Evaluating Concurrent Executions of Multiple Function-as-a-Service Runtimes with MicroVM

Jungae Park
Dept. of Computer Science
Kookmin University
Seoul, South Korea
barkjungae@kookmin.ac.kr

Hyunjune Kim
Dept. of Computer Science
Kookmin University
Seoul, South Korea
4u_olion@kookmin.ac.kr

Kyungyong Lee
Dept. of Computer Science
Kookmin University
Seoul, South Korea
leeky@kookmin.ac.kr

***Abstract*—Serverless computing and public Function-as-a-Service (FaaS) systems are gaining significant attention because they help easily build a highly available system. With recent advances in micro virtual machines (microVM), the internal architecture of FaaS systems substantially changes. This paper focuses on a thorough investigation of the recent improvement in public FaaS systems concerning numerous concurrent executions. The adoption of microVM has changed the nature of FaaS, especially for runtime reservations. As a result, the performance degradation has decreased significantly compared to the previous generation FaaS, as shown in the experiment.**

I. Introduction

Cloud computing provides great flexibility and elasticity in deploying and operating highly available applications. In a classical cloud computing model, Infrastructure-as-a-Service (IaaS), users must design a system architecture to ensure scalability and fault-tolerance by using cloud-native services, such as auto-scaling and load-balancing. Next to the IaaS, container-based technology, such as Kubernetes [11], provides scalability and fault-resiliency with few configurations. In the serverless computing, cloud service providers guarantee scaling out or in underlying resources as the number of requests changes. To handle resource failure, public serverless computing service providers embed fault-tolerance mechanisms for reliable services.

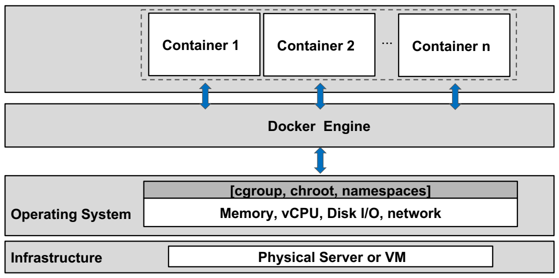
Serverless computing becomes feasible through using various fully-managed services, such as Amazon S3 for object storage, API Gateway for a managed HTTP endpoint, SQS for message queuing, and SNS for message notification. Using fully managed services, public cloud vendors abstract complex infrastructure management, necessary software stack installation, instance scaling, and replication so that users can focus on core application development. However, the high level abstraction of fully managed services lacks customization capability, and users must rely on functionalities provided by vendors. To grant flexibility in developing applications using serverless computing, FaaS allows users to write a custom code that is executed on an environment fully managed by cloud service providers.

The FaaS is developing quickly, and many public cloud vendors provide this service, such as Amazon Web Service (AWS) Lambda, Google Cloud Functions, and Microsoft Azure Functions, expediting the adoption of serverless computing. Despite its simple interface and fully managed execution environment, the applications of FaaS are limited to simple tasks, for instance, embarrassingly parallel jobs that do not need communications among function runtimes, functions compositions, and cloud service orchestrations [4]. One of the reasons that wide adoption of FaaS hindered is the inconsistent performance of the CPU and the disk and network I/O performance especially when many functions are requested concurrently in a short period [13], [6], [9]. Performance degradation in such cases happens due to the interference among the function runtimes, because cloud service vendors execute function runtimes on the same host as much as possible to maximize the server utilization and minimize cost.

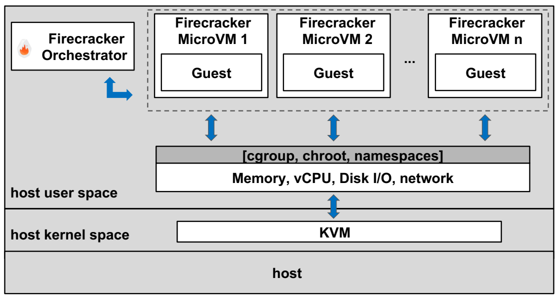
To further increase resource utilization and lessen security concerns in FaaS, microVMs have been developed by multiple FaaS vendors, such as AWS Firecracker [1] and Google gVisor [14]. By adopting a microVM, the performance characteristics of the FaaS can change, and it is crucial to understand the traits of contemporary FaaS systems. We conducted thorough evaluations of AWS Lambda and Google Cloud Functions under diverse scenarios and discovered that the interference among function runtimes reduces significantly when a user submits numerous functions. Improved performance is more noticeable for heavy disk and network I/O functions. In comparison to the previous work [13], [6], [9] that had evaluated various FaaS systems, the disk and network I/O performance with concurrent tasks improves significantly when the number of concurrent function invocations increases. We are sure that the recently stabilized FaaS concurrent execution performance helps to maintain stable service and to adopt FaaS and serverless computing for more diverse data oriented applications.

II. Evolution of Serverless Computing and FaaS

Cloud computing services have evolved in the direction of hiding complexity in maintaining the high availability



(a) Container internal



(b) microVM internal

Figure 1: FaaS internal - container and microVM technology

of applications. The first-generation cloud computing service model, IaaS, uses load-balancing and auto-scaling to guarantee fault-tolerance and scalability, respectively. They greatly help to develop and maintain highly available systems, but the proper service setting for the scaling policy, network, and security can be very challenging, even for system experts. To mitigate the burdens of architecting and maintaining highly available cloud systems, public service vendors provide fully managed services that have a higher level of resource abstraction while embedding inherent high-availability features.

A. Accelerating Serverless Adoption with FaaS

The FaaS is a fully managed cloud service that provides various programming language runtime environments, such as Python, Java, C#, JavaScript, Ruby, and Go. Public cloud service vendors provide FaaS while integrating the service with other fully managed cloud services.

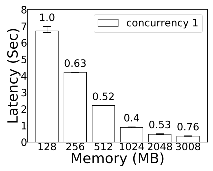
FaaS has different resource allocation and cost calculation mechanisms from IaaS. In the FaaS runtime setup, users set the maximum memory size that a function runtime can use. The CPU is allocated proportionally to the maximum memory size. Usage costs are calculated in a very fine-grained manner. For instance, AWS Lambda is billed every 100 milliseconds proportional to the allocated maximum memory size.

1) FaaS Runtime Internal: Container technology provides process-level isolation by using various Linux kernel features. Although containers of the same machine share a kernel with a host operating system, the *chroot* kernel feature lets a container to have its own exclusive file system. The *cgroup* feature isolates the CPU, memory, disk, and network I/O resources among containers. The *namespace* feature allocates the exclusive process ID tree per each container. The architecture of the container technology is illustrated in Figure 1a. Using container technology, Docker [10] provides easy-to-use interfaces and a novel image-management mechanism. The improvements have made Docker widely adopted in experimental and production environments. In addition to Docker's basic container-management features, Kubernetes [11] provides container-cluster management services that support load-balancing and auto-scaling for high availability, and many industrial applications have adopted Kubernetes in production environments.

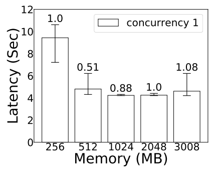
In the beginning of FaaS, the container technology was adopted to provide function runtime environments so that a runtime can be started and stopped promptly because the container has a lower management overhead than the virtualization technique [3]. Despite the light overhead of containers, sharing a kernel among containers in a host machine can cause security vulnerabilities and unexpected interruptions [12]. To mitigate security concerns, public FaaS providers designate a host machine for each account to execute functions exclusively [13]. However, such resource scheduling can result in inefficient resource usage when an account does not execute numerous functions simultaneously. Even worse, it is likely that the same function owned by an account can be invoked simultaneously on the same host that is assigned to an account. It can result in severe interruptions among function invocations because invocations of the same function necessitate accessing the same resources.

Kim et al. [8], [6] thoroughly investigated the performance of FaaS when multiple functions are invoked simultaneously. They found that, even under the same runtime configuration, simultaneous invocations of 10s of functions can result in about three times more time to download a file from a shared object storage service through a network compared to a single invocation case. For an application that performs disk I/O operations, concurrent function executions result in about six times performance degradation compared to a single function invocation. They concluded that the exclusive VM instance scheduling to enhance the security of FaaS resulted in performance degradation.

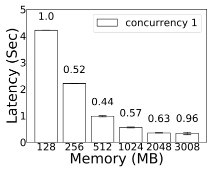
To better isolate function runtimes, public cloud service vendors have developed microVMs, including Amazon Firecracker [1] and Google gVisor [14], that enhances the security of function runtimes. To enhance security, Firecracker has adopted a VM Monitor (VMM) based on Kernel-based Virtual Machine (KVM). Using virtualization is expected to lower the security vulnerability more than the container-based approach. It is built with minimal device emulation



(a) CPU Intensive

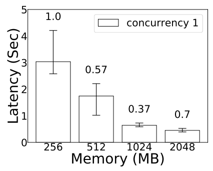


(b) Network Intensive

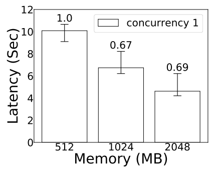


(c) Disk intensive

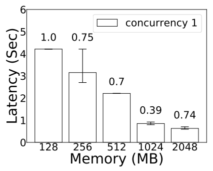
Figure 2: AWS Lambda latency to complete various tasks with different memory size configured



(a) CPU Intensive



(b) Network Intensive



(c) Disk intensive

Figure 3: Google Cloud Functions latency to complete various tasks with different memory size configured

to enhance the start-up time of a VM and reduce memory overhead [1]. The detailed architecture of Firecracker [1] is depicted in Figure 1b.

III. Evaluation

In this section, we evaluate the performance of recent FaaS systems to see whether the adoption of microVM has resulted in the performance improvement over the previous container-based approach. AWS and Google have publicly announced using microVM for FaaS. However, other major cloud service vendors, such as Microsoft Azure and IBM cloud, did not make related announcements, and we performed the experiments with AWS and Google cloud.

A. *Performance of Recent Public FaaS Systems*

We first evaluate the up-to-date performance of the publicly available FaaS: AWS Lambda and Google Cloud Functions. As described in Section II-A, the adoption of the microVM for function runtime changes the characteristics of FaaS significantly, especially for runtime scheduling. We chose AWS and Google Cloud for evaluation because they recently announced using microVMs. We are primarily interested in the performance of the recent FaaS using microVMs, and we aim to compare the current characteristics with those of the pre-microVM by referencing [13], [9], [8], [6].

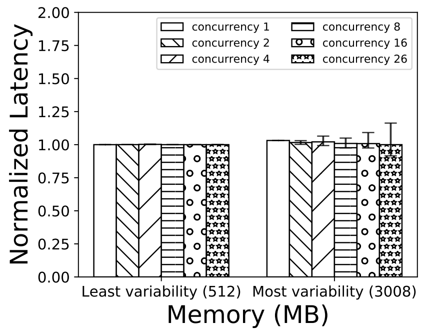
1) Performance with Different Memory Configuration: Figures 2 and 3 reveal the latency of the running CPU, network, and disk I/O intensive workloads under different memory configurations. In the experiments, we used the workloads suggested in FunctionBench [7], [5]. In FunctionBench, we used *Linkpack* for the CPU intensive workload, *object storage service downloading/uploading 100MB file* to measure the network performance, and the *dd* workload to measure the disk I/O performance. In the experiments, we configured the function memory size from 128MB to the maximally available size, that is, AWS Lambda at 3008MB and Google Cloud Functions at 2048MB, doubling it for each experiment. The missing bars in the figure indicate that the given workload cannot be completed with the configured memory size.

In each memory configuration, a single function (concurrency being one) is invoked six times, and we removed the latency of the first invocation to remove the effect of the cold-start. The cold-start is an important issue that needs to be considered to develop applications with FaaS. However, recent advancements from public vendors provide a way to overcome the cold-start problem. For example, using provisioned concurrency of AWS Lambda [2] prepares a set of pre-warmed function runtimes, and cold-start problem can be mitigated. Given this situation, we focus on evaluating FaaS under warm-start scenarios. In the figures, we show the median value in the bar with minimum and maximum latency in the error bars.

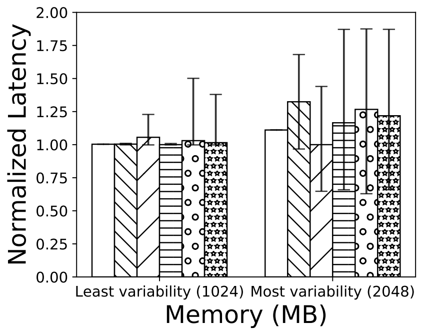
In each figure, the horizontal axis indicates the configured memory size, and the vertical axis indicates the latency in seconds. The FaaS of both AWS and Google Cloud claims that the resources are allocated proportionally to the configured memory size. On top of the bars in each figure, we list the ratio of the latency to the half memory size configuration latency. The number indicates whether the increase in the memory size results in a proportional performance gain. Ideally, the value should be inversely proportional to the memory size. For instance, an increase in the memory size from 128MB to 256MB should result in half the latency, and the value on a bar should be 0.5 if it has an exactly proportional latency reduction with the increased memory size. In Figure 2a, the value when Memory size being 256MB is 0.63, which means that the latency reduction is not as good as the increase in the configured memory size.

For the CPU intensive workloads (Figure 2a and Figure 3a), we confirm that the performance is proportional to the configured memory size, as presented in the previous work [13], [9]. However, the performance of heavy network workloads is not proportional to the configured memory size, especially for AWS Lambda. From this experimental result, we conclude that the network resource is not allocated proportionally to the configured memory size and that users should be cautious when running heavy network applications with FaaS, especially data-intensive applications.

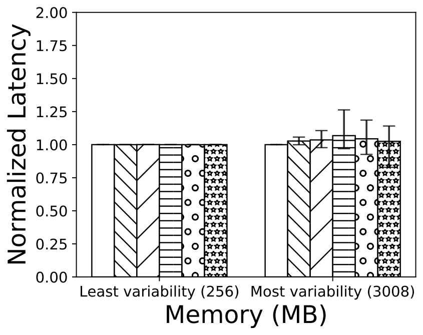
2) FaaS Performance with Different Level of Concurrent Executions: We evaluate the performance of FaaS under different degrees of concurrent executions. Figures 4 and 5 illustrate the results from AWS Lambda and Google Cloud Functions, respectively. Each figure has three sub-



(a) CPU Intensive

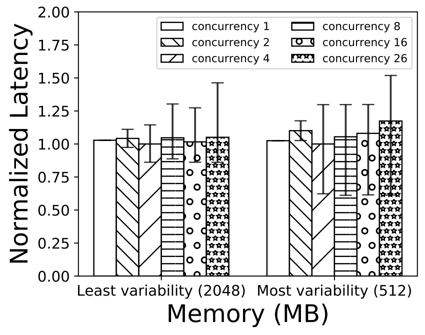


(b) Network Intensive

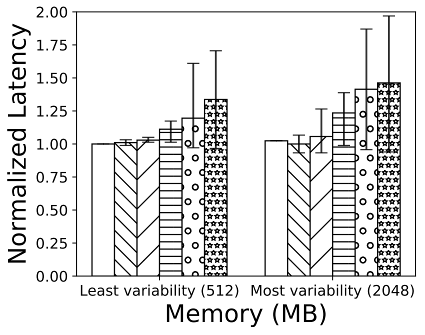


(c) Disk Intensive

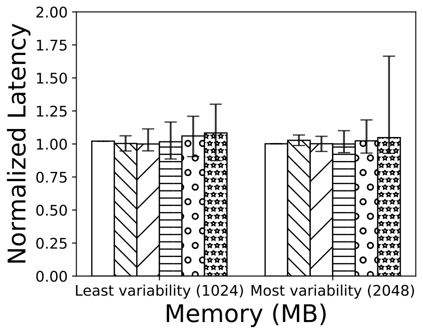
Figure 4: Normalized latency of AWS Lambda under different level of concurrency execution



(a) CPU Intensive



(b) Network Intensive



(c) Disk Intensive

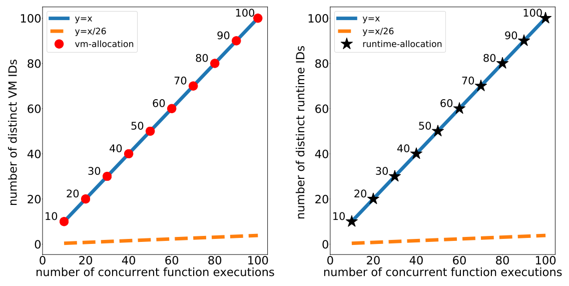
Figure 5: Normalized latency of Google Cloud Functions under different level of concurrency execution

figures that reveal the performance of the CPU, network, and disk I/O workloads under different numbers of concurrent executions. For workloads, we used the same scenarios that we used in Section III-A1. In the figures, the horizontal axis indicates different memory configurations, and the vertical axis shows the normalized latency. In the normalization step, we chose the fastest latency value among different concurrency invocations with the same memory configuration and calculated the relative value to the fastest latency. In each workload, we conducted experiments with differently configured memory sizes, and we demonstrated two cases of the experimental results: one with the least performance variance of different memory sizes (on the left) and another with the most variance (on the right).

The performance does not differ much as the concurrency level changes. This observation is contrary to the work presented by Wang et al. [13] and Kim et al. [6] presented. In their work, the performance degrades significantly as the the number of concurrent executions increases, especially for heavy disk and network I/O workloads. For heavy network workloads, the latency to download a file from a public object storage service increases more than three times as the number of concurrent functions increases from 1 to 26. For sequential disk I/O operation, the per function runtime bandwidth decreases more than six times. They concluded that packing many function runtimes into a single VM for each account to preserve security is the primary reason for such a contention.

Regardless of the workload type in Figures 4 and 5, the latency does not increase much, although the concurrency level increases. The most variability is exhibited with the Google Cloud Function Network workload (Figure 5b). However, even in the worst case, the latency increases about 50% on average, which is a significant improvement over the old generation FaaS. In summary, both AWS Lambda and Google Cloud Functions exhibited a marginal latency increase as the concurrency level increased. Comparing the AWS Lambda and Google Cloud Functions, AWS Lambda has better performance concerning the interference during concurrent function executions.

To further analyze the reason for improved concurrency execution performance, we investigated VM and the function runtime scheduling algorithm. Wang et. al. [13] proposed a mechanism to detect the VM ID of a function runtime that references the */proc/self/cgroup* file. The runtime for VM mapping is supported only in AWS Lambda. Using the



(a) Distinct VMs (b) Distinct runtimes

Figure 6: Normalized latency of Google Cloud Functions under different level of concurrency execution

method, we recorded VM IDs of the concurrent executions of the same function from 10 to 100. To improve the chances of locating a functions on the same VM higher, we set the function memory size at the minimum (i.e., 128MB).

Figure 6a illustrates the distribution of distinct VM IDs (in the y-axis) for different degrees of concurrent executions (in the x-axis). We added two reference lines. A solid line is the graph of $y = x$, which indicates that all functions are executed on different VMs. Another dotted line is the graph of $y = \frac{1}{26} \times x$, which indicates the maximum function runtimes (26) are packed into a single VM with a memory size of 128MB [13], [6]. The figure indicates that all functions are executed on different VMs. This is the primary reason that the resource contention from concurrent function executions significantly decreased. In the analysis results of the previous generation FaaS system [13], a VM is allocated to a user, and the function runtimes are packed into the VM as much as possible. The change in the function runtime to VM scheduling prevents functions with similar characteristics from running in the same VM and could lessen the overall interference among the function runtimes. Figure 6b displays the distinct number of runtime IDs and confirms that all functions are correctly executed on different runtimes.

IV. Conclusion and Future Work

Serverless computing and FaaS systems provide a new way to develop applications in cloud computing. Recently, public FaaS vendors have announced the development of a microVM that inherits the strong isolation of VMs with the minimal management overhead of containers. To understand the influence of adopting a microVM as the underlying FaaS runtime management mechanism, we thoroughly investigated performance of the public FaaS system, focusing on the interference among function runtimes. We could uncover great performance improvement that reverts claims made by previous publications.

In this paper, we aimed quantitative evaluation of serverless computing and public FaaS systems. Other than latency and scalability measurements, a fair cost evaluation is challenging. The research on the Quality of Service (QoS) of serverless computing and the cost model to maintain the desired QoS level is part of our ongoing work.

Acknowledgement

This work is supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) (No. NRF-2015R1A5A7037615), the ICT R&D program of IITP (2017-0-00396).

References

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

[2] A. Blog., "Aws lambda announces provisioned concurrency," last accessed April. 2020. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/

[3] W. Felter, A. Ferreira *et al.*, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[4] J. M. Hellerstein, J. M. Faleiro *et al.*, "Serverless computing: One step forward, two steps back," in *CIDR 2019*.

[5] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, pp. 502–504.

[6] ——, "I/o resource isolation of public cloud serverless function runtimes for data-intensive applications," *Cluster Computing*, 2020. [Online]. Available: https://doi.org/10.1007/s10586-020-03103-4

[7] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: ACM, 2019.

[8] J. Kim, J. Park, and K. Lee, "Network resource isolation in serverless cloud function service," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sep. 2019.

[9] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *IEEE CLOUD 2018*.

[10] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[11] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[12] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[13] L. Wang, M. Li *et al.*, "Peeking behind the curtains of serverless platforms," in *USENIX ATC 2018*.

[14] E. G. Young, P. Zhu *et al.*, "The true cost of containing: A gvisor case study," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/young