Online Container Scheduling With Fast Function Startup and Low Memory Cost in Edge Computing

Zhenzheng Li, Jiong Lou, *Member, IEEE*, Jianfei Wu, Jianxiong Guo, *Member, IEEE*, Zhiqing Tang, *Member, IEEE*, Ping Shen, Weijia Jia, *Fellow, IEEE*, and Wei Zhao, *Fellow, IEEE*

***Abstract*—Extending serverless computing to the edge has emerged as a promising approach to support service, but startup containerized serverless functions lead to the cold-start delay. Recent research has introduced container caching methods to alleviate the cold-start delay, including cache as the entire container or the Zygote container. However, container caching incurs memory costs. The system must ensure fast function startup and low memory cost of edge servers, which has been overlooked in the literature. This paper aims to jointly optimize startup delay and memory cost. We formulate an online joint optimization problem that encompasses container scheduling decisions, including invocation distribution, container startup, and container caching. To solve the problem, we propose an online algorithm with a competitive ratio and low computational complexity. The proposed algorithm decomposes the problem into two subproblems and solves them sequentially. Each container is assigned a randomized strategy, and these container-level decisions are merged to constitute overall container caching decisions. Furthermore, a greedy-based subroutine is designed to solve the subproblem associated with invocation distribution and container startup decisions. Experiments on the real-world dataset indicate that the algorithm can reduce average startup delay by up to 23% and lower memory costs by up to 15%.**

***Index Terms*—Serverless computing, zygote container, scheduling, online optimization.**

Manuscript received 31 January 2024; revised 25 June 2024; accepted 23 July 2024. Date of publication 12 August 2024; date of current version 8 November 2024. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62272050 and Grant 62302048, in part by Guangdong Key Lab of AI and Multi-modal Data Processing, UIC under 2023–2024 Guangdong Education Department Grants, in part by Zhuhai Science-Tech Innovation Bureau under Grant 2320004002772, and in part by the Interdisciplinary Intelligence Super Computer Center of Beijing Normal University, Zhuhai. Recommended for acceptance by W. Li. *(Corresponding author: Zhiqing Tang.)*

Zhenzheng Li and Jianfei Wu are with the School of Artificial Intelligence, Beijing Normal University, Beijing 100875, China, and also with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China (e-mail: zhenzhengli@mail.bnu.edu.cn; jianfeiwu@mail.bnu.edu.cn).

Jiong Lou is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: lj1994@sjtu.edu.cn).

Jianxiong Guo and Weijia Jia are with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China, and also with Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College, Zhuhai 519087, China (e-mail: jianxiongguo@bnu.edu.cn; jiawj@bnu.edu.cn).

Zhiqing Tang and Ping Shen are with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Zhuhai 519087, China (e-mail: zhiqingtang@bnu.edu.cn; iafn@bnu.edu.cn).

Wei Zhao is with Shenzhen University of Advanced Technology, Shenzhen 518055, China (e-mail: weizhao86@outlook.com).

Digital Object Identifier 10.1109/TC.2024.3441836

I. Introduction

EXTENDING serverless computing [1], [2], [3] to edge computing [4], [5] represents a promising approach to support service. As an innovative Function-as-a-Service (FaaS) paradigm, serverless computing attains scalability by decomposing a complex monolithic application into smaller containerized functions [6]. In this paradigm, as illustrated in Fig. 1, when the serverless functions are invoked, ❶ they are first distributed to suitable edge servers. Subsequently, ❷ the corresponding containers are started up to host and execute functions. After execution, ❸ the containers become idle and are recycled[1] to conserve hardware resources [8]. However, the startup of containerized functions entails numerous initialization operations, leading to the well-known problem of cold-start delay [9], [10]. Statistics substantiate that cold-start delay varies from a few hundred milliseconds to a few seconds, comparable to the function execution duration [11], [12]. Consequently, reducing the cold-start delay in serverless computing is critically significant.

Many existing works use caching methods to reduce the cold-start delay [13], [14]. They cache the entire idle container (① in Fig. 1) for serving the subsequent function invocations, which incurs negligible warm-start delay. However, it results in significant memory consumption, which is detrimental to the comparative resource limitations of the edge servers. Recent works reduce memory consumption by caching as the Zygote containers [15], [16] (② in Fig. 1). The Zygote[2] is a special container that pre-imports public packages shared among certain containers, thus reducing memory consumption compared to caching the entire container. Moreover, Zygotes can serve as "parent containers" for these containers. When a function is invoked, the container starts up by importing the required private packages onto the Zygote instead of starting from scratch. This approach effectively accelerates the startup process compared to cold-start.

However, it is not prudent to indiscriminately replace other container caching options with Zygotes. As shown in Fig. 1, different *container caching* decisions impact subsequent decisions regarding *invocation distribution* and *container startup*:

[1]From the perspective of container caching, recycling is equivalent to choosing ③ not-cache. Recycle is equivalent to kill and destroy, as used in some papers [7].

[2]Notably, in contrast to the container, the Zygote can not directly host function. This paper refers to the Zygote container as "Zygote" for introductory purposes.

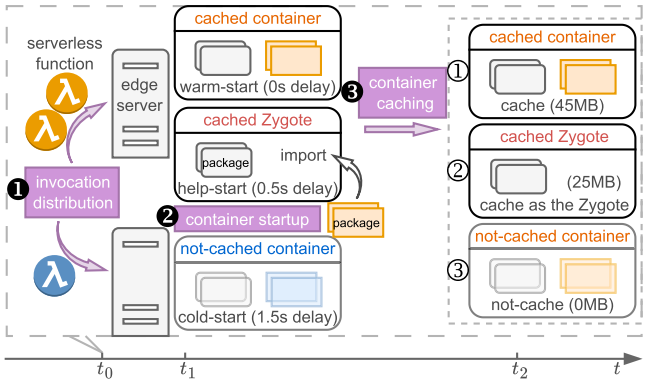


Fig. 1. Container scheduling to optimize startup delay and memory cost.

① Cache: The entire container is cached, which incurs negligible warm-start delay for subsequent invocation but at the expense of substantial memory consumption [12]; ② Cache as the Zygote: This incurs a relatively small help-start delay (inevitable delay to import private packages) for subsequent invocation but involves consuming memory resources [16]; ③ Not-cache: The container is recycled, which has no memory consumption but at the cost of significant cold-start delay for subsequent invocation. If the subsequent invocation arrives at short intervals (e.g., t_1 in Fig. 1), caching the entire container emerges as the preferred decision as it can bring negligible warm-start delay and small memory cost. Memory cost refers to the product of memory consumption and duration, which can be regarded as the price the system pays for reserving the idle containers and the Zygotes in return for accelerating the startup for subsequent invocations. Conversely, in another situation (e.g., t_2 in Fig. 1), not-cache is more advantageous due to the substantial memory costs associated with caching over a long duration. Therefore, appropriate container scheduling decisions are required to ensure fast function startup and low memory cost of the edge servers [17], [18], which has been overlooked in the literature.

To fill in such gaps, this paper aims to optimize startup delay and memory cost jointly by formulating and solving an online joint optimization problem. The objective of the optimization problem is to make container scheduling decisions to minimize startup delay and memory cost. These decisions include invocation distribution, container startup, and container caching. However, solving this problem is challenging due to the following reasons. First, considering the online nature of this problem, decisions must be made without prior knowledge of future function invocation patterns. Second, the complexity of decision variables and constraints makes solving it as a whole incur high computational costs. All these challenges contribute to the complexity of the optimization problem.

To solve the problem, this paper proposes an Online Container Scheduling (OCS) algorithm. OCS decomposes the problem into two subproblems to reduce its complexity: 1) Determining invocation distribution and container startup based on the available containers; 2) Determining container caching decisions. We analogize the container caching for a single container to the complex variant of the ski rental problem [19], [20], [21]. Then, a randomized strategy of a single container is proposed without prior knowledge of future function invocation patterns. The caching decisions are made based on the randomized strategy. OCS merges these container-level decisions to constitute overall container caching decisions. Afterward, a greedy-based subroutine is designed to solve the subproblem associated with invocation distribution and container startup decisions. OCS has theoretically guaranteed performance and low computational complexity compared to existing work. To illustrate its effectiveness, a rigorous analysis is conducted to prove the OCS's competitive ratio and computational complexity. Moreover, experiments are conducted based on the real-world dataset. The experimental results illustrate the superior performance of the OCS. Our contributions can be summarized as follows:

- This paper aims to optimize startup delay and memory cost jointly. We formulate an online joint optimization problem concerning the container scheduling.
- To solve the problem, this paper proposes an OCS algorithm with a competitive ratio and low computational complexity. OCS decomposes the problem into two subproblems and solves them sequentially.
- Experiments are carried out based on the real-world dataset. OCS can reduce up to 23% of the average startup delay and up to 15% of the memory cost.

The remainder of the paper is organized as follows. Section II reviews related work. Section III details the system model and problem formulation. Sections IV and V present the randomized strategy and the online algorithm, respectively. Section VI evaluates performance, while Section VII discusses the algorithm. Finally, Section VIII concludes the paper.

II. Related Work

Serverless computing: Serverless computing [1] has garnered substantial attention as a promising service-supporting paradigm. In recent years, various containerized serverless computing platforms have been proposed to achieve diverse objectives, including better resource efficiency [22], reduced provisioning costs [23], high-concurrency [7], decentralization [24], and lightweight isolation [25]. To reduce the container deployment time, Tang et al. [26] and Lou et al. [27] introduce layer-aware container scheduling algorithms. In Serverless, the container is recycled when the function execution is complete, and the resources allocated to the container are released. This appeals to the resource-limited edge servers. As a result, recent studies extend serverless computing to the edge to support edge services [28], [29], [30]. Despite the distinct advantages of serverless computing, it faces the well-known issue of cold-start, a concern overlooked by these works.

Cold-start problem: Considerable effort has been devoted to addressing the cold-start problem. The cold-start problem can be alleviated through container caching [12], [14], [31], [32], but at the expense of memory consumption. Moreover, Several scheduling strategies have been proposed to mitigate cold-start, including considering dependencies between

TABLE I
MAIN NOTATIONS

Parameters	Definition
$\mathcal{T}$	Set of time slots
$\mathcal{S}$	Set of servers
$\mathcal{I}$	Set of Zygotes
$\mathcal{K}$	Set of containers
λ_t^k	Number of type-k function invocations
r_k	Memory consumption of type-k container
r_i	Memory consumption of type-i Zygote
R_s	Memory capacity of server s
$e_{s,t}^k$	Number of type-k containers cached on server s
$q_{s,t}^i$	Number of type-i Zygotes cached on server s
Decisions	**Definition**
$u_{s,t}^k$	Number of type-k invocations distributed to server s
$v_{s,t}^k$	Number of Zygotes used to create the type-k containers on server s
$w_{s,t}^k$	Number of cached type-k containers recycled when memory is insufficient on server s
$x_{s,t}^k$	Number of type-k containers to be cached as the Zygotes on server s
$y_{s,t}^k$	Number of type-k containers to be recycled on server s
$z_{s,t}^i$	Number of type-i Zygotes to be recycled on server s

functions [33], runtime provisioning [34], and warm container selection [35]. To enhance memory efficiency, recent studies propose serverless computing platforms empowered by Zygotes [15], [16]. Oakes et al. [15] generalize Zygote provisioning and construct SOCK, a streamlined container and package-aware caching system. Li et al. [16] design resource-friendly Zygotes and mitigate cold-start delay by caching as the Zygotes that other functions can use. Moreover, Li et al. [36] investigate the Zygote generation and pre-warming for fast function startup in a limited resource edge cloud.

However, none of the existing studies investigate how to efficiently schedule containers and Zygotes to jointly optimize the memory cost and startup delay without prior knowledge of future function invocation patterns. Given the resource limitations of edge servers, appropriate container scheduling decisions are required to ensure fast function startup and low memory cost. To fill such a gap, this paper formulates a joint optimization problem and proposes an OCS algorithm.

III. System Model and Problem Formulation

A. System Model

We consider the configuration of an edge cluster, which comprises a set $\mathcal{S}=\{1,\cdots,|\mathcal{S}|\}$ of servers, each equipped with a finite memory capacity denoted as R_s, where $s\in\mathcal{S}$. The system works in a discrete-time manner, where the time span is denoted as $\mathcal{T}=\{1,2,\cdots,|\mathcal{T}|\}$. The duration of each time slot $t\in\mathcal{T}$ is determined by the function invocation pattern, ensuring that each function can be serviced within one slot [14]. The notation $\mathcal{K}=\{1,\cdots,|\mathcal{K}|\}$ denotes the set of containers. Notably, each container corresponds uniquely to a specific serverless function, rendering $\mathcal{K}$ suitable for representing the set of serverless functions. Zygote [15], [16], a "parent container" for certain containers, contains a bare basic container and pre-import public packages shared among these "child containers". $\mathcal{I}=\{1,\cdots,|\mathcal{I}|\}$ is used to represent the set of Zygotes. We divide the set of containers into disjoint subsets denoted as $\mathcal{K}_i$, such that $\mathcal{K}=\bigcup_{i\in\mathcal{I}}\mathcal{K}_i$. These containers share common public packages within each subset $\mathcal{K}_i$. Consequently, type-i Zygote can serve as "parent containers" for the container in $\mathcal{K}_i$ by importing private packages of that container to create the corresponding container.

The number of type-k function invocations generated at t is denoted as λ_t^k, where $k\in\mathcal{K}$. These function invocations will be distributed to the suitable servers, wherein the servers use the corresponding container to host and execute the functions. The notation $u_{s,t}^k$ denotes the number of type-k invocations distributed to server s at t. In the distribution process, it is essential to ensure that each function invocation is distributed to one server:

$$\sum_{s\in\mathcal{S}} u_{s,t}^k=\lambda_t^k,\qquad \forall k,\forall t. \tag{1a}$$

The notations $e_{s,t}^k$ and $q_{s,t}^i$ are employed to represent the number of type-k containers and the number of type-i Zygotes cached on server s, respectively. Following the distribution, a decision needs to be made on whether to start the respective container to host and execute the function. For each distributed type-k (where $k\in\mathcal{K}_i$) invocation, multiple options are available, as follows: 1) Warm-start: Directly utilize a cached container to host the function; 2) Help-start: Import the private packages into the type-i Zygote to create a type-k container; 3) Cold-start: Start up a type-k container from scratch.

The notation $v_{s,t}^k$ signifies the number of Zygotes used to create type-k containers on the server s. $w_{s,t}^k$ denotes the number of cached type-k containers that are recycled to make room for creating new containers on the server s. Note that $w_{s,t}^k$ should not exceed the number of unused cached containers. Furthermore, it is essential to ensure that $v_{s,t}^k$ concerning two constraints: it should not exceed the number of type-k invocations distributed to server s and must not surpass the number of type-i Zygotes cached on server s:

$$w_{s,t}^k\le\max\{0,e_{s,t}^k-u_{s,t}^k\},\qquad \forall k,\forall s,\forall t, \tag{2a}$$

$$v_{s,t}^k\le u_{s,t}^k,\qquad \forall k,\forall s,\forall t, \tag{2b}$$

$$\sum_{k\in\mathcal{K}_i} v_{s,t}^k\le q_{s,t}^i,\qquad \forall i,\forall s,\forall t. \tag{2c}$$

Due to the constraints on the memory capacity of server s, it should be guaranteed that the memory usage by both containers and Zygotes on server s remains within the memory capacity:

$$\sum_{k\in\mathcal{K}} r_k\big(\max\{u_{s,t}^k,e_{s,t}^k\}-w_{s,t}^k\big)+\sum_{i\in\mathcal{I}} r_i\Big(q_{s,t}^i-\sum_{k\in\mathcal{K}_i} v_{s,t}^k\Big)\le R_s,\qquad \forall s,\forall t. \tag{3a}$$

Functions are executed within their designated containers until completion. Following this, these containers enter an idle state. For the idle containers, the system needs to make decisions regarding caching the entire containers, caching as the

Zygotes, and not caching (i.e., recycle containers). We introduce the following variables to represent these decisions. $x_{s,t}^k$ signifies the number of type-k containers to be cached as the Zygotes on the server s. $y_{s,t}^k$ and $z_{s,t}^i$ denotes the number of type-k containers and the number of type-i Zygotes to be recycled on server s, respectively. Consequently, the number of type-k containers cached on server s can be calculated as follows:

$$e_{s,t}^k = \max\left\{u_{s,t-1}^k, e_{s,t-1}^k\right\} - x_{s,t-1}^k - y_{s,t-1}^k - w_{s,t-1}^k, \tag{4a}$$

$$e_{s,t}^k \geq 0, \qquad \forall k, \forall s, \forall t. \tag{4b}$$

The number of type-i Zygotes cached on server s can be calculated by the following equation:

$$q_{s,t}^i = q_{s,t-1}^i - z_{s,t-1}^i - \sum_{k\in\mathcal{K}_i} v_{s,t-1}^k + \sum_{k\in\mathcal{K}_i} x_{s,t-1}^k, \tag{5a}$$

$$q_{s,t}^i \geq 0, \qquad \forall i, \forall s, \forall t. \tag{5b}$$

B. Problem Formulation

Caching either the entire containers or the Zygotes incurs memory cost, providing the advantage of mitigating container startup delay for subsequent function invocations. For the type-k container, caching will incur a significant memory consumption denoted as r_k. If a cached type-k container hosts a subsequent type-k function invocation, the warm-start delay is considered negligible [16]. In contrast, cache as the type-i Zygote incurs a relatively lower memory consumption denoted as r_i due to the Zygote preserving only public packages. For a subsequent type-k (type-k', $k, k' \in \mathcal{K}_i$) function invocation, the Zygote can import private packages to create the corresponding type-k (type-k') container to host this function, thus incurring a help-start delay denoted as d_k ($d_{k'}$) [15]. The cold-start delay for the type-k container startup from scratch is denoted as c_k. As discussed above, these parameters' relationships are $0 < r_i < r_k$ and $c_k > d_k > 0$.

Our research aims to optimize two categories of cost, which play a substantial role in system performance: delay cost and memory cost. Delay cost is proportional to the container startup delay. The startup delay includes the help-start delay and the cold-start delay. Thus, the startup delay at t can be quantified using the following equation:

$$C_t^D = \sum_{s\in\mathcal{S}}\sum_{k\in\mathcal{K}} d_k v_{s,t}^k + \sum_{s\in\mathcal{S}}\sum_{k\in\mathcal{K}} c_k \max\left\{u_{s,t}^k - v_{s,t}^k - e_{s,t}^k, 0\right\}. \tag{6a}$$

The memory cost is proportional to the product of memory consumption and duration, and it interprets the memory resource price paid for reserving the containers and the Zygotes. The memory cost at t can be expressed as the sum of the memory consumption for all cached containers and Zygotes, defined as follows:

$$C_t^M = \sum_{s\in\mathcal{S}}\sum_{i\in\mathcal{I}} r_i q_{s,t}^i + \sum_{s\in\mathcal{S}}\sum_{k\in\mathcal{K}} r_k e_{s,t}^k. \tag{7a}$$

We formulate an online joint optimization problem that encompasses multiple decisions related to invocation distribution, container startup, and container caching, indicated by decision variables $u_{s,t}^k, v_{s,t}^k, w_{s,t}^k, x_{s,t}^k, y_{s,t}^k, z_{s,t}^i$. The problem can be formally formulated with the following integer program:

$$\mathbf{P}: \qquad \min \sum_{t\in\mathcal{T}} \left(\eta C_t^D + C_t^M\right), \tag{8a}$$

$$\text{s.t. } (1a), (2c), (2b), (2a), (3a), (4a), (4b), (5a), (5b), \tag{8b}$$

$$u_{s,t}^k, v_{s,t}^k, w_{s,t}^k, x_{s,t}^k, y_{s,t}^k, z_{s,t}^i \in \mathbb{N}, \quad \forall k, \forall i, \forall s, \forall t, \tag{8c}$$

where the weight η plays a crucial role in achieving an equilibrium between the two cost categories.

C. Problem Analysis

To solve the problem, we aim to develop an online algorithm with low computational complexity and provable performance guarantees. However, this is challenging for several reasons. Firstly, the online nature requires making decisions without prior knowledge of future function usage patterns. Moreover, complex decision variables and constraints hinder computational problem-solving. These aspects collectively contribute to the complexity of the optimization problem.

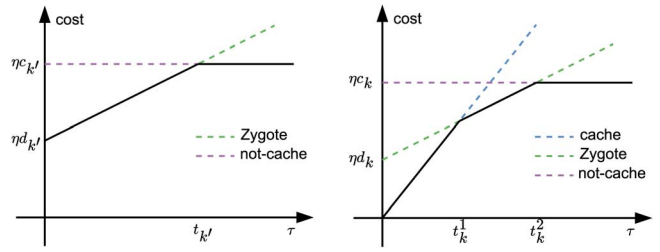
To reduce complexity, the problem is required to be decomposed. Decisions regarding invocation distribution and container startup are based on the available containers in the current time slot. In contrast, container caching decisions made in the current time slot will determine the available containers in the future. Thus, within a time slot, the problem is decomposed into two subproblems: 1) Determining invocation distribution and container startup based on the available container, and 2) Determining container caching decisions. To efficiently solve the first subproblem, a greedy-based subroutine is designed in Section V-A.

In Section IV, the container caching subproblem is first simplified by analyzing how a single container makes such decisions without future information. Then, a randomized strategy is assigned to each container. The randomized strategy determines whether to cache the entire container, cache it as the Zygote, or not cache it. These container-level decisions then constitute the overall container caching decisions. Building upon the results above, an online algorithm is proposed in Section V. This decomposition approach effectively solves the optimization problem with low computational cost and guaranteed performance. We prove the competitive ratio and computational complexity in Section V-E.

IV. Randomized Strategy for a Single Container

A. Observations and Insights

We first provide observations and insights about container caching decisions for a single type-k ($k \in \mathcal{K}_i$) container upon it becomes idle, including whether to cache it, cache as the Zygote, or not cache. From the container's perspective, it remains uncertain when the central controller distributes a function invocation for hosting. This process will incur an ongoing memory cost, which is the time interval multiplied by memory consumption. This analogies to the complex variant of a classic



(a) Subsequent invocation is type-k. (b) Subsequent invocation is type-k'.

Fig. 2. Illustration for the scenarios where the subsequent distributed invocation corresponds to type-k and type-k', respectively. $t_k^1 = \eta d_k/(r_k - r_i)$, $t_k^2 = \eta(c_k - d_k)/r_i$, and $t_{k'} = \eta(c_{k'} - d_{k'})/r_i$ as the horizontal coordinates of the intersection points of these dashed lines.

ski rental problem [19], [20], [21]. In the ski rental problem, a skier requires skiing for an unknown ski period. The skier must opt for one of several options, each characterized by an initial buying price (i.e., startup delay) and an ongoing cost related to the rental price (i.e., memory consumption). Furthermore, a more challenging aspect than the ski rental problem is that the ski rental only provides one type of service while a Zygote can serve any subsequent type-k' ($k' \in \mathcal{K}_i$) function.

Fig. 2(a) and 2(b) show the cost of the subsequent distributed invocation corresponds to type-k and type-k', respectively. $\tau = 0$ denotes the time for the container to become idle, and the three distinct decisions are each associated with a particular slope and intercept. Here, we assume that $t_k^1 < t_k^2$. This usually holds because the cold-start delay is much larger than the help-start delay [15], [16]. In Fig. 2(b), we omit the cache choice since it results in a memory consumption of r_k, while the startup delay remains $c_{k'}$. The black line shows the optimal choice if the type of subsequent distributed invocation and its arrival time t is known. However, the inherent online nature implies that the type of subsequent distributed invocation and its arrival time remain unknown.

To facilitate our analysis, we introduce the following conception: A container designated as being in the "cache state", "zygote state", and "not-cache state" signifies that it is cached, cached as the Zygote, and recycled, respectively. The profile [37] to a single container is denoted as $\boldsymbol{P} = [P_0(\tau), P_1(\tau), P_2(\tau)]$, where $P_0(\tau)$, $P_1(\tau)$, and $P_2(\tau)$ corresponds to the probability that the container is in the "cache state", the "zygote state", and the "not-cache state", respectively. For the profile, the following fundamental properties need to be satisfied:

1) For any $\tau \geq 0$, there is $P_n(\tau) \geq 0, n = 0, 1, 2$;
2) For any $\tau \geq 0$, it holds that $\sum_{n=0}^{2} P_n(\tau) = 1$, indicating that the container must be in one of the defined states;
3) For any $\tau \leq \tau'$, it should hold that $\sum_{n=0}^{m} P_n(\tau) \geq \sum_{n=0}^{m} P_n(\tau')$. As the memory cost increases over time, leading containers to increasingly favor transforming into the state with low memory consumption.

In the following subsections, we derive the profile for scenarios where the subsequent distributed invocation is type-k' and type-k, respectively. Then, we illustrate how to sample from the profile to obtain the randomized strategy.

B. Profile for Subsequent Type-k' Invocation

We begin by discussing the profile denoted as $\boldsymbol{P}^{k'} = [P_0^{k'}(\tau), P_1^{k'}(\tau), P_2^{k'}(\tau)]$ in a straightforward scenario, where the subsequent distributed invocation is type-k', and $k' \neq k, k' \in \mathcal{K}_i$. The arrival time remains uncertain. As analyzed in Section IV-A, it is evident that "cache state" will not be chosen (i.e., $P_0^{k'}(\tau) = 0, \forall \tau \geq 0$). By considering only the "zygote state" and "not-cache state", the problem can be simplified into a 2-slope ski rental problem, where the buying price is $\eta(c_{k'} - d_{k'})$, and the rental price is r_i. Correspondingly, for any arrival time t, the optimal cost for the 2-slope ski rental problem is denoted as $\mathrm{OPT}(t) = \min\{r_i t, \eta(c_{k'} - d_{k'})\}$.[3]

Considering that $P_0^{k'}(\tau) = 0$ and the fundamental property that $\sum_{n=0}^{2} P_n^{k'}(\tau) = 1$, it follows that $P_2^{k'}(\tau) = 1 - P_1^{k'}(\tau)$. When the arrival time surpasses $\eta(c_{k'} - d_{k'})/r_i$, not-cache becomes the optimal choice. Consequently, for $\tau > \eta(c_{k'} - d_{k'})/r_i$, we can ascertain that $P_2^{k'}(\tau) = 1$. Furthermore, we introduce $p(\tau) = \frac{\mathrm{d}}{\mathrm{d}\tau} P_2^{k'}(\tau)$, which signifies the probability of transform from the "zygote state" to the "not-cache state" at τ. Similar to [38], [39], let $q(t)$ denote the probability density function of the arrival time t. We define the expected arrival time for the distributed type-k' invocation as $\mu_{k'} = \int_0^{\eta(c_{k'} - d_{k'})/r_i} t q(t)\mathrm{d}t + q_\psi \psi$, where $q_\psi = \int_{\eta(c_{k'} - d_{k'})/r_i}^{+\infty} q(t)\mathrm{d}t$ and $\psi = \int_{\eta(c_{k'} - d_{k'})/r_i}^{+\infty} t q(t)/q_\psi \mathrm{d}t$, and $t > \eta(c_{k'} - d_{k'})/r_i$ with associated probability mass q_ψ. For $p(\tau)$ and $q(t)$, it is essential to ensure that the following constraints are satisfied:

$$\int_0^{\eta(c_{k'} - d_{k'})/r_i} p(\tau)\mathrm{d}\tau = 1, \tag{9a}$$

$$\int_0^{\eta(c_{k'} - d_{k'})/r_i} q(t)\mathrm{d}t + q_\psi = 1. \tag{9b}$$

The expected cost for the 2-slope ski rental problem arising from the profile within $0 \leq t < \eta(c_{k'} - d_{k'})/r_i$ can be expressed as:

$$\mathbb{E}[C_1(p(\tau), t)] = \int_0^t \Big(r_i\tau + \eta(c_{k'} - d_{k'})\Big)p(\tau)\mathrm{d}\tau + \int_t^{\eta(c_{k'} - d_{k'})/r_i} r_i t p(\tau)\mathrm{d}\tau. \tag{10a}$$

And the expected cost when $\eta(c_{k'} - d_{k'})/r_i \leq t$ can be expressed as:

$$\mathbb{E}[C_2(p(\tau), t)] = \int_0^{\eta(c_{k'} - d_{k'})/r_i} \Big(r_i\tau + \eta(c_{k'} - d_{k'})\Big)p(\tau)\mathrm{d}\tau. \tag{11a}$$

[3]Notably, the optimal cost is reduced by a constant $\eta d_{k'}$ compared to the original cost. This reduction does not impact our derivation.

The ratio between the expected cost and the optimal cost which can be expressed as follows:

$$J(p,q) = \int_0^{\eta\left(c_{k'}-d_{k'}\right)/r_i} \frac{\mathbb{E}\left[C_1\left(p(\tau),t\right)\right]}{\mathrm{OPT}(t)} q(t)\mathrm{d}t + q_\psi \frac{\mathbb{E}\left[C_2\left(p(\tau),t\right)\right]}{\mathrm{OPT}(t)} \tag{12a}$$

$$= \int_0^{\eta\left(c_{k'}-d_{k'}\right)/r_i} \frac{\mathbb{E}\left[C_1\left(p(\tau),t\right)\right]}{r_i t} q(t)\mathrm{d}t + q_\psi \frac{\mathbb{E}\left[C_2\left(p(\tau),t\right)\right]}{\eta\left(c_{k'}-d_{k'}\right)}. \tag{12b}$$

The objective is to determine the profile based on $p(\tau)$ that minimizes the ratio $J(p,q)$. The optimization problem can be obtained as follows:

$$\min_p \max_q J(p,q),$$
$$\text{s.t.} \quad (9a),(9b), \tag{13a}$$
$$\mu_{k'} = \int_0^{\eta\left(c_{k'}-d_{k'}\right)/r_i} tq(t)\mathrm{d}t + q_\psi\psi. \tag{13b}$$

To tackle this optimization problem, we establish its dual problem for the maximization problem [40], which transforms it into a linear program problem characterized by two equality constraints. The Lagrangians related to the maximization problem are as follows:

$$\mathcal{L}(q,\lambda_1,\lambda_2) = \lambda_1 + \lambda_2\mu_{k'} + \int_0^{\eta\left(c_{k'}-d_{k'}\right)/r_i} \left(\frac{\mathbb{E}\left[C_1\left(p(\tau),t\right)\right]}{r_i t} - \lambda_1 - \lambda_2 t\right) q(t)\mathrm{d}t + q_\psi\left(\frac{\mathbb{E}\left[C_2\left(p(\tau),t\right)\right]}{\eta\left(c_{k'}-d_{k'}\right)} - \lambda_1 - \lambda_2\psi\right), \tag{14a}$$

where the Lagrange multiplier λ_1 and λ_2 are correspond to the constraint (9b) and constraint (13b), respectively. The dual function $g(\lambda_1,\lambda_2) = \sup_q \mathcal{L}(q,\lambda_1,\lambda_2)$. Therefore, the dual problem becomes:

$$\min_{\lambda_1,\lambda_2} \lambda_1 + \lambda_2\mu_{k'}, \tag{15a}$$
$$\text{s.t.} \quad \frac{\mathbb{E}\left[C_1\left(p(\tau),t\right)\right]}{r_i t} - \lambda_1 - \lambda_2 t = 0, \quad 0 \le t < \eta\left(c_{k'}-d_{k'}\right)/r_i, \tag{15b}$$
$$\frac{\mathbb{E}\left[C_2\left(p(\tau),t\right)\right]}{\eta\left(c_{k'}-d_{k'}\right)} - \lambda_1 - \lambda_2\psi = 0, \tag{15c}$$
$$\lambda_1,\lambda_2 \ge 0. \tag{15d}$$

Since the constraint (15b) is valid for all $0 \le t < \eta\left(c_{k'}-d_{k'}\right)/r_i$, we can perform a double differentiation with respect to t and replace t with τ to obtain:

$$\frac{\mathrm{d}}{\mathrm{d}\tau}p(\tau) = \frac{r_i}{\eta\left(c_{k'}-d_{k'}\right)}\left(p(\tau) + 2\lambda_2\right). \tag{16a}$$

This is a first-order ordinary differential equation, the solution of which is:

$$p(\tau) = \rho e^{r_i\tau/\eta\left(c_{k'}-d_{k'}\right)} - 2\lambda_2, \tag{17a}$$

Upon introducing constraint (9a), we can derive $\rho = \left(r_i + 2\lambda_2\eta\left(c_{k'}-d_{k'}\right)\right)/\eta\left(c_{k'}-d_{k'}\right)(e-1)$. Consequently, the remaining undetermined parameter in $p(\tau)$ is λ_2. To satisfy the fundamental property, there is $\lambda_2 \le r_i/2\eta\left(c_{k'}-d_{k'}\right)(e-2)$. By further substituting $p(\tau)$ into constraints (15b) and (15c), we can obtain the following equivalent dual problem:

$$\min_{\lambda_1,\lambda_2} \lambda_1 + \lambda_2\mu_{k'}, \tag{18a}$$
$$\text{s.t.} \quad \frac{2\eta\left(c_{k'}-d_{k'}\right)(2-e)}{r_i(e-1)}\lambda_2 + \frac{e}{e-1} = \lambda_1, \tag{18b}$$
$$\left(\frac{\eta\left(c_{k'}-d_{k'}\right)(3-e)}{r_i(e-1)} - \psi\right)\lambda_2 + \frac{e}{e-1} = \lambda_1, \tag{18c}$$
$$0 \le \lambda_1, 0 \le \lambda_2 \le \frac{r_i}{2\eta\left(c_{k'}-d_{k'}\right)(e-2)}, \tag{18d}$$

where constraints (18b) and (18c) are equivalent to constraints (15b) and (15c), respectively. It is known that the solution to the linear programming problem forms a convex polyhedron, where each basic feasible solution corresponds to a vertex of this polyhedron and vice versa [40]. Therefore, the solutions are as follows:

1) $\lambda_1 = e/(e-1), \lambda_2 = 0$
2) $\lambda_1 = 1, \lambda_2 = r_i/2\eta\left(c_{k'}-d_{k'}\right)(e-2)$

The values of the objective function based on these solutions are $e/(e-1)$ and $1 + \mu_{k'}r_i/2\eta\left(c_{k'}-d_{k'}\right)(e-2)$, respectively. It is noteworthy that when $\mu_{k'} \le 2\eta\left(c_{k'}-d_{k'}\right)(e-2)/r_i(e-1)$, the latter value of the objective function is smaller than the former. Hence, for $0 \le \tau \le \eta\left(c_{k'}-d_{k'}\right)/r_i$, $p(\tau)$ can be derived by substituting λ_2 into (17a) as follows:

$$p(\tau) = \begin{cases} \dfrac{r_i}{\eta\left(c_{k'}-d_{k'}\right)(e-2)}\left(e^{r_i\tau/\eta\left(c_{k'}-d_{k'}\right)} - 1\right), \\ \quad \mu_{k'} \le 2\eta\left(c_{k'}-d_{k'}\right)(e-2)/r_i(e-1) \\ \dfrac{r_i}{\eta\left(c_{k'}-d_{k'}\right)(e-1)} e^{r_i\tau/\eta\left(c_{k'}-d_{k'}\right)}, \\ \quad \mu_{k'} > 2\eta\left(c_{k'}-d_{k'}\right)(e-2)/r_i(e-1). \end{cases} \tag{19a}$$

When $\mu_{k'} \le 2\eta\left(c_{k'}-d_{k'}\right)(e-2)/r_i(e-1)$, $P_2^{k'}(\tau)$ is obtained by integrating $p(\tau)$ as follows:

$$P_2^{k'}(\tau) = \begin{cases} \dfrac{e^{r_i\tau/\eta\left(c_{k'}-d_{k'}\right)} - r_i\tau/\eta\left(c_{k'}-d_{k'}\right) - 1}{e-2}, \\ \quad 0 \le \tau < \eta\left(c_{k'}-d_{k'}\right)/r_i \\ 1, \quad \eta\left(c_{k'}-d_{k'}\right)/r_i \le \tau. \end{cases} \tag{20a}$$

Similarly, when $\mu_{k'} > 2\eta(c_{k'} - d_{k'})(e-2)/r_i(e-1)$, $P_2^{k'}(\tau)$ is obtained as follows:

$$P_2^{k'}(\tau) = \begin{cases} \dfrac{e^{r_i\tau/\eta\left(c_{k'}-d_{k'}\right)} - 1}{e-1}, & 0 \le \tau < \eta(c_{k'} - d_{k'})/r_i \\ 1, & \eta(c_{k'} - d_{k'})/r_i \le \tau. \end{cases} \tag{21a}$$

For different $\mu_{k'}$, we can compute $P_2^{k'}(\tau)$ based on (20a) or (21a). $\mu_{k'}$ is determined by the service instance and will be expounded in the next section. The primary distinction between (20a) and (21a) lies in that (20a) prefers to persist in a high memory consumption (low startup delay) state. The smaller $\mu_{k'}$ means that the arrival time will be faster. Consequently, the profile (20a) is more suitable for scenarios where $\mu_{k'}$ is smaller. Combined with $P_0^{k'}(\tau) = 0$ and $P_2^{k'}(\tau) = 1 - P_1^{k'}(\tau)$ analyzed previously, we obtain a profile $\boldsymbol{P}^{k'}$.

C. Profile for Subsequent Type-k Invocation

The profile $\boldsymbol{P}^k = \left[P_0^k(\tau), P_1^k(\tau), P_2^k(\tau)\right]$ is explored in a more complex scenario in this subsection, where the subsequent distributed invocation is type-k, and $k \in \mathcal{K}_i$, yet its arrival time remains uncertain. As analyzed in Section IV-A, this is analog to a 3-slope ski rental problem. We simplify it into two separate 2-slope ski rental problems. In the first, the rental price is denoted as $r_k - r_i$ with a buying price of ηd_k, while in the second, the rental price is represented as r_i with a buying price of $\eta(c_k - d_k)$. With this simplification, the profile $\boldsymbol{P}^k$ can be derived based on the results presented in Section IV-B.

For the first 2-slope ski rental problem, the corresponding $p(\tau)$ represents the probability of leaving the "cache state". Hence, the integral over $p(\tau)$ in this situation is $P_1^k(\tau) + P_2^k(\tau)$. For the second 2-slope ski rental problem, the corresponding $p(\tau)$ indicates the probability of entering the "not-cache state". Hence, the integral over $p(\tau)$ in this situation is $P_2^k(\tau)$. When $\mu_k \le 2\eta d_k(e-2)/(r_k - r_i)(e-1)$, the profile is obtained by substituting the parameters (rental price and buying price) into (20a) as follows:

$$P_1^k(\tau) + P_2^k(\tau) = \begin{cases} \dfrac{e^{\left(r_k-r_i\right)\tau/\eta d_k} - (r_k - r_i)\tau/\eta d_k - 1}{e-2}, \\ \qquad 0 \le \tau < \eta d_k/(r_k - r_i) \\ 1, \qquad \eta d_k/(r_k - r_i) \le \tau \end{cases} \tag{22a}$$

$$P_2^k(\tau) = \begin{cases} \dfrac{e^{r_i\tau/\eta\left(c_k-d_k\right)} - r_i\tau/\eta(c_k - d_k) - 1}{e-2}, \\ \qquad 0 \le \tau < \eta(c_k - d_k)/r_i \\ 1, \qquad \eta(c_k - d_k)/r_i \le \tau \end{cases} \tag{22b}$$

When $2\eta d_k(e-2)/(r_k - r_i)(e-1) < \mu_k \le 2\eta(c_k - d_k)(e-2)/r_i(e-1)$, we obtain the profile by substituting the parameters into (21a) and (20a), respectively:

$$P_1^k(\tau) + P_2^k(\tau) = \begin{cases} \dfrac{e^{\left(r_k-r_i\right)\tau/\eta d_k} - 1}{e-1}, \\ \qquad 0 \le \tau < \eta d_k/(r_k - r_i) \\ 1, \qquad \eta d_k/(r_k - r_i) \le \tau \end{cases} \tag{23a}$$

$$P_2^k(\tau) = \begin{cases} \dfrac{e^{r_i\tau/\eta\left(c_k-d_k\right)} - r_i\tau/\eta(c_k - d_k) - 1}{e-2}, \\ \qquad 0 \le \tau < \eta(c_k - d_k)/r_i \\ 1, \qquad \eta(c_k - d_k)/r_i \le \tau \end{cases} \tag{23b}$$

When $2\eta(c_k - d_k)(e-2)/r_i(e-1) < \mu_k$, the profile is obtained by substituting the parameters into (21a) as follows:

$$P_1^k(\tau) + P_2^k(\tau) = \begin{cases} \dfrac{e^{\left(r_k-r_i\right)\tau/\eta d_k} - 1}{e-1}, \\ \qquad 0 \le \tau < \eta d_k/(r_k - r_i) \\ 1, \qquad \eta d_k/(r_k - r_i) \le \tau \end{cases} \tag{24a}$$

$$P_2^k(\tau) = \begin{cases} \dfrac{e^{r_i\tau/\eta\left(c_k-d_k\right)} - 1}{e-1}, & 0 \le \tau < \eta(c_k - d_k)/r_i \\ 1, & \eta(c_k - d_k)/r_i \le \tau \end{cases} \tag{24b}$$

According to the fundamental properties, we can induce $P_0^k(\tau) = 1 - P_1^k(\tau) - P_2^k(\tau)$. Consequently, the profile $\boldsymbol{P}^k$ can be derived based on the results above.

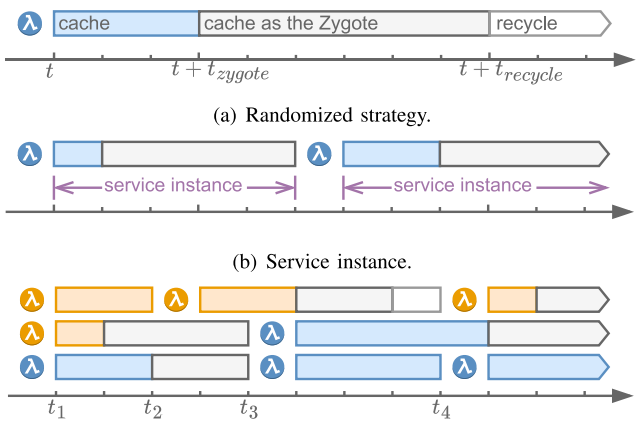
D. Profile and Randomized Strategy

Drawing upon the results in Sections IV-B and IV-C, the profile for a single type-k container can be obtained as:

$$\boldsymbol{P} = w_k \boldsymbol{P}^k + w_{k'} \boldsymbol{P}^{k'}, \tag{25a}$$

where w_k and $w_{k'}$ are adjustable parameters and $w_k + w_{k'} = 1$. The larger the $w_{k'}$, the more likely the container will be cached as a Zygote. As illustrated in the following section, we use these parameters to adjust the profile (thus adjusting the randomized strategy) based on the number of unused cached containers and the containers that have experienced cold-start. Furthermore, the profile satisfies the fundamental properties illustrated in Section IV-A.

Next, we illustrate how to obtain the randomized strategy based on the profile. The constant ξ is sampled from uniform distribution $\mathcal{U}[0,1]$, and then we set $t_{zygote} = \left\{\lfloor \min\{\tau\} \rfloor | \xi \le \sum_{n=1}^{2} P_n(\tau)\right\}$, $t_{recycle} = \left\{\lfloor \min\{\tau\} \rfloor | \xi \le \sum_{n=2}^{2} P_n(\tau)\right\}$. As shown in Fig. 3(a), if the container becomes idle at t, the randomized strategy is as follows: 1) Caching the entire container at t, 2) Caching it as the Zygote at $t + t_{zygote}$, 3) Recycling it at $t + t_{recycle}$. As the profile satisfies that $t_{recycle} \ge t_{zygote}$, it guarantees the strategy's correctness. It is noteworthy that if $t_{zygote} = t_{recycle}$ or $t_{recycle} = 0$, the container is recycled directly.



(a) Randomized strategy.

(b) Service instance.

(c) Example illustration.

Fig. 3. The circle signifies that the container is started up to host and execute the function, while bars of various colors denote the container's different states. (a) The randomized strategy is to cache the container first, caching it as the Zygote at $t + t_{zygote}$, and finally recycling it at $t + t_{recycle}$. (b) When a container is employed to host a function, its ongoing service instance ends. When the container becomes idle, a new service instance begins. (c) An example illustration, with details provided in Section V-D.

Algorithm 1 Online Container Scheduling (OCS)

Input: $t, \lambda_t^k, \forall k$
Output: $u_{s,t}^k, v_{s,t}^k, w_{s,t}^k, x_{s,t}^k, y_{s,t}^k, z_{s,t}^i, \forall k, i, s$
1: Initialize $u_{s,t}^k, v_{s,t}^k, w_{s,t}^k, x_{s,t}^k, y_{s,t}^k, z_{s,t}^i = 0, \forall k, i, s$
2: Update $e_{s,t}^k$ and $q_{s,t}^i$ by Eq. (4a) and Eq. (5a), respectively
3: $\left(u_{s,t}^k, v_{s,t}^k, w_{s,t}^k\right) = \text{IDCS}\left(\lambda_t^k, e_{s,t}^k, q_{s,t}^i, u_{s,t}^k, v_{s,t}^k, w_{s,t}^k\right)$
4: For $\forall k \in \mathcal{K}$, update $\alpha_{k,t}$ and $\beta_{k,t}$
5: **while** $|\mathcal{F}| < \sum_{k \in \mathcal{K}} \lambda_t^k$ **do**
6: Once execution is complete, add the container to $\mathcal{F}$
7: **for** $f \in \mathcal{F}$ **do**
8: Invoke $\text{RS}\left(f, \mathcal{H}, \alpha_{k,t}, \beta_{k,t}\right)$
9: Update $\mathcal{F} = \mathcal{F} - \{f\}$, $\mathcal{H} = \mathcal{H} + \{f\}$
10: **for** $h \in \mathcal{H}$ **do**
11: Update $\left(x_{s,t}^k, y_{s,t}^k, z_{s,t}^i\right) = \text{TS}\left(h, x_{s,t}^k, y_{s,t}^k, z_{s,t}^i\right)$

V. Online Algorithm

Based on the randomized strategy for a single container, an OCS algorithm characterized by low computational complexity and a provable competitive ratio is proposed in Algorithm 1. As discussed in Section III-C, the main idea of the OCS is to decompose the problem into two subproblems and solve them one by one: 1) Determining invocation distribution and container startup decisions; 2) Determining container caching decisions. For the first subproblem, the subroutine Invocation Distribution and Container Startup (IDCS) is invoked to solve it in line 3. IDCS is a greedy-based subroutine. Its details will be explained in Section V-A.

For the second subproblem, we solve it by merging container-level decisions in lines 4 to 11. In line 4, note α_k as the count of unused cached containers and β_k as the count of containers that have experienced cold-start. In line 8, the subroutine Randomized Strategy (RS) is invoked to assign a randomized strategy to the container within $\mathcal{F}$. α_k and β_k are used to determine the adjustable parameters of the profile, thus adjusting the randomized strategy. Subsequently, the subroutine Transform State (TS) decides the container caching based on the container's randomized strategy in line 11. For each container, TS is invoked one by one to update $x_{s,t}^k, y_{s,t}^k, z_{s,t}^i$. The details of RS and TS are given in Sections V-B and V-C, respectively. Finally, the OCS's computational complexity and competitive ratio are proved in Section V-E.

A. Invocation Distribution and Container Startup

The main idea of IDCS is to distribute invocations and start up the containers greedily, i.e., with the lowest startup delay based on available cached containers and Zygotes. In lines 2 to 4, the invocations are distributed, and the cached containers are employed to host functions. Subsequently, in lines 5 to 13, IDCS employs Zygotes to create the corresponding containers for hosting functions. Finally, in lines 14 to 22, the corresponding containers are started up from scratch to host the remaining undistributed invocations.

In lines 3, 7, 12, 16, and 21, IDCS ensures that constraints (1a), (2a), (2b), (2c), and (3a) are not violated, respectively. R_s' is the remaining memory space. $q_{s,t}^{i,k}$ and $q_{s,t}^{i,k'}$ denote the number of type-i Zygotes transformed by type-k and type-k' containers, respectively. To prevent the Zygotes from being preempted, the Zygotes (transformed by the type-k containers) are prioritized for creating type-k containers, followed by other types. If insufficient memory space and undistributed invocations exist (lines 18 to 22), the unused cached containers must be recycled to make room for startup containers.

To estimate the expected arrival time for the subsequent invocation (i.e., μ_k), we define the **service instance** as shown in Fig. 3(b). When a container is employed to host a function, its ongoing service instance ends. Besides, when the container becomes idle, a new service instance begins. If a container is started from scratch, it either ends the service instance of the recycled containers or initiates entirely new service instances. Consequently, the time interval between the beginning and end of a service instance is used to estimate the arrival time. As shown in line 9 of the Algorithm 1, $\mathcal{H}$ is the set of the ongoing service instances. Moreover, the priority of the containers is essential since the containers are shared across all functions of the same type. Priorities are assigned according to the beginning time of their service instances. The earlier the beginning time, the higher the priority.

B. Randomized Strategy

The subroutine RS is responsible for assigning randomized strategy to a container. The main idea of RS is to compute the profile $\boldsymbol{P}$ (lines 2 to 22) and subsequently sample a randomized strategy based on the profile $\boldsymbol{P}$ (lines 23 to 25). To compute the profile $\boldsymbol{P}$, the parameters μ_k, $\mu_{k'}$, w_k, and $w_{k'}$ need to be determined. As shown in lines 2 to 5, μ_k is set as the time interval between the service instance's beginning and ending time. Note that in line 6, we reset the beginning time of the service

IDCS: Invocation Distribution and Container Startup

Input: $\lambda_t^k, e_{s,t}^k, q_{s,t}^i, u_{s,t}^k, v_{s,t}^k, w_{s,t}^k, \forall k, i, s$
Output: $u_{s,t}^k, v_{s,t}^k, w_{s,t}^k, \forall k, s$
1: Sort all $k \in \mathcal{K}$ by $c_k - d_k$ in descending order
2: **for** $(k, s) \in \mathcal{K} \times \mathcal{S}$ **do**
3: **if** $\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k > 0$ **then**
4: Update $u_{s,t}^k = u_{s,t}^k + \min\{\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k, e_{s,t}^k\}$
5: **for** $(k, s) \in \mathcal{K} \times \mathcal{S}$ **do**
6: **if** $\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k > 0 \wedge k \in \mathcal{K}_i$ **then**
7: Set $\varphi = \min\{\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k, q_{s,t}^{i,k}, \lfloor R_s'/(r_k - r_i)\rfloor\}$, $q_{s,t}^{i,k} = q_{s,t}^{i,k} - \varphi$
8: Update $u_{s,t}^k = u_{s,t}^k + \varphi$, $v_{s,t}^k = v_{s,t}^k + \varphi$
9: **for** $(k, s) \in \mathcal{K} \times \mathcal{S}$ **do**
10: **if** $\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k > 0 \wedge k \in \mathcal{K}_i$ **then**
11: **for** $k' \in \mathcal{K}_i$ **do**
12: Set $\varphi = \min\{\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k, q_{s,t}^{i,k'}, \lfloor R_s'/(r_k - r_i)\rfloor\}$, $q_{s,t}^{i,k'} = q_{s,t}^{i,k'} - \varphi$
13: Update $u_{s,t}^k = u_{s,t}^k + \varphi$, $v_{s,t}^k = v_{s,t}^k + \varphi$
14: **for** $(k, s) \in \mathcal{K} \times \mathcal{S}$ **do**
15: **if** $\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k > 0$ **then**
16: Set $\varphi = \min\{\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k, \lfloor R_s'/r_k\rfloor\}$
17: Update $u_{s,t}^k = u_{s,t}^k + \varphi$
18: **for** $(k, s) \in \mathcal{K} \times \mathcal{S}$ **do**
19: **if** $\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k > 0 \wedge k \in \mathcal{K}_i$ **then**
20: **for** $k' \in \mathcal{K}_i$ **do**
21: Set $\varphi = \min\{\lambda_t^k - \sum_{s\in\mathcal{S}} u_{s,t}^k, \max\{0, e_{s,t}^{k'} - u_{s,t}^{k'} - w_{s,t}^{k'}\}, \lfloor r_{k'} \max\{0, e_{s,t}^{k'} - u_{s,t}^{k'} - w_{s,t}^{k'}\}/r_k\rfloor\}$
22: Update $u_{s,t}^k = u_{s,t}^k + \varphi$, $w_{s,t}^{k'} = w_{s,t}^{k'} + \varphi$

RS: Randomized Strategy

Input: $f, \mathcal{H}, \alpha_{k,t}, \beta_{k,t}, \forall k$
Output: $t_{zygote}^f, t_{recycle}^f$
1: Set $k = Type(f)$, $i = Type(k \in \mathcal{K}_i)$
2: **if** The service instance of f ends **then**
3: Set $\mu_k = t - t_{begin}^f$, update $\mathcal{H} = \mathcal{H} - \{f\}$
4: **else**
5: Set $\mu_k = 0$
6: Set $\beta_{k'} = \max_{k\in\mathcal{K}_i}\{\beta_k\}$, $t_{begin}^f = t$
7: **if** $\alpha_k > 0 \wedge \beta_{k'} > 0$ **then**
8: Set $w_{k'} = \min\{1, \sum_{k\in\mathcal{K}_i} \beta_k / \sum_{k\in\mathcal{K}_i} \alpha_k\}$, $w_k = 1 - w_{k'}$, $\mu_{k'} = t - \min\{t_{begin}\}$
9: Set $\alpha_k = \alpha_k - 1$, $\beta_{k'} = \beta_{k'} - 1$
10: **if** $\mu_{k'} \le 2\eta(c_{k'} - d_{k'})(e-2)/r_i(e-1)$ **then**
11: Compute $\boldsymbol{P}^{k'}$ by Eq. (20a)
12: **else**
13: Compute $\boldsymbol{P}^{k'}$ by Eq. (21a)
14: **else**
15: Set $w_{k'} = 0$, $w_k = 1$
16: **if** $\mu_k \le 2\eta d_k(e-2)/(r_k - r_i)(e-1)$ **then**
17: Compute $\boldsymbol{P}^k$ by Eq. (22a)
18: **else if** $2\eta d_k(e-2)/(r_k - r_i)(e-1) < \mu_k \le 2\eta(c_k - d_k)(e-2)/r_i(e-1)$ **then**
19: Compute $\boldsymbol{P}^k$ by Eq. (23a)
20: **else**
21: Compute $\boldsymbol{P}^k$ by Eq. (24a)
22: Compute the profile $\boldsymbol{P}$ by Eq. (25a)
23: Set $\xi = Sample(\mathcal{U}[0,1])$
24: Set $t_{zygote}^f = \{\lfloor \min\{\tau\}\rfloor | \xi \le \sum_{n=1}^2 P_n(\tau)\}$
25: Set $t_{recycle}^f = \{\lfloor \min\{\tau\}\rfloor | \xi \le \sum_{n=2}^2 P_n(\tau)\}$

instance, which means that a new service instance begins. In line 8, $\mu_{k'}$ is the interval between the earliest unused type-k cached container's service instance beginning and the current time.

To determine parameters w_k and $w_{k'}$, we first search for the container's type with the highest count of undergoing cold-start in line 6. If the type-k' containers experience a cold start due to the failure of timely caching the type-k containers as the Zygote in the previous time slot (line 7), we correct the probability of caching as the Zygote by adjusting the parameter $w_{k'}$. A larger $w_{k'}$ implies a greater likelihood to cache as the Zygote. In line 8, The parameter $w_{k'}$ is associated with the proportion of containers that experienced cold-start. Otherwise, no surplus unused cached containers are available, or no containers undergo cold-start. In such cases (line 15), w_k is directl yset as one.

Based on the parameters, $\boldsymbol{P}$ is Computed in line 22. Finally, in lines 23 to 25, we sample from a uniform distribution $\mathcal{U}[0,1]$ to generate the constant ξ, which determines the container's randomized strategy (as discussed in Section IV-D).

C. Transform State

The subroutine TS executes state transform based on the container's randomized strategy and updates corresponding decision variables. Lines 2 to 6 and 7 to 9 represent decisions

TS: Transform State

Input: $h, x_{s,t}^k, y_{s,t}^k, z_{s,t}^i, \forall k, i, s$
Output: $x_{s,t}^k, y_{s,t}^k, z_{s,t}^i, \forall k, i, s$
1: Set $k = Type(h)$, $i = Type(k \in \mathcal{K}_i)$
2: **if** The current state is "cache state" **then**
3: **if** $t_{begin}^p + t_{recycle}^p \le t$ **then**
4: Transform to "not-cache state", update $y_{s,t}^k = y_{s,t}^k + 1$
5: **else if** $t_{begin}^p + t_{zygote}^p \le t < t_{begin}^p + t_{recycle}^p$ **then**
6: Transform to "zygote state", update $x_{s,t}^k = x_{s,t}^k + 1$
7: **else if** The current state is "zygote state" **then**
8: **if** $t_{begin}^p + t_{recycle}^p \le t$ **then**
9: Transform to "not-cache state", update $z_{s,t}^i = z_{s,t}^i + 1$

when the container is currently in a "cache state" and "zygote state", respectively. Lines 3 to 4 signify that the container has surpassed its recycling time, necessitating recycling the container. Lines 5 to 6 represent the decision that cache as the Zygote, and the relevant decision variables are updated. Lines 8 to 9 indicate that the Zygote's recycling time has been exceeded.

Then, the Zygote is recycled. Other cases indicate no timeout or the container has been recycled. Thus, no action is needed. Notably, as long as the ongoing service instance of the container does not end, it performs state transformation based on the randomized strategy assigned at the beginning of the service instance.

D. Example Illustration

As shown in Fig. 3(c), we provide an example of how the OCS performs scheduling. At t_1, three containers become idle. The OCS determines caching decisions for each container in each time slot based on the randomized strategy. Furthermore, there are different function invocation arrivals at t_2, t_3, and t_4, respectively. OCS employs the available containers and Zygotes to host the function. For instance, at t_3, the available containers include one container and two Zygotes. OCS imports the private packages into the two available Zygotes to host the two functions.

E. Algorithm Analysis

This subsection analyzes the OCS's computational complexity and competitive ratio.

Theorem 1: The computational complexity of the OCS is $\mathcal{O}\big(|\mathcal{K}|^2|\mathcal{S}|\big)$.

Proof: The OCS's computational complexity analysis can be decomposed into two primary constituents. Firstly, OCS invokes IDCS. In a worst-case scenario, IDCS will undergo nesting and traverse the sets $\mathcal{K}$ and $\mathcal{S}$. Consequently, the worst-case computational complexity of IDCS is denoted as $\mathcal{O}\big(|\mathcal{K}|^2|\mathcal{S}|\big)$. Additionally, RS and TS are characterized by a constant computational complexity of $\mathcal{O}(1)$. OCS invokes RS once up to $|\lambda|$ times, where $|\lambda|$ denotes the total number of the serverless function invocations. Similarly, the subroutine TS is invoked up to $|\lambda|$ times. Consequently, when considering all components, the worst-case time complexity can be expressed as $\mathcal{O}\big(\max\big\{|\mathcal{K}|^2|\mathcal{S}|, |\lambda|\big\}\big)$. Given that $|\lambda|$ is essentially a constant, OCS's computational complexity can be represented as $\mathcal{O}\big(|\mathcal{K}|^2|\mathcal{S}|\big)$. □

Theorem 2: The proposed OCS is c-competitive, where $c = \max_{k\in\mathcal{K}} \Big\{\big\lceil c_k + \max_{k,k'\in\mathcal{K}_i}\big\{r_{k'}\lfloor \eta d_{k'}/\big(r_{k'}-r_i\big)\rfloor + r_i\big(\lfloor \eta\big(c_{k'}-d_{k'}\big)/r_i\rfloor - \lfloor \eta d_{k'}/\big(r_{k'}-r_i\big)\rfloor\big)\big\}\big\rceil/r_k\Big\}$.

Proof: The overall cost of the OCS can be decomposed into the sum of the costs associated with hosting functions, along with the additional memory cost denoted as ϖ stemming from unserved functions. The relationship between the overall cost incurred by the OCS and the offline optimal cost can be expressed as follows:

$$\mathrm{COST} \le c\cdot \mathrm{OPT} + \varpi. \tag{26a}$$

This decomposition allows us to establish an upper bound c for the cost of hosting a function. Based on this upper bound, the OCS's competitive ratio is derived. First, for the type-k function ($k\in\mathcal{K}_i$), we can establish a lower bound on the optimal cost required to host it, which can be expressed as $\mathrm{OPT}_k = \min\big\{r_k t, r_i t + \eta d_k, \eta c_k\big\} \ge r_k$.

Then, we need to account for the upper bound on the OCS's cost to host the type-k function. Due to memory constraints, a container might be unable to start on a particular server. The worst-case scenario unfolds when the system can only distribute the invocation to a server with adequate memory and end the service instance of the container in a "not-cache state", followed by a cold-start delay. Hence, the upper bound on the cost of serving type-k function can be bounded by $\mathrm{COST}_k \le c_k + \max_{k'\in\mathcal{K}_i}\big\{r_{k'}\lfloor \eta d_{k'}/\big(r_{k'}-r_i\big)\rfloor + r_i\big(\lfloor \eta\big(c_{k'}-d_{k'}\big)/r_i\rfloor - \lfloor \eta d_{k'}/\big(r_{k'}-r_i\big)\rfloor\big)\big\}$. Consequently, the maximum cost ratio is $c = \max_{k\in\mathcal{K}}\Big\{\big\lceil c_k + \max_{k,k'\in\mathcal{K}_i}\big\{r_{k'}\lfloor \eta d_{k'}/\big(r_{k'}-r_i\big)\rfloor + r_i\big(\lfloor \eta\big(c_{k'}-d_{k'}\big)/r_i\rfloor - \lfloor \eta d_{k'}/\big(r_{k'}-r_i\big)\rfloor\big)\big\}\big\rceil/r_k\Big\}$.

Finally, in the worst case, all the memory resources of the edge cluster are occupied up to the duration $\max_{k\in\mathcal{K}}\big\{\lfloor \eta\big(c_k-d_k\big)/r_i\rfloor\big\}$, so the additional memory cost is $\varpi = \max_{k\in\mathcal{K}}\big\{\lfloor \eta\big(c_k-d_k\big)/r_i\rfloor\big\}\sum_{s\in\mathcal{S}} R_s$. Consequently, the competitive ratio for the OCS is derived as c. □

VI. Evaluation

In this section, the performance of the OCS is validated with trace-driven simulations based on real-world datasets.

A. Experiment Setups

Simulation environments: In the experiment, we consider an edge cluster consisting of 10 servers, and the memory capacity of each server is set as $[32, 64]$ GB. 200 packages are used to build 100 serverless functions. These serverless functions share 30 Zygotes. The bare basic container requires 15MB memory [36]. The container sizes are $[25, 50]$ MB. The corresponding Zygote sizes are $[20, 35]$ MB. The cold-start delay of the container is set as $[1.2, 3.7]$ second, and the help-start delay of the container is set as $[0.1, 0.8]$ second.

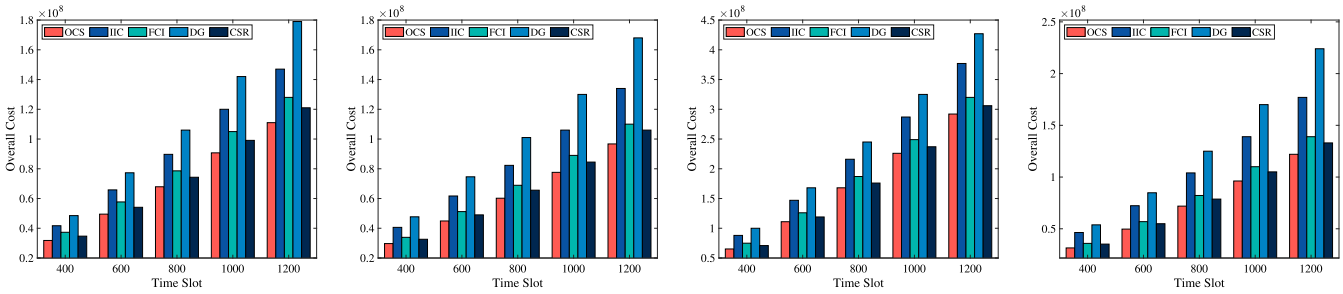
Dataset selection: The Microsoft Azure datasets [31] are used, which contain the invocations of functions on Microsoft Azure. We randomly select the trace of 400 functions from the Microsoft Azure datasets to generate traces 1 to 4, which represent the various scenarios of the invocation patterns.

Performance metrics: The performance metrics in the evaluation are as follows.

- Overall Cost: The objective function value.
- Memory Cost: The memory cost for reserving the idle containers and the Zygotes.
- Average Delay: The average startup delay per container.

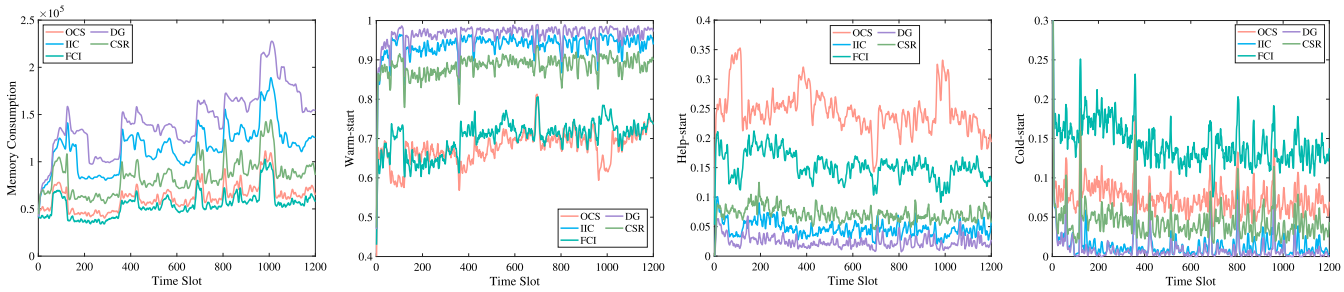
Baseline algorithms: The state-of-the-art solutions are employed as the following baseline algorithms.

- Identifying Idle Container (**IIC**) [16]: This algorithm identifies 95%-idle of the function invocation interval as a threshold to cache these idle containers as the Zygotes.
- Fixed Caching Interval (**FCI**) [41]: This algorithm caches the idle containers for a fixed time. Here, we cache each idle container for one slot, followed by caching it as the Zygote for one slot.
- Delay Greedy (**DG**): This algorithm is a greedy histogram-based strategy to minimize startup delay as much as possible. DG sets the maximal interval as a threshold to the cache container.



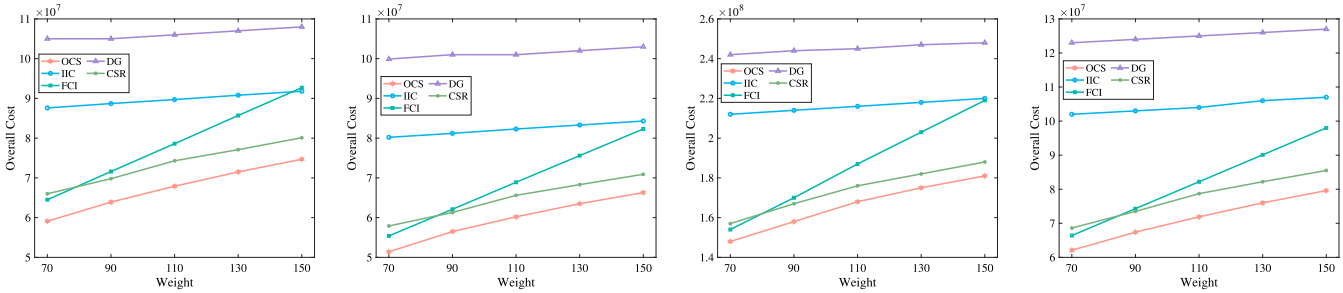
(a) Trace 1. (b) Trace 2. (c) Trace 3. (d) Trace 4.

Fig. 4. Overall cost in different function invocation patterns.



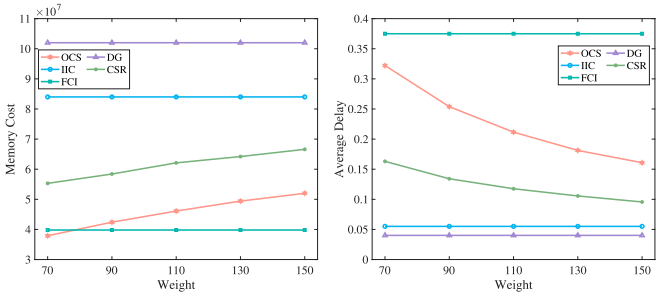
(a) Memory consumption. (b) Warm-start. (c) Help-start. (d) Cold-start.

Fig. 5. Memory consumption and container startup.



(a) Trace 1. (b) Trace 2. (c) Trace 3. (d) Trace 4.

Fig. 6. Overall cost in different weights.



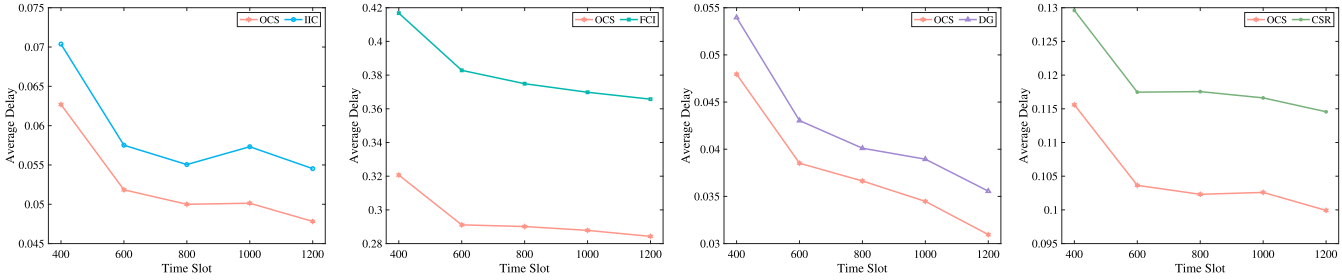
(a) Memory cost. (b) Average delay.

Fig. 7. Memory cost and average delay.

- Classic Ski Rental (**CSR**): This algorithm uses the solution of the classic ski-rental problem. The type-k container is cached as the Zygote after $\lfloor \eta d_k / (r_k - r_i) \rfloor$ time slots and recycled after $\lfloor \eta (c_k - d_k) / r_i \rfloor$ time slots.

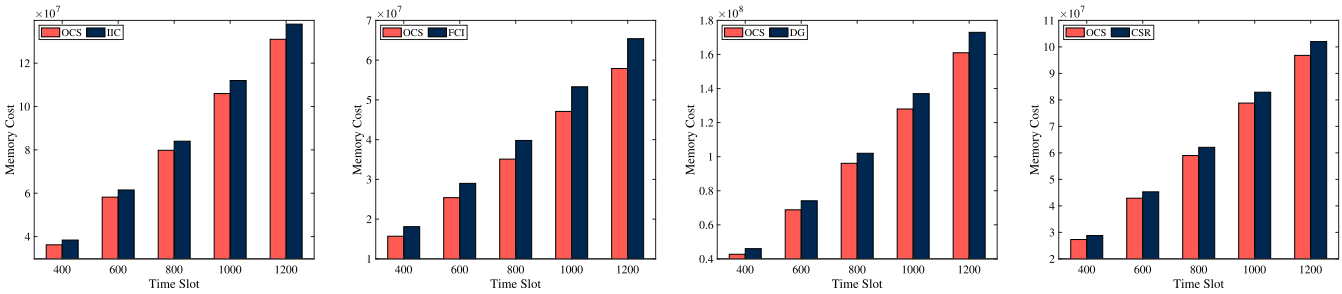
B. Experiment Results

Overall cost in different function invocation patterns: To illustrate the performance of the OCS in the scenario of various function invocation patterns, as shown in Fig. 4, we compare the overall cost of the OCS with the baseline algorithms at four traces. It can be seen that the overall cost of the OCS outperforms the baseline algorithms in all cases. Furthermore, compared to the widely used IIC and FCI, OCS demonstrates an improvement in the performance of approximately 32% and 15%, respectively. To illustrate this, we document the memory usage along with the percentage of containers experiencing warm-start, help-start, and cold-start, as shown in Fig. 5. OCS consumes relatively little memory, while only about 7% of the containers undergo cold-start. Although DG, IIC, and FCI make most containers undergo warm-start, they incur substantial memory costs to reserve the idle containers and Zygotes, consequently increasing the overall cost. In contrast, OCS



(a) OCS versus IIC. (b) OCS versus FCI. (c) OCS versus DG. (d) OCS versus CSR.

Fig. 8. Average delay under equivalent memory cost.



(a) OCS versus IIC. (b) OCS versus FCI. (c) OCS versus DG. (d) OCS versus CSR.

Fig. 9. Memory cost under equivalent average delay.

preferred to cache as Zygotes to reduce overall cost, benefiting from the Zygotes' memory efficiency and "parent container" attributes. These results show that the OCS excels without prior knowledge of future function invocation patterns. Thus, various workload fluctuations can be handled efficiently.

Impact of different weight: The weight η defined in Eq. (8a) is set from 70 to 150 to evaluate the overall cost with four traces. From Fig. 6, we can observe that the different weights significantly affect the overall cost. IIC, FCI, and DG make decisions based on fixed strategy, lacking the ability to adapt to weight changes and resulting in a degradation of the overall performance. While the CSR can perceive weight changes, it cannot adjust decisions when experiencing cold starts. OCS dynamically adapts its decisions based on the count of unused cached containers and the frequency of containers experiencing cold starts, enabling it to outperform baseline algorithms across all weight configurations. The outcomes obtained across varying weights prove the effectiveness of the OCS.

Memory cost and average delay: Fig. 7 shows the impact of weights on the algorithm's memory cost and average delay. It can be seen that as the weight increases, both the OCS and the CSR consume more memory cost (in Fig. 7(a)) to obtain lower startup delay (in Fig. 7(b)). The other baseline algorithms cannot make adaptive decisions based on the weight changes. For the widely used IIC and FCI, it is observed that IIC incurs a substantial memory cost, while FCI suffers from excessive startup latency. Consequently, both IIC and FCI exhibit higher overall costs than OCS. These results show that the OCS optimizes overall cost by incurring a relatively modest memory cost while achieving a low average delay.

To evaluate the average delay under equivalent memory cost, we change the weights to ensure the OCS incurs an equivalent memory cost to the baseline algorithms. From Fig. 8(a)–8(d), OCS can reduce up to 23% of the average delay against the baseline algorithms. To further assess the memory cost under an equivalent average delay, weights are adjusted to ensure the OCS incurs an equivalent average delay to the baseline algorithms. From Fig. 9(a)–9(d), we can observe that compared to the baseline algorithms, OCS can reduce up to 15% of the memory cost. These results demonstrate that the OCS significantly enhances resource efficiency in edge servers with constrained resources.

Execution time: As shown in Table II, we evaluate the execution time on different devices and with different problem sizes to prove the low computational complexity of the OCS. The execution time required by the OCS and the baseline algorithms is close, indicating that the OCS does not bring unacceptable execution costs. By decomposing the problem, OCS reduces complexity and solves it effectively.

VII. Discussion

Competitive ratio: Through theoretical analysis and experimental evaluation, it can be demonstrated that OCS is both efficient and has low computational complexity. Theorem 2 quantitatively assesses the algorithm's effectiveness. This assessment is valuable for designing practical containers, as it allows us to adjust the container's parameters to reduce the competitive ratio. For example, lowering c_k results in a decrease in c, suggesting that improving the theoretical performance of

TABLE II
EXECUTION TIME FOR ALGORITHMS ON DIFFERENT DEVICES

Algorithm	OCS	IIC	FCI	DG	CSR
Apple M2	27ms	21ms	27ms	20ms	22ms
Intel Xeon Silver 4210	51ms	209ms	51ms	210ms	40ms
Intel i9 14900KF	27ms	104ms	27ms	104ms	22ms
Intel i9 10900K	29ms	136ms	30ms	136ms	23ms
AMD EPYC 7742	42ms	126ms	41ms	126ms	33ms
$\|\mathcal{K}\| = 50$, $\|\mathcal{S}\| = 5$	5ms	4ms	5ms	5ms	4ms
$\|\mathcal{K}\| = 100$, $\|\mathcal{S}\| = 5$	18ms	13ms	18ms	13ms	14ms
$\|\mathcal{K}\| = 100$, $\|\mathcal{S}\| = 20$	46ms	38ms	47ms	37ms	40ms

the OCS can be achieved by minimizing the cold-start delay of containers during their design phase. Overall, Theorem 2 establishes the worst-case bound of the OCS and highlights the performance guarantees of the OCS.

Scalability and robustness: For alternative hardware resource constraints, such as CPU resources, OCS shows adaptability by incorporating CPU costs into its objective function and adjusting the container's randomized strategy. Similar to the handling of constraint (3a), OCS only needs to ensure sufficient CPU resources when distributing function invocations. This illustrates OCS's high scalability in managing different resource constraints. Additionally, OCS guarantees performance theoretically without prior knowledge of future function invocation patterns. The experimental results in Fig. 4 further demonstrate OCS's robustness, as it effectively handles various workload fluctuations.

VIII. CONCLUSION

This paper investigated the online container scheduling problem with fast function startup and low memory cost in edge computing. We formulated an online joint optimization problem. An OCS algorithm was proposed to solve the problem. A rigorous analysis was conducted to prove the OCS's computational complexity and competitive ratio. Furthermore, are conducted using the real-world dataset. The experimental results illustrated the superior performance of the OCS, which can reduce up to 23% of the average startup delay and up to 15% of the memory cost. Future work will consider using the Markov decision process to model the decision-making of container caching problems further and deploy the algorithms in the Kubernetes system.

REFERENCES

[1] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–32, 2022.

[2] "Apache openwhisk." Apache OpenWhisk. Accessed: Dec. 5, 2023. [Online]. Available: https://openwhisk.apache.org

[3] "Microsoft azure functions." Microsoft. Accessed: Dec. 5, 2023. [Online]. Available: https://azure.microsoft.com/en-us/products/functions/

[4] L. Kong et al., "Edge-computing-driven internet of things: A survey," *ACM Comput. Surv.*, vol. 55, no. 8, pp. 1–41, 2022.

[5] Y. Wu, K. Ni, C. Zhang, L. P. Qian, and D. H. Tsang, "NOMA-assisted multi-access mobile edge computing: A joint optimization of computation offloading and time allocation," *IEEE Trans. Veh. Technol.*, vol. 67, no. 12, pp. 12244–12258, Dec. 2018.

[6] A. M. Potdar, D. Narayan, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Comput. Sci.*, vol. 171, pp. 1419–1428, 2020.

[7] Z. Li et al., "{RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, USENIX, 2022, pp. 53–68.

[8] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv:1902.03383*.

[9] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Columbus, OH, USA, Apr. 2019, pp. 1063–1075.

[10] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2021, pp. 559–572.

[11] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2022, pp. 753–767.

[12] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2018, pp. 133–146.

[13] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. 21st Int. Middleware Conf. (Middleware)*, Delft, Netherlands: ACM/IFIP/USENIX, 2020, pp. 356–370.

[14] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Piscataway, NJ, USA: IEEE Press, 2022, pp. 1069–1078.

[15] E. Oakes et al., "{SOCK}: Rapid task provisioning with {Serverless-Optimized} containers," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2018, pp. 57–70.

[16] Z. Li et al., "Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2022, pp. 69–84.

[17] S. Minakova and T. Stefanov, "Memory-throughput trade-off for CNN-based applications at the edge," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 28, no. 1, pp. 1–26, 2022.

[18] S. Lee et al., "GreenDIMM: OS-assisted dram power management for dram with a sub-array granularity power-down state," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Columbus, OH, USA, 2021, pp. 131–142.

[19] S. Irani, S. Shukla, and R. Gupta, "Competitive analysis of dynamic power management strategies for systems with multiple power saving states," in *Proc. Des., Automat. Test Europe Conf. Exhib. (DATE)*, Piscataway, NJ, USA: IEEE Press, 2002, pp. 117–123.

[20] J. Augustine, S. Irani, and C. Swamy, "Optimal power-down strategies," in *Proc. 45th Annu. IEEE Symp. Found. Comput. Sci. (FOCS 04)*, Piscataway, NJ, USA: IEEE Press, 2004, pp. 530–539.

[21] Z. Lotker, B. Patt-Shamir, and D. Rawitz, "Rent, lease, or buy: Randomized algorithms for multislope ski rental," *SIAM J. Discrete Math.*, vol. 26, no. 2, pp. 718–736, 2012.

[22] I. E. Akkus et al., "{SAND}: Towards {High-Performance} serverless computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2018, pp. 923–935.

[23] A. Wang et al., "{FaaSNet}: Scalable and fast provisioning of custom serverless container runtimes at Alibaba Cloud function compute," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2021, pp. 443–457.

[24] M. Arutyunyan et al., "Decentralized and stateful serverless computing on the internet computer blockchain," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2023, pp. 329–343.

[25] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2020, pp. 419–433.

[26] Z. Tang, J. Lou, and W. Jia, "Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 22, no. 6, pp. 3444–3459, Jun. 2023.

[27] J. Lou, H. Luo, Z. Tang, W. Jia, and W. Zhao, "Efficient container assignment and layer sequencing in edge computing," *IEEE Trans. Services Comput.*, vol. 16, no. 2, pp. 1118–1131, Mar./Apr. 2022.

[28] B. Wang, A. Ali-Eldin, and P. Shenoy, "LaSS: Running latency sensitive serverless computations at the edge," in *Proc. 30th Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, New York, NY, USA: ACM, 2021, pp. 239–251.
[29] V. Mittal et al., "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA: ACM, 2021, pp. 168–181.
[30] L. Patterson et al., "HiveMind: A hardware-software system stack for serverless edge swarms," in *Proc. 49th Annu. Int. Symp. Comput. Archit. (ISCA 22)*, New York, NY, USA: ACM, 2022, pp. 800–816.
[31] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Berkeley, CA, USA: USENIX, 2020, pp. 205–218.
[32] L. Zhang et al., "Tapping into NFV environment for opportunistic serverless edge function deployment," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2698–2704, Oct. 2022.
[33] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Defuse: A dependency-guided function scheduler to mitigate cold starts on FaaS platforms," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 194–204.
[34] A. Kumari and B. Sahoo, "ACPM: Adaptive container provisioning model to mitigate serverless cold-start," *Cluster Comput.*, vol. 27, no. 2, pp. 1333–1360, 2024.
[35] B. Sethi, S. K. Addya, and S. K. Ghosh, "LCS: Alleviating total cold start latency in serverless applications with LRU warm container approach," in *Proc. 24th Int. Conf. Distrib. Comput. Netw. (ICDCN)*, New York, NY, USA: ACM, 2023, pp. 197–206.
[36] Y. Li, D. Zeng, L. Gu, M. Ou, and Q. Chen, "On efficient zygote container planning toward fast function startup in serverless edge cloud," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1–9.
[37] A. Borodin and R. El-Yaniv, "On randomization in on-line computation," *Inf. Comput.*, vol. 150, no. 2, pp. 244–267, 1999.
[38] A. Khanafer, M. Kodialam, and K. P. Puttaswamy, "To rent or to buy in the presence of statistical information: The constrained ski-rental problem," *IEEE/ACM Trans. Netw.*, vol. 23, no. 4, pp. 1067–1077, Aug. 2015.
[39] C. Dong, H. Zeng, and M. Chen, "A cost efficient online algorithm for automotive idling reduction," in *Proc. 51st Annu. Des. Automat. Conf. (DAC)*, New York, NY, USA: ACM, 2014, pp. 1–6.
[40] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge University Press, 2004.
[41] "AWS Lambda." AWS. Accessed: Dec. 5, 2023. [Online]. Available: https://aws.amazon.com/lambda/

Zhenzheng Li received the B.S. degree from the School of Information Technology, Beijing Institute of Technology, Zhuhai, China, in 2019, and the M.S. degree from the School of Information and Control Engineering, China University of Mining and Technology, China, in 2022. He is currently working toward the Ph.D. degree with the School of Artificial Intelligence, Beijing Normal University, China. His research interests include edge computing and resource scheduling.

Jiong Lou (Member, IEEE) received the B.S. and Ph.D. degrees from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016 and 2023, respectively. Since 2023, he has held the position of a Research Assistant Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He has published more than 10 papers in leading journals and conferences such as ToN, TMC, and TSC. His research interests include edge computing, task scheduling, and container management.

Jianfei Wu received the B.E. degree from the School of Computer Science and Technology, Qingdao University, China, in 2023. He is currently working toward the Ph.D. degree with the School of Artificial Intelligence, Beijing Normal University. His research interests include edge computing, large language models, and reinforcement learning.

Jianxiong Guo (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA, in 2021. Currently, he is an Associate Professor with the Advanced Institute of Natural Sciences, Beijing Normal University, and also with Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College, Zhuhai, China. His research interests include social networks, wireless sensor networks, combinatorial optimization, and machine learning. He is a member of ACM and CCF.

Zhiqing Tang (Member, IEEE) received the B.S. degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015, and the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2022. Currently, he is an Assistant Professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. His research interests include edge computing, resource scheduling, and reinforcement learning.

Ping Shen received his B.Sc. and M.Sc. degrees in communication and information system from the Southeast University, in 2001 and 2006, respectively. Currently, he is an Assistant Researcher with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University. His research interests include high performance computing, edge computing, and wireless networks. He has 11 patents and has published one research book.

Weijia Jia (Fellow, IEEE) received the B.Sc. and M.Sc. degrees from the Center South University, China, in 1982 and 1984, respectively, and Master of Applied Science and Ph.D. degrees from the Polytechnic Faculty of Mons, Belgium, in 1992 and 1993, respectively. Currently, he is a Chair Professor, the Director of BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (Zhuhai) and the VP for Research of BNU-HKBU United International College. His research interests include optimal network routing and deployment, networking, AI, and edge computing. He is the Distinguished Member of CCF.

Wei Zhao (Fellow, IEEE) received the Ph.D. degree in computer and information sciences from the University of Massachusetts, Amherst, in 1983 and 1986, respectively. He has served important leadership roles in academic including the Dean of Science with the Rensselaer Polytechnic Institute, the Director for the Division of Computer and Network Systems in the U.S. National Science Foundation, and the Senior Associate Vice President for Research, Texas A & M University. Dr. Zhao was awarded the Lifelong Achievement Award by the Chinese Association of Science and Technology in 2005.