2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3503-3071-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPADS60453.2023.00308

Online Function Caching in Serverless Edge Computing

Xuan Zhang*, Hongjun Gu†, Guopeng Li*, Xin He†, Haisheng Tan*
*School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
† School of Computer and Information, Anhui Normal University, Wuhu, China

***Abstract*—Serverless edge computing has emerged as a new paradigm for running short-lived computations on edge devices. Considering the challenges posed by multiple edge servers and non-negligible cold start latency in serverless edge computing, we investigate the problem of function caching on multiple edge servers with relaying and bypassing. Our objective is to minimize the total latency of serving all function requests, which may either be processed by an idle container on the local server, initiate a new container on the local server, relayed to other edge servers, or bypassed to the cloud server. We propose `FunCa`, a greedy-based algorithm, and `FunCa`$^+$, an extension version that supports bypassing. Large-scale simulation experiments using Azure trace and Alibaba trace demonstrate that compared to Camul, the state-of-the-art algorithm for handling requests on multiple edge servers, `FunCa` can reduce latency by 52.2% and 73.27% in the two traces, respectively.**

***Index Terms*—Serverless Compuing, Edge Computing**

I. Introduction

Serverless computing is an innovative cloud computing model in which the cloud provider manages the infrastructure and automatically allocates and manages the computing resources required to execute code [1]–[3]. One of the critical advantages of serverless computing is that it frees developers from the burden of managing infrastructure, enabling them to focus entirely on writing code. Furthermore, serverless platforms automatically scale up or down capacity to meet demand, eliminating the need for capacity planning or provisioning. This considerably simplifies and accelerates the development and deployment of applications, while also lowering costs because developers only need to pay for the resources they use.

Serverless computing has garnered attention from various communities, such as networking [4]–[7], architecture [8]–[10], system [11]–[22], Artificial Intelligence [23]–[28] and software [29], [30], however, serverless computing at the edge has not been widely studied. Edge computing involves deploying computational resources at the edge of the network close to devices and users, making it particularly well-suited for latency-sensitive or bandwidth-intensive applications [31]. *Serverless edge computing* takes the benefits of serverless computing and applies them to the edge of the network [32]–[36]. This means that code is executed closer to the user, reducing latency and improving performance. This is particularly important for applications that require real-time processing, such as Internet-of-Things (IoT) devices and mobile applications. With the rapid expansion of IoT applications, various commercial platforms have been extended to the edge of the network, e.g., AWS IoT Greengrass [37], Lambda@Edge [38], and EdgeRoutine [39].

Serverless computing enables users to execute their code in containers or virtual machines, triggered by events like HTTP requests, database triggers, or message queues. However, this approach introduces a problem known as *cold start*, which is caused by the initialization of a container before it can process a function request. The initialization of a container includes container startup, image download, dependent installation, and importing necessary packages, etc., which increases the latency to millisecond or even second level [40], [41]. Caching containers in memory proves to be an effective method to mitigate cold start, *i.e.*, after the container processes the function request, cache it in memory instead of terminating it immediately, and reuse it when the same function is requested [42]. However, the memory of server is not infinite, and it is impossible to cache all containers in memory. Therefore, this work mainly focuses on devising efficient function caching policies in serverless edge computing to fully utilize limited memory.

Designing function caching policy in serverless edge computing presents several new challenges due to various factors: **1) Limited resources**: Edge servers typically have less CPU, memory and disk space than cloud servers. This means they can handle fewer function requests, and it may take longer latency to initialize a container. **2) Non-uniform cold start latency**: Cold start latency can vary significantly depending on the programming language, deep learning framework, and package size. For example, Java or C Sharp functions tend to have longer cold start latency than those written in Python or Node.js [40]. **3) Multiple edge servers and Cloud**: In serverless edge computing, there are multiple edge servers and cloud servers, and function requests can be routed to different servers. This means that a function may not always be executed on the same server where it was previously requested or cached. Therefore,

2690-5965/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPADS60453.2023.00308

cold start may occur more frequently as functions need to be deployed or migrated across different servers. In some scenarios, a request may need to be relayed to other edge servers or bypassed to the cloud server to reduce cold start latency because of the availability of different containers on different servers.

The proposed solution in this paper for serverless edge computing aims to address the challenges of limited resources, different cold start latency, and relaying between servers. Specifically, our solution uses bypassing to utilize cloud resources to process requests when edge resources are insufficient. This method ensures that frequently used functions have idle containers available to reduce cold start latency.

In this work, we study the function caching problem with relaying and bypassing within *multiple* servers in serverless edge computing, and propose an online algorithm to minimize the total latency of serving all requests. Our contributions are summarized as follows.

- We investigate a practical online function caching problem with relaying and bypassing on multiple edge servers in serverless edge computing to minimize the total latency of serving all function requests (Sec. IV).
- We propose a greedy-based algorithm called `FunCa` to support distributed requests and non-uniform cold start latency. To the best of our knowledge, the extended version of `FunCa` , `FunCa`$^{+}$, is the first online algorithm for the online function caching problem with relaying and bypassing in serverless edge computing with multiple servers (Sec. V).
- We conduct extensive simulations on Azure Functions Trace and The Function Cold Start Traces from Alibaba Cloud Function Compute. Compared with Camul, the state-of-the-art algorithm that deals with multiple servers and bypassing, in default settings, `FunCa` can reduce the latency by 52.2% in Azure trace and 73.27% in Alibaba Trace, if the bypassing is allowed, the improvement is and 53.63% and 98%, respectively (Sec. VI).

II. Background

A. Function Execution

In a serverless computing framework, the execution process of a function involves several steps. When a function is invoked, which could be a request from an HTTP endpoint, a message from a queue, or a change to an object in a storage bucket, the first step is to pull the container image from a container registry. Once the image is pulled, a container is initialized to execute the function. This process, *i.e.*, pulling an image from a registry and initializing the container is known as a cold start and can take some latency as resources need to be allocated and initialized. After the function has finished its execution, the container may be kept alive for some time to allow for container reuse. A warm start happens if the same function is triggered again within a certain time frame when the existing container can be used to execute the function, avoiding another cold start. Container reuse can significantly reduce the latency of function execution as it eliminates the need for image pull and container initialization. However, it also introduces additional complexity in terms of managing container lifecycles and ensuring that containers are properly cleaned up when they are no longer needed. Overall, the execution process of a function in a serverless computing framework involves a balance between minimizing latency through container reuse and ensuring efficient resource utilization through proper container lifecycle management.

B. Container Keep Alive and Function Caching

When a function is invoked, the system checks if there is an available container to execute the function. If there is, it means a warm start, which allows the function to execute without the latency of a cold start. This scenario significantly reduces function latency. On the other hand, if there is no existing container, it means a cold start. In this case, a new container has to be created to execute the function, which involves pulling the container image that contains the code and dependencies for the function and initializing the runtime environment. This process is called a cold start and can introduce extra latency. When a container is not needed anymore, it may be destroyed to free up resources. Effective container lifecycle management, including proper reuse and cleanup of containers when they are not required, has the potential to minimize cold starts and enhance the overall performance of serverless systems.

III. Related Works

A. Serverless Computing

Serverless computing, an evolving paradigm, liberates developers from the intricacies of underlying infrastructure when deploying applications. This approach delegates the responsibility of executing functions on demand and managing the requisite resources for each invocation to the provider, which bestows advantages like scalability, availability, and cost-efficiency.

Jonas *et al.* [1] discussed how serverless computing simplifies cloud programming and anticipates its ascendancy in the future of cloud computing. Shahrad *et al.* [43] characterized the entire production Function as a Service (FaaS) workload of Azure Functions and proposed a practical resource management policy that reduces cold starts. Yu *et al.* [44] proposed ServerlessBench, an open-source benchmark suite for characterizing serverless platforms. Wang *et al.* [40] conducted a large measurement study of more than 50,000 function instances across AWS Lambda, Azure

Functions, and Google Cloud Functions to characterize their architectures, performance, and resource management efficiency. In the realm of serverless AI systems, Li *et al.* [23] presented Tetris, a serverless platform catered to inference services with an order of magnitude lower memory footprint. It reduces runtime redundancy through a combined optimization of batching and concurrent execution, and it eliminates tensor redundancy among instances from the same or other functions by employing a lightweight and safe tensor mapping mechanism. Tang *et al.* [32] discussed task scheduling in serverless edge computing, which has been widely adopted in several applications, especially IoT applications. Patterson *et al.* [45] presented HiveMind, the first swarm coordination platform that enables programmable executions of complex task workflows between cloud and edge resources.

B. Mitigating Cold Start

Serverless computing poses some challenges, prominently including the cold start problem. This issue is particularly relevant in serverless edge computing, which aims to deploy functions closer to users. Reducing cold start latency in serverless computing is a crucial research topic that has garnered significant attention from academia and industry. Previous studies have approached the cold start problem through architecture design [16], [46]–[48], scheduling and caching strategies [18], [49]–[51], container management [52], [53], and other perspectives. This paper focuses on caching strategies to mitigate the cold start latency in serverless edge computing.

Fuerst *et al.* [54] introduced a caching-inspired Greedy-Dual keep-alive policy, yielding over a 3× reduction in cold-start overhead compared to current approaches.

Pan *et al.* [55] addressed the issue of startup latency in containers, which significantly impairs the responsiveness of IoT services. While container caching can mitigate this latency, it necessitates resource retention, potentially compromising resource efficiency. The paper proposes to optimize container caching jointly with distributed and heterogeneous nature of edge platforms. Tan *et al.* [56] formulated an online file caching problem involving relaying and bypassing across multiple caches where a file request might be relayed to other caches or bypassed directly to memory when a cache miss happens. Chen *et al.* [57] analogized container caching to object caching and proposes an online request distribution algorithm that considers container frequency, size, and cold-start time to balance cold-start overheads with resource utilization, but it does not support bypassing.

However, to the best of our knowledge, no works on the joint optimization of concurrent requests on multiple servers with relaying and bypassing have been reported in the literature. Therefore, it is still a challenge to minimize the total latency of serving all requests in serverless edge computing.

IV. System Model and Problem Formulation

Edge System. Motivated by serverless edge computing, we consider the online function caching model with multiple edge servers and a cloud server. The system consists of N edge servers, $\mathcal{E} = \{e_1, e_2, \dots, e_N\}$, where the memory size of each server is $K_i, i = 1, 2, \dots, N$. The functions $\mathcal{F} = \{f_1, f_2, \dots\}$ are assumed to run in containers. When a function is invoked, a container is needed to execute it, and the container c_{f_i} corresponding to function f_i occupies memory of size z_{f_i}. For convenience, in the following discussion, c_f can be used instead of c_{f_i}, and z_f can be used instead of z_{f_i}. Without loss of generality, we assume all the memory sizes are integers. Naturally, the sum of sizes of containers stored in each edge server can not exceed the size of edge server, *i.e.*, $\sum_{c \text{ in server } e_i} z_f \leq K_i$.

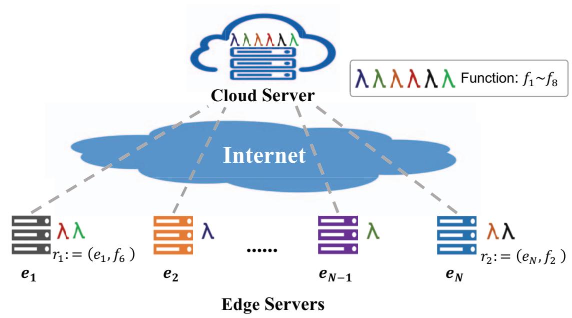


Fig. 1. The Serverless Edge Computing Model

Container State. In serverless edge computing, the corresponding container of function f, *i.e.*, c_f, have 3 different states on edge server e.

1) None: There is no c_f in e, or it has been destroyed already.
2) Running: c_f has been initialized in the memory of e, and is executing f.[1]
3) Idle: Also known as Available, c_f is in the memory of e, but is not executing or running f.

Function Processing Model. Let $\mathcal{R} = (r_1, r_2, \dots)$ be the sequence of function requests, a request r is a pair $(e, f) \in \mathcal{E} \times \mathcal{F}$, meaning a function f on edge server e is invoked. All function requests arrive in an online manner, *i.e.*, we can not get future information and no assumption is made on the arrival sequences. Time is divided into slots of unit size. Multiple different kinds of requests might come within one time slot, while each function $f \in \mathcal{F}$ can be invoked at

[1]We assume that one container can only handle one request at one time slot, *i.e.*, when a request arrives while the corresponding container is running, we have to relay the request to another server or start a new container.

most once in each slot. In the multiple edge servers system, when a request $r := (e, f)$ arrives at time T, the following 6 types of operations may be performed and result in different latencies.

- *Local Warm Start*: If container c_f corresponds to function f on server e, and the container c_f is an idle container, then it is considered a warm start. The latency of serving request r, in this case, is equal to the actual execution time of the function f, denoted as t_f^e. We use t_f^w to represent the warm start latency of the function, which is equal to t_f^e.
- *Local Cold Start*: If there is no container c_f corresponding to function f on server e or the corresponding container is running, one option is to start the container c_f on server e and execute function f. Let t_f^{cs} denote the latency for cold start. Therefore, the actual latency for local cold start of the function f on server e is $t_f^c = t_f^{cs} + t_f^e$.
- *Local queuing and Warm Start*: If there is a container c_f corresponding to function f on server e, but the container c is running, one option is to queue on the server e and wait until the container is idle before serving the request r. Therefore, the latency for a warm start after queuing on the server e is $t_f^{qw} = t_f^q + t_f^w$, where t_f^q is the queuing latency, t_f^w is the warm start latency, and $t_f^w = t_f^e$.
- *Relaying and Warm Start*: If there is no container c_f on server e or c_f is running, one option is to relay the request r to another edge server e' for processing. If there is an idle container c_f on server e', the latency is $t_f^{rw} = t_f^r + t_f^e$, where t_f^r is the relaying latency of function f.
- *Relaying queuing and Warm Start*: If there is no container c_f corresponding to function f or c_f is running on server e, one option is to relay the request r to server e' for processing. However, if all corresponding containers c_f on server e' are running, the request can wait with a latency of t_f^q. The latency after relaying and queuing is $t_f^{rqw} = t_f^r + t_f^q + t_f^e$, where t_f^r is the relaying latency of function f.
- *Bypassing and Warm Start*: Besides the above cases, another option is to bypass the request for function f to the cloud server for processing. Since the cloud server has a large amount of memory, we assume that all functions can be warm-started on the cloud server without queuing. Therefore, the latency of bypassing the request $r := (e, f)$ to the cloud center is $t_f^b + t_f^e$, where t_f^b is the bypass latency for the function f.

Problem Formulation. The objective of this problem is to minimize the total latency to serve all function requests. Let $t_{r:=(f,s)}$ denote the latency incurred to serve the request $r := (f, s)$. For simplicity, we can also use t_r to denote the latency of request r.

Problem 1:

$$\begin{aligned} &\text{minimize} \sum_{r\in\mathcal{R}} t_{r:=(s,f)} \\ s.t. \quad &\sum_{c \text{ in server } e_i} z_f \le K_i \qquad \forall i \in [1, N] \end{aligned} \tag{1}$$

V. Algorithm

In this section, we introduce the online algorithm to support distributed requests and non-uniform cold start latency. An online algorithm is an algorithm that can process the input one by one, without knowing the whole input in advance. The main algorithm, `FunCa` and `FunCa`$^+$are defined in Algorithm 2, and the operation selection algorithm is defined in Algorithm 1. In Algorithm 2, initially, the memory of edge servers in the serverless edge computing system is initialized to empty. We use a list $\mathcal{C}$ to represent the containers in the serverless edge computing system, where the elements are triplets (e, f, t) that indicate the container c_f corresponding to the function f on the edge server e will finish at time t. If t is 0, *i.e.*, $(e, f, 0)$, it indicates the existence of an idle c_f in the system.

For the request $r := (e, f)$, if there exists an idle container $(e, f, 0)$ on the server e, `FunCa` can serve the request r with the latency of the function's execution time t_f^e (Line 8). If there is no idle c_f on server e for the request $r := (e, f)$, `FunCa` greedily selects the operation that minimizes the latency for serving the request r based on the current status of containers in the serverless edge system, *i.e.*, $\mathcal{C}$ (Line 14). To describe the details of the selection operation in a more concise way, we use Alg 1. The Landlord [58] and Landlord with Bypassing (LLB) [59] are used as the container replacement algorithms in `FunCa` and `FunCa`$^+$, respectively. However, other algorithms that can handle weight and support bypassing can be used instead. To be more specific, in Algorithm 2, after determining the server for serving request r (Line 16), if a new container needs to be created, `FunCa` checks if there is enough remaining memory to start a container of size z_f. If there is not enough memory in server e, `FunCa` uses Landlord to replace containers in e (Line 20), and then creates c_f on e. On the other hand, `FunCa`$^+$first assumes that c_f is located in e^β and sets the credit for f (Line 18). If after running the replacement algorithm, f_{e^β}.weight > 0, then c_f is cold-start on server e^β. Otherwise, request r is bypassed to the cloud server with a latency of $t_f^b + t_f^e$.

Algorithm 1 is designed to select the operation for serving a function request. Algorithm 1 takes as input the latencies of cold start latency (t_f^{cs}), executing the function (t_f^e), and relaying the request to another server (t_f^r) on the edge server that receives the request, as well as the queueing latency of the function on edge server e ($t_{f_e}^q$). The algorithm returns the latency of the selected operation (t_{real}) and the edge server that actually serves the request (e^{α}). The algorithm considers 3 possible scenarios: (1) there is no container for the function in the system, (2) there are idle containers for the function on other edge servers, and (3) there are only running containers for the function on other edge servers. For each scenario, the algorithm randomly chooses one container on another edge server (if any), and selects the operation that corresponds to the minimum latency among the possible options. The algorithm appends the selected operation to a list $\mathcal{C}$ that records all the operations performed by the system.

Algorithm 1: SelectOp

Input t_f^{cs}, t_f^e, t_f^r, $t_{f_e}^q$
if There is no c_f in the serverless edge computing system **then**
  $t_{real} = T + t_f^{cs} + t_f^e$;
  $e^{\alpha} = e$;
  $\mathcal{C}.append(e, f, t_{real})$;
if There are idle containers for f in other servers in the system **then**
  Random choose one idle container on edge e';
  Select the operation corresponding to the minimum latency t_{real} *between* ($t_f^{cs} + t_f^e, t_{f_e}^q + t_f^e, t_f^r + t_f^e$);
  $\mathcal{C}.append(e', f, t_{real})$;
if There are only running containers for f on other servers in the system **then**
  Random choose one running container on edge e';
  Select the operation corresponding to the minimum latency t_{real} *between* ($t_f^{cs} + t_f^e, t_{f_e}^q + t_f^e, t_f^r + t_{f_{e'}}^q + t_f^e$);
  $\mathcal{C}.append(e', f, t_{real})$;
return t_{real}, e^{α};

Algorithm 2: `FunCa` and `FunCa`$^+$

Input *Request* $r := (e, f)$, t_f^{cs}, t_f^e, t_f^r, t_f^b, z_f, K_e;
$\mathcal{C} \leftarrow []$, Timer $T \leftarrow 0$, $t_{f_e}^q = +\infty$;
while True **do**
  for $(e, f, t) \in \mathcal{C}$ **do**
    if $t <= T$ **then**
      $(e, f, t) = (e, f, 0)$;
  while new request $r := (e, f)$ arriving at T **do**
    if There is one $(e, f, 0)$ in $\mathcal{C}$ **then**
      $(e, f, 0) = (e, f, T + t_f^e)$;
      Serve r at edge e with latency t_f^e;
    else
      if There is one (e, f, t) in $\mathcal{C}$ **then**
        $t_{f_e}^q = t - T$;
      $t_{real}, e^{\alpha} =$ *SelectOp*$(t_f^{cs}, t_f^e, t_f^r, t_{f_e}^q)$;
      if $t_{real} = t_f^{cs} + t_f^e$ and $e^{\alpha} = e$ **then**
        Let edge server e^{β} be a copy of e^{α} ;
        if `FunCa`$^+$ **then**
          Let c_f in e^{β}, $f_{e^{\beta}}$.weight $= 0$;
        if the remaining size of $e^{\beta} < z_f$ **then**
          $\mathcal{C}_{\text{evicts}} \leftarrow \text{Replace}(e^{\beta}, f)$;
          for $f' \in \mathcal{C}_{evicts}$ **do**
            Evict $c_{f'}$ from e^{α};
        if $f_{e^{\beta}}$.weight > 0 or `FunCa` **then**
          $\mathcal{C}_{\text{append}}(e^{\alpha}, f, t_{real})$;
          $f_{e^{\alpha}}$.weight $= t_{real}$;
          Start c_f on e^{β}, serve r with $t_f^{cs} + t_f^e$;
        else
          Bypass this request with t_f^b ;
      else
        Serve r at e^{α} with latency t_{real};
  $T \leftarrow T + 1$;

VI. Evaluation

In this section, we evaluate the performance of `FunCa` and `FunCa`$^+$on two traces: (1) Azure Functions Trace (Azure) [43], (2) The Function Cold Start Traces from Alibaba Cloud Function Compute (AliFC) [60]. We compare `FunCa` and `FunCa`$^+$ with several caching algorithms, *i.e.*, LRU [61], Fixed Caching (FC) [62], FaaSCache [54] and Camul [56].

A. Simulation Setup

Traces. For Azure Trace, similar to the approach used by FaaSCache, we selected 100 representative functions from the Azure Trace and used a subset that included 581,718 requests for these 100 functions. As Azure Trace is focused on cloud computing paradigm and does not contain information about the servers that handled the requests, we used server information from Google's Trace [63]. For AliFC Trace, we used the request information in "region2.csv" in the origin dataset, using the server information from Google's Trace. AliFC Trace contains 1188,492 requests for 8599 functions.

Edge Servers. By default, we set 40 edge servers with 2000MB memory on each, and let $t_r = 100ms$, $t_b = 1000ms$. We also conduct extensive experiments to study the impact of different edge server numbers and the memory size of the servers. The metric used to evaluate the performance of algorithms is the total latency to serve all requests. Furthermore, we use the latency improvement relative to LRU to measure the performance of the algorithm when the parameters change, which can be calculated by: Latency Improvement of ALG = (Latency(LRU) − Latency(ALG))/Latency(LRU). A higher latency improvement means better performance.

B. Baseline Algorithms

LRU [61]. LRU is the most widely adopted caching replacement policy used in single cache systems. When a file request is missed, LRU will fetch the file immediately. When the cache is full, LRU will discard the least recently used file to make space for the newly fetched. In the function caching problem, LRU will discard the least recently used container to make space for the newly created container.

Fixed Caching (FC) [62]. This is the caching strategy widely used in AWS Lambda. An instantiated container will be cached for a fixed long duration. For example, in our simulation, an idle container will be destroyed if it has been more than 5 minutes since it was last requested.

FaaSCache [54]. FaaSCache is a greedy duality-based approach that takes advantage of the last time the function was requested, the cold start latency of the function, the frequency of the function being requested, and the size of the container. It prioritizes the functions in the server and destroys the lowest priority container when the memory is full.

Camul [56]. Camul is an algorithm based on online file caching problems on multiple caches and takes relaying, bypassing, and fetching costs into consideration. The technical core is a novel generalization of the marking methods.

C. Experiment Results

Overall Result. We first evaluate the overall performance of `FunCa` and `FunCa`$^+$in the default setting, where there are 40 edge servers in the system, each with 2000MB of memory. The experimental results are shown in Fig. 2. In Azure trace, the latency improvement of `FunCa` to Camul is 52.2%, and `FunCa`$^+$ to Camul is 53.63%. For the results in AliFC, the latency improvement of `FunCa` to CaLa is 73.27%, and `FunCa`$^+$ to Camul is 98%. Due to the presence of multiple edge servers in the system, algorithms designed for container caching on single server, such as FC and FaaSCache, have not achieved better performance than LRU.

Number of Edge Server. Fig. 3(a) and Fig. 3(b) illustrate the effect of the number of edge servers; the range of the number of edge servers is $1, 5, 10, 15, 20, 40, 60, 80, 100$. In Azure trace, when only one edge server is present, both `FunCa` and `FunCa`$^+$perform poorly. As the number of servers increases, the algorithms suitable for multiple server scenarios, namely `FunCa`, `FunCa`$^+$, and Camul, all show improved performance. However, when there are a sufficient number of servers, the performance of `FunCa`, `FunCa`$^+$, and Camul tends to decline. This is because as the number of servers increases, the number of requests per server decreases, resulting in fewer situations where containers need to be destroyed. For AliFC, even with only one edge server present, `FunCa` and `FunCa`$^+$perform better than LRU. The difference in the performance of `FunCa` and `FunCa`$^+$on the two traces, when there is only one server, is due to the different number of functions and requests in the traces.

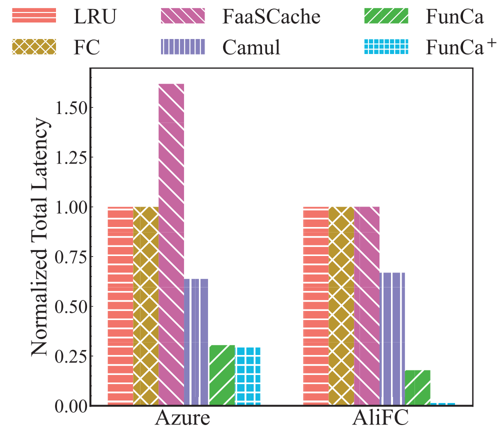
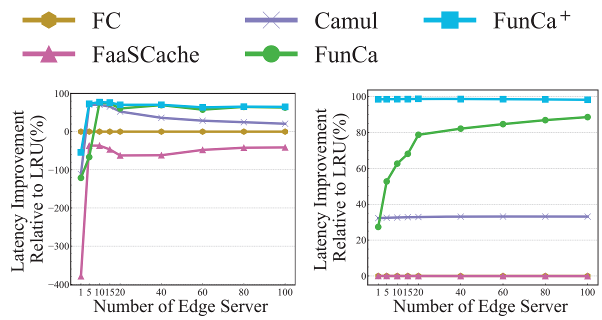


Fig. 2. Overall performance.



(a) Azure (b) AliFC

Fig. 3. Impact of Number of Edge Server.

Memory of Edge Server. We show the result of the impact of the memory of edge servers in Fig. 4, the range of memory of servers is 1000, 2000, 3000, 4000 and 5000 MB. Fig. 4(a) shows that as the memory of edge servers increases, the performance of `FunCa`, `FunCa`$^+$, and Camul decreases. A larger memory size results in fewer container evictions across multiple servers, leading to lower latency for all requests. In AliFC, however, the performance of each algorithm did not show significant changes with the increase in server memory.

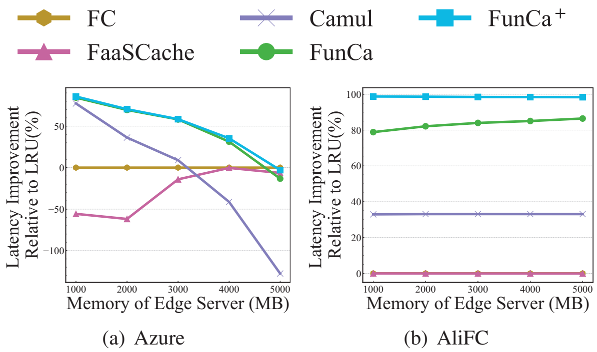


Fig. 4. Impact of Memory of Edge Server.

VII. Conclusion

In this paper, we study the function caching problem on multiple edge servers with relaying and bypassing and aim to minimize the total latency to serve all function requests. We propose a greedy-based algorithm, `FunCa`, to handle the challenges in function caching and its version that supports bypassing, called `FunCa`$^+$. We evaluate `FunCa` and `FunCa`$^+$ on Azure trace and Alibaba trace. The experiment results show that compared with Camul, in default settings, `FunCa` can reduce the latency by 52.2% in Azure trace and 73.27% in Alibaba trace, and if the bypassing is allowed, the improvement is 53.63% and 98%, respectively.

Acknowledgment

This work was supported in part by the National Key R&D Program of China under Grant 2021ZD0110400, NSFC under Grant 62132009, and the Fundamental Research Funds for the Central Universities at China. Guopeng Li is the corresponding author (email: guopengli@mail.ustc.edu.cn).

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[2] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

[3] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[4] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 780–794.

[5] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/yu-0

[6] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing more with less: Orchestrating serverless applications without an orchestrator," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1505–1519.

[7] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 406–419.

[8] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 306–318.

[9] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: characterization and optimization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 757–770.

[10] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "Mxfaas: Resource sharing in serverless environments for parallelism and efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[11] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in *OSDI 23*. Boston, MA: USENIX Association, Jul. 2023, pp. 497–517. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[12] H. Ding, Z. Wang, Z. Shen, R. Chen, and H. Chen, "Automated verification of idempotence for stateful serverless applications," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 887–910. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/ding

[13] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 46–60.

[14] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 753–767.

[15] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: running latency sensitive serverless computations at the edge," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 239–251.

[16] S. Ristov, M. Hautz, C. Hollaus, and R. Prodan, "Simless: simulate serverless workflows and their twins and siblings in federated faas," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 323–339.

[17] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 78–93.

[18] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 289–305.

[19] Z. Zhao, M. Wu, J. Tang, B. Zang, Z. Wang, and H. Chen, "Beehive: Sub-second elasticity for web services with semi-faas execution," in *ASPLOS*, 2023, pp. 74–87.

[20] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 768–781.

[21] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *SOSP*, 2021, pp. 724–739.

[22] A. Sahraei, S. Demetriou, A. Sobhgol, H. Zhang, A. Nagaraja, N. Pathak, G. Joshi, C. Souza, B. Huang, W. Cook *et al.*, "Xfaas: Hyperscale and low cost serverless functions at meta," in *Proceedings*

of the 29th Symposium on Operating Systems Principles, 2023, pp. 231–246.

[23] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[24] S. Gupta, S. Rahnama, E. Linsenmayer, F. Nawab, and M. Sadoghi, "Reliable transactions in serverless-edge architecture," *arXiv preprint arXiv:2201.00982*, 2022.

[25] D. Justen, "Cost-efficiency and performance robustness in serverless data exchange," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2506–2508.

[26] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, "Serverless data science-are we there yet? a case study of model serving," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1866–1875.

[27] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *ACM SoCC 2023*.

[28] Z. Hong, J. Lin, S. Guo, S. Luo, W. Chen, and R. Wattenhofer, "Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation," in *European Conference on Computer Systems (EuroSys), Athens, Greece*, April 2024.

[29] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Transactions on Software Engineering and Methodology*, 2023.

[30] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[31] G. Li, H. Tan, L. Liu, H. Zhou, S. H.-C. Jiang, Z. Han, X.-Y. Li, and G. Chen, "DAG scheduling in mobile edge computing," *ACM Trans. Sen. Netw.*, vol. 20, no. 1, oct 2023.

[32] Q. Tang, R. Xie, F. R. Yu, T. Chen, R. Zhang, T. Huang, and Y. Liu, "Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach," *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 19 634–19 648, 2022.

[33] P. Raith, S. Nastic, and S. Dustdar, "Serverless edge computingwhere we are and what lies ahead," *IEEE Internet Computing*, vol. 27, no. 3, pp. 50–64, 2023.

[34] F. Tütüncüoğlu, S. Jošilo, and G. Dán, "Online learning for rate-adaptive task offloading under latency constraints in serverless edge computing," *IEEE/ACM Transactions on Networking*, 2022.

[35] Q. L. Trieu, B. Javadi, J. Basilakis, and A. N. Toosi, "Performance evaluation of serverless edge computing for machine learning applications," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2022, pp. 139–144.

[36] H. Ko and S. Pack, "Function-aware resource management framework for serverless edge computing," *IEEE Internet of Things Journal*, vol. 10, no. 2, pp. 1310–1319, 2022.

[37] "Aws iot greengrass," 2023, https://aws.amazon.com/greengrass/.

[38] "Aws lambda@edge," 2023, https://aws.amazon.com/lambda/edge/.

[39] "Edge routine," 2023, https://www.aliyun.com/activity/cdn/edgeroutine.

[40] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.

[41] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling cold start of serverless applications by efficient and adaptive container runtime reusing," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 433–443.

[42] M. Abdi, S. Ginzburg, C. Lin, J. M. Faleiro, Í. Goiri, G. Chaudhry, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, 2023.

[43] M. Shahrad, R. Fonseca, Í. 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.

[44] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 30–44.

[45] L. Patterson, D. Pigorovsky, B. Dempsey, N. Lazarev, A. Shah, C. Steinhoff, A. Bruno, J. Hu, and C. Delimitrou, "Hivemind: a hardware-software system stack for serverless edge swarms," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 800–816.

[46] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[47] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 356–370.

[48] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, "From warm to hot starts: Leveraging runtimes for the serverless era," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 58–64.

[49] H. Yu, *FaaSRank: A Reinforcement Learning Scheduler for Serverless Function-as-a-Service Platforms*. University of Washington, 2021.

[50] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," in *HPDC*. New York, NY, USA: Association for Computing Machinery, 2022, p. 227239.

[51] J. Tilles *et al.*, "Serverless computing on constrained edge devices," 2020.

[52] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 53–68.

[53] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and C. Das, "Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 257–272.

[54] A. Fuerst and P. Sharma, "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*, 2021, pp. 386–400.

[55] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," *Proc. of IEEE INFOCOM, IEEE*, 2022.

[56] H. Tan, S. H.-C. Jiang, Z. Han, L. Liu, K. Han, and Q. Zhao, "Camul: Online caching on multiple caches with relaying and bypassing," in *IEEE INFOCOM 2019*.

[57] C. Chen, L. Nagel, L. Cui, and F. P. Tso, "S-cache: Function caching for serverless edge computing," in *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*, 2023, pp. 1–6.

[58] N. E. Young, "On-line file caching," *Algorithmica*, vol. 33, no. 3, pp. 371–383, 2002.

[59] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György, "Online file caching with rejection penalties," *Algorithmica*, vol. 71, no. 2, pp. 279–306, 2015.

[60] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 443–457.

[61] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Communications of the ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[62] "Aws lambda," 2023, https://aws.amazon.com/lambda.

[63] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage trace," in *Technical Report*, 2011.