S-Cache: Function Caching for Serverless Edge Computing

Chen Chen
University of Cambridge
Cambridge, UK
cc2181@cam.ac.uk

Lars Nagel
Loughborough University
Loughborough, UK
l.nagel@lboro.ac.uk

Lin Cui
Jinan University
Guangzhou, China
tcuilin@jnu.edu.cn

Fung Po Tso
Loughborough University
Loughborough, UK
p.tso@lboro.ac.uk

ABSTRACT

Serverless edge computing uses an event-driven model in which Internet-of-Things (IoT) services are run in short-lived, stateless containers only when invoked, leading to significant reduction of resource utilization. However, a cold-start of a container can take up to several seconds which significantly degrades the response time of serverless applications. Container caching can mitigate the cold-start problem at the cost of extra computing resources which violates the spirit of serverless computing. Therefore, we need to balance the cold-start overheads with the extra resource utilization for serverless edge computing. Nevertheless, the diverse ranges of containers lead to different cold-start overheads, resource consumption and invocation frequencies and these characteristics of containers are largely overlooked by existing caching policies. In this paper, we study the request distribution and caching problem for serverless edge computing. We devise an online request distribution algorithm with performance guarantee and present an adaptive caching policy which incorporates container frequency, container size and cold-start time. Via real-system implementation, the superiority of the proposed algorithm is verified by comparing with existing caching policies, including fixed caching and histogram based policies. Our results show that the proposed algorithm reduces both the average response time and cold-start frequency by a factor of 3 compared to current approaches.

CCS CONCEPTS

• **Networks** → **Cloud computing**; **Network management**.

KEYWORDS

Serverless Computing, Function as a Service, Caching

ACM Reference Format:
Chen Chen, Lars Nagel, Lin Cui, and Fung Po Tso. 2023. S-Cache: Function Caching for Serverless Edge Computing. In *6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3578354.3592865

1 INTRODUCTION

Edge computing is a cluster of distributed edge nodes with limited hardware resources where tasks are generated (e.g., sensing/actuation) and the computation are executed. Meanwhile, with the accelerated utilization of Function as a Service (FaaS) in the edge computing model, application code is executed by edge nodes in the form of user-defined "functions" [16], [18]. Serverless functions are usually single purpose and stateless containers, only created when responding to event-driven tasks. The users benefit from the fast response time and pay-as-you-use pricing model in serverless paradigm. Serverless services are provided by all major computing platforms such as Amazon Lambda [12], Google Functions [5] and Microsoft Azure [8]. The applications include API services, web services, image recognition and many others.

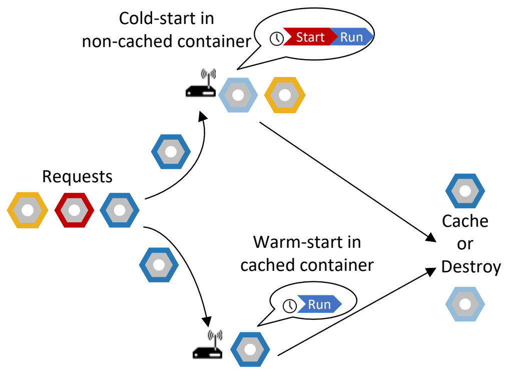

Cold-start in non-cached container
Start
Run
Requests
Cache or Destroy
Warm-start in cached container
Run

Figure 1: Request distribution and caching in serverless edge computing

A key challenge in serverless edge computing is the cold-start delay incurred by the initialization of the container. This startup process takes a considerable amount of time, which adds to the overall response time observed by the user. The cold-start issue will destroy the spirit of edge computing as an important strength of edge computing is to reduce the latency and offer real-time responsiveness [3]. To mitigate the cold-start delay, a common approach is container caching which keeps the container alive or "warm" for a short period of time after serving an invocation so that future invocations do not have to experience a cold-start. As shown in Figure 1, if a request is distributed to a newly created (i.e. non-cached) container, a cold start is required before the request is served. If the hardware resources at the edge node are saturated, a cached container with low priority must be evicted before a new container can be created. If a request is placed into a cached container, the request is processed without a cold start. After the request is finished, the container is either cached or destroyed.

The price for container caching is the use of extra computational resources at the edge nodes and thus deteriorates the resource efficiency of serverless computing where resources are only allocated when triggered by an event. Furthermore, the resource capacity in edge computing is much more limited than centralized clouds, and container caching may not suffice to satisfy all service requests due to resource availability. Some existing works [17], [16], [14] resort to container caching, but fail to capture the characteristics of different containers. In particular, the keep-alive time should depend on the characteristics of containers due to the diverse range of containers with different start-up overheads, resource footprint (e.g., memory consumption) and invocation frequencies. Hence, container caching policies must balance the overhead of caching a container with extra resource usage which has not been explicitly studied in the existing literature.

In this work, we study the problem of request distribution and caching in serverless edge computing. Our key insight is that container caching is analogous to object caching because caching a container reduces the cold-start delay in the same way as caching an object reduces the access latency. Hence, we devise a caching policy based on cold-start time, resource footprint and invocation frequency of containers to dynamically provision and destroy containers between edge nodes, aiming to minimize the average response time of requested applications. First, we use the Long-Short-Term-Memory model (LSTM) to predict the number of requests in each time interval. Then, we propose a placement algorithm to distribute the incoming requests to edge nodes with performance guarantee. After that, we devise a Cold-start-Frequency Caching policy (CFC) to decide the container priority in the cache. Our main contributions are summarized as follows.

- We formulate a request distribution problem with container caching as an Integer Linear Programming (ILP) problem which jointly considers cold-start delay, processing delay and link delay. We devise an online competitive algorithm with performance guarantee.
- We show the similarity between object caching and container caching, and devise a container caching policy based on cold-start time, size and invocation frequency of containers, aiming to reduce average response time of requests.
- We conduct extensive experiments based on real-world datasets. We implement the prototype in a real-world platform Knative and demonstrate that our algorithm outperforms the state-of-the-art caching policies by a factor of 3 in terms of average response time and cold-start frequency.

The remainder of this paper is organized as follows. We first provide an overview of the related work before discussing the system model and the problem. Subsequently, we present the proposed algorithm, the experiments conducted and their results.

2 RELATED WORK

A number of works investigate the request distribution in edge computing. Poularakis *et al.* [15] propose a service placement and flow routing algorithm based on randomized rounding to maximize the number of requests handled by base stations in mobile edge computing. Farhadi *et al.* [6] propose a two-time-scale algorithm to jointly optimize the service placement and request scheduling. Gu *et al.* [9] present a layer-aware container placement and request scheduling algorithm, aiming to maximize the number of placed containers and the overall throughput. However, all these works focus on service placement and ignore cold-start delay and container caching, which is why they are not applicable to the container caching problem for serverless computing.

In the context of edge computing, content caching is applied to reduce the latency in content delivery. A wide range of works, e.g. [19] and [2], study content caching with the aim of reducing the content fetching time. For instance, Zhou *et al.* [19] optimize service caching and task offloading by using Lyapunov optimization and the dependent rounding technique. Container caching, however, is different from content caching as cached content can serve multiple requests simultaneously while a container serves only one request Existing content caching policies are therefore not directly applicable to container caching. Shahrad *et al.* [17] propose a caching policy for mitigating the cold-start problem by characterizing the pattern of function invocations. Li *et al.* [14] map the container caching problem cost to the ski-rental problem to optimize the deployment cost. Roy *et al.* [16] use low-cost nodes to cache containers which keeps a larger number of containers warm under the same cost budget. However, none of the above consider the characteristics of containers (e.g., the cold-start time, the size and frequency of invoked containers) in the caching policies. Furthermore, our work manages request distribution by jointly considering the link delay, processing delay and cold-start delay for serverless computing, which has not been explicitly studied in existing literature.

3 SYSTEM MODEL

We consider an edge network consisting of multiple edge nodes at different geographical locations which we refer to as a cluster. These edge nodes have computational resources and serve service requests generated at the edge nodes. Each request is served in a single container. The serverless system forwards requests to edge nodes of the cluster based on its placement policy.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote the physical network. $\mathcal{V}$ denotes the set of edge nodes and $\mathcal{E}$ denotes the set of links in the system. We use integers $n \in \mathcal{N}$ to denote the type n of a container. Each edge node $v \in \mathcal{V}$ is equipped with a certain amount of hardware resource (e.g., memory) denoted by C_v. The complete time span $\mathcal{T} = \{1, ..., T\}$ of the execution is divided into time slots $t \in \mathcal{T}$. Edge nodes in the system process incoming requests by invoking serverless containers that are attached to the edge nodes. $r \in \mathcal{R}$ represents a request where $\mathcal{R}$ denotes the set of requests.

4 PROBLEM DESCRIPTION

We consider three categories of delays that are essential to the system performance: link delay, processing delay and cold-start delay.

If a request is generated at edge node v and is distributed to edge node v', we use $l_{v,v'}$ to denote the link delay between the edge node v and the edge node v'. Let $m^n_{v \to v',t}$ denote the number of type n requests generated at node v and distributed to node v'. The total link delay for requests in time interval t is given by:

$$D^L_t = \sum_{v,v' \in \mathcal{V}} \sum_{n \in \mathcal{N}} l_{v,v'} m^n_{v \to v',t} \tag{1}$$

The processing delay is incurred by the processing time at the type n container. The total processing delay for requests in time interval t is given by:

$$D_t^P = \sum_{v\in\mathcal{V}}\sum_{n\in\mathcal{N}} p^n m_{v,t}^n \tag{2}$$

where $m_{v,t}^n$ denotes the number of type n requests distributed to node v in time interval t. p^n denotes the average processing time of a type n request.

The cold-start delay is incurred by the container initialization. Hence, the cold-start delay of all requests is given by:

$$D_t^C = \sum_{v\in\mathcal{V}}\sum_{n\in\mathcal{N}} d^n max\{m_{v,t}^n - a_{v,t}^n, 0\} \tag{3}$$

where $a_{v,t}^n = \max\{a_{v,t-1}^n, m_{v,t-1}^n\} - y_{v,t-1}^n$ is the number of active type n containers at node v at the beginning of time interval t. d^n represents the cold-start delay for instantiating a container of type n. $y_{v,t-1}^n$ denotes the number of containers that were destroyed at edge node v in the end of time interval $t-1$.

The total response time of the system is the sum of all delays:

$$D_t = 2D_t^L + D_t^P + D_t^C \tag{4}$$

4.1 Problem Formulation

The objective is to minimize the average response time taken over all time slots. This problem can be formulated as an integer linear programming problem:

$$\begin{aligned}
min \quad & \frac{1}{|\mathcal{R}|}\sum_{t=1}^{T} D_t \\
s.t. \quad & \sum_{n\in\mathcal{N}} c_n max\{a_{v,t}^n, m_{v,t}^n\} \le C_v, \quad \forall v, \forall t \\
& y_{v,t}^n \in [0, max\{a_{v,t}^n, m_{v,t}^n\}], \quad \forall v, \forall n, \forall t \\
& \sum_{v'\in\mathcal{V}} m_{v\to v',t}^n = \lambda_{v,t}^n, \quad \forall v, \forall n, \forall t
\end{aligned} \tag{5}$$

The first constraint ensures that required hardware resources do not exceed the resource limit C_v where c_n denotes the amount of resource required by a type n container. The second constraint guarantees that the number of destroyed containers is not greater than the number of active containers. The third constraint ensures that each request is distributed to only one edge node; $\lambda_{v,t}^n$ is the number of type n requests generated from edge node v in time interval t.

We show that the Generalized Assignment Problem (GAP) [4], which is known to be NP-hard, can be reduced to our problem. The GAP problem is assigning a number of K tasks to a set of J agents with minimum cost. Let b_k denote the size of the task k where we can map b_k to c_n. Let each edge node v denote an agent j with a hardware resource capacity $P_j = C_v$. Assuming that the cost of assigning a job is the response time of a request, our problem becomes finding request distribution to minimize the cost. Hence, the GAP problem is a special case of our problem.

5 REQUEST DISTRIBUTION AND CACHING

5.1 Invocation Prediction Using LSTM

Nearly 98% of the serverless functions (in the Microsoft Azure serverless trace) show some kind of periodic pattern in their requests (number of a specific function requested per time interval) [16]. Most existing works only predict the inter-arrival time of a function rather than the number of requested containers [7]. Lin *et al.* propose a histogram-based prediction method that ignores the varying inter-arrival time between two successive function requests [13]. In this work, we use the Long Short Term Memory (LSTM) network to predict the number of containers invoked in the next time interval. We have used one layer, 100 features in the hidden state and 150 epochs for training the dataset. In other words, the proposed forecasting scheme is used to predict $\lambda_{v,t}^n$ which denotes the number of type n requests at edge node v. The proposed LSTM model achieves a predictive accuracy of up to 95%.

5.2 Caching-based Keep-alive Policy

Cold-start Frequency Caching (CFC): One of the key insights of this work is that caching containers is analogous to keeping object in a cache. Caching a container reduces its cold-start delay in the same way as caching an object reduces the access latency. If all server resources are occupied, the problem of which container to destroy is analogous to which object to evict from a cache. Our goal is to reduce the cold-start delay which is analogous to the high-level goal of improving the object access time.

We take advantage of the caching analogy to provide deeper insights into the trade-offs in response time optimization and caching policies. Our insight is that we can take advantage of existing observations and methods in object caching to devise effective caching policies that provide us with a well-established starting point for formulating and improving container caching. We propose a caching policy based on the characteristics of containers (e.g., invocation frequency, cold-start time and resource footprints). Our policy decides which container to destroy if a new container is to be created and there are insufficient resources available.

Priority Calculation: In the context of container caching, we create a caching priority for each container which is calculated based on the frequency of invocation, the cold-start time and the size. This is because containers with high invocation frequencies are more likely to be reused in the future. Also, the cold-start time of containers significantly impacts the response time in the system. Finally, the size (amount of required hardware resource of containers) determines how many containers can be cached as the resource capacity is scarce in edge computing.

$$Priority = Clock + \frac{Freq \times ColdStartTime}{Size} \tag{6}$$

Clock: We maintain a "logical clock" to reflect the recency of invocation. The clock is updated after every time interval. When a container is used, we assign the clock to the container and update the priority. Thus, containers, that are used recently, will have larger clock values and hence higher priorities.

Frequency: We define frequency as the number of times a particular function is requested until the current time interval. A particular function is the set of containers of the same type. The frequency is reset to zero if all the containers of a function are destroyed. The priority is proportional to the frequency, and hence more frequently invoked containers are kept alive for longer.

ColdStartTime: ColdStartTime denotes the start-up time of containers in our case. This parameter captures the benefit of caching

Algorithm 1: Request Distribution

```
foreach $t \in \mathcal{T}$ do
    foreach $v \in \mathcal{V}$ do
        foreach $n \in \mathcal{N}$ do
            Predict the number of type $n$ requests $\lambda^n_{v,t}$ at node $v$;
            Sort all nodes $v'$ by $l_{v,v'}$ in ascending order and store
              them in the sorted list $\mathcal{V}'$;
            Update the priority for type $n$ containers acc. to
              Equation 6.
            if $\lambda^n_{v,t} \leq a^n_{v,t}$ then
                Assign all $\lambda^n_{v,t}$ requests to current node $v$.
                $m^n_{v,t} \leftarrow \lambda^n_{v,t}$.
                The number of remaining to be placed containers
                  $k^n_{v,t}$ is set to zero.
                Update the priority for type $n$ containers acc. to
                  Equation 6.
            else if $\lambda^n_{v,t} \geq a^n_{v,t}$ then
                $m^n_{v,t} \leftarrow a^n_{v,t}$.
                $k^n_{v,t} \leftarrow \lambda^n_{v,t} - a^n_{v,t}$.
                foreach $v' \in \mathcal{V}'$ do
                    if $2l_{v,v',t} \leq d^n$ then
                        if $a^n_{v',t} \geq k^n_{v,t}$ then
                            $m^n_{v',t} \leftarrow k^n_{v,t}$.
                            $k^n_{v,t} \leftarrow 0$.
                            break;
                        else
                            $m^n_{v',t} \leftarrow a^n_{v',t}$.
                            $k^n_{v,t} \leftarrow k^n_{v,t} - a^n_{v',t}$.
                        end
                    end
                end
                if $k^n_{v,t} \neq 0$ then
                    Call Algorithm 2, instantiate $k^n_{v,t}$ new
                      containers at current node $v$.
                end
                Update the priority for type $n$ containers acc. to
                  Equation 6.
            end
        end
    end
end
```

a container. The priority is therefore proportional to the cold-start time of a container.

Size: Size is the memory usage of a container because cached containers are kept alive in memory. As the priority is inversely proportional to the size, larger containers are terminated before smaller containers. Thus, it is more expensive to keep larger containers alive instead of smaller containers.

Serverless-specific considerations: Recall that a specific function may have multiple containers due to concurrent function invocations. Hence, the proposed caching policy calculates termination priority at the function level but terminates at the container level. Each edge node has a priority list of functions. We assume that all containers of a function are identical because they have the same image, footprint and cold-start delay. Thus, any one of the identical containers can be terminated at a single edge node.

Algorithm 2: Instantiate New Container - INC

```
if $\sum_{n\in\mathcal{N}} c_n max\{a^n_{v,t}, m^n_{v,t}\} > C_v$ then
    Destoy the container with the lowest priority until the
      hardware resource is sufficient;
    Create a new type $n$ container;
    Update priority for the type $n$ container;
else
    Create a new type $n$ container;
    Update priority for the type $n$ container;
end
```

5.3 Request Distribution

Intuitively, requests created in a particular edge node should be served by the cached containers if there are sufficient "warm" containers at the current edge node. Otherwise, the system can either create new containers at the current edge node at the cost of cold-start delay or distribute the requests to nearby edge nodes with sufficient cached containers at the cost of extra link delay.

As illustrated in Algorithm 1, at the end of time interval $t-1$, we use the LSTM to predict the number of requests $\lambda^n_{v,t}$ generated at node v for type n requests in the time interval t. After that, we sort the edge nodes in a list based on the link delay to the current node. If the number of type n requests $\lambda^n_{v,t}$ is smaller than the number of cached containers $a^n_{v,t}$, we assign all the requests to cached containers. Otherwise, we offload the unprocessed requests to neighbor nodes in which the link delay $2l_{v,v'}$ between the current node and the neighbor node is smaller than the cold-start delay of the container. In this case, offloading some of the requests to neighbor nodes is more beneficial than creating new containers in the current node. If, after checking available nodes, there are still unprocessed requests, new containers are created at the current node which is delayed due to the cold-start. If the current node cannot create new containers due to resource constraints, low-priority containers are terminated using the proposed caching policy to free resources for the new containers.

Algorithm 2 shows the algorithm of creating a new type n container. If the residual resource capacity at node v is insufficient to create a new type n container, the algorithm terminates containers with the lowest priority in the cache, until there are sufficient resources for creating a new type n container.

Theorem 5.1. *The worst-case response time for handling a type n request is bounded by* $\max\{1+\frac{d^n}{p^n}, 1+2\frac{l_{v,v'}}{p^n}\}$.

Proof. The offline optimal solution to process a type n request is distributing the request to the generated node with an available cached type n container. In this case, the link delay is zero as the request is placed on the node that generated the request, and the cold-start delay is also zero as the request uses a cached container. Hence, we obtain the lower bound for the response time of a single request as:

$$OPT(n) \geq p^n \tag{7}$$

After that, we consider the maximum response time produced by the proposed algorithm. In the worst case, the current node has no cached type n containers. If a neighbor node meets the requirement that the extra link delay is smaller than the cold-start delay, the neighbor node processes the request by using a cached container. Otherwise, the current node creates a new type n container for the request at the cost of cold-start latency. We can define the total response time of the worst case as:

$$WORST(n) = \begin{cases} p^n + d^n & if \quad 2l_{v,v'} \geq d^n \\ p^n + 2l_{v,v'} & if \quad 2l_{v,v'} < d^n \end{cases} \tag{8}$$

We define $f(n) = WORST(n)/OPT(n)$. Hence, the maximal value of $f(n)$ denotes the worst-case competitive ratio. $f(n)$ can be represented as follows:

$$f(n) = \begin{cases} 1 + \frac{d^n}{p^n} & if \quad 2l_{v,v'} \geq d^n \\ \\ 1 + 2\frac{l_{v,v'}}{p^n} & if \quad 2l_{v,v'} < d^n \end{cases} \tag{9}$$

In both cases, $f(n)$ is a constant function. Hence, the maximum $f(n)$ is given by $\max\{1 + \frac{d^n}{p^n}, 1 + 2\frac{l_{v,v'}}{p^n}\}$. □

6 PERFORMANCE EVALUATION

6.1 Experiment Evaluation

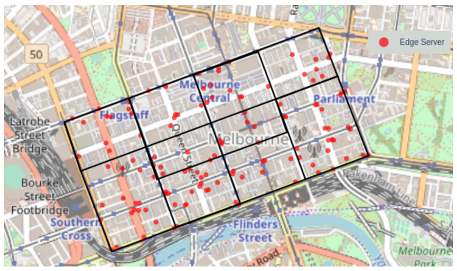


Figure 2: Graphical map of the edge servers in Melbourne CBD area

Topology: We use the EUA dataset [11] which contains the information of edge servers in the Melbourne CBD area. We divide this area into 10 zones as shown in Figure 2 and each zone is represented by one edge node in the implementation. The link delay between zones is measured by the geographical distance between the zone's center coordinates.

Requests: We use the Azure dataset [1] to create the request arrival pattern. This dataset contains the invocations of functions on Microsoft Azure for 14 days. The workload in each zone is generated using the Zipf distribution [14] with the exponent ranging from 0.5 to 1.5. We select the four applications which are most frequently invoked and map them to four containers shown in Table 1. Also, we reduce the number of invocations by 10000× to adapt to the system capacity.

Containers: We build four types of containers derived from AWS Lambda functions, as shown in Table 1. According to our measurement, the cold-start delay d^n lasts between 4.89 and 5.34 seconds, the average processing time p^n is between 20 and 2076 milliseconds and the memory sizes are between 55 and 332 MB.

Testbed experiments: To demonstrate that our algorithm can be easily implemented in real systems, we use the Knative implementation [10] which is an enterprise-level solution for serverless

Application Name	Processing Time	Cold Start Time	Memory Size
Web Server	35 ms	5.31 s	55 MB
File Processing	63 ms	5.33 s	158 MB
Supermarket Checkout	20 ms	5.34 s	332 MB
Image Recognition	2076 ms	4.89 s	92 MB

Table 1: Container Instances

applications. Our setup includes 1 master node and 10 worker nodes. The master node is a server equipped with 4 CPU cores and 4GB RAM. The worker nodes are Raspberry Pi 3Bs with 4 CPU cores and 1GB RAM, hosting the actual functions in Docker containers.

Performance Benchmarks: Histogram (**HIST**) was proposed by Shahrad *et al.* [17]. HIST identifies the invocation pattern for applications. It terminates the container after each function execution but pre-warms the container before a potential next invocation. Fixed Caching (**FC**) is widely used in AWS Lambda [12]. It keeps a container alive for a fixed period of time. In the experiments, the caching period was set to 10 minutes, and all the algorithms have the same cache size.

6.2 Performance in Knative

Average response time: As illustrated in Figure 3, we observe that the average response time of CFC is 1.04s, 1.51s and 1.17s, respectively. The average response time of HIST ranges from 2.54s to 3.7s. This is because HIST heavily relies on the distribution of histogram that exhibits a clear invocation pattern, where the pre-warm windows are ideally long and keep-alive windows are short. The average response time of FC is 3.32s, 3.79s and 4.5s, respectively. FC's performance is worse because it caches containers for a fixed period of time, FC does not consider frequency or cold-start time and cannot adapt to the frequently varying request pattern. Compared with the benchmarks, CFC reduces the average response time by a factor of up to 3. CFC clearly benefits from considering frequency, size and cold-start time of containers.

Cold-start frequency: Figure 4 illustrates the cold-start frequency of each algorithm for different Zipf-β parameters. While the cold-start frequency of baselines keeps around 40%, that of CFC ranges from 11.5% to 17.3% which results in a lower average response time. CFC reduces the cold-start frequency by a factor of 3.5 and 5.6 compared to HIST and FC, respectively. This is because CFC considers the frequency of containers, and hence reuses popular containers more frequently. In other words, CFC destroys containers less frequently because popular containers are assigned with high priority and are frequently reused.

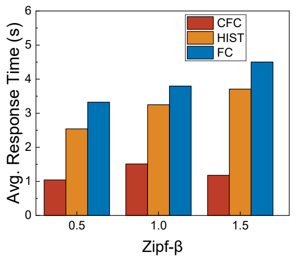


Figure 3: The average response time

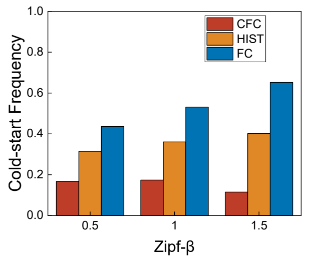
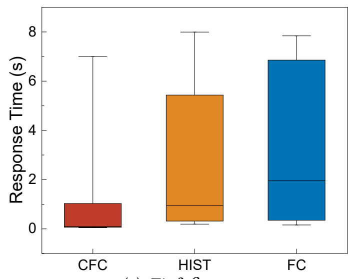
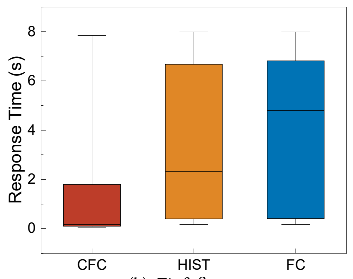


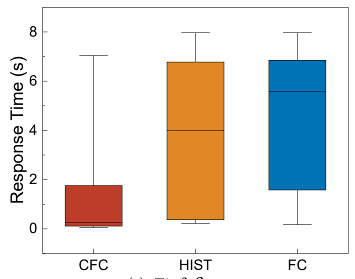
Figure 4: The cold-start frequency



(a) Zipf-β=0.5



(b) Zipf-β=1



(c) Zipf-β=1.5

Figure 5: The distributions of response time

The distributions of response time: In Figure 5, we evaluate the performance of CFC for different Zipf-β values. We use box plots to demonstrate the distributions of the request response time. The box plots show the maximum, median and minimum of the results. In Figure 5(a), the median response time of CFC outperforms HIST and FC by 90% and 95%, respectively. This is because CFC caches popular containers that are more likely to be invoked in the next time interval. Also, the 75^{th} percentile of the response time is 1.03s while that of HIST and FC is 5.43s and 6.85s, respectively. This result indicates that CFC experiences remarkably less cold-starts as most invocations are processed by cached containers.

Figure 5(b) presents the distributions of the response time for Zipf-β = 1. The response time of CFC is 1.79s at the 75^{th} percentile, which is 73.2% and 73.8% lower than that of HIST and FC, respectively. The rationale is that CFC caches frequently invoked containers and by that reduces the response time. Similarly, CFC has a better performance in reducing the median response time by 92.8% and 96.5% compared with HIST and FC, respectively.

Figure 5(c) shows the distributions of response time for Zipf-β = 1.5. We observe that CFC achieves 1.76s at 75^{th} percentile which outperforms HIST and FC by 74.1% and 74.4%, respectively. CFC reduces the median response time by 93.5% and 95.4% compared to the competitors.

When β increases from 0.5 to 1.5, CFC shows a stable performance in spite of request distribution changes because CFC caches the most frequently invoked containers and swiftly destroys containers that are not likely to be invoked in the next time interval.

7 CONCLUSION

In this paper, we studied the request distribution and caching problem for serverless edge computing. We proposed a practical caching policy for reducing the response time and the number of cold starts. The policy was evaluated using a real-world implementation and workload traces. Our results show that the proposed policy reduces the average response time and the cold-start frequency by a factor of more than 3 compared to algorithms from the literature.

8 ACKNOWLEDGEMENT

This work has been partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1, and InnovateUK grants 106199-47198 and 10040850; and the National Natural Science Foundation of China (NSFC) No. 62172189; Dr. Chen is supported by InnovateUK grant 10017032.

REFERENCES

[1] Azure. 2023. AzurePublicDataset. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md.
[2] Xuanyu Cao, Junshan Zhang, and H. Vincent Poor. 2018. An Optimal Auction Mechanism for Mobile Edge Caching. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 388–399.
[3] C. Chen, L. Nagel, L. Cui, and F. Tso. 2022. B-Scale: Bottleneck-aware VNF Scaling and Flow Routing in Edge Clouds. In *2022 IEEE Symposium on Computers and Communications (ISCC)*. 1–6.
[4] P.C. Chu and J.E. Beasley. 1997. A genetic algorithm for the generalised assignment problem. *Computers and Operations Research* 24, 1 (1997), 17–23.
[5] Google Cloud. 2023. Google Functions. https://cloud.google.com/functions.
[6] V. Farhadi, F. Mehmeti, T. He, T. F. La Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis. 2021. Service Placement and Request Scheduling for Data-Intensive Applications in Edge Clouds. *IEEE/ACM Transactions on Networking* 29, 2 (2021), 779–792.
[7] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 386–400.
[8] Azure Functions. 2023. Serverless Functions in Computing. https://azure.microsoft.com/en-gb/products/functions.
[9] Lin Gu, Zirui Chen, Honghao Xu, Deze Zeng, Bo Li, and Hai Jin. 2022. Layer-aware Collaborative Microservice Deployment toward Maximal Edge Throughput. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 71–79.
[10] Knative. 2023. Home. https://knative.dev/docs/.
[11] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. 2018. Optimal Edge User Allocation in Edge Computing with Variable Sized Vector Bin Packing. In *Service-Oriented Computing*. Springer International Publishing, 230–245.
[12] AWS Lambda. 2023. Amazon Web Services. https://aws.amazon.com/lambda/.
[13] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2021), 615–632.
[14] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. 2022. Retention-Aware Container Caching for Serverless Edge Computing. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 1069–1078.
[15] Konstantinos Poularakis, Jaime Llorca, Antonia M. Tulino, Ian Taylor, and Leandros Tassiulas. 2019. Joint Service Placement and Request Routing in Multi-cell Mobile Edge Computing Networks. In *IEEE INFOCOM 2019*. 10–18.
[16] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. New York, NY, USA, 753–767.
[17] M. Shahrad, R. Fonseca, Íñ. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*.
[18] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. 2021. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*.
[19] Ruiting Zhou, Xiaoyi Wu, Haisheng Tan, and Renli Zhang. 2022. Two Time-Scale Joint Service Caching and Task Offloading for UAV-assisted Mobile Edge Computing. In *IEEE INFOCOM 2022*. 1189–1198.