Project Report

On

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

Submitted in partial fulfilment of the requirements for the degree of

MASTER OF TECHNOLOGY

In

Computer Science and Engineering,

Submitted by

Sajja Komal Sai(212is026)

Gudiwada Raghava(212is009)

Department of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal.
2021

Declaration

I hereby declare that the Report of the P.G. Project Work entitled **Scheduling Containers Rather Than Functions for Function-as-a-Service** which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfillment of the requirements for the award of the Degree of Master of Technology (M.tech) in CS-IS in the Department of Computer Science, is a bonafide report of the work carried out by us. The material contained in this report has not been submitted to any University or Institution for the award of any degree.

Sajja Komalsai (212is026)
Gudiwada Raghava (212is009)
Department of Computer Science

Certificate

This is to certify that the P.G. Project Work Report entitled **Scheduling Containers Rather Than Functions for Function-as-a-Service** submitted by Gudiwada Raghava, Komalsai Sajja as the record of the work carried out is accepted as the P.G. Project Work Report submission in partial fulfillment of the requirements for the award of the degree of Master of Technology (M.Tech) in CS-IS in the Department of Computer Science.

Internal Guide
Sourav Kanti Addya
(Assistant Professor)

Chairman - DPGC
(Signature with Date and Seal)

Acknowledgement

This project has given me the opportunity to learn and explore the concept of server-less computing and its practical implementation in the real world. I would like to express my special gratitude to my guide for giving me the opportunity to implement this project. I would also like to thank all the members who have also supported in completion of this project.

Abstract

FaaS (Function-as-a-Service) is an appealing technology that enables users to run functions in an event-driven manner without having to worry about server maintenance. Container-based virtualization allows functions to execute in a lightweight, isolated run-time environment, but frequent function executions combined with container initialization (cold starts) cause the platform to become busy and unresponsive. Warm starts, or executing functions on previously initialised containers, are advocated for performance reasons, and so FaaS platforms make efforts to schedule functions to warm containers.

We discovered that the present scheduler had poor performance and inconsistent behaviour when dealing with multi-tenant and highly concurrent workloads from our experience running an on-premise FaaS infrastructure. Instead of scheduling functions to containers, this paper presents a new FaaS scheduling technique called FPCSch that schedules Function-Pulling-Containers. Because FPCSch allows containers to constantly fetch functions of the same kind, cold starts are significantly reduced.

Contents

List of Figures

1 Introduction

Serverless computing is a type of cloud computing that allows several tenants (users) to run billable applications without having to worry about the servers' operational logic[1]. It allows developers to concentrate on higher-level abstractions like business logic, and the billable applications are broad, encompassing everything from operating systems to functions. Functions can effectively deal with event driven requests utilised to apply their business logic because they are the smallest. As a result, FaaS cloud platforms are becoming the most popular form of serverless computing, not only in commercial clouds like Amazon AWS Lambda[2], Google Cloud Functions[3], and Microsoft Azure Functions[4], but also in open sources like Apache OpenWhisk[5], Fission, OpenFaaS, Kubeless, Knative, and OpenLambda.

While tenants of FaaS platforms define only functions, FaaS service providers provide all operational support needed to perform those functions. The quality of a FaaS service is measured by how rapidly functions can be done and how consistently FaaS platforms can adapt to fluctuating workloads, according to FaaS tenants.Meanwhile, service providers strive to maximise the efficiency of existing resources and process as many requests as possible while meeting the high standards that consumers expect.It is necessary to prepare an underlying environment, such as a certain version of an operating system, a language-specific environment, library packages, various settings, and so on, in order to execute functions that a user submits. Furthermore, virtualization (also known as sandbox) technologies are commonly used in FaaS systems to ensure strong isolation for supporting multi-tenancy, i.e. comprising of many different types of functions. Despite the fact that numerous virtualization technologies[6] [7]-[8] can be employed, we chose to create container-level virtualization using docker.

It takes a long time to build a new container and setup its run-time environment, i.e., cold start, before running any function within it for the first time. Many studies have found that cold starts decrease the performance of FaaS platforms significantly. Shortening cold starts from containers or virtual machines (VMs) is an efficient way to break a bottleneck in FaaS platforms, but there is still opportunity for cold starts to be reduced and warm begins to be promoted.

Reduced cold starts and increased warm starts can improve not just tenant satisfaction but also cluster efficiency, thanks to distributed scheduling algorithms in FaaS systems. In truth, the primary purpose of scheduling is to evenly distribute loads among available resources, however non-uniform scheduling can help with warm starts. Only rudimentary hash-based scheduling algorithms are now employed by Apache OpenWhisk and OpenLambda, proving this inconsistency. The research into FaaS schedulers is still in its early phases in academia, with only one scheduler released so far, but no scheduling methods for proprietary FaaS platforms have been completely disclosed.

In this paper,we are going to discuss about Function Pulling Container algorithm(FPSch).We schedule containers, each of which pulls functions of the same type, rather than the scheduler scheduling functions to containers. This significantly speeds up warm starts and makes uniform container deployment in a FaaS cluster.

The remainder of this work is laid out as follows. The backdrop and workloads for FaaS platforms are described in Section II. We introduce related work in Section III from two perspectives: reducing cold starts and existing scheduling techniques for FaaS systems. The FPCSch scheduler is presented in Section IV. Section V brings this paper to a close by discussing our future plans.

2 BACKGROUNDS

2.1 Scheduling Problem and Goals

Function-as-a-Service (FaaS) is a type of serverless computing in which the smallest execution unit, a function, is executed. FaaS platforms are triggered by incoming requests and allow user-defined functions to be executed on preset run-time environments. As a result, function run-time environments must be loaded and initialised with a low cost, and functions must run as fast as possible. FaaS systems can prepare runtime environments using virtualization technologies. Although hypervisor-based virtualization[9] - [10] provides more isolation, container-based virtualization such as docker, is more lightweight as a result, this study assumes container-based virtualization. In Further Sections, related studies strengthening and optimising virtualization for FaaS will be presented.

Generally it takes from a few of milliseconds to hundreds of milliseconds to execute a function and initialize a container it takes from hundreds of milliseconds to a few seconds. It is, therefore, more beneficial to run a function on a pre-executed container than on a newly created container whenever functions are requested to run.

FaaS platforms often comprise of numerous worker nodes running containers or are structured by container orchestration technologies such as Kubernetes to achieve scalability and availability. A worker node can only maintain a few containers heated due to restricted resources such as CPUs and memory. As a result, FaaS platforms must select which containers should be kept on worker nodes and which nodes should receive function requests. In a FaaS cluster, the distributed scheduling challenges of managing containers and distributing every function request arise.

2.2 System Models

The system model assumed in our FaaS scheduling algorithm is defined in this section. A function type can be registered with its container specification, which specifies the run-time contexts, and we presume that each function type has its own ID. Only containers initialised with their specifications can execute functions with the same ID. After a container has been initialised, it can be used to execute subsequent function requests of the same type in terms of the security and contention of the contexts on which functions rely. In other words, functions with different IDs never share containers, even if they have package affinity, which differs from previous works[11]-[12].

It is either a cold start or a warm start when a function is executed on a container. A cold start occurs after a function has been created and its run-time environment has been initialised, whereas a warm start occurs when a function is performed immediately on a container that has previously run the same function. The container image is pulled in advance and used to initialise a container at a worker node. Required packages or interpreters must be imported or initialised during the initialization of run-time environments. Though latency varies depending on the status of networks or worker nodes, we found that initialising a container takes more than 200 milliseconds, and initialising runtime environments takes 50 to 100 milliseconds. Meanwhile, the quickest time to perform a function was at least 2 milliseconds. A cold start is obviously one or two orders of magnitude slower than a warm start, which puts a strain on the worker node.

We presume that FaaS platforms have a load balancer, controllers, queues, and workers, similar to Apache OpenWhisk's architecture we used docker docker swarm. A load balancer distributes any function request to one of the controllers, and the controllers determine which of the dispersed queues the request is sent to using scheduling algorithms. A worker is responsible for the initialization of containers and run-time environments, as well as invoking the functionalities of requests as a cold or warm start. A worker can operate as many containers as their capacity allows, which is usually governed by memory size. Containers can be busy or warm; a busy container is one that is currently running a function, while a warm container is one that is idle and waiting for other requests. We categorise scheduling algorithms as push-based or pull-based on how requests are transmitted from queues to workers; controllers decide in the push-based scheduling algorithm, but workers decide in the pull-based one.

2.3 Characterizing Workloads

Our scheduling approach is optimised for workloads with the features listed below.

Multi-tenant workloads: Multi-tenant architecture, which supports numerous clients (tenants), is inherent in cloud services like FaaS. As a result, our scheduling method is built to ensure that different types of functions do not conflict.

Resource-intensive workloads: Because functions of the same type are invoked from the same code, they tend to use similar resources. If some resource-intensive tasks are concurrent enough to require many containers, and those containers are supplied to only a few workers, the containers will compete for the workers' limited resources. Previous research has found that commercial FaaS platforms perform differently depending on CPU/disk/network heavy workloads. When provisioning containers of the same function types to workers, it is preferable to disperse containers among a variety of workers to reduce resource competition. In the same way, it is more beneficial for a worker to execute multiple multi-tenancy containers from his or her perspective.

Another challenge with scheduling highly concurrent workloads is determining how function requests should be given to containers that have already been provisioned. Because elasticity may be easily exploited, the bin packing technique is preferred over the spreading strategy in this scenario. Some assessments and a talk addressed these scheduling (or placement) concerns among workers, containers, and requests.

3 Related Work

Other large cloud service providers have introduced their own FaaS systems since AWS Lambda was initially released in 2014. Several evaluations and comparisons have been reported as those commercial FaaS platforms compete. Interesting studies were presented in those papers, however the ability to reveal the causes of the proprietary platform outcomes w as limited. Following the validation of the benefits of FaaS platforms, various opensource FaaS solutions have exploded in popularity in recent years, and have been examined in recent articles. Cold starts are crucial to the performance and responsiveness of FaaS platforms, according to the majority of those articles.

3.1 Research on Shortening Cold Starts in Virtualization

Pipsqueak[13] and its descendants SOCK made cold starts easier by caching Python interpreters and libraries. SOCK also created a lean container by eliminating costly procedures for general-purpose containers and utilising the Zygote approach[14], in which new processes begin as forks of an existing one. SAND developed the application sand-boxing mechanism, in which a new process is forked within a container as a sandbox for executing a function, rather than a new container. Because forking a new process is obviously faster than generating a new container, SAND can reduce cold starts.

Commercial FaaS platforms, like as AWS Lambda, require hypervisor-based sandboxes that provide greater isolation than container-based sandboxes to handle multi-tenant workloads.Because hypervisor-based sandboxes like Xen and KVM/QEMU are expensive, various lightweight virtualization approaches have been developed.

3.2 Research on FaaS Scheduling

Shortening container or VM's starting times is useful and necessary, but it's a microscopic approach to enhancing FaaS platform performance. To put it another way, the aforementioned research works make it impossible to respond to the following macro questions. (1) How should containers in the FaaS cluster be provisioned when large concurrent function demands exceed the capacity of a single container? (2) How can function requests be planned to reduce cold starts when several containers have already been provisioned? (3) In order to be efficient, how should the already provisioned containers be removed? In truth, FaaS platform scheduling or placement algorithms might provide solutions to these problems, but the proprietary FaaS platforms have yet to release their methods.

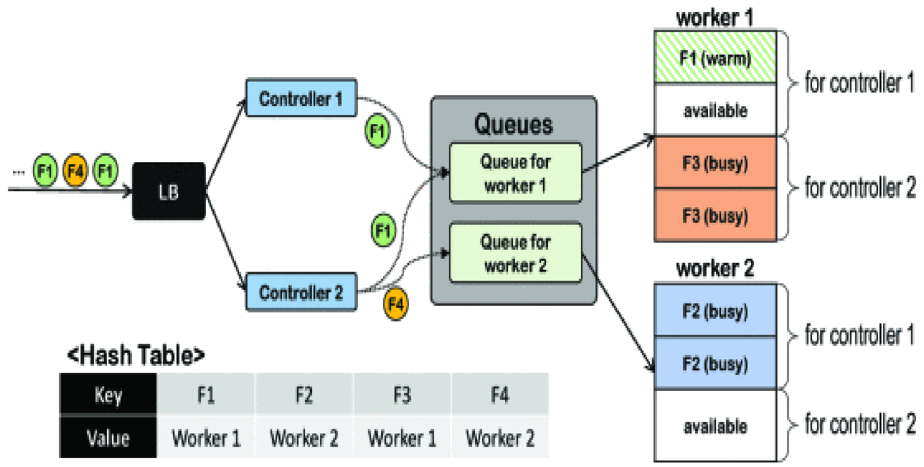
Package-aware scheduling was recently proposed as the only research of scheduling methods for FaaS platforms. In this paper, a package-aware scheduling algorithm called PASch[15] is suggested and evaluated with the push-based scheduler implementation in OpenLambda. PASch viewed enhancing package cache affinity to be a more significant aim than the inbuilt goal of any distributed scheduling algorithm, i.e. load balancing. It uses consistent hashing to distribute function requests to worker nodes and hashes the largest package as a key to maximise cache affinity. Because the larger packages are typically repeated, this strategy always results in hotspots, or load imbalance.

3.3 Hash-based Function Scheduling in OpenWhisk

This section explains the currently available hash-based scheduling method in Apache OpenWhisk.As indicated in Figure 1, function requests are sent to one of two controllers, and one queue is created for each worker. A worker can execute the same set of functions within the worker capacity because controllers push function requests to the queue of each worker associated by the hash table in Fig. 1. When the first function request F1 comes at controller 1, it is pushed to worker 1's hash table queue and delivered to F1's warm container at worker 1, resulting in a warm start. F4 is therefore executed as a cold start when it arrives at worker 2 via its queue because there is available capacity but no warm container.

Both F2 and F4 are scheduled, according to the hash table. to the second worker Assume that worker 2 gets the most F2. F4 is used a lot of the time. The capacity in this scenario is Worker 2 is primarily responsible for F2 containers, and F4 is usually executed after cleaning a warm container. F2 interference, or interference between F2 and F4. Even if none exist, As a result of this interference, HashSch is severely slowed. Re-coordinating mapping between function requests is tough.

After function requests have come at workers in HashSch, the decision to create a container is made. The time it takes to create a container is factored into the response time of a function



Key	F1	F2	F3	F4
Value	Worker 1	Worker 2	Worker 1	Worker 2

Figure 1: A scenario that shows how the hash-based scheduling algorithm works. Function requests F1, F4, and F1 arrive in sequence.

request. This results in long response times for high-variance requests, whereas our suggested technique separates container provisioning from function executions.

4 FUNCTION-PULLING-CONTAINER SCHEDULING

4.1 Overview of FPCSch Algorithm

In this section, we used a novel scheduling algorithm for FaaS platforms. The most major modifications to our method are the scheduling of containers rather than functions, and the containers pulling function requests. We make the scheduler incorporate a queue for each function type, and we distribute requests to the same function queue. Rather than scheduling functions, we schedule (provision) containers that continually pull function requests of the same type whenever they get warm hence the acronym FPCSch (Function-Pulling-Container Scheduler).

To illustrate our algorithm, Fig. 2 describes a scenario where function requests F3, F2, and F4 arrive one by one. When F3 first arrives at controller 1, it is enqueued to its designated queue. Our scheduler provisions a container of F3 since there is no container for F3. In this case, a cold start is also inevitable. However, if F2 whose containers already exist in some workers comes to its queue, one of its warm containers pulls through the priority queue of F2, which will be used for removing containers. When F4 arrives at its queue, its execution is delayed until the container of F4 becomes warm. Waiting for being warm containers, i.e., queueing delay, is shorter than a cold start time if a function execution time is shorter than a container creation time. However, high-volume function requests or long-running functions will result in lengthier queueing times, which can be alleviated by providing extra containers.

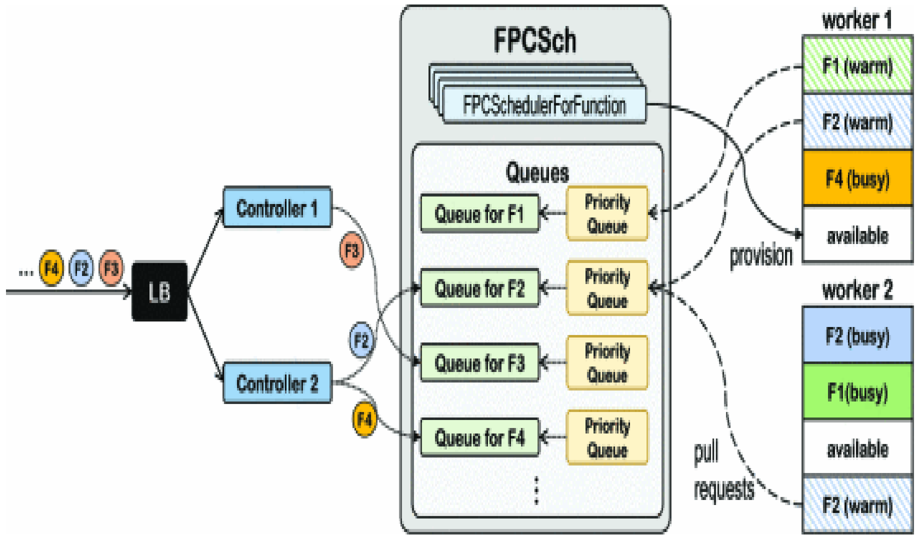


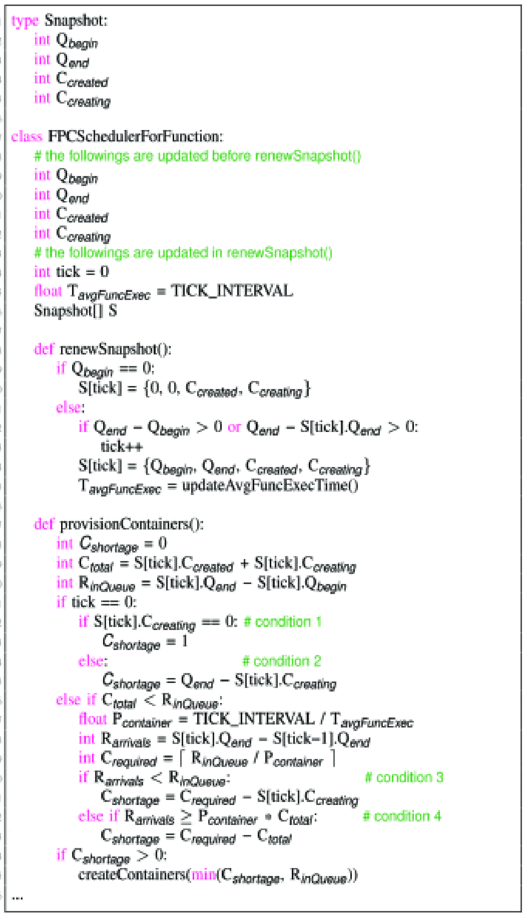
Figure 2: A scenario that shows how FPCSch algorithm works. Function requests F3, F2, and F4 arrive in sequence.

4.2 Provisioning Containers

To provide containers, our scheduler takes a snapshot for each tick to keep track of the request queue and the amount of containers for each function type. The FPCSCHEDULERFORFUNCTION class in Listing 1 renews snapshots and provides containers for each function. In the order they come in the function queue, we suppose that each function request has a monotonously growing sequence number. The following variables are used to specify the SNAPSHOT type and the FPCSCHEDULERFORFUNCTION class for each function type.

- Q_begin : the sequence number of the first request in the queue, which increases whenever a request is scheduled.
- Q_end : the sequence number of the last request arrived to the queue.
- $C\_creating$: the number of currently creating containers.
- $C\_created$: the number of already created containers.
- **TavgFuncExec**: the average execution time.

Here, we used the below algorithm in the building model can be to improve the performance of the system. It's worth noting that FPCSch only prevents function executions when there are at least one container matching the function type. While new containers are being provisioned, existing containers can continue to pull requests. As a result, despite being delayed in their queues, requests do not just wait for cold beginnings.



```
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 $C_{required}$ = $\lceil R_{inQueue} / P_{container} \rceil$
            if $R_{arrivals} < R_{inQueue}$:                  # condition 3
                $C_{shortage}$ = $C_{required}$ − S[tick].$C_{creating}$
            else if $R_{arrivals} \geq P_{container} * C_{total}$:         # condition 4
                $C_{shortage} = C_{required} - C_{total}$
        if $C_{shortage} > 0$:
            createContainers(min($C_{shortage}$, $R_{inQueue}$))
...
```

Figure 3: Provisioning containers for each function type

4.3 Removing Containers

If no requests are pulled for a given amount of time in FPCSch, the containers themselves will vanish (e.g., 10 min). Warm containers with the same function type pull requests in a different order of priority for this purpose. As shown in Fig. 2, containers requesting requests submit their container identities (CIDs) to a priority queue. The CIDs are dynamically sorted in alphabetical order in this priority queue, with the highest priority served first. The priority queue becomes filled of CIDs if the request queue is idle and many containers become warm. Arrival requests occasionally flock to a few higher-priority containers, depriving lower-priority containers of requests, resulting in their natural disappearance. This is a method of implementing the bin packing for a function type, as explained in Section II-C, so that the appropriate amount of containers are maintained for given requests while all surplus containers are reclaimed. If a new container is needed but there is no capacity available, the worker can remove one of the containers that has been warm and starving for a long time in advance.

We can store the last container for a long time, say 1 hour, to reduce cold starts for intermittently used routines. There is a trade-off between resource efficiency and performance optimization because their last containers would be idle the majority of the time.

4.4 Implications of Container Scheduling

FPCSCHEDULERFORFUNCTION is needed to keep track of the status of a function queue (Q_begin and Q_end), as well as the number of creating/created containers ($C\_created$ and $C\_creating$). They must be synchronised before the methods renewSnapshot() and provisionContainer(), because they are evaluated every tick. We allowed function queues to be incorporated in the scheduler, which instantaneously updated the state of function queues. Because containers are produced and removed less frequently, synchronising the life cycle of each container is substantially lighter than synchronising the life cycle of each function in each container. As a result, FPCSch, which schedules containers rather than functions, requires modest synchronisation.

In HashSch, scheduling a function can result in provisioning or removing any container; with other words, any function can be blocked if it is entangled in provisioning or deleting a container. FPCSch, on the other hand, schedules containers asynchronously with function executions as well as methodically according on the quantity of requests.

5 Conclusions and Future Work

Major attempts have been made in FaaS platforms to resolve the hazards of cold starts, mostly by lowering the sandbox (or container) start-up time itself via microscopic profiling. Surprisingly, the FaaS cluster has few macroscopic ways for scheduling and coordinating resources or requests. We attempted to implement FPCSch, an unique pull-based algorithm that schedules containers rather than functions, in this project. FPCSch avoids cold beginnings by asynchronously scheduling containers while warm containers pull functions continually.

References

[1] E. 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] Amazon, AWS Lambda, November, 2014 (accessed mar. 21, 2022)

[3] https://aws.amazon.com/lambda/. [3] Google, Cloud Function, February, 2016 (accessed april. 13, 2022)

[4] https://cloud.google.com/functions/. [4] Microsoft, Azure Functions, January, 2017 (accessed april. 15, 2022)

[5] https://azure.microsoft.com/services/functions/. [5] IBM/Apache, OpenWhisk, February, 2016 (accessed apri. 16, 2022)

[6] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci- Dusseau, 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.

[7] I. 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.

[8] "Docker," https://www.docker.com, (accessed april. 21, 2022).

[9] "containerd," https://containerd.io/, (accessed april. 17, 2022).

[10] A. 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.

[11] C. L. Abad, E. F. Boza, and E. van Eyk, "Package-aware scheduling of FaaS functions," in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ser. ICPE '18. New York, NY, USA: ACM, 2018, pp. 101–106.

[12] G. Aumala, E. Boza, L. Ortiz-Avil´es, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), May 2019, pp. 282–291.

[13] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), June 2017, pp. 395–400.

[14] "Overview of memory management (android developers)," https://developer.android.com/topic/performance/memory-overview, (accessed may. 07, 2022).

[15] G. Aumala, E. Boza, L. Ortiz-Avil´es, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," in 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), May 2019, pp. 282–291.

[16] Grötschel, Martin; Yuan, Ya-xiang (2012), "Euler, Mei-Ko Kwan, Königsberg, and a Chinese postman", 21st International Symposium on Mathematical Programming, Berlin, August 19–24, 2012, Documenta Mathematica, [Accessedon 04-12-2021].

[17] Arnab Chakraborty, "Fleury's Algorithm" for Euler path or Euler circuit published on 25-sep-2019.[Accessedon 11-12-2021]