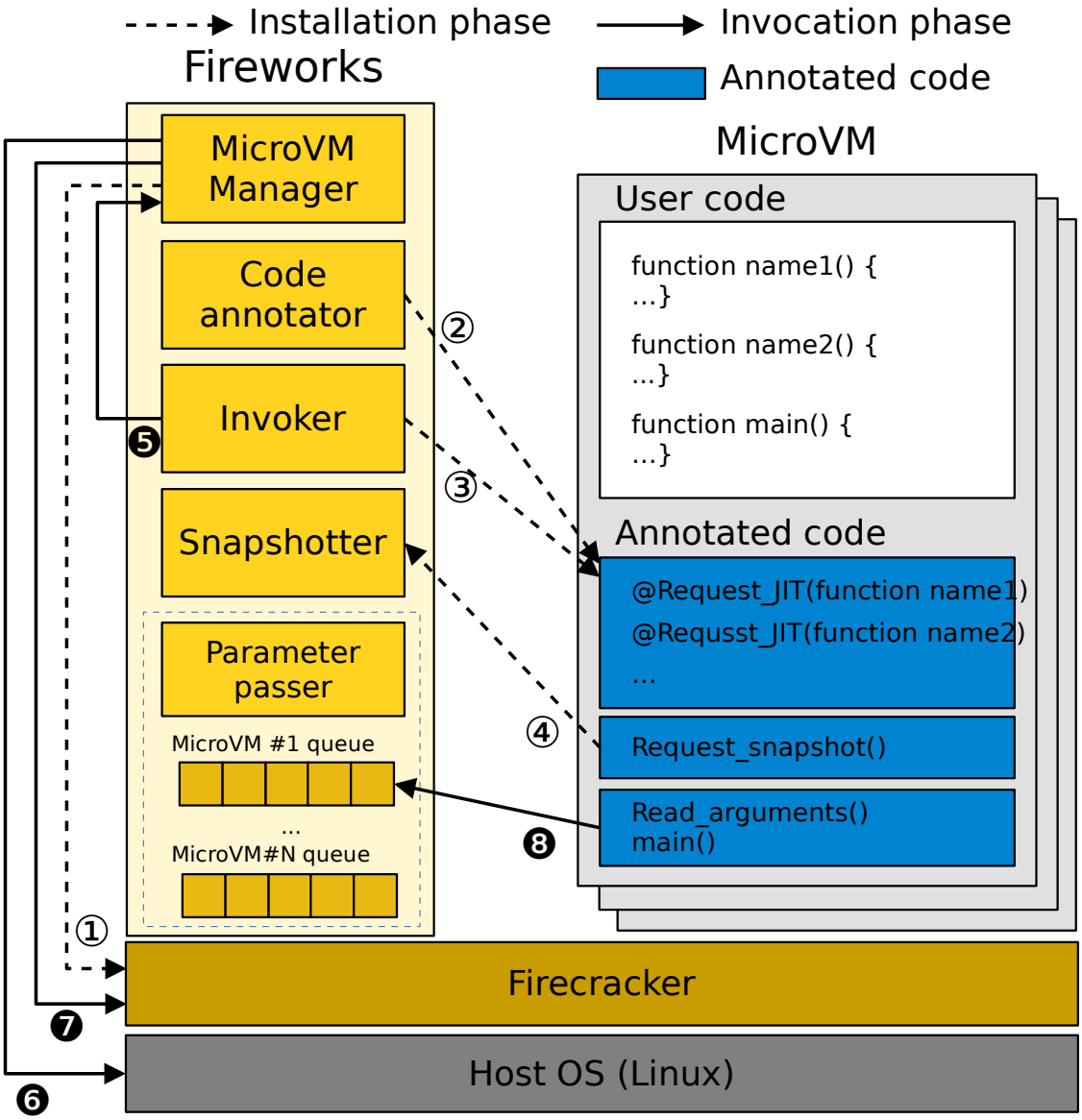


Figure 3.1: Fireworks overall architecture.

- **High performance**: Fireworks achieves high performance with an instant start-up and JITted function execution. Essentially, Fireworks uses a microVM snapshot which is created after a function is loaded and JITted. Hence, there is no initialization time – such as booting VM and OS, launching a runtime, and loading the serverless functFion to the runtime – so Fireworks achieves an instant start-up of a function. Moreover, the annotated serverless function is already JIT-compiled while creating the VM snapshot so that

the serverless function is executed with the pre-JITted machine code during the runtime. Since the JIT-compiled code is faster than the interpreter and we do not need to pay the cost of JIT compilation during the runtime, the function execution time is also significantly reduced.

- **Memory efficiency**: Multiple instances of a serverless function share the VM-level memory snapshot in a copy-on-write (CoW) manner. Fireworks uses the 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 most states of microVM, OS, library, runtime, even JITted binary. Only argument-specific execution state, which is updated during a function execution, will not be shared.

3.2 Automatic Source Code Annotation

Code annotator automatically adds additional code to a user's serverless function. Figure 3.2 shows an example of the annotated version of user code written in Node.js. We omitted too implementation details for clarity. The annotated code consists of three parts: 1) performing JIT compilation (Lines 7–9), 2) creating a VM snapshot (Lines 12–21), and 3) starting the serverless main function with new parameters upon invocation (Lines 24–39).

In order to perform JIT compilation, Fireworks leverages the extended feature in modern language runtimes. For example, V8 javascript engine used in Node.js runtime offers OptimizeFunctionOnNextCall() method, which performs the JIT compilation of the specified function. Similarly, Python Numba [5] offers a similar annotation for JIT compilation. Python Numba performs JIT compilation of functions annotated with @jit(cache=True). Fireworks adds the JIT annotation for all methods in the serverless function; In Figure 3.2, Fireworks adds ___fireworks_jit(), which triggers JIT compilation of user code, main().

The VM snapshot should be created after finishing the JIT compilation and before starting the serverless function entry point. To precisely control the snapshot point, Fireworks adds the snapshot creation code (__fireworks_snapshot() at Line 30) to the source code. The snapshot is created right after finishing the JIT compilation (Line 28) and before resuming the execution (Line 32).

The last part is the main function (__fireworks_main() at Lines 28-37), where the language runtime starts the program execution. After the JIT compilation and snapshot creation, it starts parameters from a message queue associated with this function. To distinguish the function instance (fcID, a microVM ID), Fireworks uses Firecracker's microVM Metadata Service (MMDS) [16].

3.3 Creating a pre-JITted VM Snapshot

Actual JIT compilation is performed when the generated __fireworks_jit() is called in the annotated program (Line 28 in Figure 3.2, ③ in Figure 3.1). In Node.js, the runtime first interprets the source code into the bytecode, and the Node.js JIT compiler engine, turbofan, generates the machine code from the Abstract Syntax Tree (AST) for methods annotated with OptimizeFunctionOnNextCall(). Similarly, Python Numba performs JIT compilation for methods annotated with @jit(cache=True).

After the 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 microVM, OS, libraries, language runtime, and even JITted code compiled as illustrated in Figure 3.3. Fireworks sends an HTTP request to a host to create a snapshot using Firecracker API (__fireworks_snapshot() at Line 30 in Figure 3.2). Firecracker creates a memory snapshot of an entire microVM and stores the snapshot to a file. Now Fireworks

```
1  /* A simple serverless function printing "hello world". */
2  function main(params) {
3    console.log("hello world " + params);
4  }
5  /* ===> Fireworks code annotation starts here. <=== */
6  /* Trigger JIT compilation of all user functions, "main". */
7  function ___fireworks_jit() {
8    %OptimizeFunctionOnNextCall(main);
9  }
10
11 /* Send an HTTP request to host creating a VM snapshot. */
12 async function ___fireworks_snapshot() {
13   const http = require("http");
14   const options = {
15     hostname: '172.17.0.1',
16     path: `/?snapshot=Y&name=SERVERLESS_FUNCTION_NAME`,
17     method: 'GET'
18   };
19   const req = await http.request(options, (res));
20   req.end();
21 }
22
23 /* This is where the program execution starts first time. */
24 const { exec } = require("child_process");
25 exec(`kafkacat -C -t topic${fcID}` => {
26   async function ___fireworks_main(stdout) {
27     /* First it performs JIT compilation. */
28     await ___fireworks_jit();
29     /* Then it creates a VM snapshot. */
30     await ___fireworks_snapshot();
31     /* Upon invokation, it resumes here and first gets parameters. */
32     exec(`kafkacat -C -t topic${fcID}` => {
33       params = JSON.parse(stdout);
34       params['fcID'] = fcID;
35       /* Then it start the entry point of a serverless function with given parameters. */
36       main(params);
37     });
38   }
39 }
40 /* ===> Fireworks code annotation ends here. <=== */
```

Figure 3.2: Simplified example of Fireworks source code annotation written in Node.js.

completes the installation phase of a serverless function (①–④ in Figure 3.1).

3.4 Invoking a Serverless Function

After a serverless function is installed, now Fireworks can invoke the serverless function from a request by resuming the VM snapshot (❺–❽ in Figure 3.1). When a user sends a request to the serverless function, Fireworks puts the requested information into the parameter passer

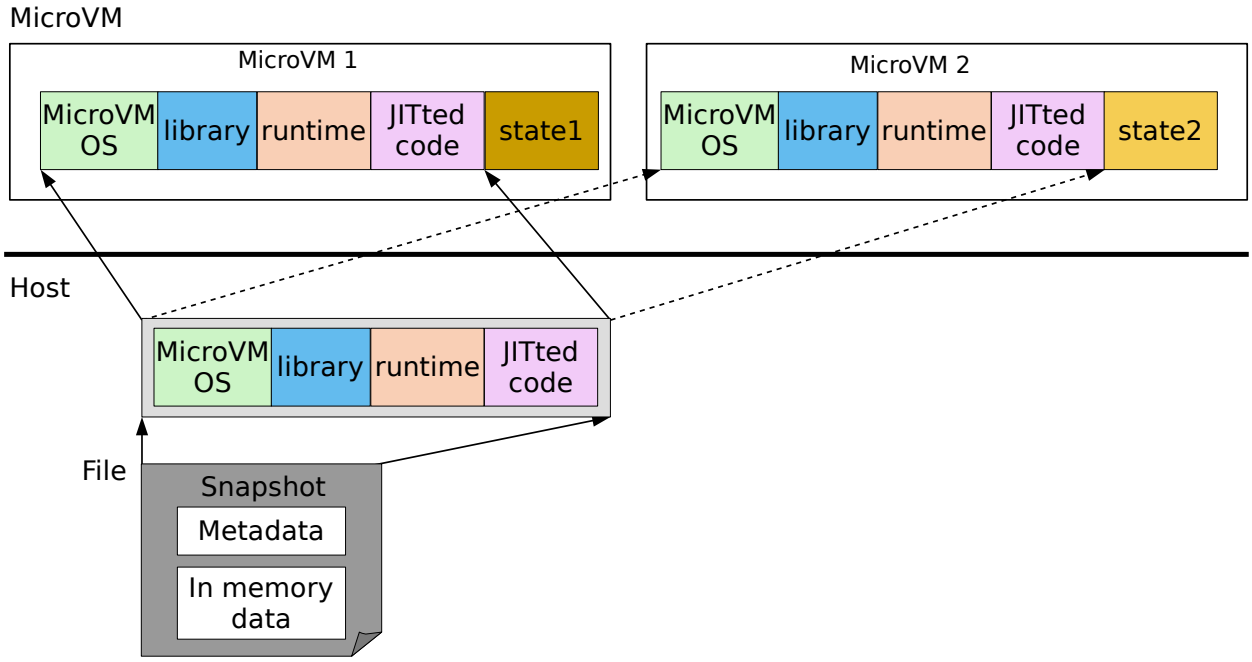


Figure 3.3: Sharing VM-level memory snapshot among multiple microVMs (*i.e.*, sandboxes for a serverless function) in Fireworks.

queue and requests to resume the microVM to Firecracker. MicroVM will restart with the corresponding snapshot image. Hence, Invoking the serverless function is to load the snapshot as a file into memory. Fireworks resumes the execution right after the pre-Jitted VM snapshot image is created. (Line 32 in Figure 3.2). Fireworks configures the network, so the resumed microVM can access the network before resumption and prepares parameter passing so the resumed function can access the parameters of the requested function. We will describe these two in detail in the following sections.

3.5 Enabling a 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 resources conflict – *i.e.*, the same

IP address and MAC address will be assigned.

To solve the network connectivity problem, Fireworks uses the network namespace [15] and Network Address Translation (NAT). Figure 3.4 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 NAT Table in iptables of network namespace 1. When the packet arrives at microVM#1 through tap device 0 (tap0), the tap device name 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 associated with its network namespace.

Note that the information for network connectivity of microVMs which are created from the same snapshot image are identical. So Fireworks needs additional information to distinguish the microVMs. We leverage Firecracker's MMDS [16] to keep the metadata (*e.g.*, microVM ID) of the microVMs.

3.6 Taking Serverless Function Arguments

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

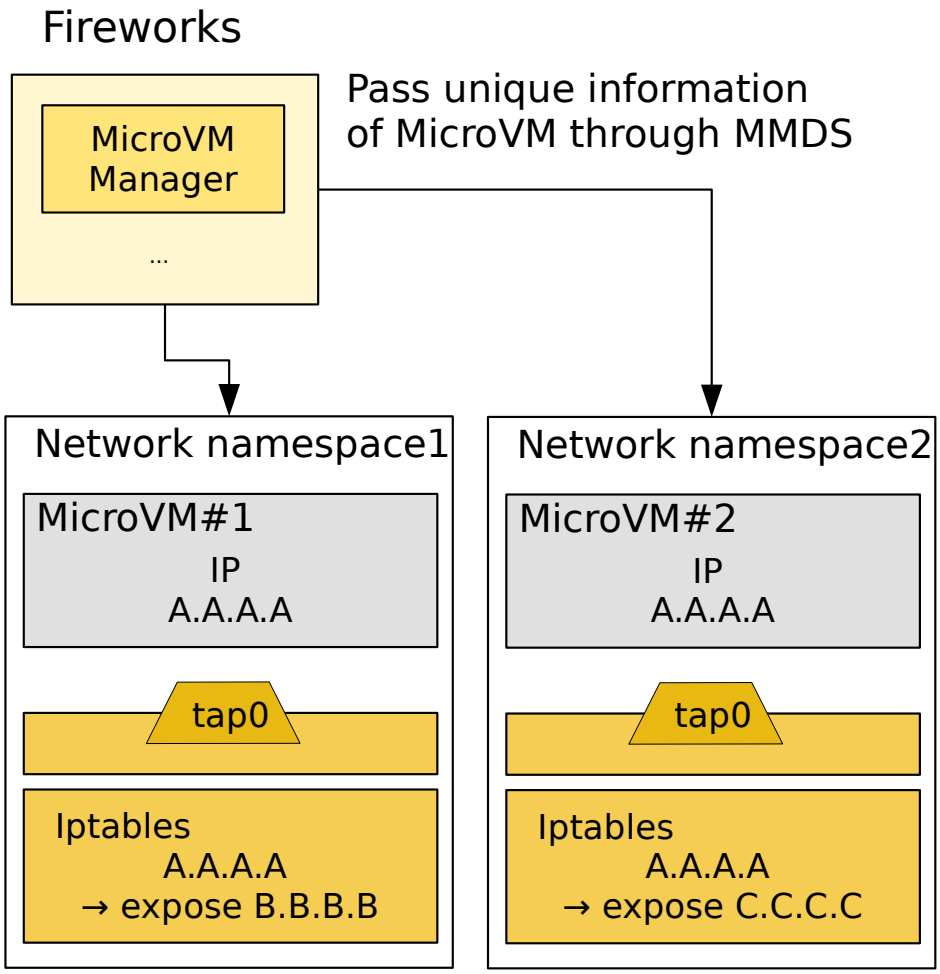


Figure 3.4: Network connectivity in Fireworks.

To solve this problem, Fireworks adds a code that fetches parameters from outside of the microVM as soon as it loads the snapshot. Fireworks exploits a Kafka queue [25] – a software message bus – to pass arguments to a serverless function. Before resuming the microVM, Fireworks first puts the function arguments to a designated Kafka queue. When the microVM resumes, it first fetches the parameters from the queue (Lines 32–34 in Figure 3.2). Since the snapshot image is identical, the function needs additional information to find the corresponding message bus. The identification information is inserted by Fireworks using MMDS after resuming the snapshot. When the snapshot image resumes successfully, the function checks its identification and reads the arguments from the message bus.

Chapter 4

Implementation

We implement Fireworks based on the Firecracker v0.24.0. Fireworks leverages Firecrackers microVM features for guaranteeing the VM-level high isolation level with snapshot function. We develop an end-to-end serverless infrastructure including invoker, code Annotator, snapshotter, and parameter passer. The parameter passer is implemented based on Kafka [25], a software message bus. The source codes of Fireworks written in Bash, Node.js, Python, and C++, which are compromising 3,500 lines of codes.

Chapter 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 IO-intensive ones by reducing start-up time and making the function execution time faster? (§5.2)
- How effective is Fireworks's approach to improve the performance of real-world serverless applications? (§5.3)
- How much memory can Fireworks save by sharing the memory snapshot across microVMs? (§5.4)
- How effective is Fireworks's design choices in improving performance and saving memory usage? (§5.5)

5.1 Evaluation Methodology

Evaluation Environment. We perform the evaluations on a server with Intel Xeon Platinum 8180 CPU (64 physical cores), 128GB memory, and 2TB SSD. For all evaluation, we configured a microVM similar to the typical configuration in serverless computing: one vCPU, 512MB memory, and 2GB of disk spaces [42].[1]

[1] Average memory size in a serverless sandbox is 170MB [42].

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 5.1: Tested serverless applications.

We compare Fireworks against the state-of-the-art serverless frameworks and sandboxes for serverless computing: OpenWhisk [8], gVisor [20], and Firecracker [1]. We use Openwhisk v20.11 with Kubernetes and gVisor runsc v1.0.2 with Docker. Regarding the runtime for the languages, we use the latest versions of this writing: Node.js v12.18.3 and Python v3.8.5. Note that we do not include Catalyzer [14] because it was not publicly available.

Workloads. We used two serverless benchmarks: FaaSdom [32] and ServerlessBench [53], shown in Table 5.1.

For micro-benchmark, we choose the FaaSdom benchmark, which consists of two compute-intensive benchmarks (faas-fact and faas-matrix-mult) and two IO-intensive benchmarks (faas-netlatency and faas-diskio) written in both Node.js and Python. faas-fact and faas-matrix-mult perform integer factorization and multiplication of large matrices many 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 the disk.

We choose two real-world serverless applications from ServerlessBench [53]: Alex Skills and data analysis, both of which are written in Node.js. Alex Skills is a collection of applications performed through the Amazon Alexa AI speaker device. The Amazone Alexa AI speaker receives the user's voice and performs the corresponding serverless functions of Alexa Skills in the Cloud, based on its analysis. The application provides the functionality of scheduler and

smart home management. Another application is the data analysis application. It calculates bonuses and taxes with employees' roles and makes statics. The wages of multiple employees are continuously inputted when a user wants. Note that these two real-world applications consist of a chain of serverless functions, as illustrated in Figure 5.3. For real-world applications, we only compare against OpenWhisk [9]. That is because sandbox managers, such as Firecracker [1] and gVisor [20], cannot process the chain of the serverless functions, and only OpenWhisk and Fireworks are able chain serverless functions.

5.2 FaaSdom Microbenchmark

We first discuss evaluation results of the Node.js version of FaaSdom benchmark (§5.2.1) 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 5.1 shows the performance comparison and latency breakdown of Node.js version of FaaSdom benchmark. We measured the latency from the function invocation to the function termination. We breakdown the latency into *start-up* latency, the function execution time (*exec*), and all other times besides the start-up and execution time (*other*). For OpenWhisk, gVisor, and Firecracker, we measured the latency for both cold start (c) and warm start (w). However, Fireworks, which relies on the snapshot restart, does not have such distinction (both).

(1) Compute-intensive benchmarks. Figure 5.1(a) is the evaluation result for CPU-intensive integer factorization benchmark (faas-fact). Fireworks shows extremely fast start-

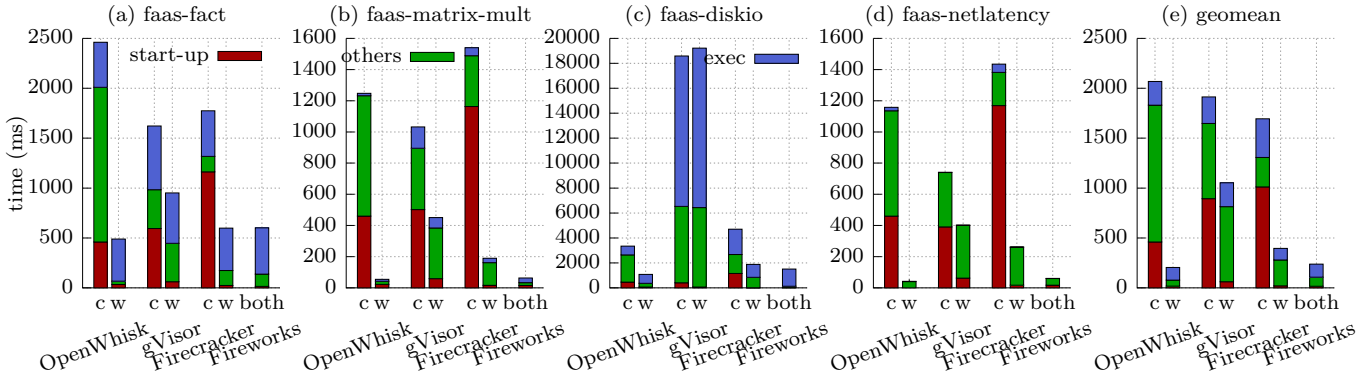


Figure 5.1: Latency comparison of the Node.js version of FaaSdom benchmark.

up time, which is comparable or even faster than warm start-up time 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 to boot VM, guest OS, 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 in case of cold start, such as authentication and message queue initialization. gVisor shows slower cold start-up time and execution time as it enforces additional security check to the container to improve the security of the runtime environment. The gap in the 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 the warm start case. The evaluation results of another compute-intensive benchmark, matrix multiplication benchmark (faas-matrix-mult), show similar performance characteristics with the integer factorization benchmark, which we discussed early.

(2) Disk-intensive benchmark. The disk-intensive benchmark, faas-diskio, performs 10KB-sized file read and write operations 100 times. Overall, Fireworks shows significantly

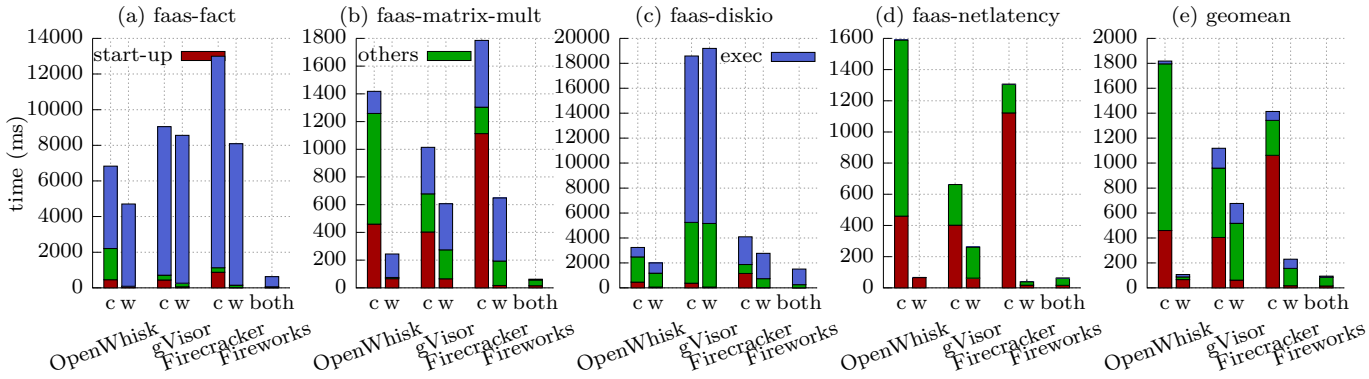


Figure 5.2: Latency comparison of the Python version of FaaSdom benchmark.

shorter latency than other serverless platforms. Specifically, as shown in Figure 5.1(c), Fireworks shows up to 68× faster cold start-up and up to 3.8× faster warm start-up time. Moreover, it shows up to 9.2× faster execution time than other serverless frameworks. Interestingly, we found that the execution time in IO-intensive workloads is mostly determined by the IO efficiency of the sandbox mechanisms. The performance impact of JIT compilation is marginal.

gVisor shows the slowest IO performance because of system call monitoring using two additional components: Sentry and Gofer. Sentry monitors system calls using seccomp filter and sends file IO requests to Gofer I/O to the Gofer to checks all access to the host resources [46]. While the cost of system call monitoring using Sentry and Gofer is much higher than IO overhead in Fireworks, the security level of 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 the heavy QEMU. Also, Firecracker removes unnecessary devices and uses a lightweight 9p filesystem in crosvm [1].

Container-based OpenWhisk uses OverlayFS [13] and chroot while it interacts directly with the host filesystem. Therefore, I/O speed is faster than microVM, which has its own VM file systems.

(3) Network-intensive benchmark. The network microbenchmark, 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 25× faster cold start-up time and up to 3.6× faster warm start-up time. Fireworks' execution time up to 22× faster in cold start cases and up to 2× faster in warm start cases. Fireworks has 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 5.1(e) shows the geometric mean of four FaaSdom benchmarks. Overall, Fireworks shows up to 8.6× shorter latency comparing to other serverless frameworks.

5.2.2 Python Benchmarks

Figure 5.2 shows the evaluation results of the Python version of FaaSdom benchmarks. The evaluation results reveal that Python is, in general, slower than Node.js, which is the same as generally known.

(1) Compute-intensive benchmarks. Fireworks shows significant higher performance in compute-intensive benchmarks comparing to other serverless frameworks. Interestingly, Fireworks dramatically reduces the execution time for Python by leveraging pre-JITted machine code in the snapshot. For the integer factorization shown in Figure 5.2(a), Fireworks shows 59.8× faster cold start-up time and 4.4× faster warm start-up time. Also, it achieves 20× faster execution time in cold start case and 14.6× faster execution times in warm-up case. For the matrix multiplication shown in Figure 5.2(b), Fireworks shows similar performance trends with the integer factorization. Fireworks shows up to 74.2× faster cold start-up time

and 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 the warm start cases.

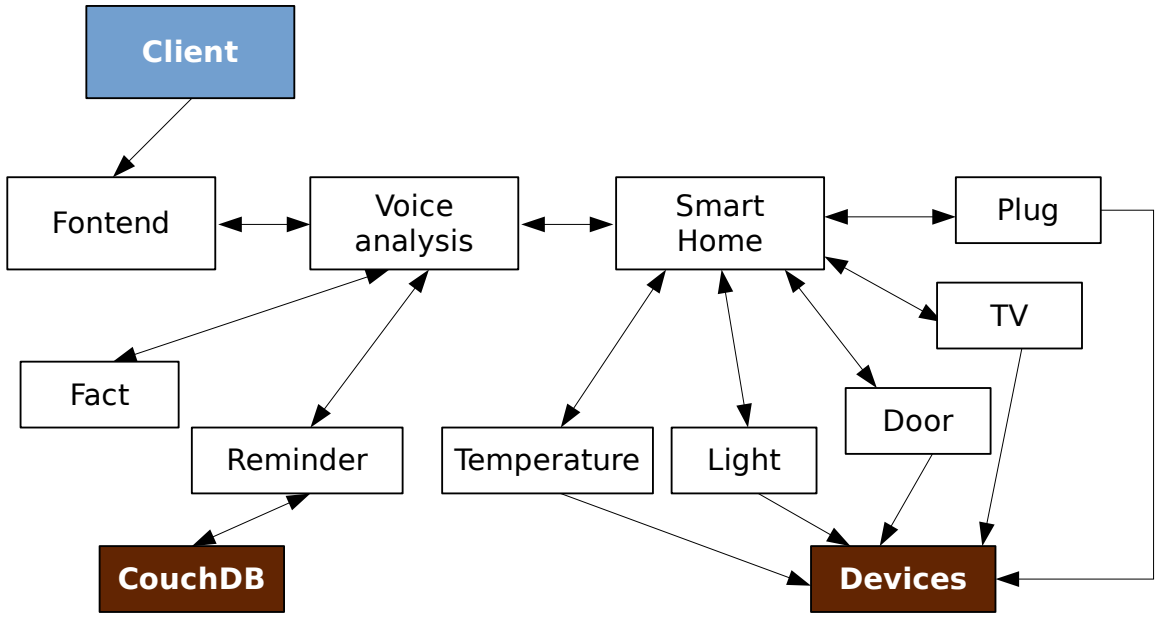
(2) IO-intensive benchmarks. Both disk-intensive and network-intensive benchmarks in Figure 5.2(c) and (d) show similar performance trends to their Node.js counterparts in Figure 5.1(c) and (d). That is because IO latency is the dominant factor in performance.

(3) Summary. Figure 5.2(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 so that the effect of pre-JIT is significant in Python. Another interesting point is that I/O performance is similar between Python and Node.js. It means that the I/O performance depends on the isolation level of the serverless platform rather than the language or runtime.

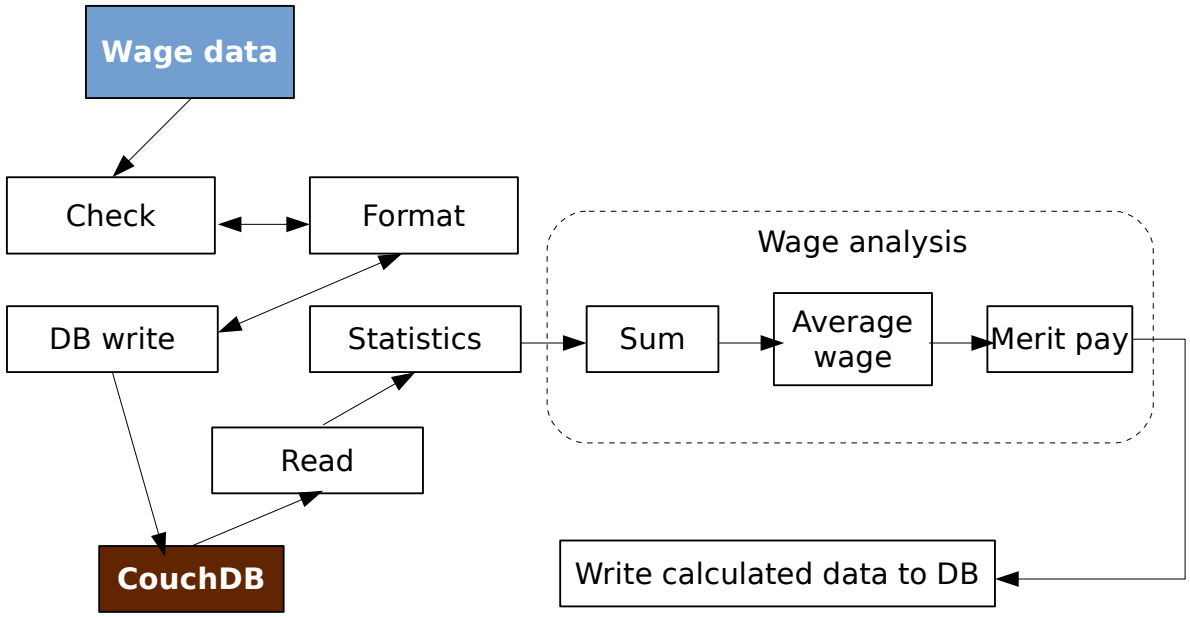
5.3 Real-World Serverless Applications

We choose two real-world applications in ServerelessBench [53] because these two are 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 the gVisor and Firecracker for the real-world workload because they cannot support the chain of functions and we compare evaluation results OpenWhisk. Figure 5.3 shows the flow of the real-world applications, where a node represents a servereless function running on a microVM and an edge represents a function invokation.

(1) Alexa Skills. Alexa skills [3] supports three skills: fact, reminder, and smarthome. It performs voice analysis using input text provided by the user and identifies the request.



(a) Alexa Skills (nested)



(b) Data Analysis (sequence + nested)

Figure 5.3: Real-World serverless applications.

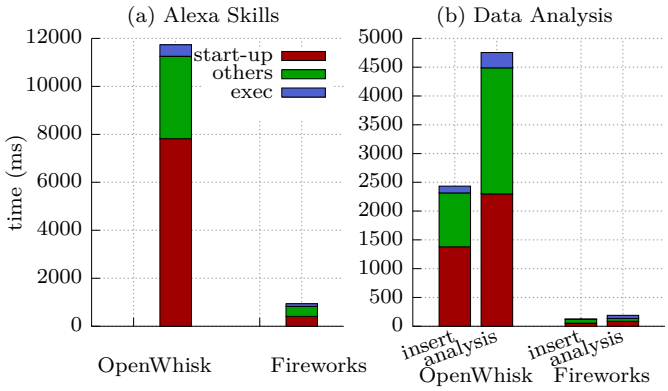


Figure 5.4: Execution time comparison of real-world applications.

The fact skill answers simple common sense, and the reminder skill searches or enters a schedule into CouchDB [7], a NoSQL database. The data inserted to CouchDB has item, place, and related URL fields. The smarthome skill notifies the on/off status of each device (*e.g.*, light, door, and TV) to the user. Figure 5.4(a) shows the performance comparison between Fireworks and OpenWhisk and its breakdown. For the evalution, we sent requests to the Alexa Skills asking for a simple fact, then checking the schedule through the reminder and checking the on/of status of home appliances through the smarthome. Fireworks shows 12.5× faster start-up time and 2.4× faster execution times 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 inserting the data to CouchDB. The data entered in the CouchDB consists of a name, ID, role, and base payment. The analysis function chain in the dashed box of Figure 5.3(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.

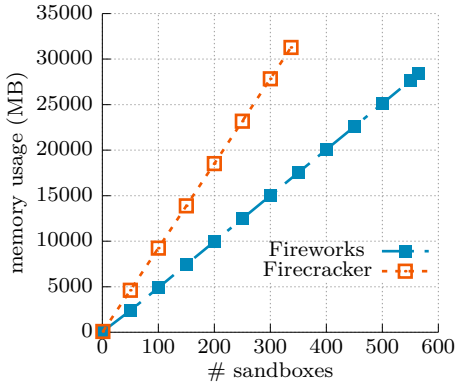


Figure 5.5: Memory usage comparison between Fireworks and Firecracker.

5.4 Memory Usage

Fireworks shares the memory snapshot 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 Proportional Set Size (PSS) in Linux using smem [28]. PSS is a physical memory consumption considering memory sharing; for the memory shared by N processes, PSS accounts 1/N to each process. Figure 5.5 shows the memory usage of Firecracker and Fireworks as microVMs increase. 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 happens to see the maximum possible consolidation. Fireworks could launch 565 microVMs, but Firecracker can only launch 337 microVMs before swapping happens. It means that Fireworks allows to consolidate 167% more sandboxes than Firecrackera since the amount of memory shared by JIT and OS snapshot increases.

5.5 Factor Analysis

We analyze how our design choices affect performance (§5.5.1) and memory consumption (§5.5.2). We separately investigate the impact of OS snapshot – creating a VM snapshot after a guest OS finishes to boot – and JIT snapshot – creating a VM snapshot after loading a serverless function and finishing JIT compilation of the function (Fireworks).

5.5.1 Performance

We conduct the factor analysis to analyze the performance improvement of each optimization we proposed as shown in Figure 5.6. We start from the original version of Firecracker.

+ OS snapshot. OS snapshot improves the performance of serverless functions by eliminating the OS boot time. For computation-intensive workload written in Node.js, adding the OS snapshot improves the performance 2.3×. Also, the network-intensive workload written in both Node.js and Python shows up to 6.1× faster performance.

+ JIT snapshot. The JIT snapshot improves the performance of computation-intensive benchmarks written in Python because the pre-JITted code significantly reduces the runtime overhead of Python interpreter. Also, it shows performance improvements for the file I/O intensive benchmarks written in both Node.js and Python up to 2× because of improvements in start-up time and execution time.

5.5.2 Memory Usage

We also conduct the factor analysis to analyze the memory efficiency improvement of each optimization. We ran 10 microVMs and reported average memory usage of one microVM

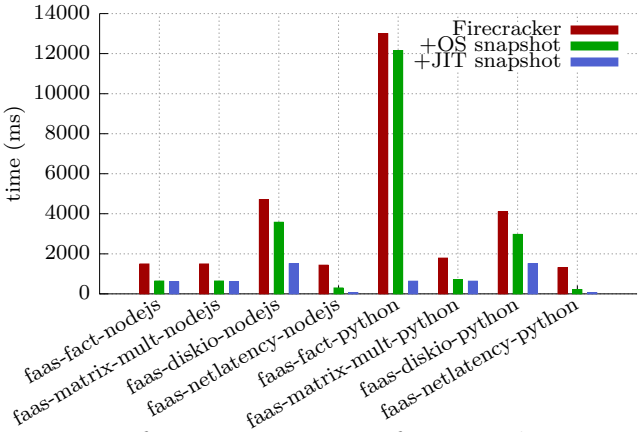


Figure 5.6: Performance impact of Fireworks optimizations.

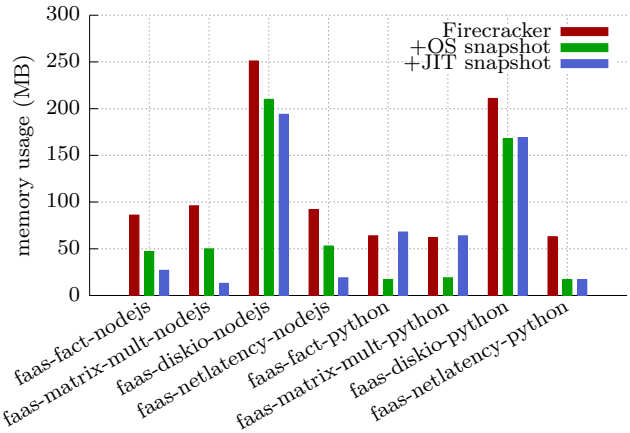


Figure 5.7: Memory saving impact of Fireworks optimizations.

in Figure 5.7. We start from the original version of Firecracker.

+ OS snapshot. All applications written in both Node.js and Python show memory efficiency improvements. OS snapshot shows that up to 73% of memory utilization improvements by sharing the resources.

+ JIT snapshot. For the case of the applications written in Node.js, adding the pre-JITting reduces memory usage up to 74%. However, there is no significant improvement in memory usage for Python applications. That is because the Python JITted codes need more memory consumption. Even though Fireworks consumes a similar amount of memory, Fireworks gains dramatic performance improvements with similar memory consumption.

Chapter 6

Discussion and Limitation

Similar to other snapshot-based approaches [47], 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 the userspace. The RNG provides enough entry when the CPU is IvyBridge or newer Intel processors. For the address space layout randomization (ASLR), we can mitigate the issue by periodically re-generating the VM snapshot similar to proposed in REAP [47].

Chapter 7

Related Work

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

Characterizing FaaS workloads and benchmarks. Serverless computing is a new paradigm, so there is not enough information to understand the characteristics of serverless workloads. Thus, there are several previous studies that conducted reverse engineering to understand the commercial serverless computing platforms. Wang *et al.* [51] 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.* [42] show characteristics of serverless workloads on MS Azure serverless computing platforms. They also propose a resource management policy to leverage the serverless computing platforms based on their observations. FaaSProfiler [41] 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, sandbox, and runtime environment. Reuse mechanisms can be caching the resources or using checkpoint/restore-based snapshots. Replayable Execution [49] proposed to use checkpoint and restore of OS kernel to reduce the start-up time latency and resource for the serverless functions. HotTub [30] eliminates JVM overhead in a parallel processing environment by

keeping the warm-up pool of JVM runtime. It is appliable to serverless environments. Mohan *et al.* [36] reduce cold start latency by pre-creating the network and attaching the network to the new function containers. The study for reusing the runtime is related to our works. SOCK [39] 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 a package-aware caching to minimize the startup latency. SAND [2] uses application-level sandboxing operation to fast-cap function and low latency of function interactions. Checkpoint/restore for instantiation is also one of the main directions to optimize the start-up latency. REAP [47] mitigate performance bottleneck from page faults when it loads the function's snapshot from disk to memory. It proposes proactively configuring the memory working set before loading a snapshot to reduce the page faults.

Another direction to optimize the serverless platform is to optimize the sandbox. SEUSS [11] leverages unikernel snapshot stacks for the execution of serverless functions. LightVM [33] propose lightweight VM through unikernel. Another approach is to place the sandbox nearby the data storage. Shredder [54] implements a sandbox near the data storage with V8:Isolate.

The main difference between Fireworks and previous works is that we pre-JIT the source codes. Our pre-JITted codes can efficiently reduce not only the cold start latency but also execution time for functions.

Extending serverless computing. Since most serverless platforms support stateless serverless functions only, it is hard to extend the applications which are disk I/O intensive to the serverless application. So, a few of previous works [40, 45] to extend serverless applications to support stateful serverless functions. Cloudburst [45] presents a stateful FaaS that can guarantee the mutable state and communication using Anna Key-Value store [52]. Anna provides autoscaling and logical disaggregation of compute node and storage node to

support the stateful function efficiently. Starling [40] is a query engine for leveraging the serverless computing platform. 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 [43] is an application that runs linear algebra operations as serverless functions on top of disaggregated storage. Excamera [17] is an application that leverages cloud functions for video processing. Excamera shows efficient video processing using the flexible use of thousands of functions. Pocket [27] designs the serverless platform to satisfy scalable demands. It automatically scales to support the applications and determines storage tier to provide high-performance with low cost. Infinicache [48] 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 [44] introduces an atomic fault tolerance shim between serverless functions and storage engines. It leverages transaction concepts for serverless functions to provide fault tolerance. Shredder[54] suggests the storage function that allows computation to be performed in a storage node with a low-latency.

Chapter 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 of serverless platform: (1) high-isolation level, (2) high performance, and (3) high memory efficiency. We show how our VM-level pre-JIT snapshot efficiently reduces memory usage without compromising the isolation level and increasing the security risk. We evaluate Fireworks against the state-of-art, including gVisor, OpenWhisk, Firecracker using two representative benchmark suites: FaaSdom and ServerlessBench. In our evaluation, Fireworks significantly outperforms the state-of-art serverless platforms, and it shows high memory efficiency with the guarantee of high isolation level.

Bibliography

[1] 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.

[2] 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.

[3] Amazon. Build An Alexa Fact Skill, 2020. https://github.com/alexa/skill-sample-nodejs-fact, visited 2021-05-18.

[4] Amazon. AWS Lambda Pricing, 2021. https://aws.amazon.com/lambda/pricing, visited 2021-05-18.

[5] Anaconda. Numba: A High Performance Python Compiler, 2018. http://numba.pydata.org/, visited 2021-05-18.

[6] 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.

[7] Apache. CouchDB, 2021. https://couchdb.apache.org/, visited 2021-05-18.

[8] Apache OpenWhisk. Apache OpenWhisk Documentation, 2016. https://openwhisk.apache.org/documentation.html#deploy_kubernetes, visited 2021-05-18.

[9] Apache OpenWhisk. Open Source Serverless Cloud Platform, 2016. https://openwhisk.apache.org/, visited 2021-05-18.

[10] AWS. AWS Lambda, 2021. https://aws.amazon.com/lambda/, visited 2021-05-18.

[11] 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.

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

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

[14] 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 *ASPLOS: Architectural Support for Programming Languages and Operating Systems*, pages 467–481. ACM, 2020.

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

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

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Marc Gendron. Ermetic Reports Nearly 80% of Companies Experienced a Cloud Data Breach in Past 18 Months, 2020. https://www.businesswire.com/news/home/20200603005175/en/Ermetic-Reports-80-Companies-Experienced-Cloud-Data, visited 2021-05-18.

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

[36] 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.

[37] NetApp Cloud Insights Team. Top Cloud Security Breaches and How to Protect Your Organization, 2021. https://cloud.netapp.com/blog/cis-blg-top-cloud-security-breaches-and-how-to-protect-your-organization#H_H3, visited 2021-05-18.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

[54] 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.