FIREWORKS: A Fast, Efficient and Safe Serverless Framework

Shin Wonseok

Master's Thesis Defense

VIRGINIA TECH™

Here is Serverless Computing!

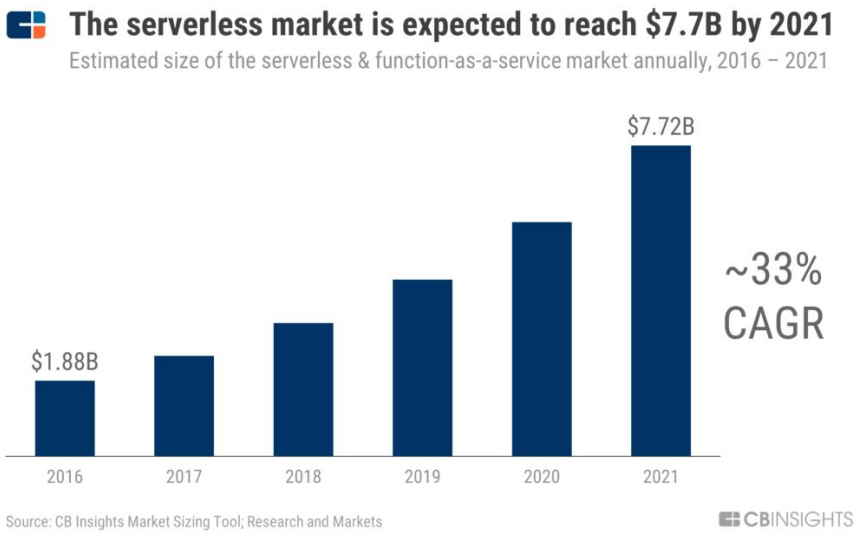
Serverless Computing is rapidly growing

The serverless introduces pay-as-you-go

Developer do not need to effort into administration

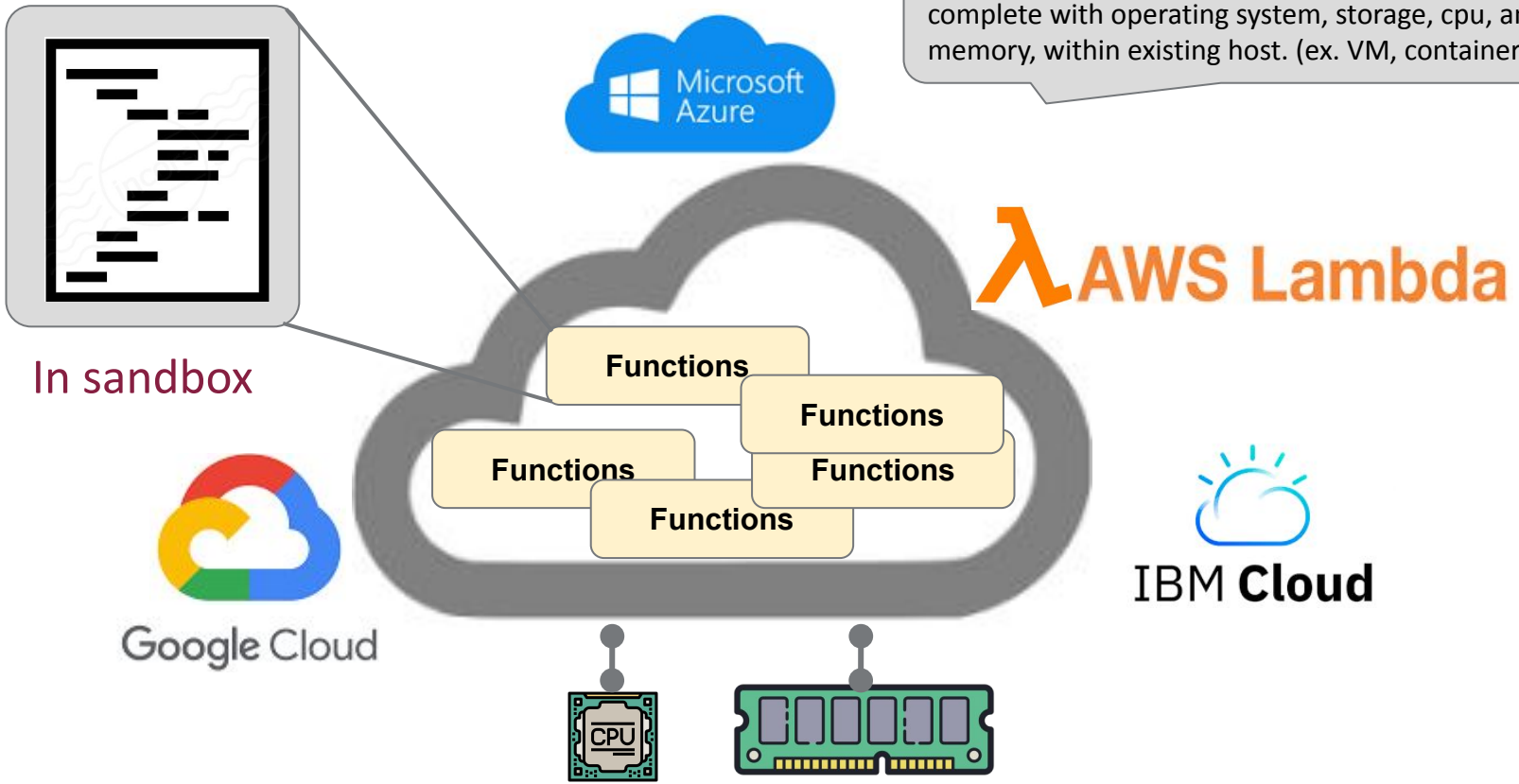
It offers significant scalability for the resource provisioning

Amazon Lambda, Microsoft Azure, Google Cloud, IBM Cloud Functions provide their own serverless computing models

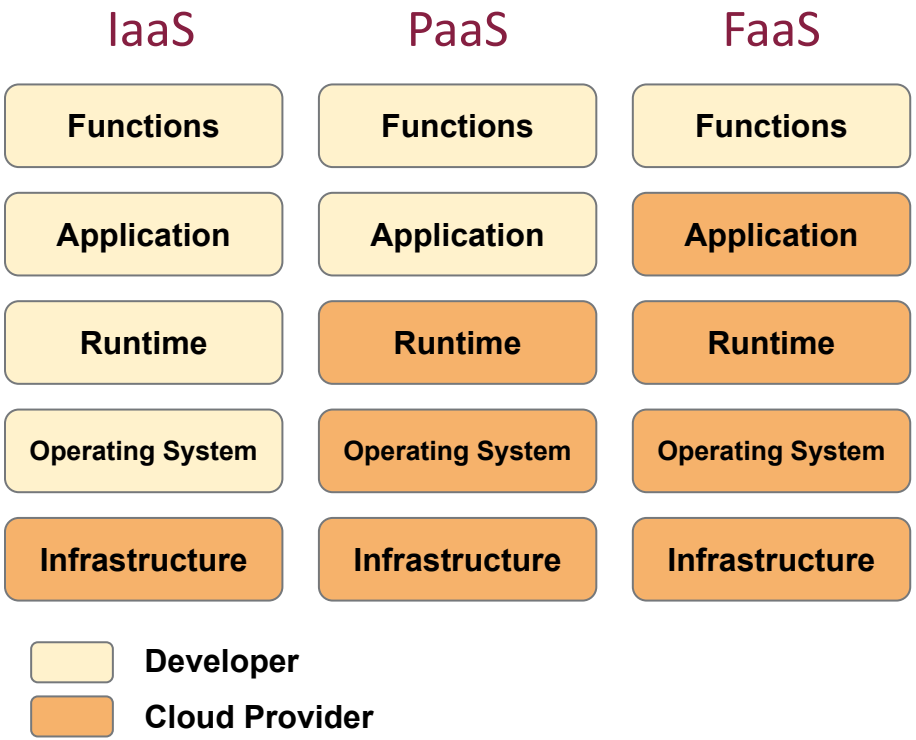


What is the serverless computing?

The **sandbox** creates an entire virtual environment, complete with operating system, storage, cpu, and memory, within existing host. (ex. VM, container)



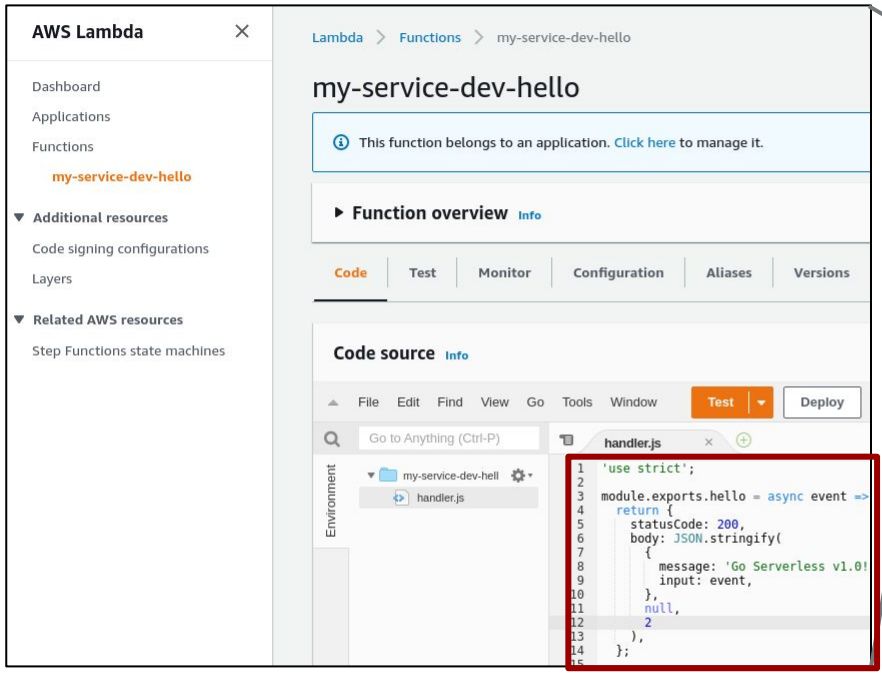
What is the serverless computing?

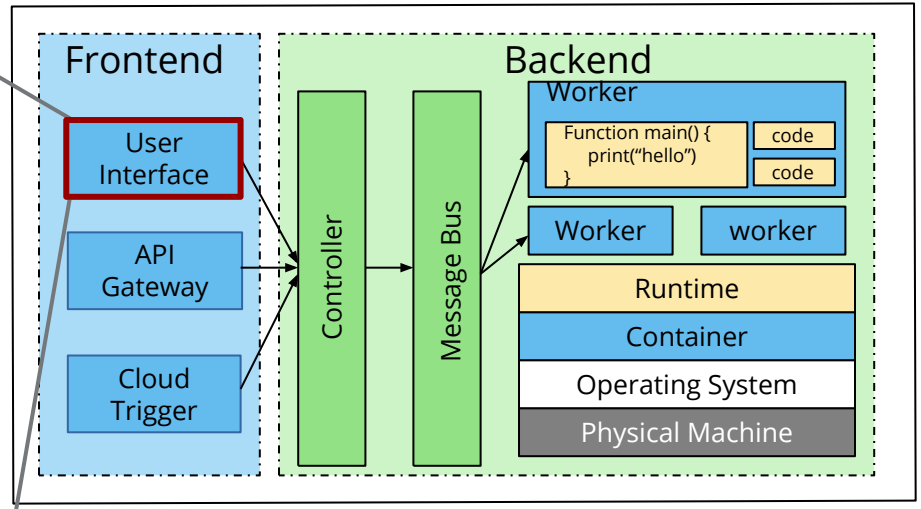


Serverless is
Function as a Service (FaaS)

What is the serverless architecture?

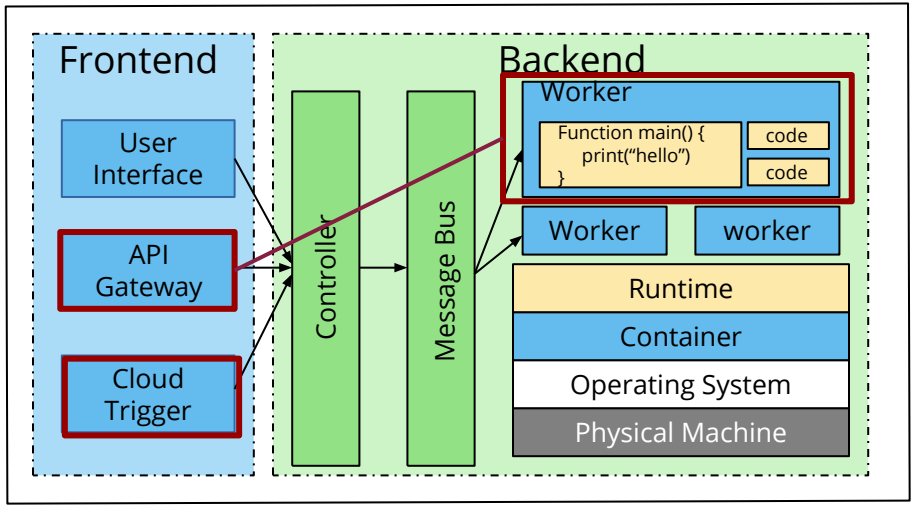
- **User interface**
  - Users write and upload the code.





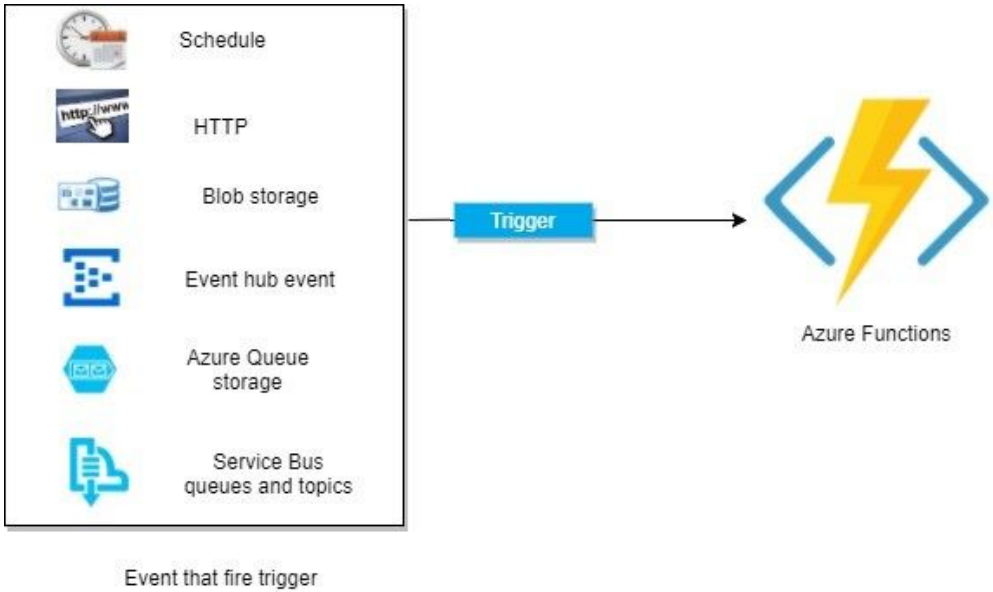
What is the serverless architecture?

- **API Gateway**
  - Several API (a REST API-type HTTP endpoint) can be created and connected to the application in the backend.
- **Cloud trigger(next slide)**
  - Provided by Cloud provider
- **Worker**
  - Run the user code



What the serverless is different from the traditional Cloud?

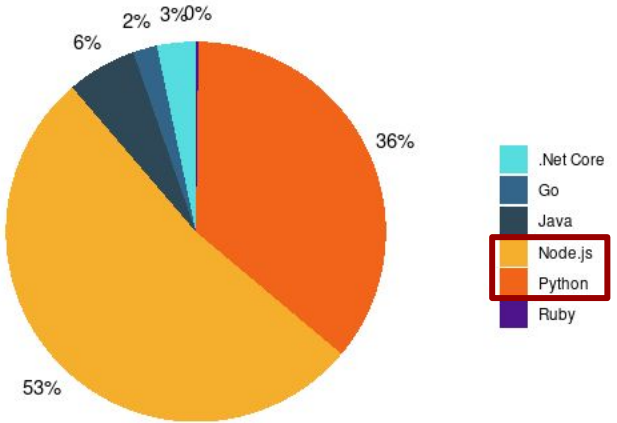
- **Triggers** invoke Functions in response to several event types such as HTTP, event, queues



- **Cold start** occurs when the function is call first, the resources for the function is not ready before
- Function **execution time** (avg. 1 sec) is short when compared to the waiting time of the resources
- It is **hard to predict** when a function will be invoked
- Cloud providers **aim to consolidate** thousands of functions in a server

What the serverless is different from the traditional Cloud?

- **Most used languages and runtime**
  - About 90% of total is interpreter language.
  - **Nodejs accounts for 53% and Python accounts for 36%.**
  - Interpreter languages use **Just-in-time(JIT) compilation**, which translate user code into machine code during execution, in improve the performance
  - However, in serverless, JIT is a known bad practice degrading performance because JIT compilation time does not pay off due to the short function running time.

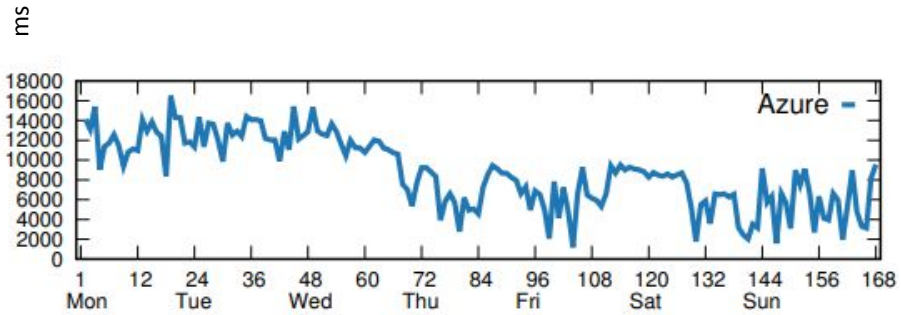


* serverless benchmark report AWS lambda 2020

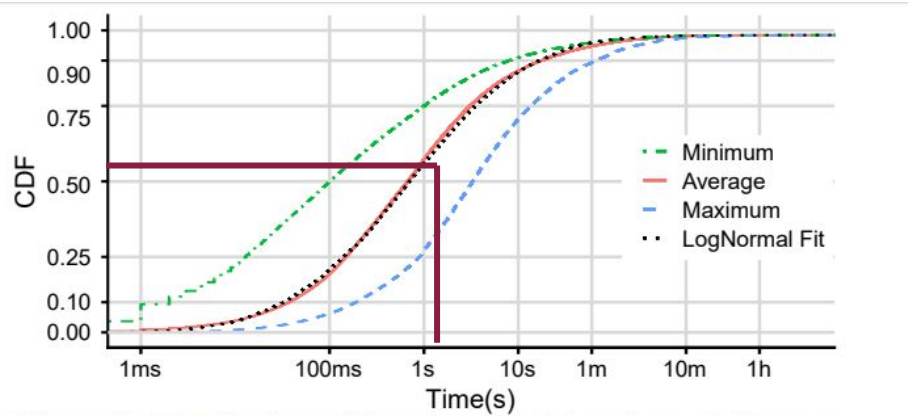
Unique Problems in the serverless computing

(1) Long start up time penalty & Not optimized execution time

- booting time of VM, OS, and containers and loading runtime
- 50% of functions take 1 second or less than 1 second



Median cold start up time per hour over 7 days (2017)[*1]



Distribution of function execution time[*2]

*1 [ATC18] Peeking Behind the Curtains of Serverless Platforms

*2 [ATC20] Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

Unique Problems in the serverless computing

(2) Waste memory

- Cloud providers use a fixed “keep-alive” policy that retains the resources in memory for 10 and 20 minutes after a function execution.*

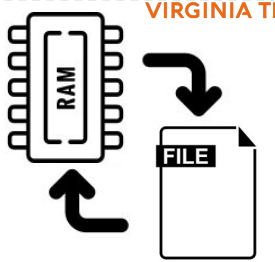
(3) Low isolation in sandbox

- Using lighter sandboxes approaches have a lower isolation level for improving the start-up time.

* Cold Starts in AWS Lambda, Cold Starts in Azure Functions. Mikhail Shilkov.

State-of-the-art approaches

snapshot:



These approaches compromise one of the three important aspects

Serverless Platform	Isolation	Performance	Memory Efficiency
Firecracker (Amazon)	VM(↑)	lightweight VM (-)	snapshot(↑)
gVisor (Google)	container(-)	container(-)	snapshot(↑)
OpenWhisk(IBM)	container(-)	container(-)	timer for hot function(↓)
Cloudflare	runtime(↓)	high(↑)	share process env(↑)
Catalyzer [ASPLOS20]	container(-)	high(↑)	snapshot(↑)
Our approach **FIREWORKS**	**VM(↑)**	**extreme(↟)**	**OS + JIT Snapshot(↟)**

↟:extreme, ↑ : high, - : medium, ↓:low

Our Approach :

Low-latency start up time and Optimized execution time → fast performance from the JITted codes

Efficient memory usage → less memory consumption from sharing the snapshot which includes JITted codes

Without compromising security level → Highest level isolation from VM level snapshot

Our Approach :

Low-latency start up time and Optimized execution time

Efficient memory usage

Without compromising security level

VM-level pre-JIT Snapshot

Executive summary

Unique Problems in the serverless computing

- **Long start up time, not optimized execution time, waste memory, low isolated sandbox**

State-of-the-art approaches can not achieve

- **the three important aspects: Isolation, Performance, Memory Efficiency**

VM level pre-JIT snapshot can achieve

- **High Isolation Level, High Performance, Memory efficiency**

Designed & implemented a new serverless platform, named FIREWORKS, based on OpenWhisk and Firecracker VM

- **20 time shorter (cold) startup time, 7 times lower memory footprint**

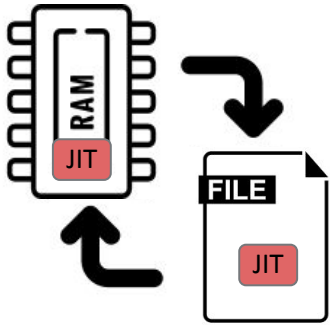
Outline

- **Design overview**
- Components
- Detail Design : Tasks
- Implementation
- Evaluation
- Conclusion

Key idea behind our design

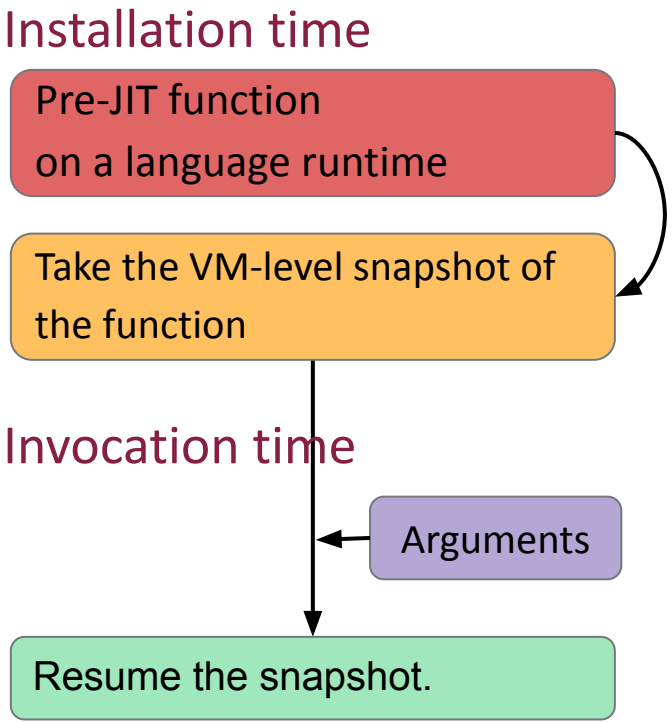
VM-level pre-JIT snapshot

- **Just-in-time (JIT) compilation** is a method to improve performance in interpreter languages by compiling user code into machine code during execution.



- JIT is usually suitable for **long-running** server applications due to reuse.
- In serverless, JIT is a known bad practice degrading performance because **JIT compilation time does not pay off due to the short function running time**.
- We **include the JIT in the snapshot** to **reuse**.

Design Overview

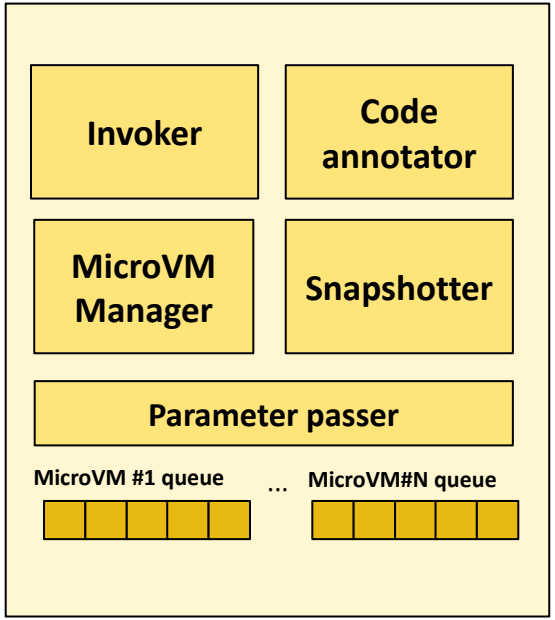


Outline

- Design overview
- **Components**
- Detail Design : Tasks
- Implementation
- Evaluation
- Conclusion

Components

- **Invoker** : receives the user's request, and transfer the request to other components.
- **Code annotator** : responsible for making JIT code by adding hints to user code.
- **MicroVM manager** : control MicroVM using Firecracker.
- **Snapshotter** : makes snapshot when serverless function request the snapshot.
- **Parameter passer** : help the main function of the snapshot get the parameters



Installation Phase

Phase

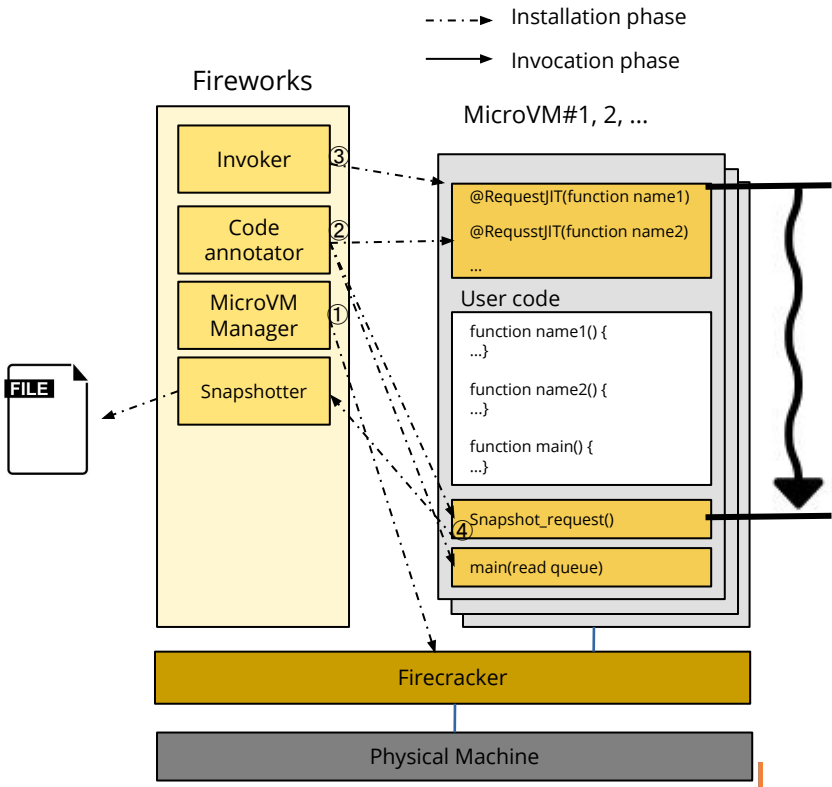
1 Installation phase

(1) FIREWORKS request to create microVM

(2) FIREWORKS gets and inputs the source code with automatic code annotation

(3) The annotated user code is executed

(4) The annotated function creates the pre-JITted snapshot



Invocation Phase

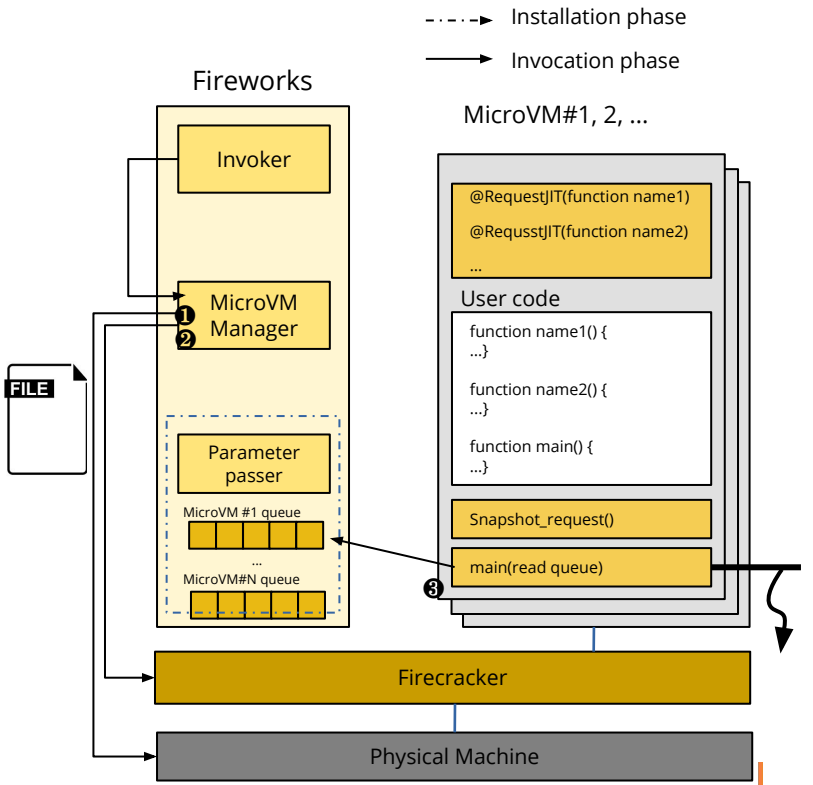
Phase

2 Invocation phase

(1) FIREWORKS setup a network for the function

(2) FIREWORKS restarts the VM snapshot

(3) The annotated function is resumed at the snap-shot point, and starts a regular entry with its parameters



How FIREWORKS meets the requirement?

Design Decision	Goal	Achieve
VM level snapshot	High Isolation Level	**Guarantee the high-level of isolation.**
Use microVM snapshot, and JIT the function before the snapshot	High Performance	**High performance in terms of both start up and execution of function.**
Sharing the snapshot among multiple instances of a serverless function	Memory Efficiency	**Sharing memory even Including JIT code**

Outline

- Design overview
- Components
- **Detail Design : Tasks**
- Implementation
- Evaluation
- Conclusion

Automatic Source Code Annotation

- Modern highly-optimized language runtimes already support annotations to trigger JIT at the program loading time.
- We re-purpose this feature to create a pre-JITted VM-level snapshot.

```
9  function optfn() {
10     try {
11         %OptimizeFunctionOnNextCall(main);
12     }
13     catch (e) {
14         console.log(e);
15     }
16 }
```

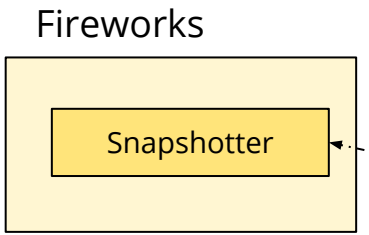
Node.js

```
@jit(cache=True)
def f(x, y):
    return x + y
```

Python

Creating a pre-JITted VM Snapshot

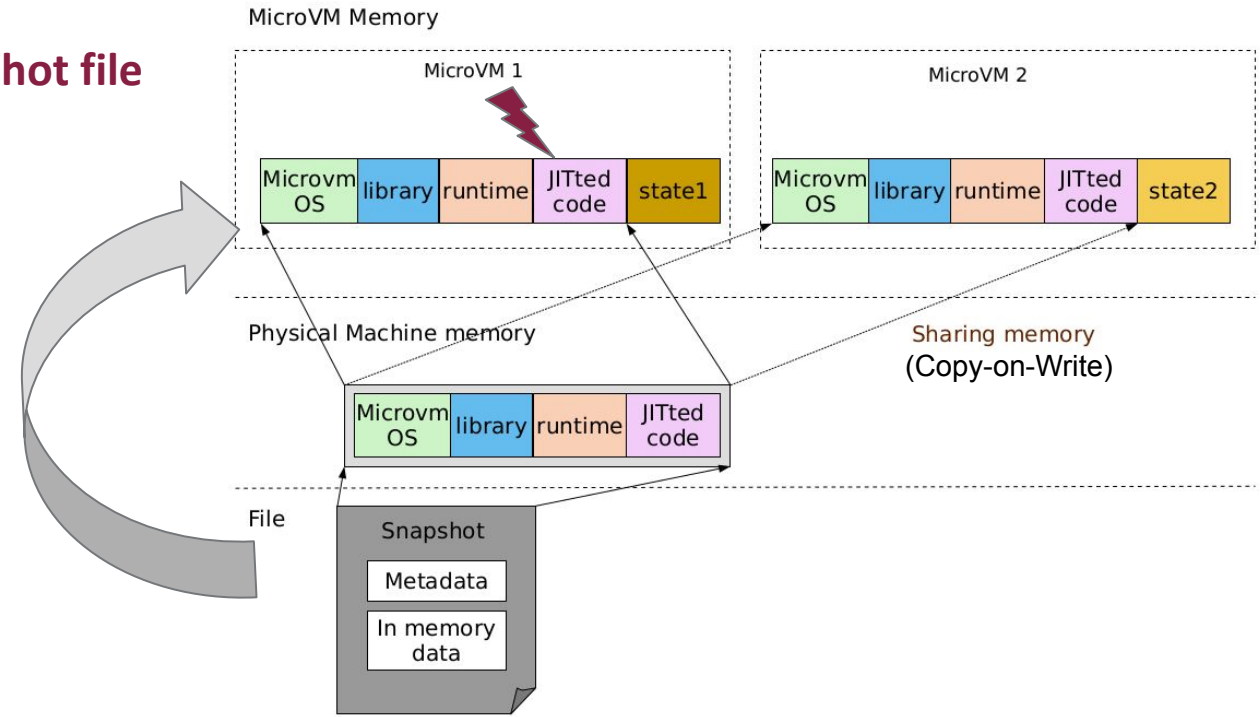
- The trigger of making snapshot is from user annotation code.




```
/* The snapshot function requests to snapshot the serverless
 * code itself to Fireworks */
async function snapshot() {
    const http = require("http");
    const options = {
        hostname: '172.17.0.1',
        port: 7777,
        path: `/?snapshot=Y&name=${actionName}&
            targetMicroVM=${microVM}`,
        method: 'GET'
    };
    const req = await http.request(options, (res));
    req.end();
}
```

Invoking a Serverless Function

The Invocation
== load the snapshot file



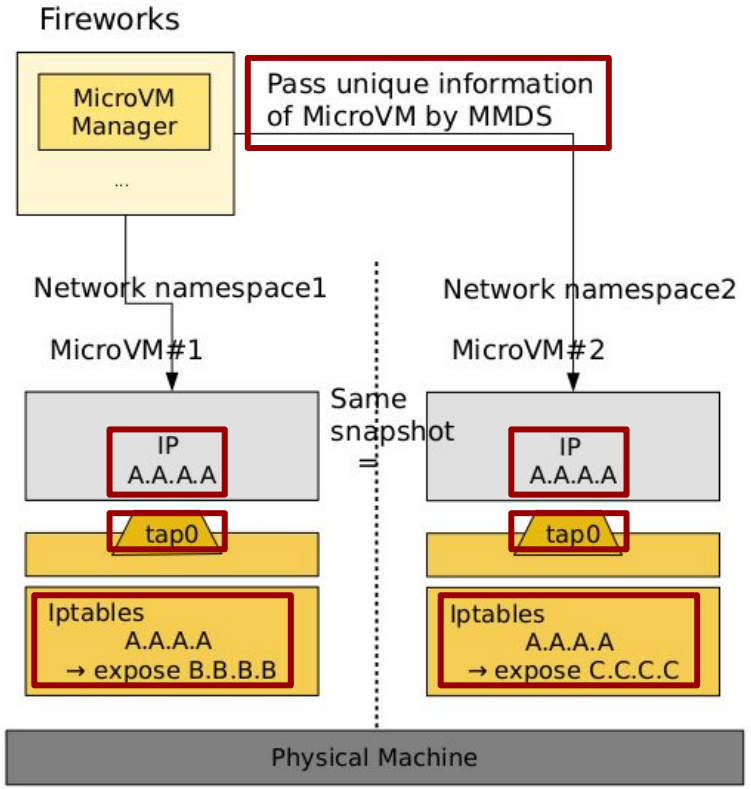
Enabling a Network Connectivity in a VM snapshot

- **conflict for the same network and the same device.**

  ⇒ own network namespace.

- **iptables NAT** map the same ip to different exposed IP

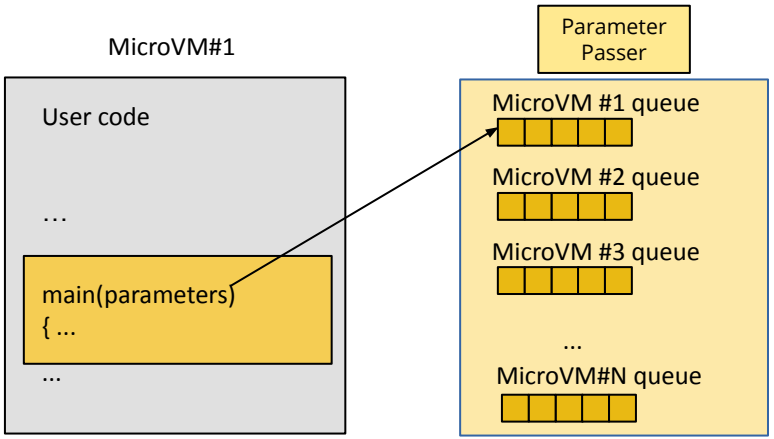
- **MicroVM metadata service (MMDS)** inserts unique information of the microVM



Taking Serverless Function Arguments

- Challenges
  - The function can not take arguments from the user
  - Resume is what is running stopped function in the memory

- FIREWORKS adds logic that checks the each microVM's repository(queue)




```
exec(`kafkacat -C -t topic${fcID}` => {
    params = JSON.parse(stdout);
    params['fcID'] = fcID;
    main(params);
});
```

Outline

- Design overview
- Components
- Detail Design : Tasks
- **Implementation**
- Evaluation
- Conclusion

Implementation

- **FIREWORKS Framework**
  - Firecracker v0.24.0
  - 3,000 lines of **Bash code for microVM manager, Invoker, Code annotator**
  - 500 lines of **Node.js, Python for Snapshotter, Parameter passer**
  - 40 lines of **C++ for Node.js V8 Source**
  - 17,480 lines of **Node.js, Python** borrowed code with modification from **FaasDom and SeverlessBench Benchmark**

Outline

- Design overview
- Components
- Detail Design : Tasks
- Implementation
- **Evaluation**
- Conclusion

Evaluation items

Cold and warm start latency with the VM-level snapshot

Execution time with preJIT

Memory usage with the OS + JIT snapshot

System Setup

- Intel(R) Xeon(R) Platinum 8180 CPU(2.50 GHz)
- 128GB DRAM, 2TB Local Storage
- The microVM has 1 vCPU, 512MB memory and 2GB of disk spaces
- **Openwhisk** v20.11 with kubernetes and **gVisor** runsc v1.0.2 with docker
- **Node.js** v12.18.3, **Python** v3.8.5

Evaluation Methodology

- **Compare** AWS Firecracker, Google gVisor, and IBM OpenWhisk
- **Measure** both cold boot latency and warm boot latency.
- The workloads of FaaSdom benchmark measures **basic performance**
- **Real-world serverless application performance** evaluation is measured by ServerlessBench

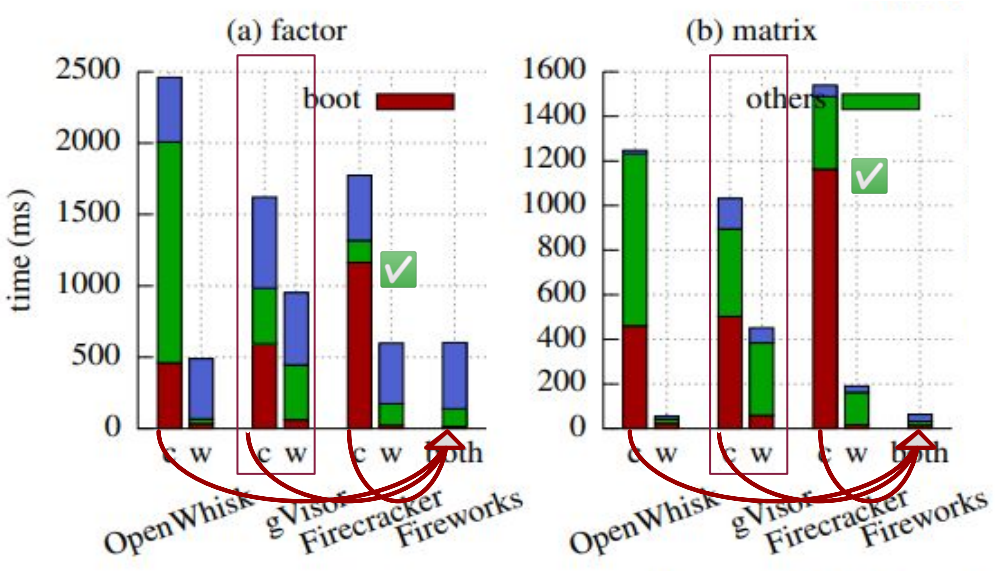
Benchmarks Overview

- **FaaSdom Benchmark** represents the performance of the application in terms of latency, CPU and file I/O
- **ServerlessBench** measures the real-world serverless application performance

Test Name	Test Type	Description	Language
FaaSdom: faas-netlatency	Micro benchmark	Network-bound test that immediately returns upon invocation	Node.js, Python
FaaSdom: faas-fact	Micro benchmark	Factorize an integer	Node.js, Python
FaaSdom: faas-matrix-mult	Micro benchmark	Multiply large integer matrices	Node.js, Python
FaaSdom: faas-diskio	Micro benchmark	An IO-bound benchmark to evaluate the performance of a disk	Node.js, Python
ServerlessBench: Alexa Skills	Real-world benchmark	Apps run through Alexa AI device	Node.js
ServerlessBench: Data Analysis	Real-world benchmark	Store, analyze and statistics employees' wage	Node.js

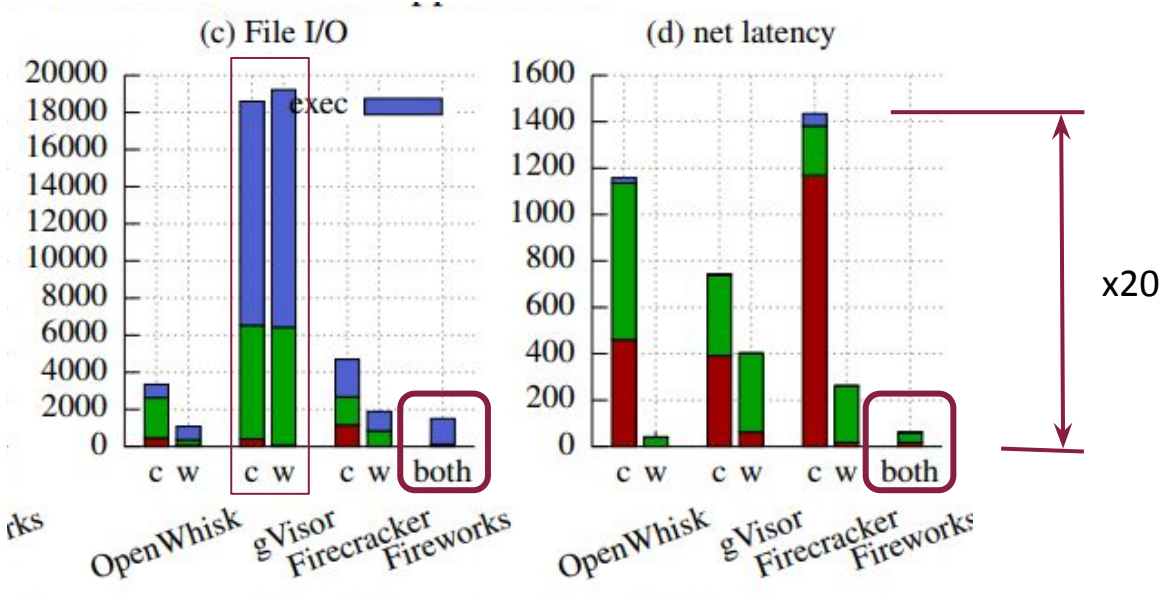
Microbenchmark

- **Latency Comparison of FaaSdom benchmarks : Node.js**
  - Compute intensive micro benchmark
    - OS snapshot includes cold boot
    - Firecracker shows a relatively faster boot time
    - gVisor's emulated kernel yields overhead in boot-up time and execution time



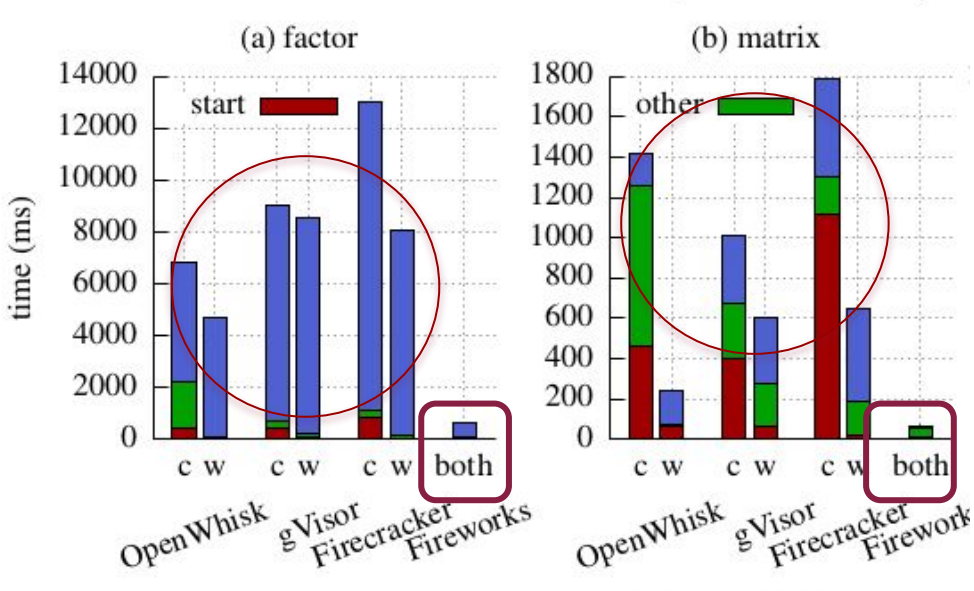
Microbenchmark

- **Latency Comparison of FaaSdom benchmarks : Node.js**
  - I/O intensive micro benchmark
    - the different isolation mechanisms have much influence on I/O performance
  - The Node.js performance improvement overall for the tests is up to 20 times.



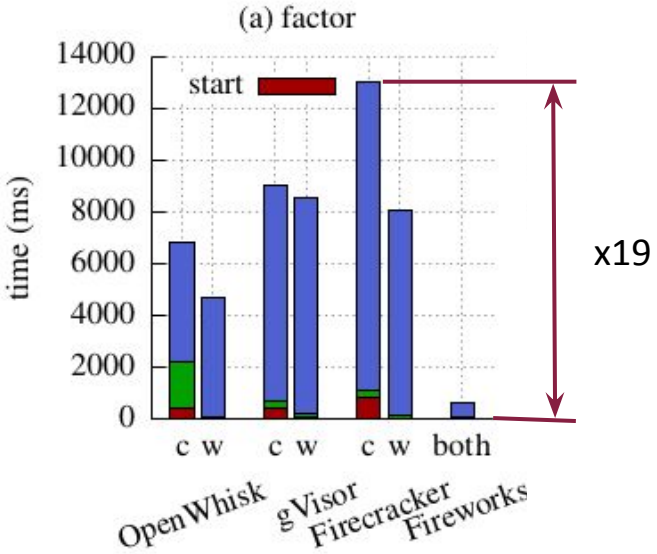
Microbenchmark

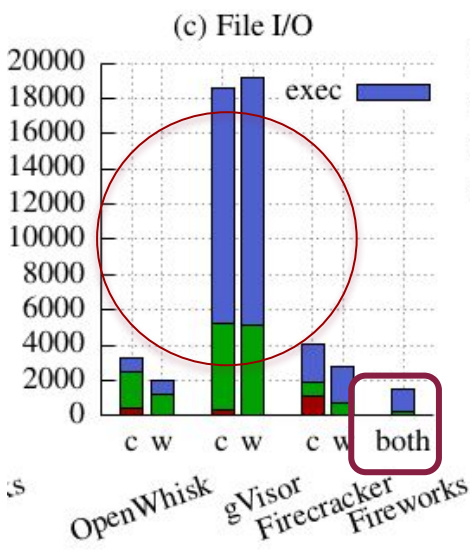
- **Latency Comparison of the FaaSdom benchmark : Python**
- The computation-intensive workload : (a), (b)
  - The execution time for the most serverless frameworks is high than that of Node.js
  - JIT dramatically reduces the execution time.



Microbenchmark

- **Latency Comparison of the FaaSdom benchmark : Python**
  - The I/O performance (c) depends on the isolation level of the serverless platform rather than the language or runtime
  - The Python performance improvement overall for the tests is up to 19 times.



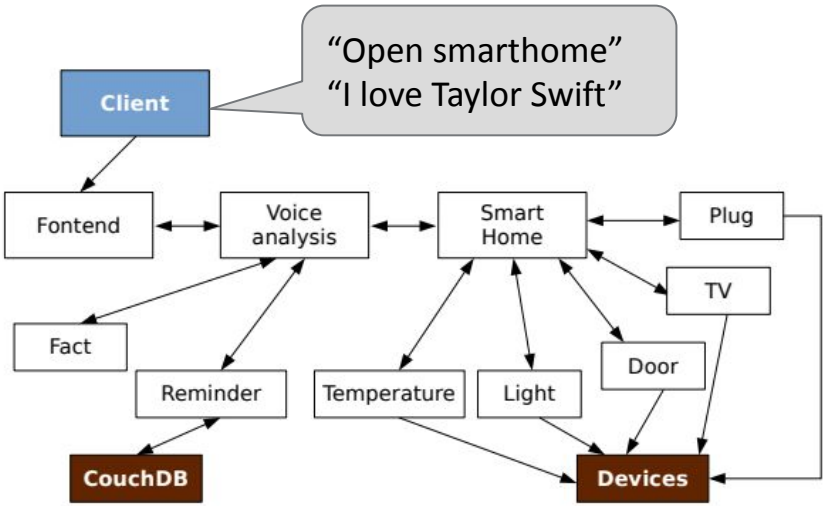


Real-World Serverless Applications

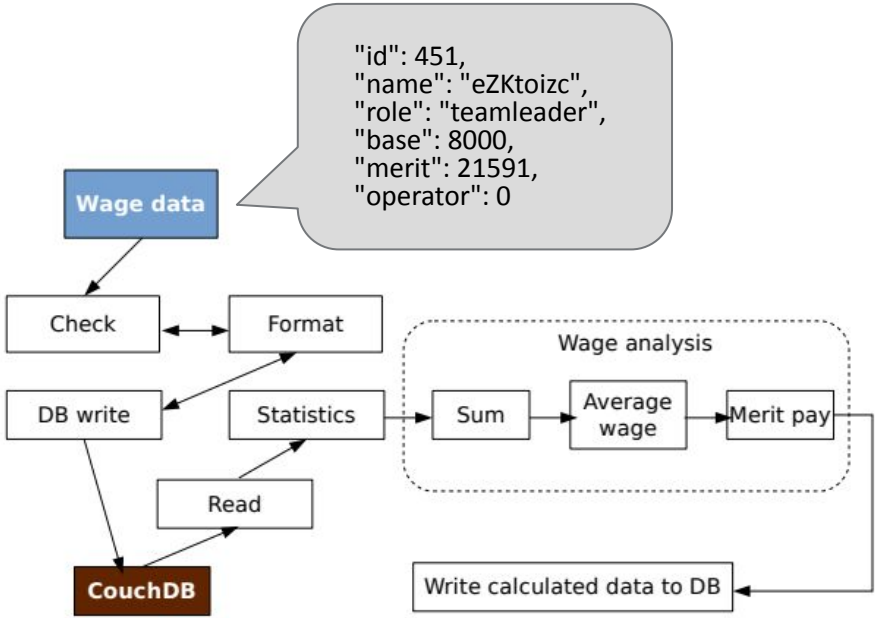
- **ServerlessBench**
  - **Real-world serverless application** consist of several functions.
  - These functions are **connected to each other** to deliver the output to the input.
  - The result will be stored to **third-party database** (e.g., CouchDB) to maintain it persistently.
  - Test these real world applications based on **OpenWhisk and FIREWORKS** to compare performance.

ServerlessBench

- **Alexa skills and Data analysis**



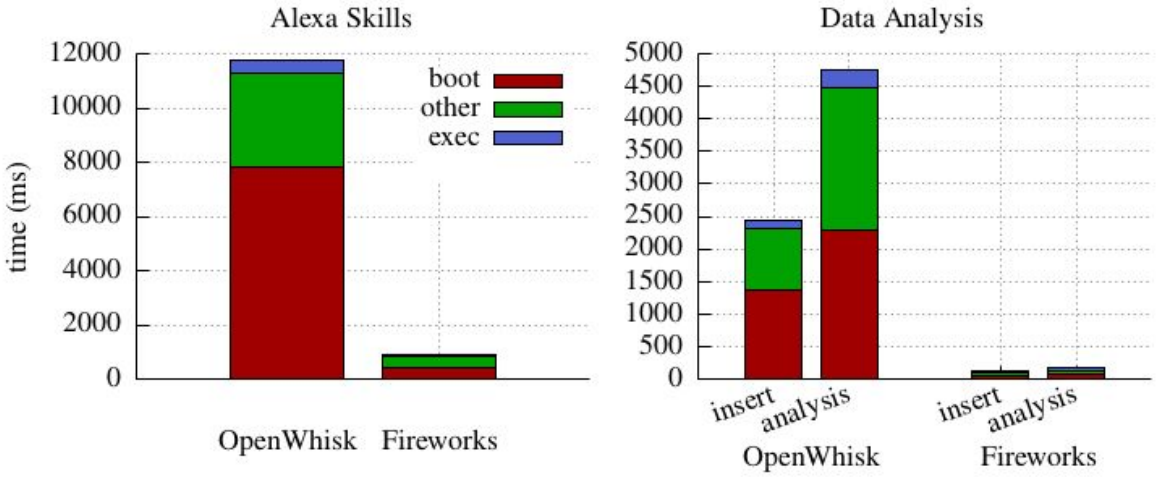
(a) Alexa skills (nested)



(b) Data analysis(sequence + nested)

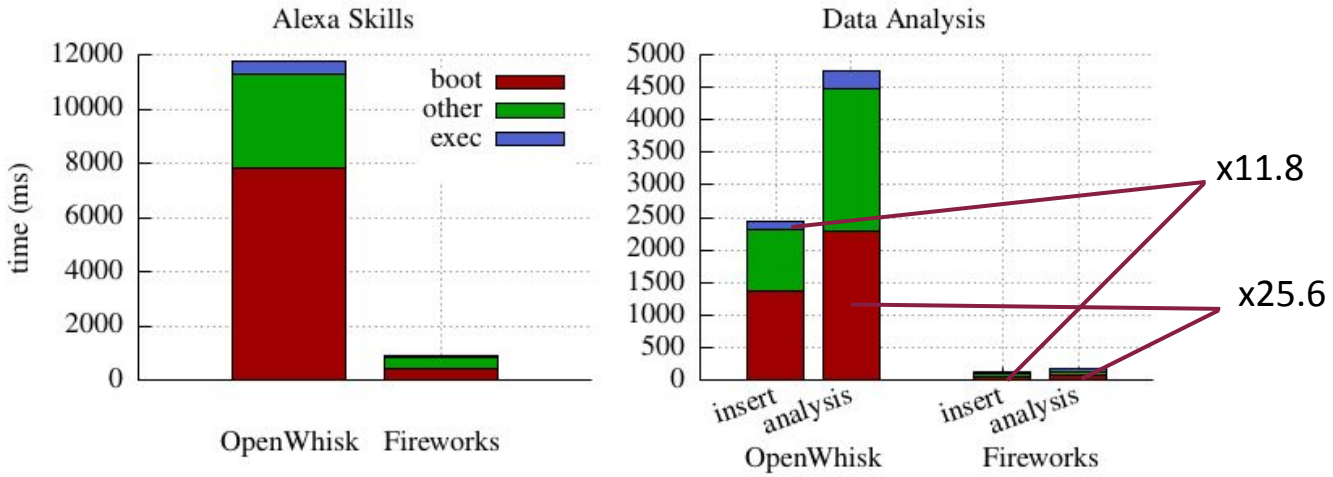
ServerlessBench

- **Latency Comparison of real-world application**
  - **Alex Skills** : asks for a simple fact, checks the schedule through the reminder, checks the on/off status of home appliances through the smarthome.
  - **Data Analysis** : input 1000 employees wage information, the data is analyzed to get the statistics.



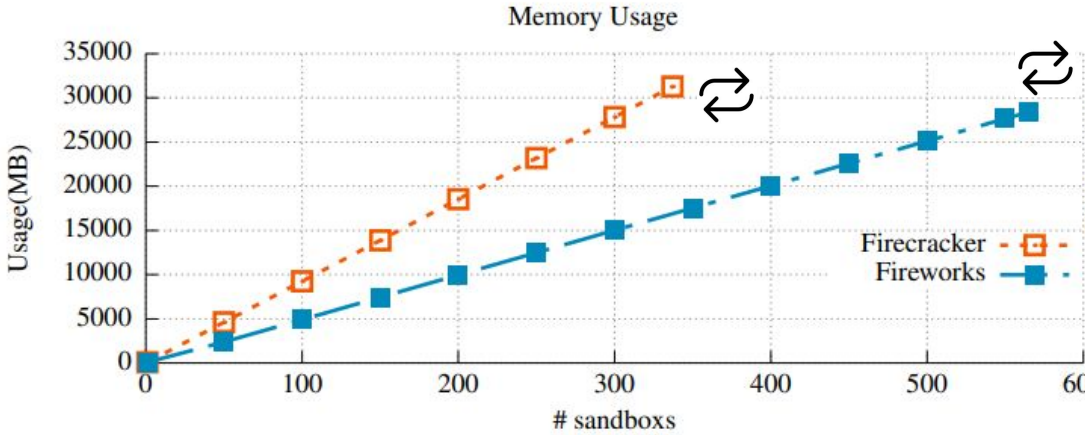
ServerlessBench

- **Latency Comparison of real-world application**
  - FIREWORKS shows faster cold start time and faster execution times in every case
  - FIREWORKS shows up to 25.6 times shorter cold start time and up to 11.8 times shorter execution time



ServerlessBench

- **Memory Usage**
  - FIREWORKS could make **565 microVMs**, and Firecracker can make **337 microVMs** without the swap memory. It allows FIREWORKS to consolidate **167%** more sandbox than Firecracker.
  - Reducing memory usage helps us consolidate more sandboxes, which means **resource efficiency** and **more profits** from the cloud provider's perspective.



ServerlessBench

- **Factor Analysis**
  - **Performance**
    - **+ OS snapshot**. adding OS snapshot improve the performance 2.3~6.1 times
    - **+ JIT snapshot.** The JIT snapshot improves the performance 2 times
  - **Memory**
    - **+ OS snapshot**. It shows that upto 73% of memory usage are reduced
    - **+ JIT snapshot.** Adding JIT snapshot reduces memory usage upto 74%

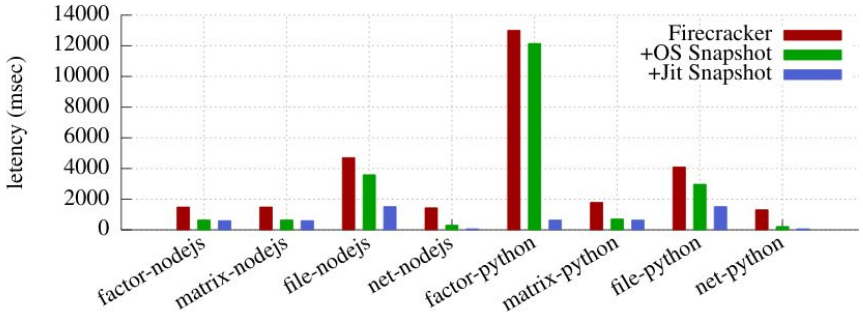


Figure 11: Performance impact of FIREWORKS optimizations

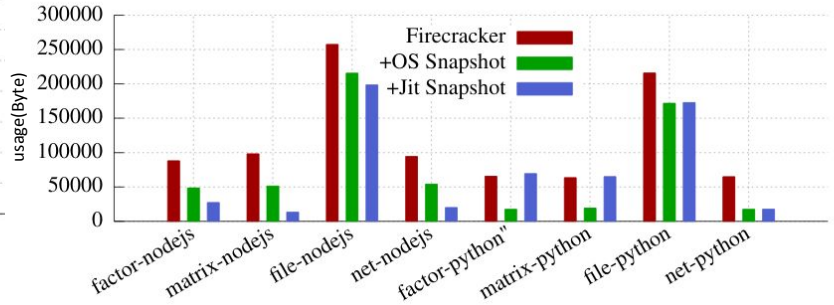


Figure 12: Memory saving impact of FIREWORKS optimizations

Outline

- Design overview
- Components
- Detail Design : Tasks
- Implementation
- Evaluation
- **Conclusion**

Conclusion

- A **safe, efficient, high performance** serverless framework does not yet exist
- **FIREWORKS** can get **all three:**

  high-isolation level, high performance, and high memory efficiency.

  pre-JIT-based approach efficiently reduces memory usage without compromising the isolation level.

- We **designed**, **implemented**, and **evaluated new serverless framework** by using VM-level snapshots and JIT-based snapshots

  20 time shorter (cold) startup time, 7 times lower memory footprint

  The achievements give a ***guidance*** to utilize **JIT** and **Snapshot** in the serverless wild.

We will plan to submit the full paper to 2021 ACM Symposium on Cloud Computing(due date : May/28/2021)

FIREWORKS: A Fast, Efficient and Safe Framework

Shin wonseok
shinws@vt.edu

Thank you!