Received: 29 October 2020 | Revised: 17 May 2021 | Accepted: 9 June 2021

DOI: 10.1002/spe.3016

SPECIAL ISSUE PAPER

WILEY

Container lifecycle-aware scheduling for serverless computing

Song Wu[1] | Zhiheng Tao[1] | Hao Fan[1] | Zhuo Huang[1] | Xinmin Zhang[1] | Hai Jin[1] | Chen Yu[1] | Chun Cao[2]

[1]National Engineering Research Center for Big Data Technology and System Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

[2]State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China

Correspondence
Hao Fan, National Engineering Research Center for Big Data Technology and System Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.
Email: fanh@hust.edu.cn

Abstract

Elastic scaling in response to changes on demand is a main benefit of serverless computing. When bursty workloads arrive, a serverless platform launches many new containers and initializes function environments (known as cold starts), which incurs significant startup latency. To reduce cold starts, platforms usually pause a container after it serves a request, and reuse this container for subsequent requests. However, this reuse strategy cannot efficiently reduce cold starts because the schedulers are agnostic of container lifecycle. For example, it may ignore soon available containers or evict soon needed containers. We propose a container lifecycle-aware scheduling strategy for serverless computing, CAS. The key idea is to control distribution of requests and determine creation or eviction of containers according to different lifecycle phases of containers. We implement a prototype of CAS on OpenWhisk. Our evaluation shows that CAS reduces 81% *cold starts* and therefore brings a 63% reduction at 95th percentile latency compared with native scheduling strategy in OpenWhisk when there is worker contention between workloads, and does not add significant performance overhead.

KEYWORDS

cold start, container, elastic scaling, scheduling, serverless computing

1 | INTRODUCTION

Serverless computing has been increasingly popular, thanks to container-based virtualization.[1-3] In serverless computing, traditional monolithic applications are broken into finer-grained functions, and each function serves one type of request by instantiating one or more specific containers on workers. Containers are easy to deploy, and have low resource footprints and fast startup times, which enables the advantages of serverless computing: no explicit provisioning, paying as you go, and elastic scaling which automatically increases or decreases the number of containers in response to an increase or decrease of the load.

However, the elastic scaling is not fast enough when it needs to increase the number of containers because of *cold starts*.[4] A *cold start* requires multiple preparations before serving a request, including launching a container, initializing specific runtime, such as python,[5] and loading associated function code. These preparations introduce serious delays to function execution when elastic scaling is needed.[4,6]

There are two primary policies to reduce *cold starts*. (1) Adopting customized sandbox technologies which have shorter startup times.[7-9] These solutions speed up container creation or runtime initialization with the cost of sacrificing isolation or flexibility, and cannot avoid the delay caused by loading function code. (2) Relying on schedulers to pause a container that has served a request, and to reuse it for the same type of requests since one type of request can only be processed in specific containers. These schedulers work well when workloads are stable. However, they are agnostic of container lifecycle and may fail to make appropriate decisions for distribution of requests and creation or eviction of containers when elastic scaling is needed for bursty workloads.

In this article, we summarize the reasons why schedulers frequently trigger *cold starts* as follows. (1) *Ignoring the worker with an available container*. When a request arrives, it is usually dispatched to a worker specified by a load balancing or hashing algorithm ignoring available containers on other workers. (2) *Evicting soon needed container*. When a container needs to be created on one worker, a container serving other type of requests is evicted if the worker's resource is insufficient even if the evicted container may be reused shortly soon. (3) *Disregarding soon available container*. While creating a new container for a request, a container serving the same type of request may be available soon. This soon available container is often disregarded even if specifying this container to the request shows a shorter latency.

To address these issues, we propose a **c**ontainer lifecycle-**a**ware **s**cheduling scheme, **CAS**, which knows when containers are created, used, paused, or evicted. The primary idea is to control distribution of requests and determine creation or eviction of containers according to different container states (i.e., starting, running, or paused). In particular, CAS reduces *cold starts* in three aspects. (1) Choosing an affinitive worker. CAS maintains container states by tracing container state changes and chooses a worker with an available container accordingly. (2) Holding a soon needed container. When CAS has to choose a worker to create a new container, the worker that does not need evicting a container is preferred over others. (3) Using a soon available container. CAS detects available containers released by recently finished requests while creating a new container. If an available container appears before the new container is ready, CAS uses the available container immediately.

We implement a prototype of CAS on the popular open-source serverless platform, OpenWhisk.[10] Our experiments show that CAS reduces 81% *cold starts* and therefore brings a 63% reduction at 95th percentile latency compared with native scheduling strategy in OpenWhisk when there is worker contention between workloads, and does not add significant performance overhead.

In summary, our contributions consist of the following:

(1) We analyze in detail how and why existing schedulers frequently trigger *cold starts*.
(2) We propose a container lifecycle-aware scheduling scheme, CAS, which controls distribution of requests and determines creation or eviction of containers according to different lifecycle phases of containers.
(3) We implement a prototype of CAS on OpenWhisk[10] and evaluate its effectiveness. The evaluation shows that CAS can efficiently reduce *cold starts*.

The rest of this article is organized as follows: We first introduce background concepts of serverless computing and analyze in detail how and why unnecessary cold starts frequently happen in Section 2. We then describe the design and implementation of CAS in Section 3. Next, we evaluate the effectiveness of CAS in Section 4. We discuss related works about the cold start problem in Section 5 before concluding with a summary in Section 6.

2 | BACKGROUND AND MOTIVATION

In this section, we introduce the architecture and scheduling schemes of serverless computing, and analyze how and why unnecessary *cold starts* frequently happen.

2.1 | Serverless computing

In serverless computing, each request is processed in a container with specified resource requirements including CPU and memory. Figure 1 illustrates the scheduling model of a typical serverless platform. An incoming request is first sent to the scheduler for security verification and load balancing. Then the scheduler chooses a worker and specifies a container for the request. And this request will be processed in the specified container.

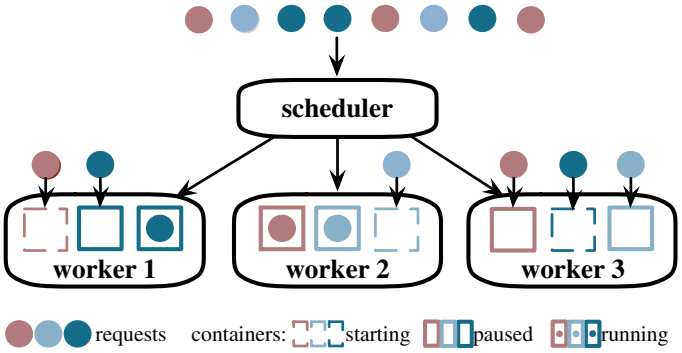


FIGURE 1 Scheduling model of a serverless platform. When a container is being launched or when initializing an execution environment, it is called a *starting container*. When a container is serving a request, it is named a *running container*. A container that is paused after serving a request is called a *paused container* [Colour figure can be viewed at wileyonlinelibrary.com]

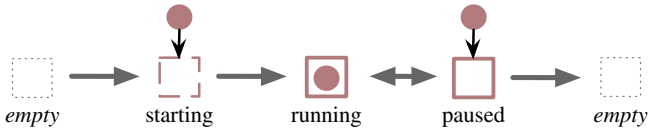


FIGURE 2 Lifecycle of a container [Colour figure can be viewed at wileyonlinelibrary.com]

As shown in Figure 2, a request can be processed in two ways. (1) The first method uses a new container and is called a *cold start*. When a request comes, a new container is created and the environment is initiated. After finishing initialization, this container starts serving the request. A *cold start* introduces undesired startup latency to request processing. (2) After serving a request, a container becomes a *paused container*. The second way reuses a *paused container* and is named a *warm start*. A *warm start* has an already prepared execution environment and serves a request immediately. Compared with a *cold start*, a *warm start* is very fast. It is important to mention that a *paused container* released by a finished request can only be reused by the same type of requests. And if a *paused container* has not been reused for a while, it will be evicted to free resources.

2.2 | Scheduling schemes of serverless computing

In this section, we discuss existing scheduling schemes of serverless computing. In general, there are two approaches to scheduling workloads in serverless computing: load balancing scheduling and affinity scheduling.

Load Balancing Scheduling. Popular load balancing solutions[11,12] spread the load among all available workers evenly. These solutions are designed to avoid worker overloading and do not take container lifecycle into account.

- *Round robin* distributes the load evenly among workers in a round-robin fashion.
- *Least load* assigns the load to the worker with the least requests.
- *Random* distributes the load randomly among workers.

Affinity Scheduling. Affinity-based solutions[13,14] dispatch the same type of requests to the same worker if the worker is not overloaded. These solutions can avoid worker overloading, and efficiently reuse *paused containers* when workloads are stable. However, these solutions are unaware of container lifecycle. They just assume that requests can easily get available containers on the specified workers.

- *Hashing* distributes the load according to hash codes of requests. A request is dispatched to the worker specified by the hash code.
- *Consistent Hashing*[15] is a special kind of hashing. Both requests and workers are mapped to the same hashing space. A request is dispatched to the nearest worker in the hashing space.

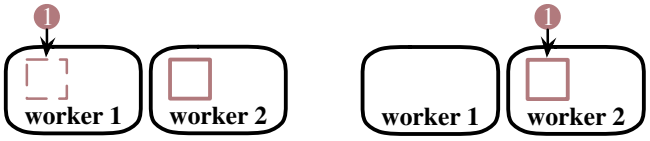
2.3 | Unnecessary cold starts

Existing scheduling approaches lack consideration or awareness of container lifecycle. *Cold starts* can be frequently triggered by these approaches for the following reasons.

Ignoring the worker with an available container. A request may be dispatched to a worker requiring a *cold start* instead of a worker with an available container (i.e., *affinitive worker*). Figure 3 shows an example of this case. In the beginning, *worker 2* hosts a *paused container* left by previous requests. As shown in Figure 3(A), when *request 1* whose preferred worker is *worker 1* arrives, it is dispatched to *worker 1* and suffers from a *cold start*. In fact, dispatching *request 1* to *worker 2* as Figure 3(B) can reuse the *paused container* to avoid a *cold start*.

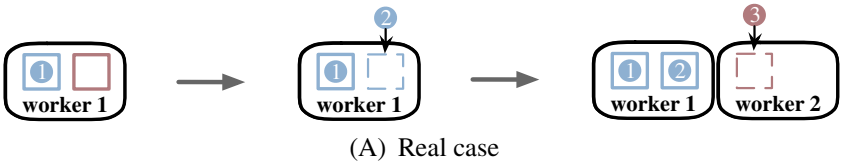
Evicting soon needed container. A container may be mistakenly evicted. When a request demanding the evicted container arrives, a *cold start* is required. Figure 4 illustrates an example. In the beginning, *worker 1* hosts a *paused container*. As shown in Figure 4(A), when *request 2* whose preferred worker is *worker 1* comes, there is no *affinitive worker* for it and *worker 1* can accommodate it by evicting a container; thus, the *paused container* on *worker 1* is evicted to create a new container for *request 2*. Then *request 3* arrives and another container is created on *worker 2* for it. Both *request 2* and *request 3* suffer from *cold starts*. If *request 2* is dispatched to *worker 2* as shown in Figure 4(B), *request 3* can reuse the *paused container* on *worker 1* to avoid a *cold start*.

Disregarding soon available container. A request may disregard an available container released by a finished request and insist on a newly created container. Figure 5 shows an example. When *request 2*arrives, there is only one

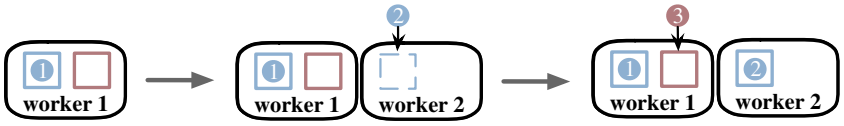


(A) Real case (B) Ideal case

FIGURE 3 Example case of ignoring the worker with an available container [Colour figure can be viewed at wileyonlinelibrary.com]

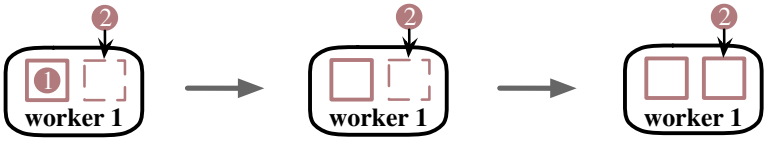


(A) Real case

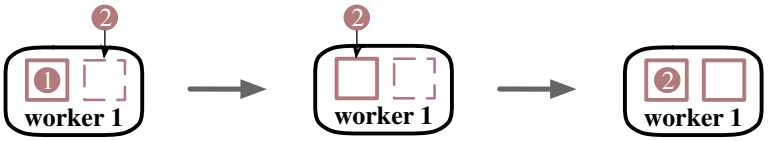


(B) Ideal case

FIGURE 4 Example case of evicting soon needed container [Colour figure can be viewed at wileyonlinelibrary.com]



(A) Real case



(B) Ideal case

FIGURE 5 Example case of disregarding soon available container [Colour figure can be viewed at wileyonlinelibrary.com]

running container and thus a new container is created. Then the *running container* finishes processing a request and becomes available. However, as shown in Figure 5(A), *request 2* waits for the newly created container rather than uses the available container. Actually, using the available container to serve *request 2* as Figure 5(B) can probably bring a shorter latency.

In summary, existing scheduling methods may make inappropriate decisions for being agnostic of container lifecycle, which can bring serious delays due to *cold starts*.[6,8]

3 | PROPOSED SOLUTION

Based on the analysis above, we propose a container lifecycle-aware scheduling scheme, CAS. In this section, we first summarize the overview of CAS and then describe the design in detail.

3.1 | Overview

In order to minimize *cold starts* while dispatching requests, first, CAS traces the changes of container state. Accordingly, CAS finds an *affinitive worker* for a request and dispatches the request to the worker. Second, CAS considers a *paused container* as a soon needed container. If there is no *affinitive worker*, CAS prefers to create a new container on the worker that does not need evicting a *paused container*. Third, CAS perceives the appearances of available containers. If an available container appears, it is immediately used for request processing.

3.2 | Choosing an affinitive worker by tracing container state changes

Dispatching a request to a worker without an available container causes a *cold start*, and resources occupied by available containers on other workers are actually wasted. Choosing an *affinitive worker* for a request can make use of an available container to avoid a *cold start*.

To do so, the scheduler needs to know where available containers are. Accordingly, CAS traces the changes of container state in a container lifecycle. The scheduler knows initial container states and updates states when container state changes happen. The scheduler can detect most container state changes by itself. When dispatching a request to a worker according to the current container states, if the chosen worker is an *affinitive worker*, container reuse happens; otherwise, container creation occurs. A request leaving a worker and returning to the scheduler indicates the availability of a container. However, when a container is evicted for expiration or resource shortage, the scheduler is unaware. Thus, when a container is evicted on a worker, CAS actively sends a report to the scheduler from this worker.

To maintain container states, two extra properties are held by each type of request: $rt_{running}$—the number of *running containers* that are serving one specific type of request on one worker, and rt_{paused}—the number of *paused containers* that can serve one specific type of request. These properties are updated according to Algorithm 1. $rt_{running}$ is increased for

Algorithm 1. State Updating

Input: csc, container state change; rt, request type
Output: void

if $csc == CREATED$ **then**
  $rt_{running} \Leftarrow rt_{running} + 1$
else if $csc == PAUSED$ **then**
  $rt_{running} \Leftarrow rt_{running} - 1, rt_{paused} \Leftarrow rt_{paused} + 1$
else if $csc == REUSED$ **then**
  $rt_{paused} \Leftarrow rt_{paused} - 1, rt_{running} \Leftarrow rt_{running} + 1$
else if $csc == EVICTED$ **then**
  $rt_{paused} \Leftarrow rt_{paused} - 1$
end if

container creation or reuse, and decreased for container pause. rt_{paused} is increased for container pause, and decreased for container reuse or eviction. Lines 1–2 increases $rt_{running}$ when a container is created. Lines 3–5 transfer $rt_{running}$ to rt_{paused} when a container is released by a finished request, and Lines 6–8 do the opposite when a container is reused. Lines 9–10 decrease rt_{paused} when a container is evicted.

According to the traced container states, CAS can choose an *affinitive worker* to avoid a *cold start* when scheduling a request.

3.3 | Holding a soon needed container by avoiding container eviction

Dispatching too many requests to a worker can evict a container serving other types of requests because of resource shortage. If a request demanding the evicted container comes, it suffers from a *cold start*. Holding a soon needed container can avoid a *cold start* for a shortly coming request.

Accurately identifying a soon needed container is hard, because workloads are usually changing and unpredictable. Currently, CAS uses a simple policy for identification: since platforms usually keep *paused containers* for a period, a *paused container* is considered as a soon needed container. In the future, CAS can integrate more complex and precise policies. For example, we can use machine learning to predict the arrival interval of requests, thereby identifying soon needed containers.

To hold a soon needed container, when a new container has to be created on a worker, CAS prefers the worker with sufficient free resources and avoids choosing a worker requiring container eviction. To determine whether a worker can accommodate an incoming request, existing schedulers check both free resources and resources occupied by *paused containers*. Instead, CAS checks free resources first and tries to find a worker with sufficient free resources. If no such worker exists and no *paused container* is suitable for the request, resources occupied by *paused containers* are considered.

Figure 6 shows how a soon needed container is held. When *request 1* and *request 2* arrive, they are dispatched to *worker 1* and *worker 2* respectively to avoid container eviction, even though *worker 1* has the capacity to accept two requests. Thus, the *paused container* on *worker 1* is held and can be reused by future requests.

By avoiding container eviction, *paused containers* can stay alive until expiration or overall resource shortage. When subsequent requests arrive, they have more chances to avoid *cold starts* by reusing *paused containers*.

3.4 | Using a soon available container in time by lazily specifying container

There are many requests whose overhead of creating a new container dominates execution time.[16] These requests probably prefer soon available containers to creating new containers, because using the soon available containers can bring shorter latency for these requests.

To use a soon available container in time, CAS intercepts event processing on a worker to perceive the appearances of available containers and specifies the first available container for a request. When there is no available container for an arriving request, CAS creates a new container on the worker, but does not immediately specify this container for the request. Instead, CAS detects an available container by intercepting event processing on the worker, while waiting for the newly created container. If an available container appears before the newly created container is ready, CAS uses this container to serve the request in time. The newly created container is not used by this request, CAS keeps it as a *paused container* to serve subsequent requests. Otherwise, the request will be processed in the newly created container when the container is ready.

Figure 7 illustrates an example of how CAS lazily specifies a container for a request. When *request 2*arrives, a new container is created. If *request 1* leaves before the new container is available, the container that has served *request 1* is specified to *request 2*. Otherwise, the newly created container will serve *request 2*.

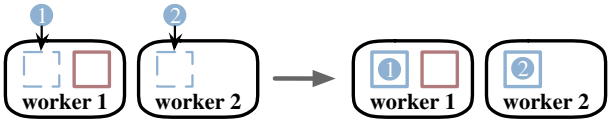


FIGURE 6 Avoiding container eviction [Colour figure can be viewed at wileyonlinelibrary.com]

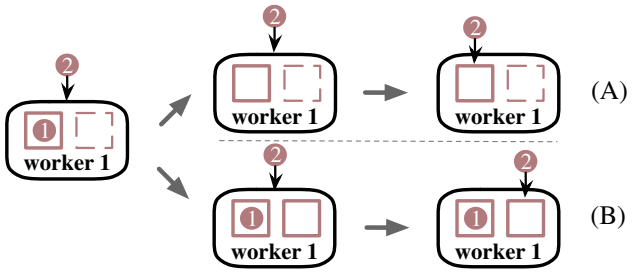


FIGURE 7 Lazily specifying container [Colour figure can be viewed at wileyonlinelibrary.com]

By lazily specifying a container for a request, CAS can use a soon available container in time to provide a shorter latency for the request, and thus less *cold starts* are required.

3.5 | Implementation in OpenWhisk

We implement a prototype of CAS on top of the open-source serverless platform, Apache OpenWhisk,[10] which is very mature and compatible with multiple infrastructures (physical machines, virtual machines, container platforms, etc.). OpenWhisk is very popular and backed by many active developers. CAS can easily be integrated into other serverless platforms with similar architecture to OpenWhisk. Recently, some platforms use service mesh[17] to manage containers and shield workers from the scheduler. For these platforms, the idea of container lifecycle awareness can also help them dispatch requests to containers more efficiently.

There are two important components in OpenWhisk: Controller and Invoker. The Controller component is the scheduler responsible for processing the HTTP method and choosing a worker to execute the function code based on a global view of available workers in the system. The workers are named Invokers, which create Docker[18] containers to run user functions.

As shown in Figure 8, there are four important components in CAS. (1) The Reporter is responsible for sending a report when a container is evicted. (2) The Tracer maintains and updates container states. (3) The Decider chooses a worker for a request. (4) The Binder is responsible for specifying a container for a request. CAS works as follows. When a request arrives, the Decider first chooses a worker for the request according to the current container states maintained by the Tracer, and the Tracer updates container states using the choice made by the Decider. Next, the Binder creates a new container for the request if no available container exists, and dispatches the request to the newly created container or an existing container in due course. When a container on a worker is evicted, the Reporter sends a report to the Tracer, and then the Tracer updates the container states.

Algorithm 2 illustrates how the Decider chooses a worker for a request. As OpenWhisk allocates CPU in proportion to memory, only memory resource is considered here. An *affinitive worker*, a worker with sufficient free resources, a worker that can accept the request by container eviction, and a random worker are tried in turn. Lines 9–10 choose an *affinitive worker* for the request. Lines 12–13 find the worker with sufficient free resources and the most containers that serve the same type of request. Lines 14–15 find a container that can accept the request by container eviction. Lines 22–23 choose a random worker.

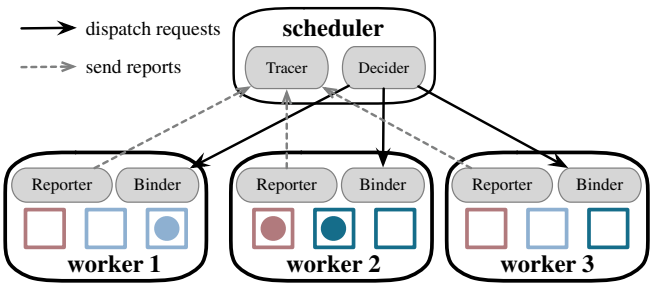


FIGURE 8 Prototype of CAS [Colour figure can be viewed at wileyonlinelibrary.com]

Algorithm 2. Worker Choosing

Input: r, request; W, workers
Output: chosen worker

```
1: hash, step ⇐ Generate_Hash(r)
2: A ⇐ null, N ⇐ −1, B ⇐ null
3: for i ⇐ hash; i ⇐ i + step; i<Size(W) do
4:     w ⇐ W[i]
5:     rt ⇐ Get_Request_Type_Info(w, r)
6:     ▷ rt_paused - number of paused containers for rt; rt_running - number of running containers for request type rt
7:     ▷ w_free - amount of free memory on w; w_free_and_cached - amount of free memory and cached memory on worker w
8:     ▷ r_required - required memory by request r
9:     if rt_paused>0 then
10:        return w
11:    end if
12:    if w_free> = r_required and rt_running>N then
13:        A ⇐ w, N ⇐ rt_running
14:    else if w_free_and_cached> = r_required and B == null then
15:        B ⇐ w
16:    end if
17: end for
18: if A! = null then
19:    return A
20: else if B! = null then
21:    return B
22: else
23:    return Random_Worker()
24: end if
```

Algorithm 3. Container Specifying

Input: r, request
Output: specified container

```
1: C ⇐ Find_Available_Container(r)
2: if C! = null then
3:     return C
4: else
5:     Create_Container(r)
6:     return Wait_For_Available_Container(r)
7: end if
```

Algorithm 3 illustrates how the Binder specifies a container for a request. If there is no available container when a request arrives, the Binder creates a new container. And the request is dispatched to the first available container by the Binder. Lines 1–3 find and specify an available container for the request. Lines 4–6 create a new container, wait for an available container to appear, and specify the first available container for the request.

There can be multiple Controllers (i.e., schedulers) in a large cluster, and each manages a few Invokers.[13] Making optimal scheduling decisions with multiple schedulers is out of the scope of this article.

4 | EVALUATION

In this section, we evaluate the effectiveness of CAS in reducing *cold starts*. First, we evaluate the overall performance of CAS using the workloads generated from real-world taxi data.[19] Then, we use synthetic workloads to further evaluate the

effectiveness of CAS in three aspects: choosing an *affinitive worker*, holding a soon needed container, and using a soon available container in time. Finally, we evaluate the overhead caused by CAS.

4.1 | Experimental setup

Platform settings. We deploy the prototype of CAS on a Kubernetes[20] cluster with 5 VMs, each with 4 cores and 8GB memory. One dedicated VM is used to run the core components of OpenWhisk including the Controller. The remaining 4 VMs serve as Invokers (workers). The user functions hosted on these Invokers are deployed on Docker[18] containers.

Parameter settings. The duration of each experiment does not exceed 5 minutes. To allow the platform to actively evict containers during these experiments, the expiration time of *paused containers* is set to 1 minute. To facilitate the statistics of the overall performance of multiple applications, each function has the same configuration. Since Singhvi et al.[16] show that most functions have small memory footprints and short execute times, each function is configured with 256MB memory and executes 500ms for processing a request. The CPU share of a function is proportional to its memory share.

Baselines. We compare CAS with the following baselines.

- Native scheduler in OpenWhisk (i.e., OW) with a hashing algorithm.
- PASch[14] (i.e., PAS), a scheduler using consistent hashing[15] and power of two choices.[21] PAS is a typical affinity-based scheduler, and it has been proved that PAS performs better than load balancing algorithms.

Metrics. To evaluate the performance of different scheduling policies, we use two key metrics: the number of *cold starts* and tail latency.

Workloads and traces. To evaluate the overall performance of CAS, we use a dataset containing data on the number of vehicles and passengers[19] to generate four workloads. This dataset can represent the traces of object or face recognition applications. The four workloads show different in two aspects. First, each workload has a different request arrival interval as shown in Figure 9. Second, each workload dispatches the same type of request. To evaluate the effectiveness of CAS in three aspects, we use synthetic workloads that send concurrent requests.

4.2 | Overall performance of CAS

We measure the overall performance of CAS with or without worker contention by concurrently running 4 workloads.

4.2.1 | Without worker contention

When the load pressure is low, each workload may have a different preferred worker, and worker contention is rare. To evaluate the performance of CAS without worker contention, we assign each workload a different preferred worker. So, there is no worker contention.

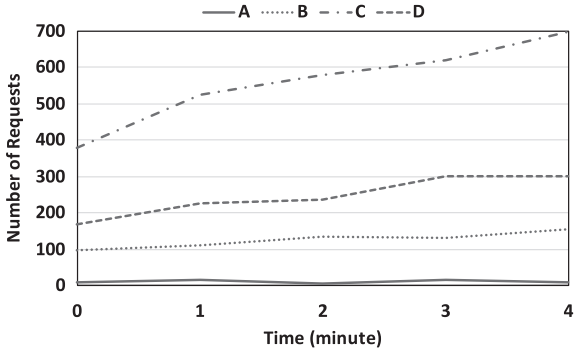
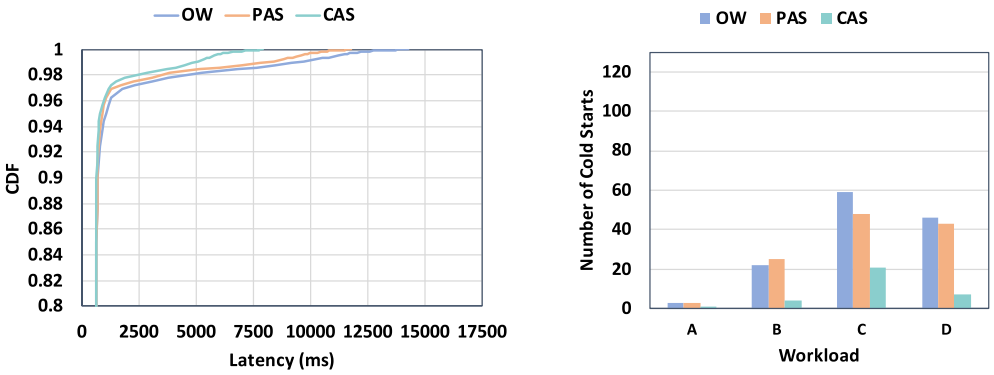


FIGURE 9 Workloads used for overall performance evaluation. The y-coordinate represents the number of requests sent in one minute for each workload



(A) Latencies in three polices. Latencies below 80th percentile are equal

(B) Cold starts in three polices

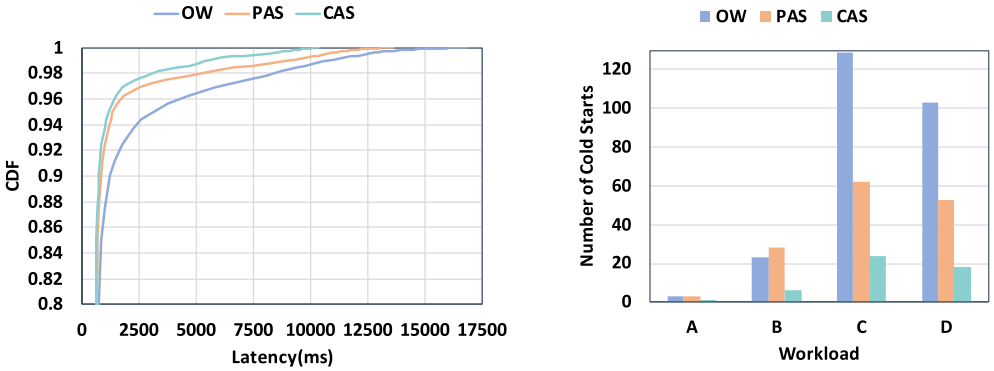
FIGURE 10 Overall performance of 4 workloads without worker contention [Colour figure can be viewed at wileyonlinelibrary.com]

As shown in Figure 10(A), CAS performs lightly better than OW and PAS, and brings a 22% reduction at 95th percentile latency compared with OW. The reason is that CAS reduces 75% *cold starts* compared with OW, as shown in Figure 10(B). When a request that cannot get an available container arrives, OW or PAS uses a new container to serve the request, and the request suffers from a *cold starts*. However, CAS uses the first available container. If an existing container becomes available before the newly created container is ready, the request can be processed with a shorter latency.

4.2.2 | With worker contention

When the load pressure is high or the platform concentrates the load on part of workers to save costs, some workloads can be assigned to the same preferred worker and compete for the worker's resources. To evaluate the performance of CAS with worker contention, workload *C* and *workload D*are assigned the same preferred worker. There exists worker contention between these two workloads.

As shown in Figure 11(A), CAS outperforms OW and PAS, and brings a 63% reduction at 95th percentile latency compared with OW. This is because CAS reduces 81% *cold starts* compared with OW, as shown in Figure 11(B). The reduction of *cold starts* is mainly reflected in *workload C* and *workload D*. If the preferred worker of workload C and *workload D* cannot accommodate more requests, containers are created on other workers. When the preferred worker has the capacity, OW and PAS create containers again on the preferred worker, while CAS reuses previously created containers on *affinitive workers*. In OW and PAS, *workload C* and *workload D* evict each other's containers for resource contention. However, CAS prefers the workers that do not need container eviction for container creation and uses soon available containers in time to provide shorter latencies.



A) Latencies in three polices. Latencies below 80th percentile are equal

(B) Cold starts in three polices

FIGURE 11 Overall performance of 4 workloads with worker contention [Colour figure can be viewed at wileyonlinelibrary.com]

4.3 | Effectiveness in three aspects

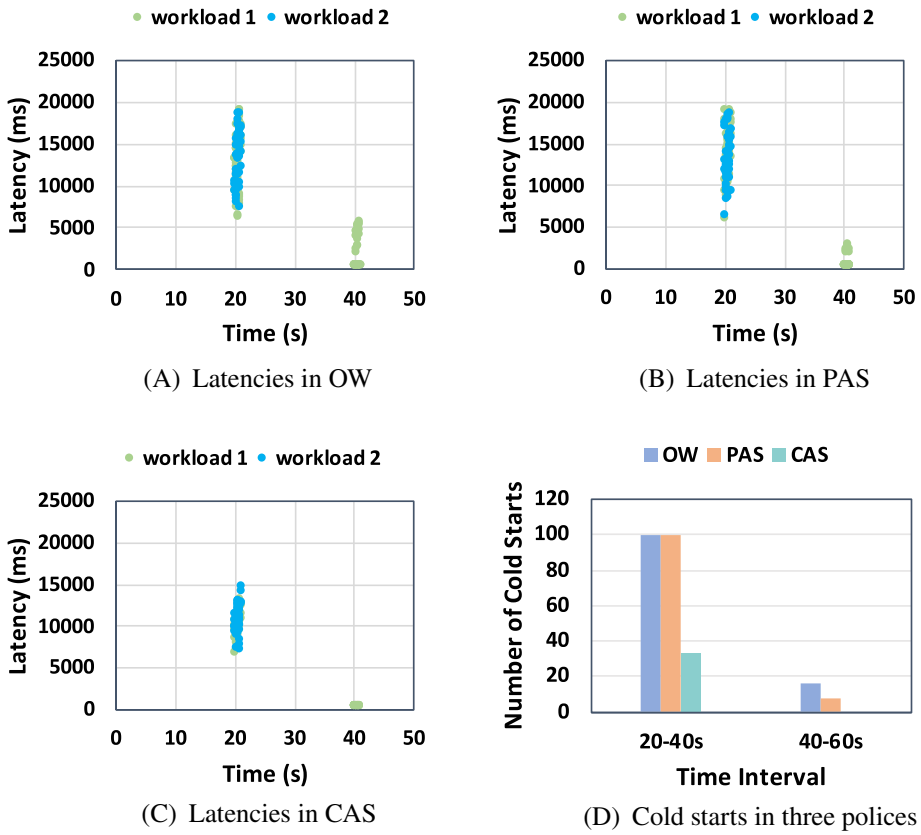
In this section, each experiment starts at 20s to facilitate the display of experimental results, and workloads send requests every 20 or 40 seconds to make sure that previous requests have been processed.

4.3.1 | Choosing an affinitive worker

Improper worker choosing cannot reuse available containers of one workload on non-preferred workers to avoid *cold starts*. To evaluate the effectiveness of CAS in choosing an *affinitive worker*, we use two workloads that have the same preferred worker. When the experiment starts, two workloads concurrently send lots of requests to allow non-preferred workers to host containers.

- *workload 1* sends 50 concurrent requests every 20 seconds from 20s.
- *workload 2* sends 50 concurrent requests every 40 seconds from 20s.

At time 20s to 40s, a few containers of *workload 1* are created on non-preferred workers, and this phase will be evaluated in section 4.3.3. At time 40s to 60s, requests of *workload 1* have chances to reuse previously created containers. As shown in Figure 12(A–C), CAS outperforms two baselines and brings a 90% reduction at 95th percentile latency compared with OW, because CAS completely avoids *cold starts*as shown in Figure 12(D). OW and PAS dispatch most requests to the preferred worker. Some available containers on non-preferred workers are not reused and thus some requests suffer from *cold starts*. Instead, CAS reuses all available containers by finding *affinitive workers*, and thus there is no *cold start*.



(A) Latencies in OW (B) Latencies in PAS

(C) Latencies in CAS (D) Cold starts in three polices

FIGURE 12 Latencies and cold starts of two workloads in three polices. The x-coordinate of a point in (A–C) is the start time of a request [Colour figure can be viewed at wileyonlinelibrary.com]

4.3.2 | Holding a soon needed container

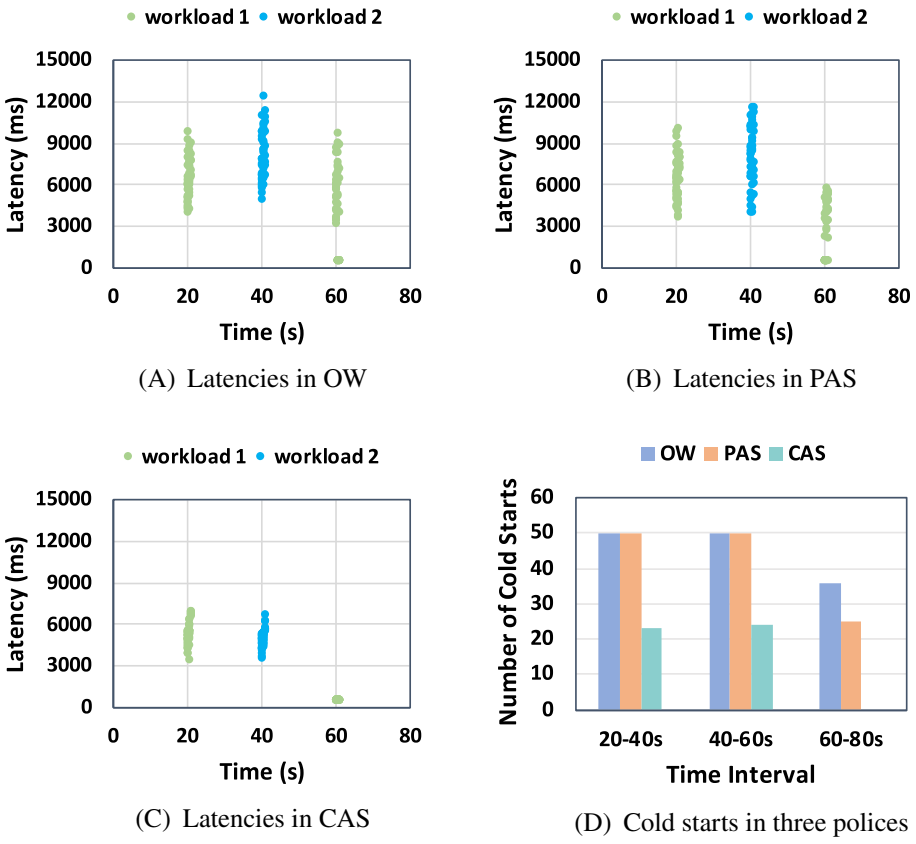
Dispatching too many requests to one worker may evict soon needed containers on this worker. To evaluate the effectiveness of CAS in holding a soon needed container, we use two workloads that have the same preferred worker to send requests alternatively. If too many requests of one workload are dispatched to a worker with soon needed containers of another, the soon needed containers may be evicted.

- *workload 1* sends 50 concurrent requests every 40 seconds from 20s.
- *workload 2* sends 50 concurrent requests every 40 seconds from 40s.

At time 20s to 40s, many containers of *workload 1* are created on the preferred worker, and this phase will be evaluated in section 4.3.3. If some containers of *workload 1* are evicted at time 40s to 60s, *cold starts*will happen at time 60s to 80s. As shown in Figure 13(A–C), CAS has the shortest latencies and brings a 94% reduction at 95th percentile latency compared with OW at time 60s to 80s. Because there is no *cold start* in CAS as shown in Figure 13(D). At time 40s to 60s, OW and CAS dispatch most requests of *workload 2* to the preferred worker, and some containers of *workload 1* are evicted to create containers for *workload 2*. When requests of *workload 1* arrive at 60s, many *cold starts* happen. However, CAS holds previously created containers of *workload 1* by creating containers for *workload 2* on workers with sufficient free resources. Therefore, *cold starts* are completely avoided.

4.3.3 | Using a soon available container in time

While creating a container for a request, another request may release an available container, and using this container can provide a shorter latency for the request. To evaluate the effectiveness of CAS in using a soon available container in time,



(A) Latencies in OW

(B) Latencies in PAS

(C) Latencies in CAS

(D) Cold starts in three polices

FIGURE 13 Latencies and cold starts of two alternative workloads in three polices. The x-coordinate of a point in (A–C) is the start time of a request [Colour figure can be viewed at wileyonlinelibrary.com]

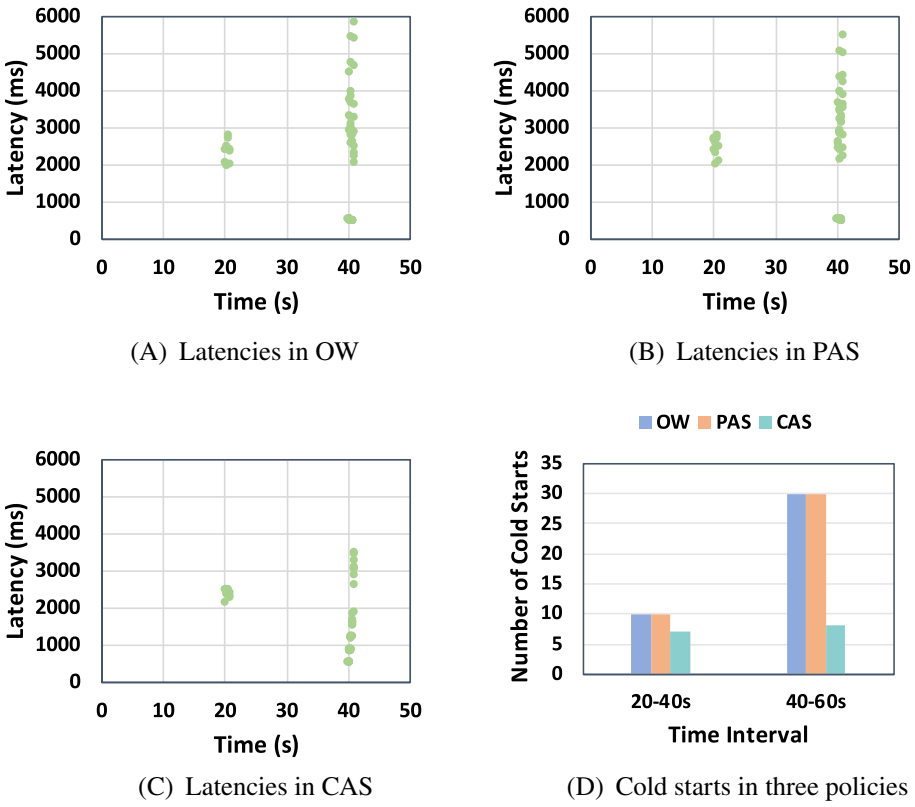


FIGURE 14 Latencies and cold starts in three polices. The x-coordinate of a point in (A–C) is the start time of a request [Colour figure can be viewed at wileyonlinelibrary.com]

we use one workload sending multiple requests each time. A previously created container may finish processing a request before a newly created container is ready.

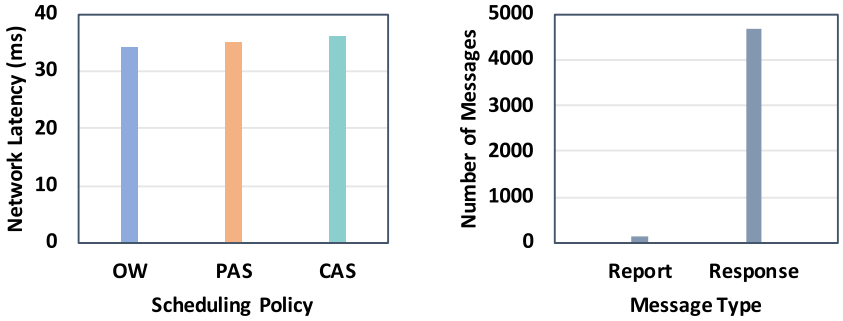
- *workload 1* sends 10 concurrent requests at 20s, and then sends 50 concurrent requests every 20 seconds.

As shown in Figure 14(A–C), CAS outperforms OW and PAS, especially at time 40s to 60s. At time 20s to 40s, CAS brings a 10% reduction at 95th percentile latency compared with OW. And the reduction is 39% at time 40s to 60s. Because there are the least *cold starts* in CAS, as shown in Figure 14(D). OW or PAS creates a new container for an arriving request that cannot get an available container, and the request suffers from a *cold start*. However, CAS detects an available container released by a finished request while creating a container. If an available container appears, the request is immediately processed in this container instead of suffering from a *cold start*. At time 20s to 40s, the number of containers is small and the appearances of available containers are rare. At time 40s to 60s, the appearances are common and *cold starts* are efficiently reduced.

4.4 | Overhead of CAS

In CAS, the operation of worker choosing is $O(n)$, which is the same as OW. Hence, CAS does not add extra CPU overhead. To maintain container states of one type of requests on one worker, two integers are needed. So, CAS requires a little extra memory. CAS sends extra report messages, which can cause network overhead. Report messages are sent in the same way as the response messages of requests are sent. We measure the network latencies in three policies by calculating the difference between the total request latency and the processing time, and count the number of report messages and response messages in CAS. The experimental setup is the same as section 4.2.2.

As shown in 15(A), the network latencies are very close in three policies, and CAS only adds 2 ms extra latency compared with OW. We can observe that the ratio of report messages to response messages is 3% in Figure 15(B). Each



(A) Network latencies (B) Number of Messages

FIGURE 15 Network latencies in three policies and number of report messages and response messages in CAS. The latencies include the time spent in the scheduler [Colour figure can be viewed at wileyonlinelibrary.com]

report message uses several bytes to describe container eviction and the amount of data is small, which does not cause significant network overhead.

5 | RELATED WORKS

There are a lot of works about the cold start problem in serverless computing. These efforts are focused on three aspects: speeding up the startup of containers with optimized sandbox technologies, reusing containers by efficiently scheduling requests, and pre-creating containers to hide long latency.

In general, sandbox optimizations can be divided into three types: container-based, VM(Virtual Machine)-based, and unikernel-based.[22] (1) *Container-based*. Oakes et al.[7] optimize containers by removing two namespaces.[23] Akkus et al.[8] run multiple functions in a single container. Pre-warming[9] pre-creates containers with customized language runtime. These solutions weaken the isolation or flexibility of containers. (2) *VM-based*. AWS uses a light VM monitor to speed up the startup of VMs. However, VMs have poor performance compared with containers.[2] (3) *Unikernel-based*. Cadden et al.[24] use snapshots to provide fast startup for unikernels. However, unikernels are not flexible compared with containers. For example, it is difficult to debug applications in unikernels. These sandbox technologies may sacrifice isolation, flexibility, or performance, and cannot avoid the delay caused by loading function code.

Existing works for scheduling mainly focus on function interference on a worker. Suresh et al.[25] dynamically adjust the CPU share of function. Kaffes et al.[26] dispatch requests according to idle CPU cores on workers. For container reuse, OpenWhisk[10] uses hashing to maximize container reuse among the same type of requests. Aumala et al.[14] use consistent hashing and power of two choices[21] to make a trade-off between load balancing and container reuse. Zhang et al.[27] manage to predict the latency of workers and try to dispatch requests to workers that show the lowest latency. Jindal et al.[12] design FDN that provides Function-Delivery-as-a-Service (FDaaS), delivering the function to the right target worker in homogeneous clusters. These solutions lack consideration or awareness of container lifecycle and may make improper scheduling decisions.

Besides, some works[28-31] manage to schedule pre-warm containers which are pre-created with running environment and language runtime to prevent the latency caused by *cold starts*. However, these works mainly focus on when containers should be started and how to dispatch requests to these containers. *Soon needed containers* and *soon available containers* are not considered.

Different from existing schedulers, CAS is aware of container lifecycle, and schedules requests according to different lifecycle phases of containers to avoid *cold starts* caused by improper worker choosing or container specifying.

6 | CONCLUSIONS AND FUTURE WORKS

Cold starts hamper the elasticity of serverless computing. A primary policy for reducing *cold starts* is relying on schedulers to reuse *paused containers*. In this research, we analyze how and why existing schedulers fail to work well. Based on our analysis, a container lifecycle-aware scheduling strategy named CAS is proposed. When scheduling a request, CAS

prefers the worker that hosts available containers or does not need container eviction to avoid *cold starts* for the current request and shortly coming requests. On one worker, CAS uses the first available container to serve a request. Hence, many requests can be served by available containers or soon available containers instead of suffering from *cold starts*. CAS is implemented on OpenWhisk.[10] The experiment results show that CAS reduces 81% *cold starts* and therefore brings a 63% reduction at 95th percentile latency compared with native scheduling strategy in OpenWhisk when there is worker contention between workloads, and does not add significant performance overhead.

Currently, the policies of identifying soon needed containers and evicting containers are simple. In the future, we will integrate request prediction into them. The scheduler dispatches request on the manner of FIFO (First In First Out), and reordering to dispatch requests that can directly get available containers first may be a better choice. We will explore reordering requests with deadline control.

ACKNOWLEDGMENT

This work is supported by National Key Research and Development Program under grant 2018YFB1004805, National Science Foundation of China under grants No.62032008 and 61872155.

ORCID

Song Wu https://orcid.org/0000-0001-8690-127X

REFERENCES

1. Dua R, Raja AR, Kakadia D. Virtualization vs containerization to support PaaS. Paper presented at: Proceedings of the International Conference on Cloud Engineering, Boston, MA, USA; 2014:610-614.
2. Morabito R, Kjällman J, Komu M. Hypervisors vs. lightweight virtualization: a performance comparison. Paper presented at: Proceedings of the International Conference on Cloud Engineering, Tempe, AZ, USA; 2015:386-393.
3. Thalheim J, Bhatotia P, Fonseca P, Kasikci B. Cntr: lightweight OS containers. Paper presented at: Proceedings of the Annual Technical Conference, Boston, MA, USA; 2018:199-212.
4. Wang L, Li M, Zhang Y, Ristenpart T, Swift MM. Peeking behind the curtains of serverless platforms. Paper presented at: Proceedings of the Annual Technical Conference, Boston, MA, USA; 2018:133-146.
5. Python. 2019. https://www.python.org/;
6. I'm afraid you're thinking about AWS Lambda cold starts all wrong. 2019. https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-%25starts-all-wrong-7d907f278a4f
7. Oakes E, Yang L, Zhou D, et al. SOCK: rapid task provisioning with serverless-optimized containers. Paper presented at: Proceedings of the Annual Technical Conference, Boston, MA, USA; 2018:57-70.
8. Akkus IE, Chen R, Rimac I, et al. SAND: towards high-performance serverless computing. Paper presented at: Proceedings of the Annual Technical Conference, Boston, MA, USA; 2018:923-935.
9. Squeezing the milliseconds: how to make serverless platforms blazing fast!; 2019. https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-ser%verless-platforms-blazing-fast-aea0e9951bd0
10. Apache OpenWhisk; 2019. https://github.com/apache/openwhisk
11. Using NGINX as HTTP load balancer; 2019. http://nginx.org/en/docs/http/loadbalancing.html
12. Jindal A, Gerndt M, Chadha M, Podolskiy V, Chen P. Function delivery network: extending serverless computing for heterogeneous platforms. *Softw Pract Exper*. 2021. https://doi.org/10.1002/spe.2966
13. Design consideration of apache OpenWhisk; 2019. https://cwiki.apache.org/confluence/display/OPENWHISK/Design+considerat%ion
14. Aumala G, Boza EF, Ortiz-Avilés L, Totoy G, Abad C. Beyond load balancing: package-aware scheduling for serverless platforms. Paper presented at: Proceedings of the International Symposium on Cluster, Cloud and Grid Computing, Larnaca, Cyprus; 2019:282-291.
15. Karger DR, Sherman A, Berkheimer A, et al. Web caching with consistent hashing. *Int J Comput Telecommun Netw*. 1999;31(11-16):1203-1213.
16. Singhvi A, Houck K, Balasubramanian A, Shaikh MD, Venkataraman S, Akella A. Archipelago: a scalable low-latency serverless platform. *Comput Res Repository*. 2019;abs/1911.09849. http://arxiv.org/abs/1911.09849.
17. Service mesh; 2019. https://istio.io/latest/docs/concepts/what-is-istio/
18. Boettiger C. An introduction to docker for reproducible research. *Oper Syst Rev*. 2015;49(1):71-79.
19. TLC trip record data; 2019. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
20. Kubernetes; 2019. https://github.com/kubernetes/kubernetes
21. Mitzenmacher M. The power of two choices in randomized load balancing. *Trans Parallel Distrib Syst*. 2001;12(10):1094-1104.
22. Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: library operating systems for the cloud. Paper presented at: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, TX, USA; 2013:461-472.
23. NAMESPACES(7); 2019. http://man7.org/linux/man-pages/man7/namespaces.7.html
24. Cadden J, Unger T, Awad Y, Dong H, Krieger O, Appavoo J. SEUSS: rapid serverless deployment using environment snapshots. *Comput Res Repository*. 2019;abs/1910.01558. http://arxiv.org/abs/1910.01558.

25. Suresh A, Gandhi A. FnSched: an efficient scheduler for serverless functions. Paper presented at: Proceedings of the International Workshop on Serverless Computing, Davis, CA, USA; 2019:19-24.
26. Kaffes K, Yadwadkar NJ, Kozyrakis C. Centralized core-granular scheduling for serverless functions. Paper presented at: Proceedings of the Symposium on Cloud Computing, Santa Cruz, CA, USA; 2019:158-164.
27. Zhang M, Krintz C, Wolski R. Edge-adaptable serverless acceleration for machine learning Internet of Things applications. *Softw Pract Exper*. 2020. http://dx.doi.org/10.1002/spe.3012.
28. Ling W, Ma L, Tian C, Hu Z. Pigeon: a dynamic and efficient serverless and faas framework for private cloud. Paper presented at: Proceedings of the International Conference on Computational Science and Computational Intelligence, Las Vegas, NV, USA; 2019:1416-1421.
29. Mohan A, Sane H, Doshi K, Edupuganti S, Nayak N, Sukhomlinov V. Agile cold starts for scalable serverless. Paper presented at: Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing, Renton, WA, USA; 2019.
30. Fuerst A, Sharma P. FaasCache: keeping serverless computing alive with greedy-dual caching. Paper presented at: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA; 2021:386-400.
31. Kumara I, Han J, Colman A, van den Heuvel WJ, Tamburri DA, Kapuruge M. SDSN@RT: a middleware environment for single-instance multitenant cloud applications. *Softw Pract Exper*. 2019;49(5):813-839.

How to cite this article: Wu S, Tao Z, Fan H, et al. Container lifecycle-aware scheduling for serverless computing. *Softw Pract Exper*. 2021;1–16. https://doi.org/10.1002/spe.3016