Online Container Caching with Late-Warm for IoT Data Processing

Guopeng Li[†], Haisheng Tan[†‡], Xuan Zhang[†], Chi Zhang[†], Ruiting Zhou[§], Zhenhua Han[¶], Guoliang Chen[†‡]
[†]University of Science and Technology of China, China, [‡]USTC-Deqing Alpha Innovation Research Institute, China
[§] Southeast University, China [¶]Microsoft Research Asia, China
guopengli@mail.ustc.edu.cn, hstan@ustc.edu.cn, {xuanzhang, gzhnciha}@mail.ustc.edu.cn
ruitingzhou@seu.edu.cn, zhenhua.han@microsoft.com, glchen@ustc.edu.cn

***Abstract*—Serverless edge computing is an efficient way to execute event-driven, short-duration, and bursty IoT data processing tasks on resource-limited edge servers, using on-demand resource allocation and dynamic auto-scaling. In this paradigm, function requests are handled in virtualized environments, *e.g.*, containers. When a function request arrives online, if there is no container in memory to execute it, the serverless platform will initialize such a container with non-negligible latency, known as cold start. Otherwise, it results in a warm start with no latency in previous studies. However, based on our experiments, we find there is a remarkable third case called Late-Warm, *i.e.*, when a request arrives during the container initializing, its latency is less than a cold start but not zero. In this paper, we study online container caching in serverless edge computing to minimize the total latency with Late-Warm and other practical issues considered. We propose `OnCoLa`, a novel $O(T_c^{3/2}K)$-competitive algorithm supporting request relaying on multiple edge servers. Here, T_c and K are the maximum container cold start latency and the memory size, respectively. Experiments on Raspberry Pi and Jetson Nano with OpenFaaS and faasd using common IoT data processing tasks show that `OnCoLa` reduces latency by up to 21.38% compared with representative lightweight policies. Extensive simulations on two real-world traces demonstrate that `OnCoLa` consistently outperforms the state-of-the-art container caching algorithms and reduces the latency by 27.8%.**

I. Introduction

Recently, the Internet-of-Things (IoT) has enabled new applications for many domains, such as healthcare [1], public transport [2], and the energy industry [3]. These applications fundamentally rely on the processing of IoT data from IoT devices like sensors, cameras, and vehicles [4]–[6]. However, uploading IoT data to the remote cloud faces challenges like privacy leaks, network congestion, and high transmission latency. Since IoT data are usually generated far from the cloud, edge computing is a natural alternative, which executes *IoT data processing tasks (IDPTs)* on edge servers close to the data sources [7]–[12]. IDPTs are event-driven, short-duration, and have bursty workloads [13]–[15]. For instance, an object detection and recognition task using a motion-activated camera is triggered by motion, completes within 5 seconds, and experiences a surge in frequency when more objects are detected [16], [17]. When executing IDPTs on the resource (CPU, memory)-limited edge servers, it is challenging to handle the task bursts and prevent resource waste during idle periods between tasks. Adopting the serverless paradigm [18], which offers on-demand resource allocation and dynamical auto-scale policy, is a promising approach for executing IDPTs on resource-limited edge servers, which can be called *serverless edge computing* [19]–[22].

Serverless computing, also known as Function-as-a-Service (FaaS) has attracted attention from various communities, such as data [23]–[28], networking [29]–[33], architecture [34]–[37], and system [38]–[42]. In FaaS, developers can implement tasks as *functions*, and execute functions within a virtualized environment, such as a container. Before executing a function, a container will go through an initialization, which involves launching the container, preparing the program language runtime, and installing necessary libraries. This initialization is known as a *cold start* with non-negligible latency. If the container is already initialized in memory before the function request, it results in a *warm start* with no latency. One way to mitigate cold starts is to cache initialized containers in memory so that they are warm for future function requests. However, caching containers in memory is costly, as about 50% of containers need more than 100 MB of memory [43]. Therefore, we need to investigate the container caching policy that decides which containers should be cached in memory to make full use of the limited memory and reduce the latency.

Container caching is a fundamental problem in serverless computing, which is non-trivial due to variable memory demands, diverse cold start latencies, and skewed function popularity [43], [44]. When taking into account the practical issues in edge computing, we reveal extra challenges as follows.

Late-Warm. In serverless computing, containers have to suffer the cold start latency before they are fully initialized in memory and ready to execute functions. Function requests arrive in an *online* manner, meaning that we cannot get future information and no assumption is made on the arrival patterns. When a function request arrives during its container initializing, which strictly is neither a cold nor a warm start, we call this case *Late-Warm*, as shown in Fig. 1. Since Late-Warm is non-negligible as it adds extra latency caused by a cold start, the optimal policy (*i.e.*, Bélády [45]) for traditional online caching cannot be applied directly here, as shown in Fig. 2. Moreover, traditional online caching mainly focuses on files, each with a static size [46]–[48], while we cache containers, each of which is much more dynamic, such as its cold start latency discussed below and the memory size occupied during its lifecycle in Sec. III. In our experiments

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00127

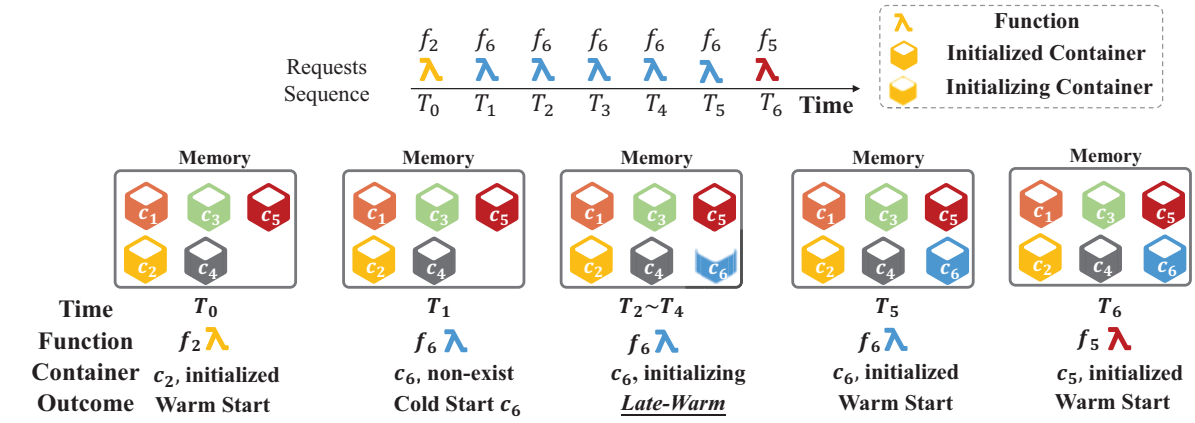


Fig. 1. Late-Warm in serverless edge computing. c_i represents the container used to execute f_i, for $1 \le i \le 6$. Before T_0, containers c_1, c_2, c_3, c_4 and c_5 have been initialized in memory. c_6 starts initializing (*i.e.*, cold start) at T_1 until T_5, and therefore the outcomes of requests arriving at T_2,T_3, and T_4 are all *Late-Warm*.

on edge devices, Late-Warm is more prevalent as the longer cold start latency due to limited edge resources increases its occurrences.

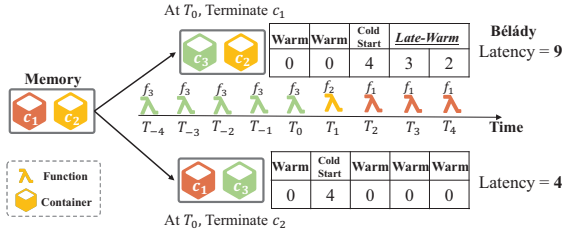


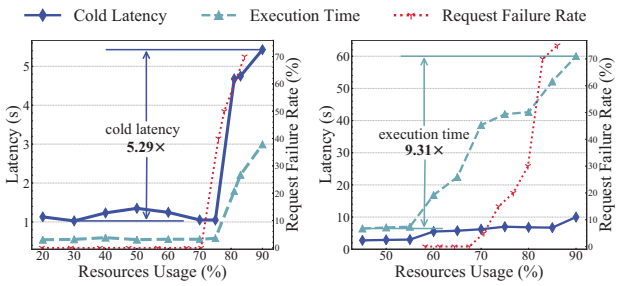
Fig. 2. Bélády is Latency-suboptimal in online caching with Late-Warm. In this example, the size of memory is 2, for any container, the memory footprint is 1, and the initialization time is 4, that is, the initialization starts at T_i and finishes at T_{i+4}. At T_0, since c_3 is initialized, one must choose to terminate either c_1 or c_2. Bélády terminates c_1. The result shows that terminating c_1 leads to a latency of 9, while terminating c_2 results in a latency of 4. Therefore, Bélády is Latency-suboptimal.

Memory Sensitivity. Edge servers are resource-limited, *e.g.*, Raspberry Pi 4B (PI4B) and Jetson Nano (Nano) have CPU clock rates no higher than 1.5 GHz and main memory no more than 8 GB. In our experiments in Example 1, we observed significant variations in cold start latency and function execution time with different memory usage percentages.[1] In addition, with a high memory usage percentage, function requests might even result in failures.[2] These dynamics in latency introduce new challenges for container caching. Moreover, as illustrated in Fig. 3, the latency and request failure rate might sharply increase at some specific memory usage in both PI4B and Nano, further inspiring us to avoid such a sudden increase adaptively.

[1]Memory usage percentage: the ratio of the currently used memory size to the total available memory size on the server. In Linux, memory usage and total available memory size can be obtained using commands like `htop` and `glances`.

[2]If there is insufficient memory to initialize a container for a new function on any server, and terminating executing containers is not permitted, or if the function's execution memory requirements exceed available memory, this will result in a request failure.

Example 1 (Memory Sensitivity)**.** Fig. 3 shows our experimental results on PI4B with 1GB memory and Nano with 4GB memory. We invoke a matrix multiplication function on PI4B and an image classification function (using ShuffleNetV2 and TensorRT) on Nano. When memory usage percentage varies, on PI4B, the cold start latency and function execution time can change by up to 5.29× and 5.31×, respectively. For Nano, these can vary up to 3.64× and 9.31×. Moreover, the latency and request failure have a sharp increase when memory usage reaches nearly 80% and 70% for PI4B and Nano, respectively. We also conducted experiments on the influence of CPU usage on latency and failure rate, and found it much less sensitive compared with the factor of memory usage. Due to limited space, we omit the result here.



(a) Matrix multiplication on PI4B (b) Image classification on Nano

Fig. 3. The variation of cold start latency, function execution time, and request failure rate with the change of Memory usage percentage.

Request Relaying. In serverless edge computing, function requests arrive at each edge server in a distributed manner. When an infrequent function request arrives on one edge as a cold start, a more cost-effective way might be not to initialize its container locally but to send it to a nearby edge, which is called *request relaying*. This becomes especially challenging with the additional latency from Late-Warm and the variable cold latency and execution time caused by Memory Sensitivity, elevating the difficulty of decision-making in the online container caching algorithm.

To address the above practical challenges in serverless edge computing for executing IoT data processing tasks, we study the online container caching problem with Late-Warm on multiple edge servers. We propose a novel online algorithm named OnCoLa to minimize the total latency of function requests. In OnCoLa, we assign a priority to each container to indicate its cost-effectiveness for reducing the total latency. We implement and evaluate it with small-scale testbed experiments using common IoT data processing tasks and large-scale simulations based on real-world traces.

Our technical contributions are summarized as follows:

- Under a novel model taking Late-Warm and other practical issues into account in executing IoT data processing tasks on edge servers, we investigate the online container caching problem on multiple edge servers to minimize the total latency. We analyze its hardness and prove the lower bound of the competitive ratio as $\Omega(T_c K)$ for all deterministic online algorithms, where T_c is the maximum cold start latency of all containers, and K is the memory size. To the best of our knowledge, we are the first to explicitly consider Late-Warm for the online container caching problem.
- We propose an Online Container Caching policy with Late-Warm, named OnCoLa, taking Late-Warm, memory sensitivity, and request relaying into account. We further theoretically prove its competitive ratio as $O(T_c^{3/2} K)$.
- We implement OnCoLa on PI4B and Nano with the representative serverless platforms as OpenFaaS and faasd, and evaluate it under workloads consisting of common IoT data processing tasks. The results demonstrate that OnCoLa significantly reduces latency by 21.38% and reduces request failure rate by up to $2.3\times$ compared with the commonly used fixed-duration container caching policy. Through extensive large-scale simulations with AliFC trace and Azure traces, we demonstrate that OnCoLa outperforms the SOTA solution GD [49] and reduces the latency by up to 27.8%.

The remainder of this paper is organized as follows. Sec. II introduces the background. Sec. III formalizes the problem, while Sec. IV demonstrates OnCoLa and theoretical analysis. Sec. V and Sec. VI further show the results of the experiment on edge devices and the extensive simulations, respectively. Sec. VII reviews the related work and we discuss the future work in Sec. VIII. Finally, Sec. IX concludes this paper.

II. Background

A. Serverless Computing

Serverless computing platforms, offering Function-as-a-Service (FaaS) abstraction, enable tasks to be deployed as serverless functions that are invoked by events such as request arrivals or new data production. These serverless function requests are managed in virtualized environments like containers [50]. In serverless computing, developers write function codes for specific tasks. These functions are then deployed to serverless platforms, which may be commercial (*e.g.*, AWS Lambda [51], Azure Functions [52]) or open-source (*e.g.*, OpenWhisk [53], OpenFaaS [54]). The platform automatically handles function initialization and execution in response to events using containers and manages resource scaling. Post-execution, the platform's container caching policy determines whether to retain or terminate the container. Unlike traditional models, serverless functions are event-triggered, operating only when needed, thus optimizing resource usage. This model also spares developers from handling dependencies and allows them to benefit from fine-grained dynamic resource scaling.

B. IoT Data Processing in serverless edge computing.

The event-driven execution paradigm and dynamically auto-scaling resource management policy of serverless computing motivate us to apply serverless to edge computing for executing event-driven and bursty IoT data processing tasks.

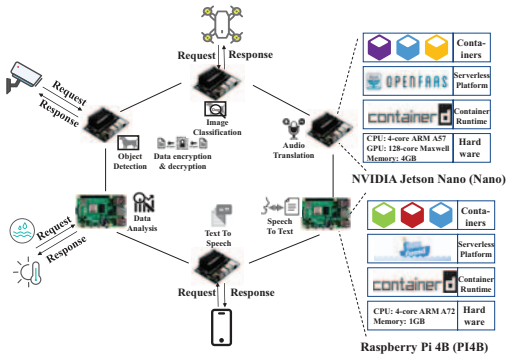


Fig. 4. IoT Data Processing in serverless edge computing.

Fig. 4 shows the components for executing IoT data processing tasks in serverless edge computing that we discuss in this paper. There are some IoT devices, such as two sensors, one camera, and one smartphone. We use PI4B and Nano as edge servers and employ lightweight containerd [55] as the container runtime. We use OpenFaaS [54] as the serverless platform on Nano, and its lighter version, faasd [56] on PI4B. In this setup, when an IoT device generates a data processing event, it triggers a request for a specific function (such as Object Detection, Text to Speech and etc.). Upon receiving this request, PI4B and Nano execute the function by initializing a container in memory to handle the task. Serverless Edge Computing [57]–[59] extends serverless computing to the edge, where data is processed on edge servers near the source. This approach reduces latency and bandwidth usage, which are crucial for low-latency, real-time applications in IoT and mobile computing. Moreover, serverless edge computing enhances privacy by enabling local data processing.

C. Cold Start

Both the serverless in the cloud and at the edge face cold start issues, which introduces extra latency for serverless functions. For example, Fig. 5 shows the composition of latency when we first request an image classification function on Nano. A cold start typically involves several stages: checking the availability of an initialized container, startup of the container, preparing the language environment, and importing the libraries. The extra cold start latency is 2.75 seconds and

the function execution time is 6.45 seconds. The extra cold start latency is 29.89% of the total latency. This additional latency is critical in low-latency applications like real-time data processing and interactive services, impacting user experience and performance. Therefore, optimizing serverless computing requires mitigating cold starts, which can be achieved by two main methods: reducing the occurrences of cold starts, and reducing the duration of each cold start through accelerating container initialization. In this paper, we focus on the first. These two methods are orthogonal and complementary in achieving the overall goal of cold start mitigation.

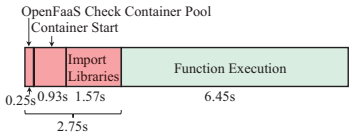


Fig. 5. The composition of latency.

III. System Model and Problem Formulation

We provide the system model and problem formulation in this section. Commonly used symbols are listed in Table I.

A. Model

System. Motivated by serverless edge computing, this study focuses on a system comprising multiple edge servers. Specifically, the system consists of N edge servers, $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$, where the memory size of each server is $K_i, i = 1, 2, \ldots, N$. We set $K = \max_i K_i$. The request of function $f_i \in \mathcal{F}$ ($\mathcal{F} = \{f_1, f_2, \ldots\}$) is assumed to execute in its own container c_{f_i}. Whenever a function f_i is requested, its corresponding container c_{f_i} needs to be initialized to execute the function. We assume that edge servers are interconnected by a local area network and thus, the communication latency between them is negligible.

Container. In this paper, we assume that each function request is executed in its own container. A container in memory can be in one of three states at any time: *initializing*, *initialized*, or *executing*. An *initializing* container is a container that has not finished its initialization and cannot execute any functions. An *initialized* container is a container that has been initialized, but has no function requests at the moment. It can also be called an idle container. An *executing* container is a container that is executing a function.[3] A container that is initialized or executes a function is called a *warm container*, which represents an already initialized environment for the requests of the same function. Generally, we use z_{f_i} to represent the memory footprint of container c_{f_i}. Specifically, $z^e_{f_i}$ denotes the memory footprint when the container is in the executing state, and $z^p_{f_i}$ represents the footprint when the container is in the initializing or initialized state. For convenience, we use c_f to represent c_{f_i}, z_f to denote z_{f_i}, z^e_f to denote $z^e_{f_i}$, and z^p_f to denote $z^p_{f_i}$. We assume that all memory sizes are integers without loss of generality. The sum of the container sizes on each edge server must not exceed its memory capacity.

[3]In serverless computing, multiple concurrent requests of the same function can be handled in a single container or by initializing multiple containers, depending on the auto-scaling policy of the serverless platform.

TABLE I
List of Symbols

Notation	Description
$\mathcal{S}$	The set of edge servers, $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$
$\mathcal{F}$	The set of functions, $\mathcal{F} = \{f_1, f_2, \ldots\}$
$\mathcal{R}$	The sequence of function requests, $\mathcal{R} = (r_1, r_2, \ldots)$, $r := (s, f) \in \mathcal{S} \times \mathcal{F}$
c_f	The corresponding container to execute function f
z^e_f	The memory footprint when container c_f is in the executing state
z^p_f	The memory footprint when container c_f is in the initializing or initialized state
t^e_f	The function execution time of f
t^c_f	The latency for initializing container c_f, *i.e.*, cold latency
p_{c_f}	The priority of container c_f

Request. Let $\mathcal{R} = (r_1, r_2, \ldots)$ be the sequence of function requests. We represent a request as a pair $(s, f) \in \mathcal{S} \times \mathcal{F}$, meaning the request of function f on edge server s. All function requests arrive in an online manner, that is, we can not get future information and we make no assumptions on the arrival patterns. We divide time into slots of unit size. Multiple different kinds of function requests might come within one time slot, however, each function $f \in \mathcal{F}$ can be requested at most once in each slot. We use t^e_f to indicate the execution time of f, and t^c_f to indicate the latency for initializing the container c_f (*i.e.*, cold latency), and t^e_f and t^c_f vary with memory usage. We set $T_c = \max_i t^c_{f_i}$. As shown in Fig. 6, in a serverless edge computing system that has multiple edge servers, there are four different outcomes for processing request $r := (s, f)$ based on the state of c_f, and resulting in different latency[4]:

- *Cold Start* (*e.g.*, r_2): If c_f is not in the memory of s, one option is to initialize a new container on s, known as Cold Start. If there is insufficient memory available, containers will be terminated[5]. The latency for processing request r consists of the execution time of function f and the initialization latency, represented as $t^e_f + t^c_f$.
- *Late-Warm* (*e.g.*, r_4): If the state of c_f in the memory of edge server s is *initializing*, we call it Late-Warm. The latency for processing request r includes the execution time of function f and the waiting time for c_f to finish initializing, denoted as $t^e_f + t^q_f$, where $0 < t^q_f < t^c_f$.
- *Warm Start* (*e.g.*, r_3): When there is an already initialized container c_f for the request of f, it is known as a Warm Start. The latency for processing this request is the execution time of f, t^e_f.

[4]In this paper, relaying request r to another server and initializing a new container is not allowed, as it may fill up all servers' memory, causing high execution time and cold latency.

[5]If all the containers are executing and not allowed to terminate, resulting in a request failure, we discuss this in section VIII.

- *Relay* (*e.g.*, r_1): If c_f does not exist in the memory of edge server s, but there is an initialized c_f on another edge server, s', one option is to relay request r to s' for processing, referred to as Relay. The latency for processing r would then be the time to execute function f, denoted as t_f^e.

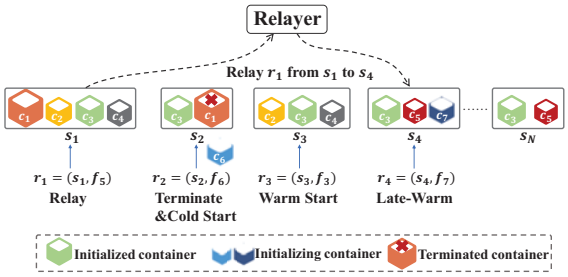


Fig. 6. Container caching on multiple edge servers with 4 different cases: Relay, Terminate & Cold Start, Warm Start and Late-Warm.

B. Problem Formulation

The objective of this problem is to minimize the total latency for processing all function requests. Let $t_{r:=(s,f)}$ denote the latency incurred by processing the request $r := (s, f)$.

Problem P:

$$\begin{aligned} &\text{minimize} \sum_{r\in\mathcal{R}} t_{r:=(s,f)} \\ s.t. \quad &\sum_{c_f \text{ in server } s_i} z_f \le K_i \qquad \forall i \in \{1, 2, \ldots, N\} \end{aligned} \tag{1}$$

For the hardness of Problem P, its simplified version, where each container is of uniform size, has been proved NP-Complete [60]. Further, we have the following theorem.

Theorem 1. *All online algorithms for Problem P have a competitive ratio lower bound as $\Omega(T_c K)$, T_c and K are the maximum container cold start latency and the memory size, respectively.*

Proof. We use *pure* and *bursty* requests. A pure request for f_i on server s has $T_c + 1$ slots, with f_i on s in the first slot and no requests in the rest. A bursty has $2T_c$ slots, with f_i on s in the first T_c slots and no requests in the rest. The latency is t_f^e for warm pure requests, $t_f^e + t_f^c$ for cold pure requests, $T_c t_f^e$for warm bursty requests, and $T_c t_f^e + \frac{t_f^c(t_f^c+1)}{2}$ for cold bursty requests. Let r_i^p and r_i^b be pure and bursty requests for f_i. Assume $K + 1$ different functions are requested.

Let $\mathcal{A}$ be an online algorithm for problem P. We assume that the containers of functions $f_1, \ldots, f_K$, *i.e.*, $c_{f_1}, \ldots, c_{f_K}$ have been initialized initially. The constructor first pure requests r_{K+1}^p, which terminates one container from $c_{f_1}, \ldots, c_{f_K}$. Then it repeats bursty requests for K times. The j-th bursty request is $r_{i_j}^b$, where c_{f_j} is the terminated container before the request. So, for $\mathcal{A}$, each bursty request has latency $T_c t_f^e + \frac{t_f^c(t_f^c+1)}{2}$, and the total latency of $\mathcal{A}$ is $t_f^c + t_f^e + K(T_c t_f^e + \frac{t_f^c(t_f^c+1)}{2})$, the total cold latency is $t_f^c + K\frac{t_f^c(t_f^c+1)}{2}$. However, for the optimal algorithm, the total latency is $t_f^c + t_f^e + KT_c t_f^e$, and the total cold latency is t_f^c. Therefore, the competitive ratio is bounded by $\Omega(T_c K)$. □

IV. Online Algorithm

In this section, we propose OnCoLa, an online container caching algorithm that supports relaying on multiple edge servers with Late-Warm. We design a priority p_{c_f} to represent the cost-effectiveness of container c_f in reducing total latency. We first present the framework of OnCoLa to handle relaying and terminating containers by using priority in Sec. IV-A. Then, we introduce the detailed design of the priority in Sec. IV-B. We also prove that OnCoLa is $O(T_c^{3/2}K)$-competitive in Sec. IV-D.

A. Online Container Caching on Multiple edge servers

In the online container caching problem on multiple edge servers, the key challenges involve relaying or locally processing requests and selecting containers to terminate under memory insufficiency. The primary approach used by OnCoLa to address these challenges at hand is that it prioritizes caching containers that offer a greater reduction in latency for each unit of memory used. To accomplish this, we assign a priority value p_{c_f} to each container c_f, and the specific details on how we determine this priority are described in Sec. IV-B. Request frequency is also an important factor to consider while selecting containers for termination when memory is insufficient [49], [61], [62]. OnCoLa takes this into account in two ways, via Line 26 in Algorithm 1 for containers cached in memory but not currently requested, and through Algorithm 2 for containers corresponding to the current request.

Algorithm 1 shows the details of OnCoLa, for an online arriving function request $r := (s, f)$, based on the state of c_f, and each container's priority, executes r result in Warm, Late-Warm, Cold Start or Relay. Initially, the memory for caching containers is empty (Line 2). At any time T, the container states are updated, and it is checked if all buffered requests for f on s can be executed (Line 6 to 11). When a new request $r := (s, f)$ for function f arrives at s, the state of container c_f on s is checked. If the state is INITED, *i.e.*, the state of c_f is initialized or executing, it is a Warm start (Line 14). If c_f is still initializing, it is a Late-Warm (Line 16). When c_f does not exist on s, if there is an edge server s' that has c_f and c_f with the lowest priority on s, r is relayed to s' (Line 18 to 19). Next, if it has not been relayed to other servers, and there is insufficient memory to start a new container, the lowest priority container(s) are terminated to release memory (Line 23 to 25). The priority of each container is decreased by p_{min}, *i.e.*, decrease the priorities of the containers cached in memory but not currently requested (Line 26). Then c_f is initialized on s, which is a Cold Start (Line 28 to 30).

B. Priority

In the online container caching problem on edge servers, since containers have varied cold start latency, execution time, and memory footprints, it is crucial to evaluate how caching different containers impacts the total latency quantitatively. In OnCoLa, we address this by assigning a priority p_{c_f} to each container c_f to indicate its cost-effectiveness, represented as the ratio of latency reductions to memory footprint. The

Algorithm 1: OnCoLa

Input *Request* $r := (s, f)$, *Priority* p_{c_f}, $z_f = z_f^e$;
$\mathcal{C} \leftarrow \varnothing$, $\mathcal{C}$ represents the containers cached in the memory of s;
Initializing containers $\mathcal{C}_{\text{Init}} \leftarrow \varnothing$, $(s, c_f, t) \in \mathcal{C}_{\text{Init}}$ means container c_f will be fully initialized on s at time t;
Timer $T \leftarrow 0$;
while True **do**
  for $(s, c_f, t) \in \mathcal{C}_{\text{Init}}$ **do**
    if $t <= T$ **then**
      if c_f.state = INITING **then**
        c_f.state $\leftarrow$ INITED;
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{c_f\}$;
      Serve all the buffered requests for f on s;
  while new request $r := (s, f)$ for function f on s arrive at T **do**
    if c_f.*state* = INITED **then** // Warm
      Execute f in c_f on s with latency t_f^e;
    if c_f.*state* = INITING **then** // Late-Warm
      Execute f in c_f on s at time t with latency $t - T + t_f^e$;
    if c_f.*state* = OUT **then**
      if there is a server s' has c_f and p_{c_f} is the lowest in s **then** // relay
        Relay $r := (s, f)$ to s' with latency t_f^e;
      else
        while remain size of $s < z_f$ **do**
          if $p_{c_{f'}}$ is the lowest priority **then**
            $p_{min} = p_{c_{f'}}$;
            Terminate $c_{f'}$ on s, $\mathcal{C} \setminus \{c_{f'}\}$;
            $c_{f'_{\text{state}}} \leftarrow$ OUT;
        For container $c_f \in \mathcal{C}$, $p_{c_f} = p_{c_f} - p_{min}$;
        $p_{min} \leftarrow 0$;
        c_f.state $\leftarrow$ INITING// ColdStart;
        $\mathcal{C}_{\text{Init}} \leftarrow \mathcal{C}_{\text{Init}} \cup \{(s, c_f, T + t_f^c)\}$;
        Initializing c_f on s with $t_f^e + t_f^c$;
    UpdatePri(s, f);
  $T \leftarrow T + 1$;

intuition behind the priority is OnCoLa prefers to cache those containers that offer greater latency reduction per unit of memory used in memory. When faced with memory insufficient, OnCoLa terminates the container with the lowest priority, allowing for greater latency reduction with less memory use. The method to calculate p_{c_f} is detailed in Eqn. 2, incorporates estimates of latency reduction and memory footprint as the numerator and denominator, respectively. Next, we detail how we estimate the latency reduction and the memory footprint.

As shown in Eqn. 2, $(1-\gamma)\cdot({t_f^c}^2 + t_f^c t_f^e) + \gamma \cdot c_f$.AvgLate represents the estimation of latency reduction for c_f, while z_f denotes the estimation of its memory footprint. The rationale for calculating these estimations is detailed as follows.

$$p_{c_f} = \frac{(1-\gamma)\cdot({t_f^c}^2 + t_f^c t_f^e) + \gamma \cdot c_f.\text{AvgLate}}{z_f}. \quad (2)$$

$\underline{{t_f^c}^2 + t_f^c t_f^e}$: In serverless edge computing, before the container c_f is fully initialized, the outcome of requests for function f is Late-Warm rather than Warm Start. Requests might experience a maximum waiting time of t_f^c before processing begins. So, for a Cold Start, in the worst case, if t_f^c requests for f within the next t_f^c time, the total latency is $(t_f^c+1)t_f^c/2+t_f^e t_f^c$. This means that if c_f is cached in memory, the maximum latency reduction could be $(t_f^c+1)t_f^c/2+t_f^e t_f^c$. To simplify the expression, we use ${t_f^c}^2 + t_f^c t_f^e$ to estimate the latency reduction for caching c_f in memory.

$\underline{c_f.\text{AvgLate}}$: As Fig. 3 shows, the cold latency t_f^c varies. Moreover, the skewed popularity of requests for different functions means that not all requests will experience the worst case. Hence, using only ${t_f^c}^2+t_f^c t_f^e$ to estimate the latency reduction for caching c_f in memory is not appropriate. We calculate c_f.AvgLate in Line 3 to 8 in Algorithm 2, representing the average latency caused by a cold start of c_f during online function request processing. This value is based on the actual latency observed during online execution rather than a fixed value. On the other hand, ${t_f^c}^2 + t_f^c t_f^e$ captures the maximum case. To incorporate both maximum case and online execution, we compute c_f.cost $= (1-\gamma)\cdot({t_f^c}^2+t_f^c t_f^e)+\gamma\cdot c_f$.AvgLate as the estimation of latency reduction of caching c_f in memory, using γ to balance between the two methods.

$\underline{z_f}$: We use z_f to estimate the memory footprint of container c_f, which depends on its state. When a container is initialized but not executing, its memory footprint z_f^p is the minimum required to store its virtual environment configuration and metadata. When a container is executing, its memory footprint z_f^e depends on the function code running inside. Our experiments on Nano show that the difference between z_f^e and z_f^p can be up to $100\times$, as Table II shows. Therefore, it is not appropriate to use z_f^e or z_f^p alone as the memory footprint of c_f, but instead, we use z_f. Specifically, $z_f = R_{\text{run}} \cdot z_f^e + \max(1 - R_{\text{run}}, 0) \cdot z_f^p$. Here, R_{run} represents the proportion of time that c_f has been in the executing state since it was fully initialized.[6]

C. Memory Adjustment

Additionally, as a component of OnCoLa, we propose a conservative *memory adjustment method* to prevent sudden increases in executing time t_f^e, cold latency t_f^c, and request failures. This method comprises two components: profiling memory thresholds for different servers, and enabling *memory growth* on servers experiencing dense request arrivals to prevent overly conservative memory thresholds. Specifically, we profile the memory usage percentage M_{th} of different servers

[6] R_{run} can exceed 1 when a container in the executing state processes multiple function requests for the same function by forking new processes, such as "fork fprocess" in OpenFaaS [63].

Algorithm 2: UpdatePri

Input *Edge Server s, function f*
if c_f.state $=$ OUT **then**
  c_f.cumLate $\leftarrow c_f$.cumLate $+ t_c^f$;
  c_f.numLate $\leftarrow c_f$.numLate $+ 1$;
if c_f.state $=$ INITING **then**
  c_f.cumLate $\leftarrow$ Total Late-Warm latency of all buffered requests;
  c_f.numLate $\leftarrow c_f$.numLate $+ 1$;
$c_f.\text{AvgLate} = \frac{c_f.\text{cumLate}}{c_f.\text{numLate}}$;
$c_f.\text{cost} \leftarrow (1-\gamma)\cdot(t_f^{c\,2} + t_f^c t_f^e) + \gamma \cdot c_f.\text{AvgLate}$;
$R_{run} = \frac{\text{Total time of } c_f \text{ executing } f}{\text{Duration since } c_f \text{ initialized}}$;
$z_f = R_{run}\cdot z_f^e + \max(1-R_{run}, 0)\cdot z_f^p$;
$p_{c_f} = \frac{c_f.\text{cost}}{z_f}$;

during sudden increases, defined as more than a 20% growth in t_f^e, t_f^c, or request failures. We set the memory size K_i^a for caching containers on server s_i as $K_i \cdot M_{\text{th}}$. Here, M_{th} is referred to as the initial *memory threshold*. To adjust the memory size online, we maintain a ghost list [62]. When container c_f on server s is terminated, we add f to the ghost list (without keeping c_f in memory). If $r = (s_i, f)$ re-arrives within the cold start latency t_f^c and before this, there have been no sudden increases in t_f^e, t_f^c, and request failures, we increase K_i^a by z_f^e if $K_i^a + z_f^e \le K_i$, we call it *memory growth*.

D. Theoretical Analysis

Lemma 1. *OnCoLa is $O(K)$-competitive for container caching without Late-Warm.*

Proof. We use the potential function method [64] to prove this Lemma. OPT is the optimal algorithm. We define the potential function as follows:

$$\Phi = (K-1)\cdot \sum_{c_f \in \text{Mem}} p_{c_f} + K \cdot \sum_{c_f \in \text{OptMem}} p_{c_f}^{init} - p_{c_f}$$

Here Mem and OptMem indicate the memories of OnCoLa and OPT, respectively. For containers not in memory, $p_{c_f} = 0$, and $p_{c_f}^{init}$ is the priority of c_f when it starts cold start. Initially, Φ is zero, and finally, $\Phi \ge 0$, satisfying the requirements of a potential function. For each request, we have:

- If OnCoLa initializes a container with $p_{c_f}^{init}$, Φ decreases by at least $p_{c_f}^{init}$.
- If OPT initializes a container with $p_{c_f}^{init}$, Φ increases by at most $K \cdot p_{c_f}^{init}$.
- Otherwise, Φ does not increase.

These facts imply that the cost incurred by OnCoLa is bounded by K times the cost incurred by OPT.

Next, we analyze in detail the impact of different cases on Φ after receiving one request.

- OPT terminates a container c_f: Φ does not increase.
- OPT initializes a container c_f: Φ increases by at most $K \cdot p_{c_f}^{init}$.
- OnCoLa reduces the priority for all containers in Mem: Φ decreases by at least 0.
- OnCoLa terminates a container c_f: Φ is unchanged.
- OnCoLa realying the request from s to s': Φ is unchanged.
- OnCoLa initializes the request container c_f and sets $p_{c_f}^{init}$: Φ decreases by $(K-1)p_{c_f}^{init} + K p_{c_f}^{init} = p_{c_f}^{init}$.

Thus, the cost incurred by OnCoLa is bounded by K times the cost incurred by OPT, OnCoLa is $O(K)$ - competitive for container caching without Late-Warm. □

Theorem 2. *OnCoLa is $O(T_c^{3/2}K)$-competitive for container caching with Late-Warm.*

Proof. We define some notations for this proof. Let $\text{ALG}(t_f^c)$ and $\text{OPT}(t_f^c)$ be the total latency of OnCoLa and the offline optimal in the online container caching model with Late-Warm. Let $\text{MALG}(t_f^c)$ and $\text{MOPT}(t_f^c)$ be the total cost of the online algorithm MALG and the offline optimal of online container caching on multiple edge servers, where MALG is c-competitive and t_f^c is the cost to start c_f. We have $\text{MALG}(t_f^c) \le c\cdot\text{MOPT}(t_f^c)$. In the proof, we use t_c to present t_f^c and t_e to present t_f^e.

1. $\text{ALG}(t_c) \le \text{MALG}(t_c^2 + t_c t_e)$.

We define the request sequence of a function as all requests to f from the cold start of c_f to the next cold start or a relaying for f. Each request sequence of f has one cold start of c_f and zero or more Late-Warm of f. Each initialization of f causes at most $t_c - 1$ Late-Warm, so the initialization latency of each request sequence of f of $\text{ALG}(t_c)$ is at most $\frac{t_c\cdot(t_c+1)}{2}$. The initialization cost of each request sequence of f of $\text{MALG}(t_c^2 + t_c t_e)$ is $t_c^2 + t_c t_e$. For each relaying of f, the latency of f in OnCoLa is t_e, and the cost of f in MALG is t_e. Thus, $\text{ALG}(t_c) \le \text{MALG}(t_c^2 + t_c t_e)$.

2. $\text{MOPT}(t_c^2 + t_c t_e) \le T_c \cdot \text{MOPT}(t_c + t_e)$.

In the model of online container caching on multiple edge servers, let $\mathcal{S}_1$ and $\mathcal{S}_2$ request the same functions, with initialize costs $(w_1, w_2, \ldots, w_n)$ and $(\alpha w_1, \alpha w_2, \ldots, \alpha w_n)$ in $\mathcal{S}_1$ and $\mathcal{S}_2$. Then $\text{MOPT}(\mathcal{S}_1) = \alpha \cdot \text{MOPT}(\mathcal{S}_2)$. If $(w_1, w_2, \ldots, w_n)$ and $(w_1', w_2', \ldots, w_n')$ are the cold start costs in $\mathcal{S}_1$ and $\mathcal{S}_2$, and $w_i \le w_i'$ for all i, then $\text{MOPT}(\mathcal{S}_1) \le \text{MOPT}(\mathcal{S}_2)$. So, $\text{MOPT}(t_c^2 + t_c t_e) \le T_c \cdot \text{MOPT}(t_c + t_e), T_c = \max t_c$.

3. $\text{MOPT}(t_c + t_e) \le \sqrt{T_c}\cdot\text{OPT}(t_c)$.

Like the proof above, we define the request sequence of f as all requests to f from a cold start of c_f to the next one. Each request sequence of f has a cold start and zero or more Late-Warm. Let n be the number of Late-Warm ($0 \le n \le t_c - 1$) and $d_1, d_2, \ldots, d_n$ be their latencies. The average latency of cold start and Late-Warm is at least $\frac{t_c + t_e + \sum_{i=1}^n d_i}{n+1} \ge \frac{t_c}{n+1} + t_e + \frac{n}{2} \ge \sqrt{t_c} + t_e$ in $\text{OPT}(t_c)$. In the model without Late-Warm, the average cost is $t_c + t_e$ for $\text{MOPT}(t_c + t_e)$, so $\text{MOPT}(t_c + t_e) \le \frac{t_c + t_e}{\sqrt{t_c} + t_e}\cdot\text{OPT}(t_c)$. Since $t_c \ge 1, t_e \ge 1$, the maximum of $\frac{t_c + t_e}{\sqrt{t_c} + t_e}$ is $\sqrt{t_c}$, so $\text{MOPT}(t_c + t_e) \le \sqrt{T_c}\cdot\text{OPT}(t_c)$, where T_c is the maximum of t_c for all containers.

Thus, $\mathrm{ALG}(t_c) \le c \cdot \mathrm{MOPT}(t_c^2 + t_c t_e) \le T_c \cdot c \cdot \mathrm{MOPT}(t_c + t_e) \le T_c^{3/2} \cdot c \cdot \mathrm{OPT}(t_c)$. *i.e.*, $\mathrm{ALG}(t_c) \le T_c^{3/2} \cdot c \cdot \mathrm{OPT}(t_c)$.

Since the competitive ratio of OnCoLa on the model without Late-Warm is $O(K)$, OnCoLa is $O(T_c^{3/2}K)$-competitive for container caching with Late-Warm. □

Message complexity. Since the serverless edge computing system consists of N edge servers and one Relayer as shown in Fig. 6. The Relayer maintains the container states of all edge servers. According to Theorem 3, the message complexity is $O(c_\alpha \cdot \min(N, m_i))$. Since the N edge servers and the Relayer are connected in a local area network (LAN), the latency of transmitting $O(c_\alpha \cdot \min(N, m_i))$ messages is considered negligible compared to the execution time and cold latency.

Theorem 3. *The message complexity of* OnCoLa *is* $O(c_\alpha \cdot \min(N, m_i))$ *at any timeslot* i*, where* c_α *is a constant,* $m_i \le |\mathcal{R}|$.

Proof. At times i, let the number of requests be m_i, and we have $\sum m_i = |\mathcal{R}|$. For m_i requests arriving on N edge servers, assume each request leads to an edge server updating container states at the Relayer, with a constant c_α messages per request. Then, with m_i requests on N edge servers, the message complexity is $O(c_\alpha \cdot \min(N, m_i))$. □

V. Implementation on Real Edge Devices

In this section, we implement and evaluate OnCoLa on an edge cluster with PI4B and Nano [65]–[67]. We conducted experiments on three types of workloads: Low, Medium and High, consisting of 10 commonly used functions for processing IoT data. The experimental results demonstrate that OnCoLa significantly reduces latency by 21.38% and reduces the request failure rate by up to 2.3× compared to the commonly used fixed-duration container caching policy (*i.e.*, TTL policy [53], [54]).

A. Experimental Setup

Device and Platform. As shown in Fig. 7, our experiments are conducted on an edge cluster of 4 PI4B and 4 Nano, with another PI4B as the Relayer. PI4B is an edge device with an ARM CPU (1.5GHz) and 1GB RAM, and Nano has an ARM CPU (1.43GHz), 4GB RAM, and a GPU (921 MHz). By default, the swap is turned off on all devices.

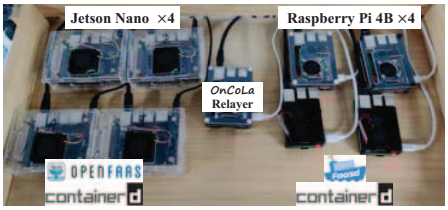


Fig. 7. Overview of the edge cluster.

We deploy OpenFaaS [54] on Nano and faasd [56] on PI4B, both using containerd [55] as the container runtime. We enable GPU supported in OpenFaaS by mounting the nvidia-container-runtime on Nano [68], [69]. We measure the relay time for relaying requests from one device to the Relayer and then to the target. Through hundreds of measurements, the average relay time is only 16.96 milliseconds (ms), and even under high memory usage percentage, it does not exceed 100 ms. We consider this negligible compared to cold latency and function execution time.

Functions. In this experiment, as shown in Table II, we use 10 commonly used functions for IoT data processing as the composition of the workload.

Workload. To evaluate the performance of OnCoLa under different workload types, we generate three types of workloads, with each workload containing 80,000 function requests across the 10 functions and 8 devices. The IC function requests cannot be processed on PI4B. The three workload types are:

- *Low*: 80% of requests are for 2 functions with small memory footprints and short execution times, *i.e.*, Node and Curl, while the other 20% of requests are for the other functions. The average inter-request interval is 0.5 seconds.
- *Medium*: Each function has 8000 requests, which are evenly distributed to the 8 devices (except for IC). The average inter-request interval is 0.5 seconds.
- *High*: The number of function requests is the same as *Medium*. The average inter-request interval is 0.2 seconds.

B. Experimental Results

Average Latency. We compare OnCoLa ($\gamma = 0.6$, the memory threshold for PI4B and Nano is 40% and 60%, respectively) with the widely used TTL policy (which caches the containers for 5 minutes). And the metric is the average latency of all successfully completed requests. As Fig. 8 shown, OnCoLa reduces the latency by 10.16%, 21.38% and 14.75% for Low, Medium and High workloads, respectively.

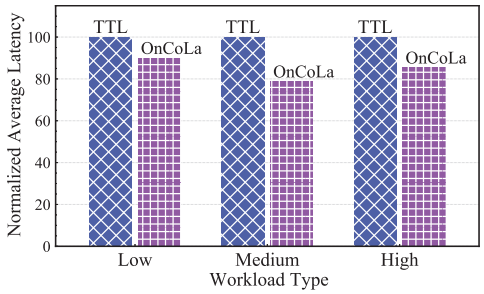


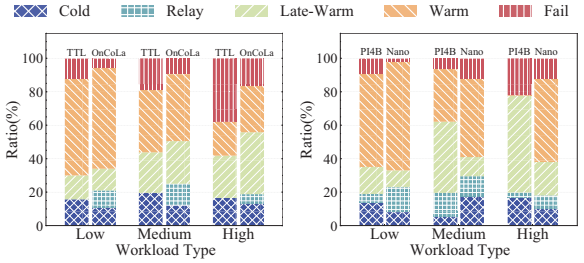
Fig. 8. The normalized average latency under three different workloads.

The Ratio of Outcomes. Fig. 9(a) shows the ratios of Cold, Relay, Late-Warm, Warm, and Fail for OnCoLa and TTL under the three workloads. Fail represents the request failure rate, indicating the percentage of failed function requests out of the total number of requests, with their latency excluded from the total. We observe that the request failure rate gradually increases from Low to High workload for both OnCoLa and TTL, and OnCoLa reduces the request failure rate by up to 2.3× compared to TTL. In the Low workload, both OnCoLa and TTL exhibit a higher Warm Start ratio. In the High workload, with an average inter-request interval of 0.2s and more requests for functions with longer cold start latency, the Late-Warm ratio for OnCoLa and TTL increases by 184.62% and 76.92%, respectively, compared to the Low workload.

TABLE II
FUNCTIONS USED IN EXPERIMENT ON PI4B AND NANO

Function	PI4B				Nano				Description
	Cold	Exec	Idle Size	Exec Size	Cold	Exec	Idle Size	Exec Size	
MM	1.312s	0.543s	20MB	110MB	2.125s	1.525s	25MB	104MB	Matrix Multiplication.
FFT	1.420s	0.462s	22MB	69MB	1.620s	0.672s	25MB	78MB	Fast Fourier transform.
STT	1.232s	0.925s	18MB	54MB	1.318s	1.120s	22MB	58MB	Speech to Text.
AD	1.125s	0.8s	13MB	56MB	0.825s	0.625s	10MB	63MB	Audio Denoising.
RSA	2.866s	1.2s	12MB	47MB	1.658s	1.336s	11MB	58MB	Data Encryption and Decryption.
PCA	1.623s	3.623s	15MB	70MB	1.225s	2.812s	13MB	72MB	Dimensionality Reduction.
RE	1.699s	2.256s	12MB	50MB	1.350s	1.813s	13MB	56MB	Resizing images.
IC	-	-	-	-	2.749s	6.446s	10MB	1060MB	Image Classification.
Node	1.010s	0.051s	16MB	20MB	1.323s	0.025s	19MB	21MB	Monitor the cpu and memory of device.
Curl	1.332s	0.603s	6MB	8MB	1.414s	0.204s	6MB	8MB	Get information about other devices.

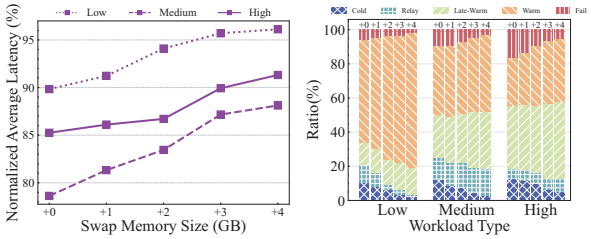
Cold start latency, execution time (both under 60% memory usage), idle memory usage, executing memory usage and the description of the functions.



(a) Different type workloads (b) Different devices

Fig. 9. Ratio of outcomes for processing requests.

The Ratio of Outcomes on Different Devices. Fig. 9(b) presents the ratio of the outcomes on PI4B and Nano when using OnCoLa. Under the Low workload, where 80% of requests are for Node and Curl, both PI4B and Nano exhibit a high Warm ratio of 55.77% and 65.12%, respectively. However, under the Medium workload, the Fail ratio on Nano reaches 12%, primarily due to more IC requests. Under the High workload, the Fail ratio is 22%, and Late-Warm increases to 57.78%, while only 0.44% are Warm on PI4B.

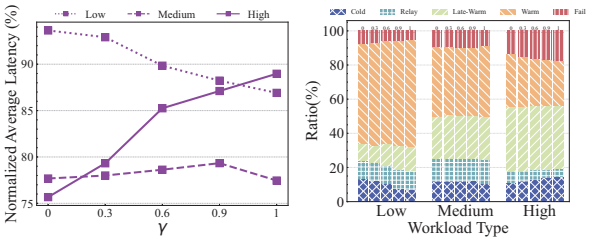


(a) Latency (b) Ratio of Outcomes

Fig. 10. Impact of memory size of devices.

Impact of Memory Size of Devices. To evaluate the scalability of OnCoLa regarding the memory size, we enabled swap to extend the memory sizes of PI4B and Nano, incrementally adding from 1GB to 4GB, termed as swap memory. Results under three workloads are depicted in Fig.10. As the swap memory size increases, allowing more containers to be cached in memory, OnCoLa's performance decreases but remains at least 3.88% better than TTL. Fig.10(b) illustrates that with the increase in swap memory, the proportion of Warm increases while the proportion of Fail decreases.

Impact of γ. We vary the parameter γ from 0 to 1 to evaluate the scalability of OnCoLa, as shown in Fig. 11. Fig. 11(a) shows that a smaller γ results in better performance of OnCoLa under the High workload. This is attributed to the fact that in Eqn. 2, the latency reduction with the maximum case is multiplied by $1-\gamma$. The High workload is similar to the maximum case. Conversely, for the Low workload, larger γ enhances the performance of OnCoLa. For the Medium workload, the performance of OnCoLa remains relatively stable, showing no significant variation.

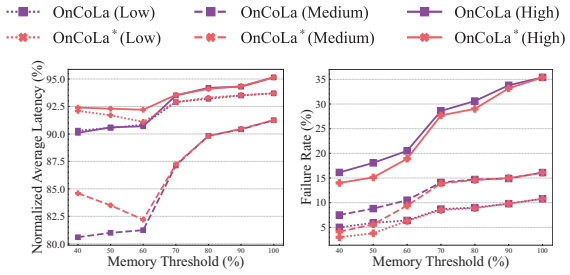


(a) Latency (b) Ratio of Outcomes

Fig. 11. Impact of γ.

Impact of Memory Adjustment. In Sec. IV-C, we introduce a memory adjustment method. To assess its effectiveness and the scalability of OnCoLa in relation to the initial memory threshold, we vary the threshold from 40% to 100%. In addition, we evaluate the efficacy of memory growth by comparing OnCoLa with its variant, OnCoLa*, which does not incorporate memory growth. We conduct these experiments under three workloads, setting the TTL strategy's initial memory threshold in the same range. As Fig. 12(b) demonstrates,

when we set the initial memory threshold no more than 60%, OnCoLa experiences a higher failure rate than OnCoLa*. However, OnCoLa consistently achieves lower latency than OnCoLa*. This is because OnCoLa grows its memory size under dense requests, leading to a larger memory size. Moreover, when the initial memory threshold exceeds 60%, we observe a sudden increase in both latency and failure rates, as OnCoLa stops further memory growth. Overall, Fig. 12 confirms the effectiveness of the memory adjustment method. By limiting the initial memory threshold, we successfully prevent sudden increases, effectively reducing the proportion of request failures and the overall latency. Additionally, the implementation of memory growth in OnCoLa plays a crucial role in decreasing latency.



(a) Latency (b) Failure Rate

Fig. 12. Impact of memory adjustment.

VI. Evaluation

We evaluate the performance of OnCoLa using the AliFC Trace [14], and the Azure Trace [43]. Compared with GD, the state-of-the-art algorithm that deals with container caching in serverless computing, OnCoLa can reduce the latency by up to 27.8%. Compared with LLB, the algorithm that supports relaying requests to other servers, it improves by 23.23%, under the default setting. Through scalability analysis on the number of edge servers, total memory size, initial memory threshold, and γ, OnCoLa consistently outperforms baselines.

A. Methodology

The total memory size of edge servers in OnCoLa is determined similarly to [70], which is the sum of the sizes of containers corresponding to the most active functions. The default configuration consists of 200 edge servers, and their total memory size is calculated as the sum of the initialized memory footprint of the top 40% active functions' containers. The default value for γ is 0.6. To handle varying memory sizes among edge servers, we allocate the total memory size to N edge servers using Eqn. 3, and the memory threshold is 60% for each server. We categorize all edge servers into 5 types numbered i, where edge servers with the same $i\%N$ have the same memory size.

$$K_i = (i\%N + 1)\left\lceil \frac{\text{Total Memory Size}}{(15\lfloor (N/5) \rfloor + \frac{(1+N\%5)(N\%5)}{2})} \right\rceil . \quad (3)$$

The metrics used to evaluate the performance of algorithms is the total latency incurred of all requests, including the function execution time and cold start latency.

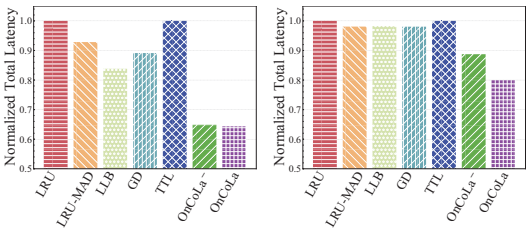
Workloads. The AliFC trace contains 398,172 requests for 3122 functions, while the Azure trace has 59,312 requests for 200 functions used in our simulations. Since both traces originate from serverless computing and lack edge server information for the requests, we use the Machine ID from Google's trace [71], and use Machine ID modular N as the edge server. The two traces differ in average request locality and average Late-Warm intensity. We define the request locality of a request sequence as the ratio of consecutive requests for the same function to the total requests on an edge server, with average request locality as the mean across total servers. The Late-Warm intensity is the ratio of Late-Warm requests out of all requests when caching all requested containers in memory on an edge server, and the average Late-Warm intensity is the mean across total servers. With 200 servers, the AliFC trace has an average request locality of 0.161 and a Late-Warm intensity of 0.159, while the Azure trace has a request locality of 0.096 and a Late-Warm intensity of 0.485.

B. Baselines

We compare the performance of OnCoLa with LRU [72], TTL [53], LRU-MAD [70], GD [49] and LLB [73]. To verify the effectiveness of OnCoLa in handling memory-sensitive, we also include OnCoLa$^-$ as a baseline.[7]

C. Experiment Results

Overall performance. We first evaluate the overall performance of OnCoLa and compare it with baselines, using default setting. The experimental results are shown in Fig. 13, where the total latency of each algorithm is normalized relative to LRU (set to 100%). The experiments in both traces show that among all baselines except OnCoLa$^-$, LLB performs the best, with its improvement attributed to relaying requests to other servers. Compared to LLB, OnCoLa demonstrates performance improvements of 23.23%, adopting priorities more suitable for multiple servers with Late-Warm. Furthermore, compared to GD, OnCoLa leverages cooperation among edge servers and utilizes priorities tailored for serverless edge computing, achieving latency improvements of 27.8%.



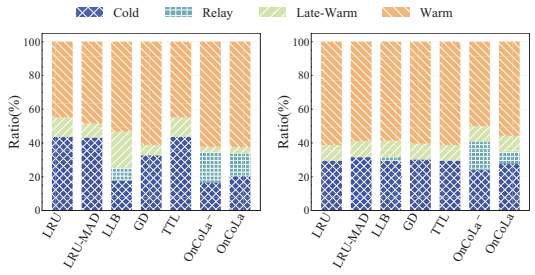
(a) AliFC (b) Azure

Fig. 13. Overall performance.

Ingredient of Latency. Fig. 14 shows the ratios of Cold Start, Relay, Late-Warm, and Warm Start for each algorithm, revealing that Warm and Relay ratios significantly impact performance. In AliFC trace, GD, OnCoLa$^-$, and OnCoLa have the highest warm ratios (around 62%). OnCoLa has a higher

[7] OnCoLa$^-$ does not consider the impact of memory usage on execution time and cold start latency, and it does not estimate the actual container footprint.

Cold Start ratio due to its memory threshold, accommodating fewer containers. However, it estimates the actual memory footprint after the container has been initialized, ensuring its Warm Start ratio is no less than OnCoLa$^-$'s.



(a) AliFC (b) Azure

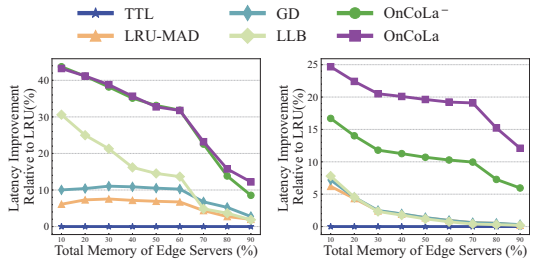
Fig. 14. Ratio of Cold-Start, Relay, Late-Warm, and Warm.

D. Scalability Analysis

In this subsection, we use the latency improvement relative to LRU to measure the performance of the algorithm, a higher latency improvement means better performance.

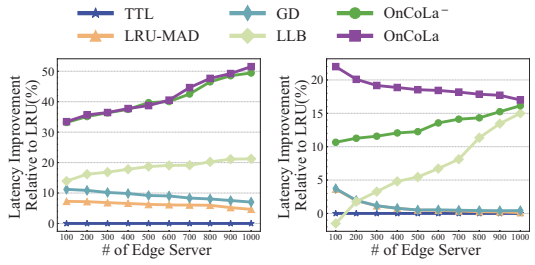
$$\text{Latency Improvement of A} = \frac{\text{Latency(LRU)} - \text{Latency(A)}}{\text{Latency(LRU)}}.$$

Total Memory Size. To assess the scalability of OnCoLa in terms of total memory size, we vary it from 10% to 90% and display the results in Fig. 15. When the total memory size is small, OnCoLa exhibits higher improvement than other baselines. However, with sufficient memory to cache frequent containers, all algorithms' performances converged, especially in the AliFC trace.



(a) AliFC (b) Azure

Fig. 15. Impact of total memory size.

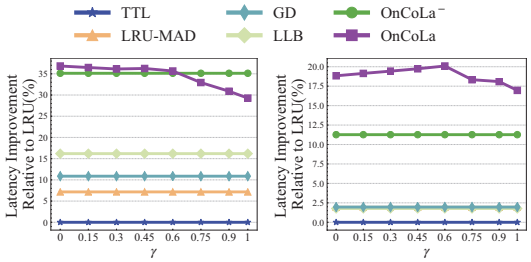


(a) AliFC (b) Azure

Fig. 16. Impact of the number of edge servers.

Number of Edge Servers. To assess OnCoLa's scalability, Fig. 16 examines the effect of changing the number of edge servers. OnCoLa consistently outperforms other algorithms with server counts ranging from 100 to 1000. With total memory size constant, more servers mean less memory per server. In the AliFC trace, OnCoLa gains from request relaying among multiple servers, enhancing performance as server numbers increase. In the Azure trace, however, performance drops due to the reduced per-server memory limiting container storage. This experiment, conducted under a fixed total memory size, primarily explores how increasing server numbers affect algorithm performance. If the per-server memory also increases with more servers, *i.e.*, the total memory size expands, the results would mirror those in Fig. 15.

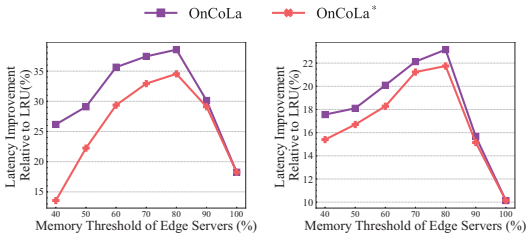
Parameter γ. We vary γ from 0 to 1 to investigate the scalability of OnCoLa, as shown in Fig. 17. In the Azure trace, the performance of OnCoLa fluctuates with changes in γ, reaching its optimum at $\gamma = 0.6$. However, in the AliFC trace, OnCoLa begins to underperform compared to OnCoLa$^-$ when γ exceeds 0.6, and the performance degrades further as γ increases. The role of γ is to adjust the estimation of latency reduction based on online execution from a cold start. The limited impact of γ in the AliFC trace is reasonable due to the smaller average Late-Warm intensity compared to the Azure trace.



(a) AliFC (b) Azure

Fig. 17. Impact of γ.

Memory Adjustment. To assess the effectiveness of the memory adjustment method and the scalability of OnCoLa, we vary threshold from 40% to 100%. We also assess the impact of memory growth by comparing OnCoLa with OnCoLa*, which lacks the memory growth feature. This evaluation involved setting LRU's memory threshold within the same range. As shown in Fig.18, OnCoLa's performance starts to decline when the threshold exceeds 80%, indicating that limiting the initial memory threshold is effective. Fig.18(a) shows that the impact of memory growth is more pronounced in the AliFC trace due to its higher average request locality compared to the Azure trace, making memory growth more likely to occur.



(a) AliFC (b) Azure

Fig. 18. Impact of memory adjustment.

VII. Related Works

A. Data Management and Processing in Serverless Computing

Serverless computing, known for its elasticity, ease of use, and fine-grained billing, is increasingly adopted for analytical query engines. For instance, Pixels-Turbo, proposed by Bian *et al.* [74], is a hybrid query engine that leverages scalable VM clusters for standard operations and functions for unpredictable workload spikes. Justen [24] employs elastic query processors in a serverless framework to improve data exchange efficiency. Spiegelberg *et al.* [75] enhance performance and cost efficiency in serverless data analytics through adaptive code generation and compilation. Burckhardt *et al.* [27] introduce Netherite for optimizing serverless workflow execution on elastic clusters. Meanwhile, Yu *et al.* [29] focus on data-centric function orchestration for complex data processes, and Ali *et al.* [26] optimize batching for ML inference requests.

B. Container Caching

Major serverless platforms like AWS and Azure use a fixed duration caching policy [51]. FaaSCache [49] uses a Greedy-Dual keep-alive policy considering the request frequency and function patterns. Shahrad *et al.* [43] propose a practical resource management policy for container caching and pre-warming. Since the requests for functions are arriving in an online manner, the online file caching policies can be used in container caching. The classical work by Sleator and Tarjan [72] introduced the notion of competitive analysis for online paging algorithms. Later works extended the problem to non-uniform file size and fetch cost [46], [76]–[79], multiple caches and request relaying [47], and machine learning advice [48], [80]. In this paper, we focus on container caching in serverless edge computing, which faces challenges from varying cold start latency and execution time due to limited resources (Memory Sensitivity), as well as the need for Request Relaying across multiple servers and the additional latency caused by Late-Warm. Existing online file caching algorithms typically assume constant retrieval latency, and container caching policies for powerful servers often neglect edge-specific constraints. We introduce `OnCoLa`, a novel approach that assigns a priority to each container, effectively dealing with these three challenges all at once with the competitive ratio. We summarize the related work in Table III.

TABLE III
Related Work of Caching Problems

Algorithms	Container footprint	Late-Warm	Request Relaying	Memory Sensitivity	Performance Guarantee
LRU [72]	Uniform	✘	✘	✘	$O(K)$
GD [49]	Uniform	✘	✘	✘	-
Landlord [46]	Non-uniform	✘	✘	✘	$O(K)$
LLB [73]	Non-uniform	✘	✔	✘	$O(K)$
LRU-MAD [70]	Uniform	✔	✘	✘	-
`OnCoLa` [this work]	Non-uniform	✔	✔	✔	$O(T_c^{3/2}K)$

VIII. Discussion

More Computing Model. In this paper, we investigate the online container caching problem to improve the performance of executing IoT data processing tasks on multiple edge servers. In this scenario, we observe that resource-limited edge servers introduce additional challenges to designing the online container caching algorithms, such as Late-Warm, Memory Sensitivity, and Request Relaying. To address these challenges, we propose a novel algorithm `OnCoLa`. `OnCoLa` can be applied to various scenarios with multiple servers or resource-limited servers, such as serverless computing, cluster computing, cloud-edge hybrid computing, and mobile computing. `OnCoLa` can adapt the priority calculation and updating algorithm to suit the needs of each scenario to ensure performance. Additionally, this paper focuses on IoT and edge devices within a LAN, overlooking dependencies and data transmission latency critical in broader applications like machine learning. Future work could integrate dependencies and data transmission based on network conditions.

Request Queuing and Waiting Latency. As shown in Fig. 1, requests arriving at T_2, T_3, and T_4 are all Late-Warm, and we assume they will all be executed at T_5 with no additional queuing latency. This assumption is reasonable as multiple requests for the same function can be processed concurrently in a single container [63]. Depending on the auto-scaling policy of the platform, it is also possible to execute multiple requests by initializing multiple containers or queuing a process. Moreover, if all edge servers' memory is used by executing containers, requests for new functions fail due to memory insufficient, *i.e.*, we adopt a strict timeout policy in this paper. If the timeout policy is relaxed, requests can wait for memory to be freed with waiting latency. A future direction is considering queuing and waiting latency when auto-scaling and request timeout policies change.

IX. Conclusion

This paper investigates the online container caching problem in serverless edge computing. We highlight the new challenges of designing an online caching algorithm with resource-limited edge servers, including Late-Warm, Memory-sensitivity, and Request Relaying. We propose an $O(T_c^{3/2}K)$-competitive algorithm, `OnCoLa`, to address these challenges. We implement `OnCoLa` and conduct experiments on edge devices to validate the improvement over the current policy. We conduct extensive simulations based on real-world traces and show that `OnCoLa` outperforms baselines. Serverless edge computing is still in its early stages, we hope this work can contribute to implementing the serverless paradigm in large-scale edge systems.

Acknowledgement

This work was supported in part by the National Key R&D Program of China under Grant 2021ZD0110400, NSFC under Grant 62132009, 62072344, U20A20177, 62232004 and the Fundamental Research Funds for the Central Universities at China. Haisheng Tan (email: hstan@ustc.edu.cn) and Ruiting Zhou (email: ruitingzhou@seu.edu.cn) are the corresponding authors of this paper.

REFERENCES

[1] M. Adil, M. Attique, M. M. Jadoon, J. Ali, A. Farouk, and H. Song, "Hopctp: a robust channel categorization data preservation scheme for industrial healthcare internet of things," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 10, pp. 7151–7161, 2022.

[2] P. Killeen, I. Kiringa, and T. Yeap, "Unsupervised dynamic sensor selection for iot-based predictive maintenance of a fleet of public transport buses," *ACM Transactions on Internet of Things*, vol. 3, no. 3, pp. 1–36, 2022.

[3] J. E. Tate, "Preprocessing and golomb–rice encoding for lossless compression of phasor angle data," *IEEE transactions on smart grid*, vol. 7, no. 2, pp. 718–729, 2015.

[4] M. Mazaheri, R. Ruiz, D. Giustiniano, J. Widmer, and O. Abari, "Bringing millimeter wave technology to any iot device," in *ACM MobiCom 2023*, pp. 1–15.

[5] Y. Luopan, R. Han, Q. Zhang, C. H. Liu, G. Wang, and L. Y. Chen, "Fedknow: Federated continual learning with signature task knowledge integration at edge," in *IEEE ICDE 2023*, pp. 341–354.

[6] H. Li, B. Tang, H. Lu, M. A. Cheema, and C. S. Jensen, "Spatial data quality in the iot era: management and exploitation," in *ACM SIGMOD 2022*, pp. 2474–2482.

[7] H. Tan, G. Li, Z. Shen, Z. Wang, Z. Han, M. Xiao, X.-Y. Li, and G. Chen, "Edge-centric pricing mechanisms with selfish heterogeneous users," *Journal of Computer Science and Technology*, 2024.

[8] 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.

[9] C. Zhang, F. Zhang, K. Chen, M. Chen, B. He, and X. Du, "Edgenn: Efficient neural network inference for cpu-gpu integrated edge devices," in *IEEE ICDE 2023*, pp. 1193–1207.

[10] T. Shaowang, N. Jain, D. D. Matthews, and S. Krishnan, "Declarative data serving: the future of machine learning inference on the edge," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2555–2562, 2021.

[11] J. Liu, Y. Xu, H. Xu, Y. Liao, Z. Wang, and H. Huang, "Enhancing federated learning with intelligent model migration in heterogeneous edge computing," in *IEEE ICDE 2022*, pp. 1586–1597.

[12] Z. Jiang, Y. Xu, H. Xu, Z. Wang, C. Qiao, and Y. Zhao, "Fedmp: Federated learning through adaptive model pruning in heterogeneous edge computing," in *IEEE ICDE 2022*, pp. 767–779.

[13] R. Jeyaraj, A. Balasubramaniam, A. K. MA, N. Guizani, and A. Paul, "Resource management in cloud and cloud-influenced technologies for internet of things applications," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–37, 2023.

[14] 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 *USENIX ATC 2021*.

[15] U. Tadakamalla and D. A. Menascé, "Characterization of iot workloads," in *EDGE 2019*, pp. 1–15.

[16] M. Yuan, L. Zhang, X. You, and X.-Y. Li, "Packetgame: Multi-stream packet gating for concurrent video inference at scale," in *ACM SIGCOMM 2023*, pp. 724–737.

[17] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *ACM MobiCom 2019*, pp. 1–16.

[18] 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.

[19] 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 *ISCA 2022*.

[20] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, "When serverless computing meets edge computing: architecture, challenges, and open issues," *IEEE Wireless Communications*, vol. 28, no. 5, pp. 126–133, 2021.

[21] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: running latency sensitive serverless computations at the edge," in *ACM HPDC 2021*.

[22] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *ACM/IFIP Middleware 2020*.

[23] S. Gupta, S. Rahnama, E. Linsenmayer, F. Nawab, and M. Sadoghi, "Reliable transactions in serverless-edge architecture," in *IEEE ICDE 2023*, pp. 301–314.

[24] D. Justen, "Cost-efficiency and performance robustness in serverless data exchange," in *ACM SIGMOD 2022*, pp. 2506–2508.

[25] 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 *ACM SIGMOD 2022*, pp. 1866–1875.

[26] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Optimizing inference serving on serverless platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, 2022.

[27] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1591–1604, 2022.

[28] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *ACM SIGMOD 2021*, pp. 857–871.

[29] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *USENIX NSDI 2023*, pp. 1489–1504.

[30] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing more with less: Orchestrating serverless applications without an orchestrator," in *USENIX NSDI 2023*, pp. 1505–1519.

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

[32] 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 *ACM SIGCOMM 2022*.

[33] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *USENIX NSDI 2021*, pp. 653–669.

[34] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "Mxfaas: Resource sharing in serverless environments for parallelism and efficiency," in *IEEE/ACM ISCA 2023*, pp. 1–15.

[35] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: characterization and optimization." in *ACM/IEEE ISCA 2022*, pp. 757–770.

[36] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *ACM/IEEE ISCA 2021*, pp. 306–318.

[37] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "Specfaas: Accelerating serverless applications with speculative function execution," in *IEEE HPCA 2023*, pp. 814–827.

[38] Z. Hong, J. Lin, S. Guo, S. Luo, W. Chen, and R. Wattenhofer, "Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation," in *EuroSys 2024*.

[39] 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 *USENIX OSDI 2023*, pp. 497–517.

[40] H. Ding, Z. Wang, Z. Shen, R. Chen, and H. Chen, "Automated verification of idempotence for stateful serverless applications," in *USENIX OSDI 2023*, pp. 887–910.

[41] 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 *SOSP 2023*, pp. 231–246.

[42] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in *ACM PPoPP 2022*, pp. 46–60.

[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 *USENIX ATC 2020*.

[44] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *ACM SoCC 2022*.

[45] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

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

[47] 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*.

[48] D. Rohatgi, "Near-optimal bounds for online caching with machine learned advice," in *SIAM SODA 2020*.
[49] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ACM ASPLOS 2021*.
[50] "Use containers to build, share and run your applications," 2023, https://www.docker.com/resources/what-container/.
[51] "Aws lambda," 2023, https://aws.amazon.com/lambda.
[52] "Azure functions," 2023, https://azure.microsoft.com/en-us/services/functions/.
[53] "Apache openwhisk," 2023, https://openwhisk.apache.org.
[54] "OpenFaaS, serverless functions made simple," 2023, https://github.com/openfaas.
[55] "Containerd, an industry-standard container runtime with an emphasis on simplicity, robustness and portability," 2023, https://containerd.io.
[56] "faasd,a lightweight and portable faas engine," 2023, https://github.com/openfaas/faasd.
[57] "Aws lambda@edge," 2023, https://aws.amazon.com/lambda/edge/.
[58] "Edge routine," 2023, https://www.aliyun.com/activity/cdn/edgeroutine/.
[59] 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 *IEEE CLUSTER 2021*.
[60] P. Manohar and J. Williams, "Lower bounds for caching with delayed hits," *arXiv preprint arXiv:2006.00376*, 2020.
[61] T. Johnson, D. Shasha *et al.*, "2q: a low overhead high performance bu er management replacement algorithm," in *VLDB 1994*, pp. 439–450.
[62] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache." in *USENIX FAST 2003*.
[63] "of-watchdog," 2023, https://github.com/openfaas/of-watchdog.
[64] S. Albers, "Brics, mini-course on competitive online algorithms," *Aarhus University*, vol. 32, 1996.
[65] "End-to-end aiot w/ sagemaker and greengrass 2.0 on nvidia jetson nano," 2023, https://github.com/aws-samples/aiot-e2e-sagemaker-greengrass-v2-nvidia-jetson.
[66] C. Zhang, H. Tan, H. Huang, Z. Han, S. H.-C. Jiang, G. Li, and X.-Y. Li, "Online approximation scheme for scheduling heterogeneous utility jobs in edge computing," *IEEE/ACM Transactions on Networking*, vol. 31, no. 1, pp. 352–365, 2023.
[67] S. Kurkovsky and C. Williams, "Raspberry pi as a platform for the internet of things projects: Experiences and lessons," in *ACM ITiCSE 2017*, pp. 64–69.
[68] "Nvidia container runtime," 2023, https://developer.nvidia.com/nvidia-container-runtime.
[69] D. M. Naranjo, S. Risco, C. de Alfonso, A. Pérez, I. Blanquer, and G. Moltó, "Accelerated serverless computing based on gpu virtualization," *Journal of Parallel and Distributed Computing*, vol. 139, pp. 32–42, 2020.
[70] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *ACM SIGCOMM 2020*.
[71] C. Reiss, J. Wilkes, and J. Hellerstein, "Google cluster-usage trace," in *Technical Report*, 2011.
[72] 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.
[73] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György, "Online file caching with rejection penalties," *Algorithmica*, vol. 71, pp. 279–306, 2015.
[74] H. Bian, T. Sha, and A. Ailamaki, "Using cloud functions as accelerator for elastic data analytics," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–27, 2023.
[75] L. Spiegelberg, T. Kraska, and M. Schwarzkopf, "Hyperspecialized compilation for serverless data analytics," 2023.
[76] S. Irani, "Page replacement with multi-size pages and applications to web caching," in *ACM STOC 1997*.
[77] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, "A unified approach to approximating resource allocation and scheduling," *Journal of the ACM*, vol. 48, no. 5, pp. 1069–1090, 2001.
[78] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke, "An O(log k)-competitive algorithm for generalized caching," *ACM Transactions on Algorithms*, vol. 15, no. 1, pp. 1–18, 2018.
[79] C. Zhang, H. Tan, G. Li, Z. Han, S. H.-C. Jiang, and X.-Y. Li, "Online file caching in latency-sensitive systems with delayed hits and bypassing," in *IEEE INFOCOM 2022*.
[80] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *Journal of the ACM (JACM)*, vol. 68, no. 4, pp. 1–25, 2021.