Virtualization and Cloud Computing Assignment Report

On

Scheduling Containers Rather Than Functions for Function-as-a-Service

Submitted by

212IS007: Chandan Kumar Sah
212IS016: Mohammad Asif Pir

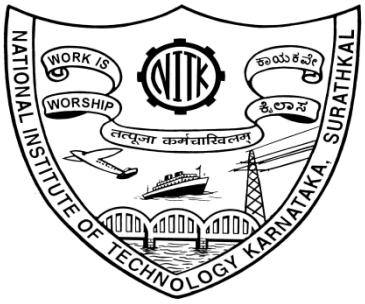
Under the Guidance of

Dr. Sourav Kanti Addya

Dept. of Computer Science,

NITK, Surathkal

Date of Submission: 19-05-2022



Department of Information Technology
National Institute of Technology Karnataka, Surathkal.

Abstract

Function-as-a-Service (FaaS) is a compelling technology that allows users to run functions in an event-driven way without concerns about server management. Containerbased virtualization enables functions to run in a lightweight and isolated run-time environment, but frequent function executions accompanied with container initialization (cold starts) make the platform busy and unresponsive. For performance sake, warm starts, which is to execute functions on already initialized containers, are encouraged, and thus FaaS platforms make efforts to schedule functions to warm containers. From our experience operating an on-premise FaaS platform, we found that the existing scheduler showed poor performance and unstable behavior against multi-tenant and highly concurrent workloads. This paper proposes a novel FaaS scheduling algorithm, named FPCSch, that schedules Function-PullingContainers instead of scheduling functions to containers. As FPCSch lets containers continuously pull functions of the same type, cold starts decrease dramatically. Our evaluations show that Apache OpenWhisk equipped with FPCSch has many desirable features for FaaS platforms; (1) quite stable throughput against the multi-tenant workloads mixed by the increasing numbers of function types, (2) growing throughput for increasing concurrency, (3) uniformly load-balancing resource-intensive workloads, and (4) nearly proportional performance for scale-out.

Declaration

I, the undersigned solemnly declare that the project report **Scheduling Containers Rather Than Functions for Function-as-a-Service** is based on my own work carried out during the course of our study under the supervision of **Dr. Sourav Kanti Addya**. I assert the statements made and conclusions drawn are an outcome of my research work. I further certify that work contained in the report is original and has been done by me under the general supervision of my supervisor. The work has not been submitted to any other Institution for any other degree/diploma/certificate in this university or any other University of India or abroad. We have followed the guidelines provided by the university in writing the report. Whenever we have used materials (data, theoretical analysis, and text) from other sources, we have given due credit to them in the text of the report and giving their details in the references.

Certificate

This is to certify that **Chandan Kumar Sah**, M.tech first year student of Computer Science and Engineering department has submitted the mini-project Report entitled **Scheduling Containers Rather Than Functions for Function-as-a-Service** as the course work of CS-852 in second semester of Mtech in Computer Science (Information Security) course at National Institute of Technology Karnataka, Surathkal in session 2021-2022.

Contents

1 Introduction

Serverless computing is emerging as a form of cloud computing that enables multiple tenants (users) to run billed applications without concerns about the operational logic of servers. It helps developers to focus on high-level abstractions, such as business logic, and the billed applications are comprehensive; i.e., from operating systems to functions. As the smallest one, functions can effectively deal with eventdriven requests used to implement their business logic. For this reason, Function-as-a-Service (FaaS) cloud platforms are getting the most popular form of serverless computing not only in public clouds, such as Amazon AWS Lambda, Google Cloud Function, and Microsoft Azure Functions, but also in open sources: for example, Apache OpenWhisk, Fission, OpenFaaS, Kubeless, Knative, and OpenLambda. While tenants of FaaS platforms only define functions, FaaS service providers serve with all the operational supports to run those functions. In FaaS tenants' point of view, the quality of FaaS service is determined by how quickly functions can be executed and how stably FaaS platforms can respond to varying workloads. Meanwhile, what the service providers want is to maximize the efficiency of available resources and to process as many requests as possible, satisfying the quality users demand. In addition, from our experience as an onpremise FaaS service provider operating Apache OpenWhisk, it is also highly desired to ensure predictable and stable performance against highly concurrent multi-tenant workloads and cluster node changes.

For executing functions a user submits, it is required to prepare an underlying environment, such as a particular version of an operating system, language-specific environment, library packages, various settings, and etc. Moreover, to provide strict isolation for supporting multi-tenancy, i.e. consisting of many different types of functions, FaaS platforms generally make use of virtualization (also known as sandbox) technologies.

To run any function within a container for the first time, it takes a long time to create a new container and to prepare its run-time environment, i.e., cold start. Many research works revealed that cold starts significantly degrade the performance of FaaS platforms. Shortening cold starts from containers or virtual machines(VMs) is an effective approach to breaking a bottleneck of FaaS platforms, but there is considerable room for reducing the occurrence of cold starts and promoting warm starts. Reducing cold starts and promoting warm starts can enhance not only the quality for tenants but also the efficiency of clusters, which can be embodied by distributed scheduling algorithms of FaaS platforms. In fact, the innate goal of scheduling is to uniformly distribute loads to given resources, but warm starts could be boosted by non-uniform scheduling. This contradiction has been substantiated by only primitive hash-based scheduling algorithms currently used by Apache OpenWhisk and OpenLambda. Our experiments show that the existing hash-based scheduler of OpenWhisk fails to guarantee stable performance for highly-concurrent and multitenant workloads. Nevertheless, the study on FaaS schedulers is still in the early stages in academia, and only one scheduler has been introduced yet, but no scheduling algorithms of proprietary FaaS platforms have been fully disclosed.

This paper presents a novel scheduling algorithm for FaaS platforms. As an on-premise FaaS service provider, we have gone through poor performance and unpredictable behavior against highly concurrent multi-tenant workloads, which motivated our scheduling algorithm. Our scheduling algorithm is characterized by the fact that it schedules FunctionPulling-Containers, hence the name "FPCSch". Instead of the scheduler scheduling functions to containers, we schedule containers, each of which pulls functions of the same type. This considerably accelerates warm starts and makes it easy to deploy containers uniformly in a FaaS cluster.

2 Literature Survey

Shortening startup times of containers or VMs is effective and necessary, but a microscopic approach to improving the performance of FaaS platforms. In other words, the aforementioned research works are difficult to give answers to the following macroscopic questions. (1) When high concurrent function requests exceeding the capacity of a single container are arriving, how should containers be provisioned in the FaaS cluster? (2) When multiple containers have already been provisioned, how should function requests be scheduled to minimize cold starts? (3) How should the already provisioned containers be removed so as to be efficient? In fact, scheduling or placement algorithms of FaaS platforms can be answers to these questions, but the proprietary FaaS platforms have hardly revealed their algorithms yet. Instead, there were few attempts to infer such algorithms by well-designed experiments as in [23], [41]. Recently, as the only study of the scheduling algorithms for FaaS platforms, package-aware scheduling has been introduced by a research group [30], [31]. In their latest work [31], a package-aware scheduling algorithm, called PASch, is proposed, and evaluated with the implementation of the push-based scheduler in OpenLambda. Beyond the inborn goal of any distributed scheduling algorithms, i.e, load balancing, PASch considered maximizing the cache affinity of packages as the more important goal. To maximize the cache affinity, it makes use of consistent hashing that distributes function requests to worker nodes by hashing the largest package as a key. As the largest packages tend to be repeated, this approach obviously causes hot-spots, i.e., load imbalance. A prerequisite of PASch is a solution to cache packages for programming languages such as Pipsqueak [29]. The effect of caching, therefore, might be available only for few languages. In addition, the hash-based scheduling is vulnerable to multitenant workloads as we will show in Section V, and PASch also depends on consistent hashing by using conflict-prone keys. Though no multi-tenant workload was evaluated in the papers [30], [31], it is reasonable to expect that PASch will produce similar results.

3 Motivation

Since the use of cloud computing and virtualization has been explained in the above section and we also know the scenario behind the evolution of cloud services above the traditional on-premises data centres. Now, as we have shifted to off-premises service, there are certain thing that we should been working on to get better results. Such as, efficiently use of server resources to maximize the resource utilization and server space. This research is purely based on the different techniques of containers Scheduling algorithm for FaaS and we motivated our scheduling algorithm. Our scheduling algorithm is characterized by the fact that it schedules Function-Pulling-Containers, hence the name "FPCSch". Instead of the scheduler scheduling functions to containers, we schedule containers, each of which pulls functions of the same type. This considerably accelerates warm starts and makes it easy to deploy containers uniformly in a FaaS cluster. The upcoming section will state the problems and proposed solutions.

4 Problem Statement

Function-as-a-Service (FaaS) is a representative of the serverless computing which runs the smallest execution unit, i.e., a function. Triggered by incoming requests, FaaS platforms enable user-defined functions to be executed on the predefined run-time environments. Consequently, it is necessary to load and initialize run-time environments of functions at cost, and functions should run as soon as possible. Virtualization technologies enable FaaS platforms to prepare runtime environments. Although the hypervisor-based virtualization provides stronger isolation, container-based virtualization is more lightweight; thus, this paper assumes to adopt container-based virtualization. Related works enhancing and optimizing the virtualization for FaaS. While it generally takes from a few of milliseconds to hundreds of milliseconds to execute a function, it takes from hundreds of milliseconds to a few seconds to initialize a container. It is, therefore, more beneficial to run a function on a pre-executed container (warm start) than on a newly created container (cold start) whenever functions are requested to run. To achieve scalability and availability, FaaS platforms generally consist of many worker nodes running containers or are organized by container orchestration tools such as Kubernetes. Due to the limited resources such as CPUs or memory, a worker node keeps only limited containers warm. FaaS platforms, therefore, have to decide which containers should be kept on worker nodes and to which nodes function requests should be sent.

Here arise the distributed scheduling problems of how to manage containers and where to distribute any function request in a FaaS cluster. This paper addresses those distributed scheduling problems between containers and function requests. Since the scheduling components, presented in Section II-B, are distributed in a FaaS cluster, and scheduling decisions have to be made in real time, our scheduling problems are challenging.

5 Methodology

To provision containers, our scheduler keeps track of the request queue and the number of containers for each function type by taking a snapshot for every tick. Listing 1 presents a class FPCSCHEDULERFORFUNCTION that renews snapshots and provisions containers for each function. We assume that every function request has a monotonously increasing sequence number in the order they arrive in its function queue. SNAPSHOT type and FPCSCHEDULERFORFUNCTION class are defined with the following variables for each function type.

- Qbegin: the sequence number of the first request in the queue, which increases whenever a request is scheduled.
- Qend : the sequence number of the last request arrived to the queue.
- Ccreating: the number of currently creating containers.
- Ccreated : the number of already created containers.
- TavgFuncExec: the average execution time.

For every fixed interval, e.g., 100 ms, renewSnapshot() and provisionContainer() are executed one after another. Using the member variables Qbegin, Qend , Ccreated , and Ccreating that were updated with the latest values, TavgFuncExec, tick, and snapshot S are updated in renewSnapshot(). We assign TICK-INTERVAL to TavgFuncExec initially.

In renewSnapshot(), the first condition Qbegin == 0 means that no request has been pulled, and S[tick].Qbegin and S[tick].Qend for tick = 0 remain zeros. After any requests have been pulled (Qbegin 6= 0), tick increases only when its queue is not empty (Qend Qbegin ¿ 0) or any new requests arrive in the previous interval (Qend S[tick]. Qend ¿ 0). At that time, the snapshot of the current tick is renewed. TavgFuncExec is calculated by averaging the execution times of the last N function requests (e.g., N = 10) in updateAvg-FuncExecTime(), which returns TICK-INTERVAL if no execution time is available. To this end, containers send every pull request piggybacked with the last execution time.

The method provisionContainer() provisions containers based on the following variables.

- Ctotal: the number of containers that have been created and are creating.
- RinQueue: the number of requests that currently exist in the queue.
- Pcontainer : the power that indicates how many function requests a container can execute in an interval.
- Rarrivals: the number of requests that arrive in the last interval.
- Crequired : the required number of containers to process all the requests in the queue in a tick interval.

```
type Snapshot:
    int Q_begin
    int Q_end
    int C_created
    int C_creating

class FPCSchedulerForFunction:
    # the followings are updated before renewSnapshot()
    int Q_begin
    int Q_end
    int C_created
    int C_creating
    # the followings are updated in renewSnapshot()
    int tick = 0
    float T_avgFuncExec = TICK_INTERVAL
    Snapshot[] S

    def renewSnapshot():
        if Q_begin == 0:
            S[tick] = {0, 0, C_created, C_creating}
        else:
            if Q_end - Q_begin > 0 or Q_end - S[tick].Q_end > 0:
                tick++
            S[tick] = {Q_begin, Q_end, C_created, C_creating}
            T_avgFuncExec = updateAvgFuncExecTime()

    def provisionContainers():
        int C_shortage = 0
        int C_total = S[tick].C_created + S[tick].C_creating
        int R_inQueue = S[tick].Q_end - S[tick].Q_begin
        if tick == 0:
            if S[tick].C_creating == 0: # condition 1
                C_shortage = 1
            else:                       # condition 2
                C_shortage = Q_end - S[tick].C_creating
        else if C_total < R_inQueue:
            float P_container = TICK_INTERVAL / T_avgFuncExec
            int R_arrivals = S[tick].Q_end - S[tick-1].Q_end
            int C_required = ⌈ R_inQueue / P_container ⌉
            if R_arrivals < R_inQueue:                          # condition 3
                C_shortage = C_required - S[tick].C_creating
            else if R_arrivals ≥ P_container * C_total:         # condition 4
                C_shortage = C_required - C_total
        if C_shortage > 0:
            createContainers(min(C_shortage, R_inQueue))
```

Provisioning containers for each function type can execute during the tick interval; for instance, if TICK-INTERVAL = 100 ms and TavgFuncExec = 20ms, Pcontainer is 5.0, which means that a container can handle 5 requests for the interval. If the average execution time is not yet available, a container is assumed to execute only a request per tick. In provisionContainers(), containers are provisioned differently according to the four ramified conditions.

- **Condition 1:** This is the initial condition to create the first container. If no request has processed and no container is being created yet, this condition is satisfied. Once the container creation starts, this condition is never met again.
- **Condition 2:** To cope with the sudden appearance of requests while the first container is being created, as many containers as requests in the queue are created. If the first container is not created in an interval, this condition can be repeatedly satisfied; at that time, creating containers are excluded as Cshortage = Qend - S[tick].Ccreating.
- **Condition 3:** Congestion arises when enough containers have not been provisioned at the previous tick. If it takes more than one tick to provision containers, this condition can be repeatedly met while new requests are still incoming. So this condition should consider such a case, and more containers are provisioned according to the available requests in the queue. Since S[tick]. Ccreating reflects the required number of containers in the previous tick, we need to exclude this from Cshortage to consider newly arrived requests in the last interval. As Cshortage can be rewritten as Crequired Ctotal +S[tick].Ccreated , S[tick].Ccreated containers are additionally created in this condition, compared to Condition 4.
- **Condition 4:** This denotes the state that the current number of containers fail to pull all the arrival requests. Intuitively, Cshortage is calculated as Crequired Ctotal. Finally, createContainers() creates the given number of containers in randomly selected workers. Unlike HashSch, FPCSch allows any worker to have containers of every function type. This makes the FaaS platform not only flexible in provisioning containers but also load-balanced.

It is worth noting that FPCSch rarely blocks any function executions due to container creations, only if there is at least one container corresponding to the function type. Every existing container can keep requests being pulled while new containers are being provisioned. For this reason, requests do not purely wait for cold starts, though delayed in their queues.

6 Conclusion

Significant attempts have been made in FaaS platforms to resolve the hazards of cold starts, mainly by lowering the sandbox (or container) start-up time via microscopic profiling. Surprisingly, the FaaS cluster has few macroscopic ways for scheduling and coordinating resources or requests. FPCSch is a unique pull-based algorithm that schedules containers rather than functions, as suggested in this paper. FPCSch avoids cold beginnings by asynchronously scheduling containers while warm containers pull functions continually. We are convinced that there are no critical restrictions to applying FPCSch to any sandbox-based FaaS platforms due to the versatility of our scheduler. Another contribution is to replace Apache OpenWhisk's rudimentary hash-based scheduler.

Our testing with real-world implementations reveals that FPCSch beats the competition, proving that our scheduler can handle multi-tenant, highly concurrent, resource-intensive, and scaling-out workloads. We intend to use our scheduler to optimize intermittent workloads in the future. Shahrad et al. [25] proposed a resource management policy that prevents cold starts for intermittently used routines. We anticipate that, due to its efficient resource utilization, FPCSch will have greater room to adopt such a strategy.

References

[1] . van Eyk, L. Toader, S. Talluri, L. Versluis, A. Ut,a, and A. Iosup, "Serverless is more: From PaaS to present cloud computing," IEEE Internet Computing, vol. 22, no. 5, pp. 8–17, 2018.

[2] mazon, AWS Lambda, November, 2014 (accessed Sept. 13, 2020), https://aws.amazon.com/lambda/.

[3] oogle, Cloud Function, February, 2016 (accessed Sept. 13, 2020), https://cloud.google.com/functions/.

[4] icrosoft, Azure Functions, January, 2017 (accessed Sept. 13, 2020), https://azure.microsoft.com/services/functions/.

[5] BM/Apache, OpenWhisk, February, 2016 (accessed Sept. 13, 2020), https://openwhisk.apache.org/.

[6] Fission," https://fission.io/, (accessed Sept. 13, 2020).

[7] OpenFaaS," https://www.openfaas.com/, (accessed Sept. 13, 2020).

[8] Kubeless," https://kubeless.io/, (accessed Sept. 13, 2020).

[9] Knative," https://knative.dev/, (accessed Sept. 13, 2020).

[10] .Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda," in 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). Denver, CO: USENIX, Jun. 2016.

[11] Docker," https://www.docker.com, (accessed Sept. 13, 2020).

[12] gVisor," https://gvisor.dev, (accessed Sept. 13, 2020).

[13] . Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. ArpaciDusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in 2018 USENIX Annual Technical Conference. Boston, MA: USENIX, Jul. 2018, pp. 57–70.

[14] . E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in 2018 USENIX Annual Technical Conference (USENIX ATC 18). Boston, MA: USENIX Association, Jul. 2018, pp. 923–935.

[15] . Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), 2019, pp. 209–214.