2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-7281-9586-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCGrid51090.2021.00056

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

Dong Kyoung Kim
NAVER Corporation
Seongnam-si, Gyeonggi-do, Republic of Korea
Email: dominic.kim@navercorp.com

Hyun-Gul Roh
NAVER Cloud Corporation
Seongnam-si, Gyeonggi-do, Republic of Korea
Email: hyungul.roh@navercorp.com

***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. Container-based 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-Pulling-Containers* 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.

I. 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 [1]. 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 event-driven 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 [2], Google Cloud Function [3], and Microsoft Azure Functions [4], but also in open sources: for example, Apache OpenWhisk [5], Fission [6], OpenFaaS [7], Kubeless [8], Knative [9], and OpenLambda [10].

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 on-premise 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. Though various virtualization technologies [11], [12], [13], [14], [15], [16], [17] can be used, this paper assumes to provide *lightweight isolation* based on the container-level virtualization [11], [18], [19], [15].

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 [20], [13], [21], [22], [23], [24], [25] revealed that cold starts significantly degrade the performance of FaaS platforms. Shortening cold starts from containers or virtual machines (VMs) [26], [27], [12], [28], [29], [13], [14], [16] 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 [5] and OpenLambda [10]. Our experiments show that the existing hash-based scheduler of OpenWhisk *fails* to guarantee stable performance for *highly-concurrent and multi-tenant workloads*. Nevertheless, the study on FaaS schedulers is still in the early stages in academia, and only one scheduler has been introduced yet [30], [31], 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 *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.

We implemented `FPCSch` on Apache OpenWhisk, and have operated it in production since 2019. In this paper, we performed several evaluations to compare `FPCSch`-equipped OpenWhisk with that of the existing hash-based scheduling algorithm (`HashSch`) in terms of the four workload characterizations: multi-tenancy, high concurrency, spreading workloads, and scale-out. In all the cases, `FPCSch` outperformed `HashSch` and overcame all the weaknesses of `HashSch` against the workloads of multi-tenancy and high concurrency. As a result, `FPCSch` algorithm provides essential features required for production-level FaaS platforms.

The rest of this paper is organized as follows. Section II introduces backgrounds and characterizes workloads for FaaS platforms. In Section III, we introduce related work in two perspectives; shortening cold starts and existing scheduling algorithms for FaaS platforms. Section IV presents our proposed scheduler named `FPCSch`. In section V, we provide detailed evaluations with intensive workloads. Section VI finally concludes this paper, and discusses our future work.

II. Backgrounds

A. Scheduling Problem and Goals

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 run-time environments. Although the hypervisor-based virtualization provides stronger isolation, container-based virtualization [11], [19], [18], [13], [15], such as docker [11], is more lightweight; thus, this paper assumes to adopt container-based virtualization. Related works enhancing and optimizing the virtualization for FaaS will be introduced in Section III.

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 [6], [7], [8], [9]. 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.

Goal #1: One of their goals is to improve the responsiveness of function requests by maximizing warm starts with pre-executed containers. *Routing* or *pushing* function requests to corresponding containers is a straightforward approach, and the hash-based function scheduling (see Section III-C) is one example of this approach. In this paper, we achieve this goal in a completely different way.

Goal #2: The second goal is to balance loads among all worker nodes, which is an innate goal of most of the scheduling problems for scalability and stability. If the first goal is achieved by making similar function requests cluster into specific worker nodes, i.e., load imbalance, the second goal could be impeded. Hence, a solution to meet these two goals at once is demanded. Apparently, containers of a function type can be distributed across all worker nodes, and function requests should be uniformly delivered to them.

Goal #3: The last goal is to guarantee stable performance and to facilitate resource prediction and planning against multi-tenant and high-concurrent workloads (see Section II-C). Public FaaS platforms are supposed to serve numerous types of functions submitted by many customers, i.e., *multi-tenancy*, and most of their resources are consumed by requests of high concurrency, as described in Section II-C.

B. System Models

In this section, we define the system model assumed in our FaaS scheduling algorithm. A function type can be registered with its container specification defining the run-time environments, and we assume that each function type is identified with a unique ID. Functions of the same ID can be executed only on the containers initialized with their specifications. Once a container is initialized, it can be reused for executing other function requests of only the same type in the aspects of the security and contention of the contexts which functions depend on. In other words, functions of different IDs never share containers even if they have affinity in packages, which is different from the prior works [30], [31] (see Section III).

When a function is executed on a container, its starting is either a *cold start* or a *warm start*. A cold start is what a function begins after creating its container and initializing its run-time environment, while a warm start denotes that a function runs immediately on the container already run the same function before. A container is initialized at a worker node with the container image pulled in advance. During the initialization of run-time environments, required packages or interpreters need to be imported or initialized. Though it varies depending on the states of networks or worker nodes, we observed that it takes about more than 200 *ms* to initialize a container, and 50 $\sim$ 100 *ms* to initialize run-time environments. Meanwhile, the shortest execution time to complete a function was observed at least 2 *ms*. Obviously, a cold start is by one or two orders of magnitude slower than a warm start, burdening the worker node.

Following the architecture of Apache OpenWhisk, we assume that FaaS platforms consist of a load balancer, controllers, queues, and workers. Any function request is distributed to one of the controllers by a load balancer, and controllers decide to which of distributed queues the request is sent according to scheduling algorithms. A worker manages the initialization of containers and run-time environments, and finally invokes the functions of requests as either a cold start or a warm start. A worker runs as many containers as *worker's capacity*, which is generally determined by memory size. Containers are either *busy* or *warm*; a busy container is the one where a function is currently running, and a warm container is the idle one waiting for other requests. According to the way how requests are delivered from queues to workers, we categorize scheduling algorithms as push-based or pull-based ones; it is decided by controllers in the push-based scheduling algorithm, but by workers in the pull-based one.

C. Characterizing Workloads

Our scheduling algorithm is specialized for the workloads of the following characteristics.

Multi-tenant workloads: Cloud services like FaaS intrinsically support *multi-tenant architecture* that serves multiple customers, i.e., tenants. Our scheduling algorithm is, therefore, designed so that different types of functions do not interfere.

Highly concurrent workloads: Shahrad et. al [25] characterized the real workloads of Azure Functions service, and presented a resource management policy preventing intermittently invoked functions from cold starts. Nonetheless, the paper revealed that frequently invoked functions are the majority of the invocations; e.g., functions invoked more than once per minute occupy the 99.6% of total invocations. Apparently, the more frequently functions are invoked, the more resources they use. Such highly concurrent workloads cannot be handled by a single container, leading to provisioning multiple containers. To achieve a better overall performance of FaaS platforms, therefore, our proposed scheduling algorithm should efficiently handle workloads of high-concurrency, while intermittent workloads can be orthogonally managed by the policy proposed in [25].

Resource-intensive workloads: In general, the same type of functions tend to require similar resources, since functions of the same type are invoked from the same code. If some resource-intensive workloads are highly concurrent enough to demand many containers, and if those containers are provisioned to only a few workers, they would compete for the limited resources of the workers. Some prior works [32], [23] showed that commercial FaaS platforms show significant differences in performance according to CPU/disk/network-intensive workloads. In this regard, when containers of the same function types are provisioned to workers, it is more recommended to spread containers over diverse workers in order to avoid resource competition. For similar reasons, it is more desirable for a worker to run various containers of multi-tenancy in the viewpoint of a worker.

Another issue of scheduling highly concurrent workloads is how function requests should be delivered to already provisioned containers. In this case, the bin packing strategy is more recommended rather than the spreading strategy because elasticity can be easily exploited [33]. These scheduling (or placement) issues among workers, containers, and requests were addressed in some evaluations [32], [23] and a talk [33].

III. Related Work

Since the first release of AWS Lambda [2] in 2014, other giant cloud service providers have launched their own FaaS platforms [3], [4], [5]. As those commercial FaaS platforms are competing, several evaluations and comparisons have been reported [20], [32], [21], [23]. In those papers, interesting experiments were given, but there was a limit to reveal the causes of the results from the *proprietary* platforms. After benefits of FaaS platforms have been proven, many open-source FaaS solutions are flourishing in recent years [5], [6], [7], [8], [9], [10], and were evaluated in recent papers [34], [35], [22], [24]. Commonly in most of those papers [20], [21], [23], [22], [24], [25], it was revealed that *cold starts are critical* to performance and responsiveness of FaaS platforms. To shorten cold start time in virtualization, therefore, many efforts have been made as in the following subsection.

A. Research on Shortening Cold Starts in Virtualization

Pipsqueak [29] and its following SOCK [13] improved cold starts by caching Python interpreters with necessary libraries. SOCK also built a lean container by avoiding expensive operations for general-purpose containers and applied Zygote technique [36] in which new processes start as forks of an initial process. SAND [14] proposed the application sandboxing mechanism where a new process, instead of a new container, is forked *within* a container as a sandbox for executing a function. SAND can shorten cold starts because forking a new process is obviously faster than creating a new container.

To deal with multi-tenant workloads, commercial FaaS platforms, such as AWS Lambda, require hypervisor-based sandboxes [37], [12], [16] that provide stricter isolation than container-based ones. However, hypervisor-based sandboxes, such as Xen [38] and KVM/QEMU [37], [39], are costly, and thus many lightweight virtualization designs have been introduced [12], [26], [27], [28], [16]. Recently, Firecracker [16] and Catalyzer [17] were developed to ensure not only strong isolation, based on KVM [37] and gVisor [12], respectively, and but also a fast startup. To reduce the overhead of those hypervisor-sandboxes, Firecracker replaced heavy QEMU [39] from KVM/QEMU, and simplified the device model with *virtio* [40], which leads to short boot time. Meanwhile, Catalyzer achieved a fast function startup through init-less booting. Init-less booting is accomplished with function images generated offline, from which sandboxes are restored.

B. Research on FaaS Scheduling

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?

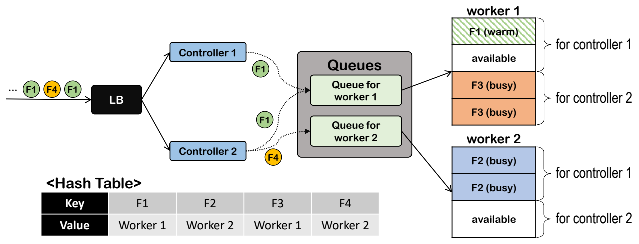
In fact, scheduling or placement algorithms of FaaS platforms can be answers to these questions, but the proprietary FaaS platforms have hardly revealed their algorithms yet. Instead, there were few attempts to infer such algorithms by well-designed experiments as in [23], [41].

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 multi-tenant 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.

C. Hash-based Function Scheduling in OpenWhisk

This section describes the existing hash-based scheduling algorithm currently presented in Apache OpenWhisk[1], abbreviated as `HashSch` for convenience's sake. It strives to meet the first goal, aforementioned in Section II-A, by scheduling function requests of the same types to the same worker. Fig. 1 depicts a simple scenario showing how `HashSch` works. As stated in Section II-B, function requests are distributed to one of the two controllers, and one queue is prepared for every worker. As controllers push function requests to the queue of each worker associated by the hash table in Fig. 1, a worker can execute the same set of functions within the worker capacity. When the first function request `F1` arrives at controller 1, `F1` is pushed to the queue of worker 1 according to the hash table and is delivered to the warm container of `F1` at worker 1, resulting in a warm start. After then, when `F4` arrives at worker 2 via its queue, `F4` is executed as a cold start because there is available capacity but no warm container.



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

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

According to the hash table, both `F2` and `F4` are scheduled to worker 2. Let us assume that worker 2 receives `F2` most of the time and `F4` occasionally. In this case, the capacity of worker 2 is mostly occupied with the containers of `F2`, and then `F4` can be executed mostly after wiping a warm container of `F2`; that is, *interference* between `F2` and `F4`. Even if any function seriously lags due to such interference, `HashSch` is difficult to re-coordinate mapping between function requests and workers. The workloads described in Section II-C can seriously exacerbate such interference.

When the worker targeted by the hash table lacks any warm containers and available capacity, controllers take a *workaround* which schedules functions to other available workers selected. To avoid the difficulty from *synchronization among controllers*, the capacity of a worker is divided evenly by the number of controllers, each of which is exclusively designated for controllers as in Fig. 1. Every controller traces the usage of its designated capacity whenever a function is scheduled and completed. When controller 2 receives `F1`, it cannot schedule `F1` at worker 1 because its designated capacity is full of busy containers of `F3`. Therefore, `F4` is finally executed at worker 2.

This workaround for overloaded functions is advantageous to the second goal mentioned in Section II-A, i.e., load balancing, but hinders other function types from the first goal because the capacity of a worker can be encroached regardless of the hash table. In this regard, it is expected that heavy real-world workloads might give rise to lots of cold starts, as proven by our experiments in Section V.

In `HashSch`, creating a container is decided after function requests have arrived to workers. If a container is created, its creating time is included in the response time of a function request. This causes lengthy response times of requests with

[1]The hash-based scheduling algorithm described in this paper is based upon the commit version 322d832857667a825ed4a3a0aa892a4a3fbce9f3

a high deviation, whereas our proposed algorithm separates container provisioning from function executions.

D. Synchronization Issue in FaaS Scheduling

Probably, instead of pre-defined hash functions, it would be *plausible* to design alternative scheduling algorithms that keep track of the states of all busy/warm containers whenever functions start and end, from which controllers schedule to push requests to the proper workers. However, the synchronized status of containers might be instantly outdated, considering the fact that functions can be executed in 2 ms, as mentioned in the II-B. The status at the moment one of controllers made a decision might be different with the actual container status, and such inconsistency leads to undesirable results. We analyzed more pathological synchronization issues in our early proposal to the OpenWhisk community [42]. From this point of view, we think the hash-based scheduler presented in the previous section is designed with less synchronization in mind. In this paper, we propose a novel scheduling algorithm requiring moderate synchronization.

IV. Function-Pulling-Container Scheduling

A. Overview of Proposed Algorithm

In this section, we propose a novel scheduling algorithm for FaaS platforms. The most significant changes in our algorithm are to schedule containers rather than functions, and for the containers to pull function requests. We make the scheduler embed a queue per function type, and deliver requests of the same function type to the same function queue. Instead of scheduling functions, we schedule (provision) containers that *pull* function requests of the same type continuously whenever they become warm; hence the name `FPCSch` abbreviated by Function-Pulling-Container Scheduler.

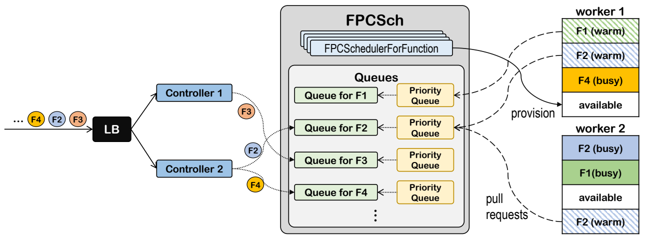


Fig. 2: A scenario that shows how FPCSch algorithm works. Function requests F3, F2, and F4 arrive in sequence.

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 (see Section IV-C). When `F4` arrives at its queue, its execution is delayed until the container of `F4` becomes warm. If a function execution time is shorter than a container creation time, waiting for being warm containers, i.e., queueing delay, is shorter than a cold start time. Nonetheless, highly concurrent function requests or long-lasting functions would cause longer queueing delays, which should be eased by provisioning more containers.

B. Provisioning Containers

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.

- $\mathbf{Q}_{begin}$: the sequence number of the first request in the queue, which increases whenever a request is scheduled.
- $\mathbf{Q}_{end}$: the sequence number of the last request arrived to the queue.
- $\mathbf{C}_{creating}$: the number of currently creating containers.
- $\mathbf{C}_{created}$: the number of already created containers.
- $\mathbf{T}_{avgFuncExec}$: the average execution time.

For every fixed interval, e.g., 100 ms, renewSnapshot() and provisionContainer() are executed one after another. Using the member variables Q_{begin}, Q_{end}, $C_{created}$, and $C_{creating}$ that were updated with the latest values, $T_{avgFuncExec}$, $tick$, and snapshot S are updated in renewSnapshot(). We assign `TICK_INTERVAL` to $T_{avgFuncExec}$ initially.

In renewSnapshot(), the first condition $Q_{begin} == 0$ means that no request has been pulled, and $S[tick].Q_{begin}$ and $S[tick].Q_{end}$ for $tick = 0$ remain zeros. After any requests have been pulled ($Q_{begin} \neq 0$), $tick$ increases only when its queue is not empty ($Q_{end} - Q_{begin} > 0$) or any new requests arrive in the previous interval ($Q_{end} - S[tick].Q_{end} > 0$). At that time, the snapshot of the current tick is renewed. $T_{avgFuncExec}$ is calculated by averaging the execution times of the last N function requests (e.g., $N = 10$) in updateAvgFuncExecTime(), 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.

- C_{total}: the number of containers that have been created and are creating.
- $R_{inQueue}$: the number of requests that currently exist in the queue.
- $P_{container}$: the power that indicates how many function requests a container can execute in an interval.
- $R_{arrivals}$: the number of requests that arrive in the last interval.
- $C_{required}$: the required number of containers to process all the requests in the queue in a tick interval.

$P_{container}$ that is obtained by dividing `TICK_INTERVAL` by $T_{avgFuncExec}$ represents *how many functions a container*

```
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($C_{shortage}$, $R_{inQueue}$))
...
```

Listing 1: Provisioning containers for each function type

can execute during the tick interval; for instance, if `TICK _INTERVAL` = 100 ms and $T_{avgFuncExec}$ = 20ms, $P_{container}$ 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` is when there is no creating (and also created) container and also no processed function request. `Condition 2` is prepared to deal with bursts of requests until the first container is created. `Condition 3` addresses congestion of the queue, as requests in the queue $R_{inQueue}$ are more than arrival requests $R_{arrival}$. `Condition 4` corresponds to the case when arrival requests $R_{arrivals}$ are beyond the power of the total containers. For each condition, the number of containers that should be created is calculated as follows:

`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 $C_{shortage}$ = Q_{end} - $S[tick].C_{creating}$.

`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 $S[tick].C_{creating}$ reflects the required number of containers in the previous tick, we need to exclude this from $C_{shortage}$ to consider newly arrived requests in the last interval. As $C_{shortage}$ can be rewritten as $C_{required} - C_{total} + S[tick].C_{created}$, $S[tick].C_{created}$ 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, $C_{shortage}$ is calculated as $C_{required} - C_{total}$.

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.

C. Removing Containers

In `FPCSch` containers themselves spontaneously vanish if they pull no request for a certain period (e.g., 10 min). For this purpose, warm containers of the same function type pull requests in different order of priority. Containers asking for requests submit their container identifiers (CIDs) to a *priority queue*, as shown in Fig. 2. This priority queue dynamically sorts the CIDs in alphabetical order and serves the highest priority first. If the request queue is idle and many containers become warm, the priority queue becomes full of their CIDs. Intermittently arrival requests flock to a few of containers of higher priority, and containers of lower priority are *starved* of requests, resulting in their natural vanishment. This is a way to implement the bin packing for a function type, as discussed in Section II-C, so that the adequate number of containers should be maintained for given requests with reclaiming all surplus containers. In case that a new container needs to be provisioned at the worker of *no available capacity*, the worker can remove in advance one of the containers that remains warm and starved for a long time.

To reduce cold starts for intermittently invoked functions, we can keep the *last* container for a long period, e.g., $1 hour$. Since their last containers would be idle most of the time, there is a trade-off between resource efficiency and performance optimization. As a future work, we plan to apply the sandbox (or container) management policy proposed in [25], which can predict when the intermittently invoked function containers should be loaded again for warm starts.

Our approach to removing containers is fully distributed, and our scheduler needs little synchronization to keep any information for removing containers. In addition, removing containers in `FPCSch` incurs no delay in executing any function. Let us recall the hash-based one in which containers of any function type are deleted due to contentions with other function types at a worker. During fierce contentions of several function types in a worker, highly demanded containers can be deleted in spite of available capacity in other workers; in this case, the execution time of a function request might include both the delay for removing the existing container of other type and the delay for creating its container. On the other hand, in `FPCSch` algorithm, deleting containers never directly affects the time for executing any functions.

D. Implications of Container Scheduling

FPCSCHEDULERFORFUNCTION is required to keep track of the status of a function queue, i.e., Q_{begin} and Q_{end}, and the number of creating/created containers, i.e., $C_{created}$, and $C_{creating}$. Since they are evaluated in the methods renewSnapshot() and provisionContainer() at every tick, they need to be synchronized before the methods. We let the function queues to be embedded in the scheduler, thereby updating the status of function queues immediately. Synchronizing the life cycle of every container is much more lightweight than synchronizing the life cycle of every function in every container because containers are less frequently created and deleted. For this reason, `FPCSch`, which schedules containers rather than functions, requires moderate synchronization, as stated in Section III-D.

In `HashSch`, provisioning/removing any container can result from scheduling a function; in other words, any function can be blocked if entangled in provisioning or removing a container. On the other hand, `FPCSch` schedules containers not only systematically based on the number of requests, but also asynchronously with function executions. As a result, `FPCSch` outperforms `HashSch` for all sorts of workloads, as shown in the experimental results of the next section.

V. EVALUATION

In this section, we present the evaluation results of `FPCSch`, compared to the existing hash-based scheduler of Apache OpenWhisk, denoted as `HashSch`. Please note that we choose only `HashSch` as a counterpart because there is no additional FaaS scheduler available for comparison. As stated in Section III, `PASch` is the only published FaaS scheduler, but we cannot apply it to our evaluation because `PASch` works for only Python workloads. One of our major contributions is to submit the real implementation of `FPCSch` to the Apache OpenWhisk community, which is currently in service by our company, and used in this evaluation.

A. Implementation Details

Our implementation was carried out based on OpenWhisk Git commit 322d832. We have newly implemented our scheduler by embedding a queue per function. Since our schedulers run in multiple nodes for the sake of high availability, we introduced ETCD [43] to *coordinate* scheduling information consistently to multiple schedulers. We modified the controller component so that a function request is distributed to its queue per function instead of any worker's queue.

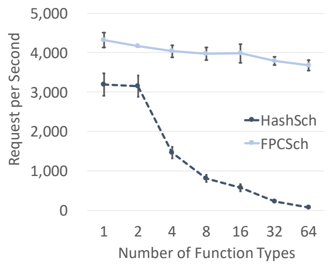
Each warm container adopts the long polling with a scheduler in order to send a *pull request* that includes a function ID (FID), a container ID (CID), and a pure execution time of the last request for the purpose of routing to its own queue, prioritizing in the priority queue, and calculating the average execution time, respectively (see Section IV). The average is updated with the last 10 executions for every function type. A container is backed-off if it pulls no request for 10 *sec* and finally removed if no request is pulled for another 10 *mins*. More detailed descriptions on the implementation can be found in our early proposal [42].

B. Experimental Settings

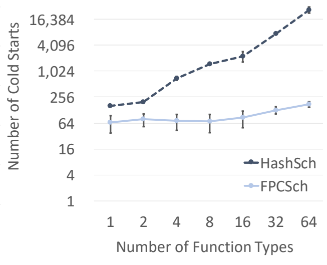
In our evaluation, FaaS platforms run on two types of machines: virtual machine (VM) and physical machine (PM). Each VM has an 8-core Intel Xeon E5-2660v4 CPU and 16GB of DDR4 SDRAM, while each PM has two 10-core Intel Xeon E5-2630v4 CPU and 128GB of 2.4GHz DDR4 SDRAM. For both `FPCSch` and `HashSch`, we prepared 3 VMs for controllers, 8 VMs for workers, and 3 PMs for the worker queues (Kafka) and the schedulers. They run on CentOS 7.4.1708 with docker 18.06.3. Every worker has pulled the container images of all function types in advance, and can simultaneously run only 40 containers whose memory is limited to 256MB.

As an *on-premise* FaaS provider, we have upgraded our platform from `HashSch` to `FPCSch`, and have considerably benefited in performance for our real workloads. Since our real workloads are obtained from intra-company clients, they are rather biased, compared to those of the giant public FaaS providers, presented in [25]. Above all, such real-world workloads did not help much to understand the characteristics of the scheduling algorithms. Since this evaluation is designed to characterize not our real workloads but our schedulers, the evaluation is performed with highly intensive workloads characterized in Section II-C.

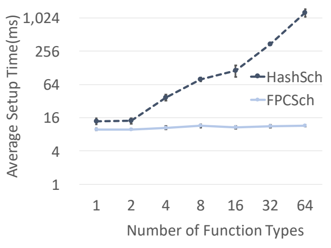
For generating workloads, we use an open-source benchmark solution nGrinder [44]. It can simulate and control concurrency by adjusting the number of *virtual clients*. It issues function requests in a *synchronous* manner; that is, a client can send a new request just upon receiving the response of its previous one. As a result, the faster the platform responds, the more requests the clients send. We have performed every



(a) throughput



(b) cold starts



(c) average setup time

Fig. 3: The effect of Multi-tenancy: (a) throughput, (b) cold starts, and (c) the average setup time according to the number of functions [NW: 8, NC: 100, NR: 100,000]

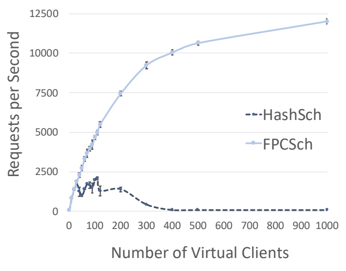


Fig. 4: Throughput according to concurrency [NW:8, NT:4, DR: 5 *min*]

experiment 5 times, except in Section V-F, and present the averaged result with standard deviation (STDDEV).

We prepared a Nodejs:8 function code that returns 10 bytes of dummy string. Note that this simple code enables to produce intensive workloads since its short execution time leads to more frequent scheduling. We regard the function types based on this code as different ones if they have different FIDs; thus, we let them never share containers and allow every container to pull function types of only the same FID. In Section V-E, we also use other longer-lasting function codes.

We denote the setting of each experiment in its result graphs with the following notations.

- **NW**: the number of workers.
- **NC**: the number of virtual clients.
- **NT**: the number of function types.
- **NR**: the number of generated requests.
- **DR**: the duration of generating requests.

C. The Effect of Multi-tenancy

To provide FaaS service as a commodity, it is essential for a single FaaS cluster to serve many sorts of functions together, i.e., multi-tenancy. We, therefore, first investigate the effects of multi-tenancy over `FPCSch` and `HashSch`. In this experiment, workloads consist of the exponentially increasing number of function types up to 64 to show the tendency more explicitly; i.e., 1, 2, 4, ... 64. With the experimental setting, we present the result in Fig. 3

Fig. 3a shows the average throughput of `FPCSch` and `HashSch`, measured by the number of processed requests per second (RPS). We observed no significant decrease in the throughput of `FPCSch`, but the throughput of `HashSch` suffers a radical degradation that comes from cold starts, as shown in Fig. 3b, where the log-2 scale is used for the y-axis. In `HashSch`, the number of cold starts with 64 function types is about 169 times greater than that with one function type, but about 2.6 times in `FPCSch`. StdDev also soars from 6.18 to 4161.99 in `HashSch`. Fig. 3a and 3b prove that there is an obvious correlation between throughput and cold starts.

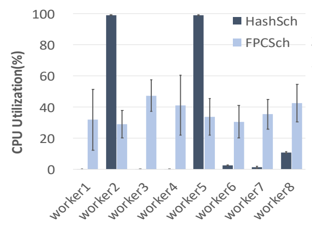
Fig. 3c observes the average setup times. A setup time is the elapsed time from the time a request arrives at a load balancer immediately before the function code is initialized. During the setup time, a controller and a worker carry out the following steps: authentication, authorization, fetching function information from the database, waiting in the queue, pushing or pulling a function request, removing/creating containers if necessary. These steps vary depending not only on cold and warm starts but also on `FPCSch` and `HashSch` (see Section III-C and IV). Hence, setup times are determined by the scheduling algorithms. `FPCSch` has quite stable average setup times varying from 9.83 *ms* to 11.50 *ms* with STDDEV less than 1.0, whereas `HashSch` brings a heavy increase in both the setup times and their StdDev as much as about 91 times and 114 times, respectively. `FPCSch` achieves such short and stable setup times not only because it causes less cold starts, but also because it provisions and removes containers asynchronously, i.e., not in the setup time.

D. The Effect of High Concurrency

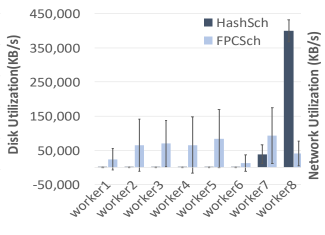
In the previous experiment, `HashSch` reaches the maximum of the cluster utilization, but `FPCSch` did not due to lack of concurrency. In this section, we go over the effect of concurrency by increasing virtual clients and find the maximum throughput of `FPCSch`. We have tested with the following numbers of virtual clients: 1, 10, 20, 30, ..., 120, 200, 300, 400, 500, 1000. Those numbers are chosen in order to find the maximum throughput and to exhibit tendency. This experiment allows only 4 function types since `HashSch` is out of control with more function types.

Fig. 4 shows the throughput of the two algorithms in terms of concurrency. Up to 30 clients, the throughput of `HashSch` are similar to that of `FPCSch`, but fluctuates seriously between 30 and 200 clients. `HashSch` reached the top, i.e., 2090.98 RPS with 110 clients, but with higher concurrency, its throughput is seriously degraded. In `HashSch`, if concurrency goes above a certain level, all the function types spread across all workers, incapacitating the hash-based scheduling. This leads to the repetition of creating and removing containers, resulting in such an unpredictable performance; thus, we had gone through difficulty in serving `HashSch`-based OpenWhisk.

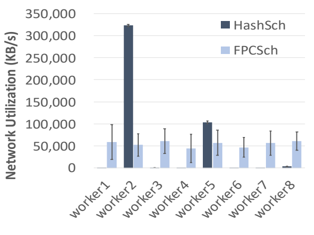
Meanwhile, our `FPCSch` shows increasing performance for the workloads of growing concurrency, though the performance seems saturated eventually. In contrast with `HashSch`, therefore, `FPCSch` has a desirable feature for serving FaaS



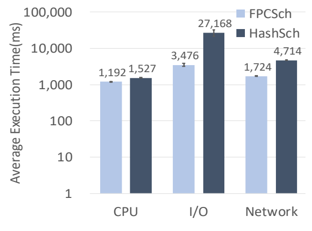
(a) Average CPU utilization



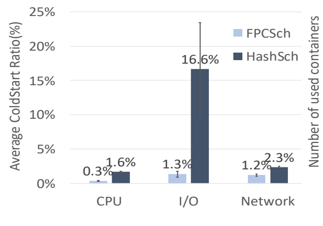
(b) Average disk utilization



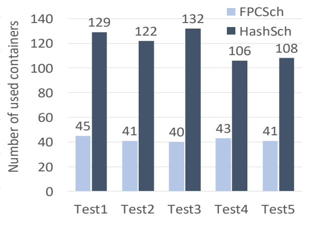
(c) Average network utilization



(d) Average execution time



(e) Cold start ratio in each test.



(f) The number of created containers in each test.

Fig. 5: The effect of resource-intensive workloads [NW:8, NT:3, DR:10 *min*]

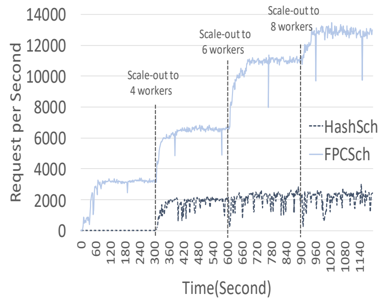


Fig. 6: Throughput according to scale-out [NW:4, NT:4, NC:110 for `HashSch`, and NC:1000 for `FPCSch`]

service; i.e., the throughput of `FPCSch` is predictable in terms of concurrency as well as multi-tenancy.

E. The Effect of Resource-intensive Workloads

Stated in Section II-C, it is encouraged to spread resource-intensive workloads evenly over all workers for better performance. This section shows how the two algorithms spread resource-intensive workloads. We prepare three types of resource-intensive function types as follows; (1) CPU-intensive function calculates Fibonacci numbers and distance matrix for 1000 *ms*, (2) I/O-intensive function writes 512 *KB* to a file 1000 times by using `dd` command, (3) Network-intensive function sends and receives back 3 *MB* string 15 times to/from a random prearranged HTTP server. We designed the workloads to concurrently send 60 requests/sec for each of CPU and network-intensive functions, and 6 requests/sec for I/O-intensive function, i.e., 126 requests/sec, so that the workload can be handled by all the worker without difficulty, but not by any single worker alone.

Fig. 5a, 5b, and 5c show the resource utilization at the eight workers for the two scheduling algorithms. Due to the nature of `HashSch`, one or two workers are over-utilized for each resource; remind that requests of a certain type are sent to another worker if the capacity of the hash target worker is exhausted (see Section III-C). As `HashSch` works with the fixed hash function in five tests, the graphs show small STDDEVs. Meanwhile, as `FPCSch` provisions containers to random worker nodes, containers in most of workers take part in pulling requests with their CIDs. Due to the properties of the priority queues based on randomly generated CIDs, STDDEVs in `FPCSch` results are relatively high (see Section IV-C).

Fig. 5d presents the average execution times of each function type. On average, CPU, I/O, and network-intensive functions of `HashSch` run 1.2, 7.8, and 2.7 times longer than those of `FPCSch`, respectively. It is obvious that `FPCSch` ensures more resources than `HashSch`. Considering the solitary execution times of CPU, I/O, and network-intensive functions are about 1.2, 3.3, and 1.5 *sec*, respectively, we can infer that `FPCSch` enables them to run with enough resources.

If any function lasts longer, it is required to create more containers for concurrent requests, incurring more cold starts. This is proven by Fig. 5d and 5e, where long-lasting I/O-intensive functions of `HashSch` result in a very high ratio of cold starts. Fig. 5f finally shows that `FPCSch` requires considerably fewer containers, i.e., cold starts, than `HashSch`.

F. The Effect of Scale-Out

The essential feature for the production-level FaaS platform is linear scalability. In this experiment, we observe the effect of scale-out by adding two workers every 5 *mins* from the initial two workers and up to eight workers. We impose the workloads of different concurrency on the two schedulers by finding the *affordable* number of virtual clients (see Section V-D); i.e., NC:110 for `HashSch` and NC:1000 for `FPCSch`.

In Fig. 6, `HashSch` is out of control with two workers, which means two workers are impossible to manage such concurrent workloads. Though `HashSch` can handle the workloads with from 4 workers, but throughput comes to a standstill in spite of increasing workers. On the other hand, we can confirm that `FPCSch` guarantees linear scalability. `FPCSch`, therefore, enables to make the FaaS platform elastic, which is a key feature of cloud computing for cost-saving.

VI. CONCLUSIONS AND FUTURE WORK

In FaaS platforms, major efforts have been put into resolving pitfalls of cold starts mainly by reducing the sandbox (or container) start-up time itself based on microscopic profiling. Surprisingly, there are few macroscopic approaches that schedule and coordinate resources or requests in the FaaS cluster. This paper proposed `FPCSch`, a novel pull-based algorithm that schedules containers rather than functions. `FPCSch` can avoid cold starts by asynchronously scheduling containers, while functions are continuously pulled by warm containers. Due to the generality of our scheduler, we are confident that there is no critical restriction to apply FPCSch to any sandbox-based FaaS platforms. Our another contribution is to replace the primitive hash-based scheduler of Apache OpenWhisk. Our evaluation with the real implementations in service shows that FPCSch scheduler noticeably outperforms the existing one, and verifies that our scheduler has effective enough to deals with intensive workloads that are multi-tenant, highly concurrent, resource-intensive, and scaling-out.

As a future work, we plan to optimize intermittent workloads with our scheduler. Shahrad et. al [25] presented a resource management policy preventing intermittently invoked functions from cold starts. We expect that FPCSch can provide more room to adopt such a policy thanks to the efficient resource utilization.

REFERENCES

[1] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, 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 Sept. 13, 2020), https://aws.amazon.com/lambda/.

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

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

[5] IBM/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] S. 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] 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.

[14] 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.

[15] 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.

[16] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "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.

[17] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481.

[18] "containerd," https://containerd.io/, (accessed Sept. 13, 2020).

[19] "CRI-O," https://cri-o.io/, (accessed Sept. 13, 2020).

[20] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017.

[21] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 159–169.

[22] S. Quevedo, F. Merchán, R. Rivadeneira, and F. X. Dominguez, "Evaluating apache openwhisk - FaaS," in *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)*, 2019, pp. 1–5.

[23] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference*. Boston, MA: USENIX Association, Jul. 2018.

[24] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY: ACM, 2019, p. 1063–1075.

[25] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC 20*. USENIX, Jul. 2020.

[26] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ASPLOS '13*. New York, NY: ACM, 2013, p. 461–472.

[27] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72.

[28] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 218–233.

[29] 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.

[30] 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.

[31] G. Aumala, E. Boza, L. Ortiz-Avilés, 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.

[32] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 442–450.

[33] A. W. Service, *AWS re:Invent 2018: [REPEAT 1] A Serverless Journey: AWS Lambda Under the Hood (SRV409-R1)*, Dec., 2018 (accessed Sept. 13, 2020), https://youtu.be/QdzV04T_kec.

[34] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An evaluation of open source serverless computing frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, pp. 115–120.

[35] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services*, 2019, pp. 206–211.

[36] "Overview of memory management (android developers)," https://developer.android.com/topic/performance/memory-overview, (accessed Sept. 13, 2020).

[37] Kivity, Avi, "KVM: the Linux Virtual Machine Monitor," in *OLS '07: The 2007 Ottawa Linux Symposium*, July 2007, pp. 225–230.

[38] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Operating System Review*, vol. 37, no. 5, 2003.

[39] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[40] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Operating System Review*, vol. 42, no. 5, Jul. 2008.

[41] M. Shilkov, *Serverless tags*, (accessed Sept. 13, 2020), https://mikhail.io/tags/serverless/.

[42] D. K. Kim, *OpenWhisk Wiki*, November, 2019 (accessed Dec. 15, 2020), https://cwiki.apache.org/confluence/display/OPENWHISK/New+architecture+proposal.

[43] "ETCD," https://etcd.io/, (accessed Sept. 13, 2020).

[44] "nGrinder," http://naver.github.io/ngrinder/, (accessed Sept. 13, 2020).