Retention-Aware Container Caching for Serverless Edge Computing

Li Pan[1] Lin Wang[2,3] Shutong Chen[1] Fangming Liu*[1]
[1]National Engineering Research Center for Big Data Technology and System,
Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]VU Amsterdam, The Netherlands [3]TU Darmstadt, Germany

***Abstract*—Serverless edge computing adopts an event-based model where Internet-of-Things (IoT) services are executed in lightweight containers only when requested, leading to significantly improved edge resource utilization. Unfortunately, the startup latency of containers degrades the responsiveness of IoT services dramatically. Container caching, while masking this latency, requires retaining resources thus compromising resource efficiency. In this paper, we study the retention-aware container caching problem in serverless edge computing. We leverage the distributed and heterogeneous nature of edge platforms and propose to optimize container caching jointly with request distribution. We reveal step by step that this joint optimization problem can be mapped to the classic ski-rental problem. We first present an online competitive algorithm for a special case where request distribution and container caching are based on a set of carefully designed probability distribution functions. Based on this algorithm, we propose an online algorithm called O-RDC for the general case, which incorporates the resource capacity and network latency by opportunistically distributing requests. We conduct extensive experiments to examine the performance of the proposed algorithms with both synthetic and real-world serverless computing traces. Our results show that O-RDC outperforms existing caching strategies of current serverless computing platforms by up to 94.5% in terms of the overall system cost.**

***Index Terms*—edge computing, serverless computing, container caching, ski-rental problem**

I. Introduction

With the accelerated penetration of Internet-of-Things (IoT) services in contexts such as smart cities and smart factories [1], cloud computing, the norm of the traditional computing paradigm, has shown its clear limitations and has been evolving towards a more distributed form called edge computing [2], [3], [4], [5]. With edge computing a general computing platform is constructed, where computing/storage resources are distributed at the network edge, in close proximity of the end devices for IoT services. As a result, reliable network connections with high bandwidth and low latency can be established between IoT devices and edge computing platforms, allowing IoT services to be hosted in a seamless and transparent fashion.

Unlike cloud computing where abundant computing resources are centralized at huge data centers, edge computing relies on a large number of distributed computing nodes (co-located at wireless access points for example), each equipped with *limited* computing capability [2]. This difference brings up questions over the suitable service model for edge computing. As many IoT services follow periodical/intermittent request patterns [6], reservating resources at the granularity of virtual machines as done in infrastructure-as-a-service (IaaS) will lead to considerable potential resource waste, deteriorating the resource limitation of edge nodes. On the other hand, software-as-a-service (SaaS) achieves finer-grained resource provisioning, but at the cost of programmability.

Serverless computing is a service model originally proposed for the cloud [7]. Serverless computing enables high elasticity of cloud resources, as apposed to IaaS, by following an event-based model, while still retaining the high programmability of the cloud as apposed to SaaS. With serverless computing, an IoT service, when deployed, is encapsulated in a lightweight container and is invoked only when triggered by an event requesting the service. The container is destroyed when the processing of the event is completed and the resources allocated to the container are released. Popular implementations of serverless computing are mostly focused on function-as-a-service (FaaS) such as AWS Lambda [8]. This trend has also been extended to the edge and *serverless edge computing* has received some initial attentions recently [9], [10], [11].

One fundamental challenge in serverless edge computing is the startup latency incurred by cold-starting the container upon a triggering event. As shown in Table 7 in [7], the cold-start latency of a Node.js-based function on current serverless platforms including AWS, Google, and Azure varies from few hundred milliseconds to a few seconds. Meanwhile, many IoT services perform only short computation when triggered, e.g., more than half of the functions execute for less than one second on average in Azure [12], so that the high cold-start latency is detrimental. While several techniques for accelerating the cold start of containers have been discussed [13], [14], such latency is still non-negligible.

The cold-start latency can be masked effectively by con-

*The corresponding author is Fangming Liu (fangminghk@gmail.com).

This work is supported by Huawei, by National Key Research & Development (R&D) Plan under grant 2017YFB1001703, by NSFC under grant 61722206 and 61761136014, and by National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals. Lin Wang is supported partially by the German Research Foundation (DFG) Collaborative Research Center (CRC) 1053 – MAKI subproject B2.

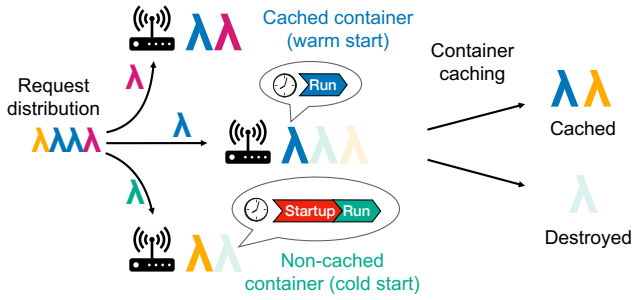


Fig. 1. Container caching jointly with request distribution for optimizing system efficiency in serverless edge computing.

tainer caching, i.e., keeping the container instance alive in memory after serving a request to avoid cold start for later requests. Reusing a cached container (referred to as warm start) for the next triggered call for the same IoT service can reduce the startup latency by at least an order of magnitude [7]. However, container caching is not free: First, container caching requires resource occupation, degrading resource utilization and thus violating the key elasticity premise of serverless computing where resources should be occupied only when necessary [15]. Second, memory media is still relatively expensive and the memory lifetime, usually measured by the P/E cycle endurance, is extended exponentially with the decrease of data retention time [16]. Larger memory capacity is required if retaining more data in memory, which increases the energy consumption of the edge node [17]. Therefore, deciding the keep-alive time for a container leads to the tradeoff between average startup latency and resource utilization, which has to be examined carefully but has been overlooked in the literature so far.

In this paper, we investigate the container caching problem in serverless edge computing. Our main idea is to leverage the flexibility provided by the distributed and heterogeneous nature of edge computing platforms and study the problem of *container caching jointly with request distribution (C2RD)*. As illustrated in Figure 1, when requests are distributed to the cached container, the container will serve these requests with a warm start. Instead, the non-cached container will incur a cold start before providing service. After all requests are completed, container must be destroyed or cached in memory. More specifically, we aim to make informed decisions on both distributing service requests to appropriate edge nodes and caching containers on these edge nodes. Our goal is to achieve optimal system efficiency defined by a combination of startup latency and resource utilization. C2RD concerns online decision making over time without a priori information on service request arrivals. To the best of our knowledge, we are among the first to study this joint optimization problem in serverless edge computing.

The C2RD problem is nontrivial mainly due to the following challenges: (1) Edge nodes are distributed and heterogeneous, resulting in varying startup latencies and container caching costs in terms of resource occupation on different edge nodes. Therefore, different nodes may prefer different container caching strategies. (2) Retention-aware container caching decisions have to be made a priori without knowing the future request arrival pattern. (3) Request distribution is hard in nature and this becomes even worse when intertwined with container caching on edge nodes. All these contribute to the extraordinary complexity of C2RD.

Prior research falls short of fully addressing the above challenges. Many works have focused on request distribution and/or service placement in edge computing [18], [19], [20], [21], [22], [23], [24], without considering service caching. Studies on content caching usually assume retention non-awareness [25], [26], meaning that caching the content incurs no cost. Retention-aware content caching in wireless networks have also been discussed in [27], [28], [29]. However, content caching has a fundamental difference to our problem that cached content can serve multiple requests simultaneously while a cached container can only serve one request at a time, rendering these solutions inapplicable. Most recently, Shahrad et al. characterize the serverless workload and propose a hybrid histogram policy for caching (keep-alive) serverless functions [12]. However, their focus is on central clouds and does not consider the heterogeneity of the distributed edge environment.

Overall, this paper makes the following contributions:

- We provide a comprehensive model to capture C2RD and formulate it with an integer linear program. Our model characterizes all the aforementioned challenges and features tradeoffs between startup latency cost and container retention cost in the objective function. We also present complexity analysis to C2RD, where we show that C2RD is NP-hard even in a simplified form.
- We reveal step by step that special cases of C2RD can be mapped to the classic ski-rental problem. We start from the simplest case where we assume a single edge node, serial request arrivals, and no resource capacity limits on edge nodes, and show that the problem can be treated equivalently to the classic ski-rental problem. We further show that concurrent request arrivals can be handled as a set of separate problem instances with serial request arrivals. We finally show that the multi-node case can be transformed into a modified version of the multi-shop ski-rental problem and we present an online competitive algorithm named RDC with performance guarantee.
- Based on the competitive algorithm, we propose an online algorithm called O-RDC with theoretically proved worst-case performance bound for the original C2RD problem, where we introduce opportunistic behaviors to the request distribution process in response to resource capacities and network latencies between edge nodes.
- We carry out extensive experiments based on both synthetic and real-world (Azure) datasets. We demonstrate that our algorithm outperforms the state-of-the-art container caching strategies used in current serverless computing systems, including fixed caching and warm queue, by up to 94.5% in terms of the overall system cost.

II. The Model

We provide the system model and problem formulation in this section. Important notations are listed in Table I.

A. System Model

We consider an edge computing system consisting of a number of edge nodes dispersed in multiple geographical zones, serving different IoT services. Here, the zone is an area with practical usages, such as enterprise campus, residential area or business area, an example of multiple zones is given in the following Sec. V-A and Figure 3. All the edge nodes in the system are connected by a network. We assume that edge nodes in a zone are interconnected by a local area network and thus, the communication latency between them is negligible. Each edge node serves as a network gateway as well as a computing facility for peripheral IoT devices. We denote the set of edge nodes in the system by $\mathcal{N} = \{1, ..., N\}$. Each edge node n is equipped with a certain amount of hardware resources (e.g., memory) denoted by U_n. We consider the case where the system works in a time-slotted fashion within a large time span $\mathcal{T} = \{1, ..., T\}$ and a time slot is denoted by $t \in \mathcal{T}$. We assume requests can be served within one time slot, which is in consistent with general service management frameworks [22], [30].

All the edge nodes in the system receive service requests of different types from IoT devices that are attached to the edge nodes. A central controller is in charge of distributing requests to appropriate edge nodes for handling in the system. We denote the set of request types in the system by $\mathcal{K} = \{1, ..., K\}$. The number of requests of type $k \in \mathcal{K}$ that are generated at edge node n in time slot t is given by $\lambda_{k,t}^n \in \mathbb{Z}_0^+$. All edge nodes follow the serverless computing paradigm, where they handle requests distributed to them in an event-based manner. Upon the arrival of a request of a new type $k \in \mathcal{K}$, the edge node instantiates a container encapsulating the corresponding IoT service and handles the request with the container. A container of type k demands u_k amount of hardware resources. As discussed, the container instantiation process typically incurs a considerable delay, known as the startup latency, before the container can actually serve the request. Once the processing of the request is completed, it is decided whether the container should be destroyed immediately or be cached for later requests of the same type. If a request is served by a cached container, the startup latency is considered negligible. We denote the number of requests of type k distributed to edge node n in time slot t by $m_{k,t}^n$. The number of requests of type k generated at edge node n_1 and distributed to edge node n_2 in time slot t is given by $m_{k,t}^{n_1 \to n_2}$. We have $m_{k,t}^n = \sum_{n' \in \mathcal{N}} m_{k,t}^{n' \to n}$.

B. Cost Model

We consider two categories of costs that are critical to the performance of the system: *service latency cost* and *container retention cost*. The service latency cost is composed of two parts: network communication cost and container instantiation cost, which are all proportional to the actual latency. The container retention cost is proportional to the retention time.

TABLE I
List of Notations

Symbol	Definition
$\mathcal{N}$	Set of edge nodes
$\mathcal{K}$	Set of service types
$\mathcal{T}$	Set of time slots
U_n	Hardware resource capacity of node n
u_k	Hardware resource demand of a type-k container
$l(n_1, n_2)$	Communication cost between nodes n_1 and n_2
d_k^n	Container instantiation cost of type k on node n
r_k^n	Container retention cost of type k on node n
$\lambda_{k,t}^n$	Number of type-k requests generated at node n at time t
$m_{k,t}^n$	Number of type-k requests distributed to node n at time t
$m_{k,t}^{n_1 \to n_2}$	Number of type-k requests distributed from n_1 to n_2
$a_{k,t}^n$	Number of cached type-k containers on node n before t
$y_{k,t}^n$	Number of type-k containers to be destroyed after serving requests on node n after t

We denote the communication cost between two edge nodes n_1 and n_2 by $l(n_1, n_2) \geq 0$, which is proportional to the network communication latency, and $l(n_1, n_2) > 0$ if n_1 and n_2 locate in different zones. The total network communication cost for all requests in time slot t is given by

$$\sum_{k \in \mathcal{K}} \sum_{n_1, n_2 \in \mathcal{N}} l(n_1, n_2) m_{k,t}^{n_1 \to n_2}. \tag{1}$$

The container instantiation cost for serving a request depends on the state of the container as discussed. Denote the number of containers of type k that are active at the beginning of time slot t (i.e., cached from last time slot) on edge node n by $a_{k,t}^n$ and the cost of instantiating a container of type k on node n by d_k^n. The total container instantiation cost for serving all requests in time slot t is given by

$$\sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}} \max\{m_{k,t}^n - a_{k,t}^n, 0\} d_k^n, \tag{2}$$

meaning that only the requests that are not served by cached containers will incur container instantiation costs. Overall, the service latency cost can be expressed as $C_t^L = (1) + (2)$. The container retention cost interprets the hardware resource price paid for caching a container (i.e., maintaining its active state), which is proportional to the container retention time. The total retention cost for all the cached containers at t is given by

$$\begin{aligned} &C_t^R = \textstyle\sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}} a_{k,t}^n r_k^n, \text{ where} \\ &a_{k,t}^n = \max\{a_{k,t-1}^n, m_{k,t-1}^n\} - y_{k,t-1}^n, \end{aligned} \tag{3}$$

r_k^n is the price for caching a container of type k on node n for a time slot duration, and $y_{k,t-1}^n$ is the number of containers to be destroyed after serving requests on node n at the end of time slot $t-1$. As discussed in Sec. I, container caching affects the resource utilization, resource attrition rate, and energy consumption. Hence, r_k^n can be set according to the resource occupation of the type-k container and the hardware specification of node n.

The total cost of the system is a sum of the service latency cost and container retention cost:

$$C_t = C_t^L + \alpha C_t^R. \tag{4}$$

The parameter α can be used to tune the tradeoff between the two types of cost. Note that $\alpha r_k^n \leq d_k^n$ should always be followed and we will focus on this case in the rest of the paper; otherwise, caching containers will cost more than instantiating containers, making container caching unhelpful.

C. Problem Formulation

Our problem is to make two decisions: request distribution (which node to handle a request, i.e., $m_{k,t}^{n_1 \to n_2}$) and container caching (how many containers of each type to be destroyed on each node in each time slot, denoted by $y_{k,t}^n$). The objective is to minimize the total cost C_t over time. The problem can be formally formulated with the following integer linear program:

$$\begin{aligned} (\mathbf{P}_1) \quad & \min \sum_{t=1}^{T} C_t \\ \text{s.t.} \quad & \textstyle\sum_{k \in \mathcal{K}} u_k \max\{a_{k,t}^n, m_{k,t}^n\} \leq U_n, \forall n, \forall t, \\ & \textstyle\sum_{n' \in \mathcal{N}} m_{k,t}^{n \to n'} = \lambda_{k,t}^n, \forall k, \forall n, \forall t, \\ & y_{k,t}^n \in \left[0, \max\{a_{k,t}^n, m_{k,t}^n\}\right], \forall k, \forall n, \forall t. \end{aligned} \tag{5}$$

The first constraint ensures that the hardware resource limit is respected on every node. The second constraint states that every request will be distributed to exactly one node. The last constraint restricts that the number of destroyed containers is no more than the number of already instantiated containers. The problem is hard in nature and we prove that

Theorem 1. *The simplified C2RD problem is NP-hard.*

We provide the proof of Theorem 1 in Appendix A.

III. Algorithms for Special Cases

In this section, we first tackle the problem in some special cases, from which we gain insights for designing our algorithm for the general case. All the special cases discussed in this section assume that the hardware resource capacity constraint is relaxed and all the nodes (if more than one) are in the same zone (i.e., the network latency is omitted because edge nodes are interconnected by a local area network). We start from the simplest case with only one node receiving serial requests in the system. We then expand to cases with concurrent requests per service type and further with multiple nodes. We finally provide an online algorithm and rigorously prove its optimality under these special cases. The proposed optimal online algorithm provides insights for designing the heuristic algorithm for the general case where hardware resource constraint and multiple zones are considered, which will be discussed in the following Sec. IV. The algorithm for the general case deviates from the optimal solution only under a few extreme conditions.

A. Single Node Serial Requests

We consider there exists only one node receiving serial requests in the system. In a single time slot, there is at most one request of the same type arriving and the arrival of requests is completely random. In this case, request distribution is irrelevant and decisions for caching containers of different types are decoupled since we omit the resource capacity constraints. We focus on a single container type and show that this problem can be mapped to the classic *ski-rental* problem.

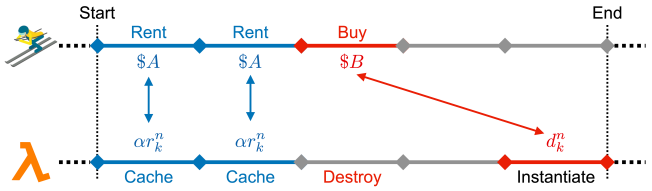


Fig. 2. Mapping of the single node serial requests container caching problem to the ski-rental problem.

Assume in time slot t we receive a request and there is no cached container to serve it. A container is thus instantiated to process the request. Upon completion, we have to make a decision on whether we should cache the container or destroy it. If we knew that there would be another request arriving before or in time slot $t + d_k^n/\alpha r_k^n$, the optimal decision would be to cache the container; otherwise, the decision is to destroy the container immediately after processing the current request. Unfortunately, such information about request arrivals cannot be known a priori in most online scenarios.

The *ski-rental* problem describes such a scenario where a skier is faced with a to-rent-or-to-buy struggle. Since the suitable days for skiing is unknown, the skier has to choose between renting a pair of skis or simply buying it. Assume the cost of renting the ski is $\$A$ and the cost of buying it is $\$B$. The skier has to make the decision for the current skiing season such that the total cost is minimized. We argue that

Theorem 2. *The single node serial requests container caching problem is equivalent to the ski-rental problem.*

Proof. As shown in Figure 2, we can build a mapping between the two problems as follows: The sequence of time slots are separated into several ski-rental instances. When a container has served a request, how to serve the next request is just like a ski-rental instance which starts with $A = \alpha r_k^n$ and $B = d_k^n$. The unpredictable time interval of next request is just like the unknown suitable days for skiing. Each time slot is mapped to a ski day. On the one hand, caching the container corresponds to renting the ski, which incurs a cost of αr_k^n. On the other hand, destroying the container corresponds to buying the ski in the ski-rental problem, because when the container is destroyed, an instantiation cost of d_k^n will have to be paid for the next request anyway. Therefore, we can bring forward the instantiation cost of the upcoming request to the point when we destroy the container. Whenever a request arrives, the current ski-rental instance ends and a new ski-rental instance begins at the next time slot. This completes the proof. □

There are in general two classes of online algorithms for the ski-rental problem: deterministic and probabilistic [31], [32]. We use *competitive ratio* – the ratio between the objective of a solution generated by the considered online algorithm and the objective of the offline optimal solution (assuming perfect knowledge about the future), to measure the performance of an online algorithm.

B. Single Node Concurrent Requests

We now discuss the case with multiple requests of the same type arriving in a time slot. Since the containers are shared among all the requests of the same type, the caching decisions for all the containers will be dependent on each other if not handled appropriately. We propose to assign decreasing priorities to the containers of the same type where requests of the same type, upon their arrivals, are always served by available containers with higher priorities. Thus, the i-th request will be served by the container with the i-th highest priority and the decisions for container caching will be separated from each other. The results presented above will apply directly.

C. Multiple Nodes Concurrent Requests

Now we consider a more general scenario where we assume multiple nodes and allow multiple requests of the same type to be processed by these nodes simultaneously. We show that this problem can be transformed into a variant of the multi-shop ski-rental problem [33]. Without loss of generality, we assume all nodes are heterogeneous in hardware specification and their processing capabilities differ from each other. One observation is that a higher specification would provide lower container instantiation cost, but the container retention cost will be larger. Assume we have in total N nodes (with varying specifications) in the system and the nodes are ranked in an increasing order with respect to their specifications, the following relationships follow for any service type $k \in \mathcal{K}$.

$$d_k^1 > d_k^2 > ... > d_k^N > 0, r_k^N > ... > r_k^1 > 0. \tag{6}$$

With request distribution, a new dimension of decision making (i.e., distributing requests to appropriate nodes), is of relevance. However, jointly making decisions for both request distribution and container caching is non-trivial. To this end, we propose an online algorithm by assigning each node a probability density function. The request distribution and container caching strategy will be based on this function. We will show that by following our strategy the competitive ratio of our online algorithm can be minimized in expectation.

Before we present the online strategy, we discuss the optimal offline strategy and its total cost. We know that if the request arrival time interval $\delta \in \mathbb{Z}^+$ is pre-given, the optimal offline decision can be easily made: If $\delta > d_k^N/\alpha r_k^1$, the container should be destroyed immediately when the handling of the previous request is completed and a new container is instantiated for the next request after δ time slots. The total cost for this decision is d_k^N as the container retention cost is zero. On the other hand, if $1 \leq \delta < d_k^N/\alpha r_k^1$, the optimal decision is to cache the container during the whole δ time slots, but the optimal location of the container being cached cannot be decided easily. Fortunately, we know that the optimal total cost $OPT(\delta)$ can be lower bounded by the total cost assuming the node with the cheapest retention cost is used, i.e.,

$$OPT(\delta) \geq \begin{cases} \alpha r_k^1 \delta, & \text{if } 1 \leq \delta \leq d_k^N/\alpha r_k^1, \\ d_k^N, & \text{if } \delta > d_k^N/\alpha r_k^1. \end{cases} \tag{7}$$

According to the case discussed in Section III-B, this result can be directly extended to multiple concurrent requests of the same type, since these requests can be treated separately. Note that services of different types will be handled separately as well, so index k can be set to any valid value in $\mathcal{K}$ and the same analysis will apply. Without loss of generality, we assume k is fixed to a certain value in the rest of this section.

Now let us discuss the online scenario where δ is unknown a priori. We denote by (n, x) the pair of decisions for request distribution and container caching (i.e., a request with decision pair (n, x) will be distributed to node n for processing and the container for serving this request will be destroyed after being cached for x time slots). We define a function $p_n(x)$ to represent the probability that the decision pair (n, x) is chosen for a request. Hence, we have

$$\textstyle\sum_{n\in\mathcal{N}} \int_0^\infty p_n(x)dx = 1. \tag{8}$$

Let $c_n(x, \delta)$ denote the total cost for this request under the decision pair (n, x). We know

$$c_n(x,\delta) = \begin{cases} \alpha r_k^n \delta, & \text{if } \delta \leq x, \\ \alpha r_k^n x + d_k^n, & \text{otherwise.} \end{cases} \tag{9}$$

The expected total cost for handling a request under the decision pair (n, x) can be expressed by

$$\begin{aligned} C_n(p_n(x),\delta) =& \textstyle\int_0^\infty c_n(x,\delta)p_n(x)dx \\ =& \textstyle\int_0^\delta (\alpha r_k^n x + d_k^n)p_n(x)dx + \int_\delta^\infty \alpha r_k^n \delta p_n(x)dx. \end{aligned} \tag{10}$$

The competitive ratio γ of an algorithm based on strategy (n, x) with function $p_n(x)$ is defined as

$$\gamma = \max\left\{\textstyle\sum_{n\in\mathcal{N}} C_n(p_n(x),\delta)/OPT(\delta)\right\}. \tag{11}$$

Our goal is to find out a function $p_n^*(x)$ such that the resulting strategy (n, x) based on $p_n^*(x)$ can minimize the worst-case competitive ratio γ.

$$\begin{aligned} (\mathbf{P}_4) \quad & \min \gamma \\ \text{s.t.} \quad & \textstyle\sum_{n\in\mathcal{N}} C_n(p_n(x),\delta)/OPT(\delta) \leq \gamma, \\ & \textstyle\sum_{n\in\mathcal{N}} \int_0^{+\infty} p_n(x)dx = 1, \\ & x \in \mathbb{Z}, \delta \in \mathbb{Z}^+. \end{aligned} \tag{12}$$

The following theorem provides the properties that $p_n^*(x)$ has to follow and sheds some light on how $p_n^*(x)$ can be computed.

Theorem 3. *The optimal request distribution and container caching function $p_n^*(x)$ satisfies the following properties:*

$$p_n^*(x) = \begin{cases} \varphi_n e^{\frac{x}{\theta_k^n}}, & \textit{if } x \in (b_{n+1}, b_n), \\ 0, & \textit{otherwise}, \end{cases} \tag{13}$$

where θ_k^n and φ_n satisfy that

$$\begin{aligned} \theta_k^n &= d_k^n/\alpha r_k^n, \\ \varphi_n d_k^n e^{\frac{b_n}{\theta_k^n}} &= \varphi_{n-1} d_k^{n-1} e^{\frac{b_n}{\theta_k^{n-1}}}, \forall n \in [2, N]. \end{aligned} \tag{14}$$

Algorithm 1: Optimal Probability Distributions (DIST)

$D_N \leftarrow 0$
for $n = N$ ***to*** 2 **do**
  Compute D_n using equation (20)
  Compute b_n using equation (22)
$\varphi_1 \leftarrow 1$
for $n = 1$ ***to*** N **do**
  Compute φ_n using equation (16)
  $\omega_n \leftarrow \int_{b_{n+1}}^{b_n} p_n(x)dx/\varphi_1$
$\omega \leftarrow \sum_{n\in\mathcal{N}} \omega_n$
for $n = N$ ***to*** 1 **do**
  $\varphi_n \leftarrow \varphi_n/\omega$

Algorithm 2: Random Distribution & Caching (RDC)

$p_n^*(x) \leftarrow \text{DIST}(d_k^n, \alpha r_k^n), \forall k \in \mathcal{K}$
for $k \in \mathcal{K}$ **do**
  $\lambda_{k,t} \leftarrow \sum_{n\in\mathcal{N}} \lambda_{k,t}^n$
  for $n \in \mathcal{N}$ **do**
    $m_{k,t}^n \leftarrow \lambda_{k,t} \int_{b_{n+1}}^{b_n} p_n^*(x)dx$
    if $m_{k,t}^n \le a_{k,t}^n$ **then**
      Compute $y_{k,t}^n$ by $a_{k,t}^n - m_{k,t}^n$ and x_n^*
      $a_{k,t+1}^n \leftarrow a_{k,t}^n - y_{k,t}^n$
    else
      $y_{k,t}^n \leftarrow 0$
      $a_{k,t+1}^n \leftarrow m_{k,t}^n$

The range (b_{n+1}, b_n) represents the caching duration on node n, where caching breakpoints b_n satisfy that $d_k^N/\alpha r_k^1 = b_1 \ge b_2 \ge ... \ge b_N \ge b_{N+1} = 0$. Solving problem $\mathbf{P}_4$ is equivalent to computing caching breakpoints $\{b_1, b_2, ..., b_{N+1}\}$.

We provide the proof of Theorem 3 in Appendix B.

The algorithm for computing the breakpoints and the optimal probability functions $p_n^*(x)$ is listed in Algorithm 1. Here we introduce some equations that will be used in the algorithm.

$$D_n = \frac{\varphi_{n+1}}{\varphi_n} D_{n+1} - \theta_k^n \left(e^{\frac{b_{n+1}}{\theta_k^n}} - 1\right) + \frac{\varphi_{n+1}}{\varphi_n} \theta_k^{n+1} \left(e^{\frac{b_{n+1}}{\theta_k^{n+1}}} - 1\right), \tag{15}$$

where we have

$$\frac{\varphi_{n+1}}{\varphi_n} = \frac{d_k^n}{d_k^{n+1}} \left(e^{\frac{b_{n+1}}{\theta_k^n} - \frac{b_{n+1}}{\theta_k^{n+1}}}\right). \tag{16}$$

Consequently, b_n can be computed as

$$b_n = \theta_k^n \ln\left(\frac{(d_k^{n-1} r_k^n - d_k^n r_k^{n-1})\left(1 - D_n \alpha r_k^n/d_k^n\right)}{d_k^n (r_k^n - r_k^{n-1})}\right). \tag{17}$$

The probability distribution functions $p_n^*(x)$ generated by the DIST algorithm will be used for making request distribution and container caching decisions and the competitive ratio following these decisions will be minimized. For each node n the approximate optimal decision for container caching duration x_n^* can be computed as $x_n^* = \lfloor (b_{n+1} + b_n)/2 \rfloor$.

Our algorithm for optimal request distribution and container caching in each time slot is listed in Algorithm 2. We distribute requests to each node based on the probability computed by $\int_{b_{n+1}}^{b_n} p_n^*(x)dx$. Once the requests are served, we decide which containers to be destroyed. If the number of cached containers is more than the number of generated requests in this time slot, we destroy the containers that have been cached longer than x_n^* time slots since last serving a request. This algorithm serves as a fundamental component of our proposed online algorithm described in the next section.

IV. Opportunistic Request Distribution and Container Caching for The General Case

Now we consider the general case where there exist multiple geographical zones and each node has hardware resource constraints. We denote the set of all zones by $\mathcal{Z}$ and the set of nodes in zone $z \in \mathcal{Z}$ by $\mathcal{N}_z$. Requests generated in each zone should all be served by the cached containers in that zone if there are already enough cached containers; otherwise, if requests cannot be fully handled in a zone by cached containers, the nodes in neighbor zones with enough cached containers will be employed, at the expense of extra network communication cost. In our system, the communication cost is among minimal and maximal instantiation cost of k types of containers. Our algorithm for this problem is highly based on the RDC algorithm for a single zone and follows an opportunistic manner, as shown in Algorithm 3.

For a zone z where the number of requests generated in the zone is smaller than the number of cached containers (i.e., $\sum_{n\in\mathcal{N}_z} \lambda_{k,t}^n \le \sum_{n\in\mathcal{N}_z} a_{k,t}^n$), we distribute the requests to each node based on the probability distribution functions $p_n^*(x)$. The extra cached containers can be used for requests distributed from other zones.

For a zone z where the number of requests generated in the zone is greater than the number of cached containers (i.e., $\sum_{n\in\mathcal{N}_z} \lambda_{k,t}^n > \sum_{n\in\mathcal{N}_z} a_{k,t}^n$), we first use all the cached containers in the zone to serve requests and distribute the extra requests to neighbor zones which satisfy that the network communication cost between the current zone and the neighbor zone is smaller than the minimal container instantiation cost in the current zone. In this case, the requests which are served by small-size containers with smaller instantiation cost, would have less chances to be served by containers in neighbor zones. Therefore, we handle requests with small-size containers and cache corresponding containers preferentially. After traversing all the neighbor zones, if extra requests are not all handled successfully, new containers have to be instantiated in the current zone with instantiation cost anyway. If a node cannot instantiate more containers due to resource constraints, another node with minimal container instantiation cost will be chosen to handle the requests.

Theorem 4. *The time complexity of O-RDC is $O(KZ^2N^2)$.*

We provide the proof of Theorem 4 in Appendix C.

Theorem 5. *The worst-case cost for serving a type-k request is bounded by $\max\{(\alpha r_k^n x_n^* + d_k^1)/\alpha r_k^1 (x_n^* + 1)\}, n \in \mathcal{N}$.*

Algorithm 3: Opportunistic RDC (O-RDC)

$p_n^*(x) \leftarrow \text{DIST}(d_k^n, \alpha r_k^n), \forall k \in \mathcal{K}$
for $z \in \mathcal{Z}$ **do**
  for $k \in \mathcal{K}$ **do**
    $\lambda_{k,t} \leftarrow \sum_{n \in \mathcal{N}_z} \lambda_{k,t}^n$
    $a_{k,t} \leftarrow \sum_{n \in \mathcal{N}_z} a_{k,t}^n$
    if $\lambda_{k,t} \leq a_{k,t}$ **then**
      for $n \in \mathcal{N}_z$ **do**
        $m_{k,t}^n \leftarrow \lambda_{k,t} \int_{b_{n+1}}^{b_n} p_n^*(x)dx$
        Compute $y_{k,t}^n$, $a_{k,t+1}^n$
    else
      for $n \in \mathcal{N}_z$ **do**
        for $z' \in \mathcal{Z}$ ***and*** $z' \neq z$ ***and*** $n' \in \mathcal{N}_{z'}$ **do**
          Compute $m_{k,t}^{n \to n'}$
          $\lambda_{k,t} \leftarrow \lambda_{k,t} - m_{k,t}^{n \to n'}$
      for $n \in \mathcal{N}_z$ **do**
        $m_{k,t}^n \leftarrow \lambda_{k,t} \int_{b_{n+1}}^{b_n} p_n^*(x)dx$
        Compute $y_{k,t}^n$, $a_{k,t+1}^n$

Proof. Firstly, we recall the offline optimal solution to serve a type-k request. If we know the request arrival time interval $\delta \in \mathbb{Z}^+$, we can obtain the lower bound of total cost to serve this request as:

$$OPT(\delta) \geq \begin{cases} \alpha r_k^1 \delta, & \text{if } 1 \leq \delta \leq d_k^N / \alpha r_k^1, \\ d_k^N, & \text{if } \delta > d_k^N / \alpha r_k^1. \end{cases} \tag{18}$$

Then, we consider the maximum cost produced by O-RDC. We cache the type-k container on node n for $x_n^* \in [0, d_k^N/\alpha r_k^1]$ time slots then destroy it. Thus, if the time interval δ of type-k request is less than x_n^*, the total cost consists of only the retention cost $\alpha r_k^n \delta$; otherwise, we destroy the container and instantiate it when the request arrives and cannot be handled by nodes in neighbor zones, so the total cost consists of retention cost $\alpha r_k^n x_n^*$ and instantiation cost d_k^n.

With hardware resource constraints, sometimes the container cannot be instantiated on node n. The worst case happens when the container can only be instantiated with the largest instantiation cost d_k^1 on node 1. We formulate the total cost of worst case as follows:

$$WORST(\delta) = \begin{cases} \alpha r_k^n \delta, & \text{if } 1 \leq \delta \leq x_n^*, \\ \alpha r_k^n x_n^* + d_k^1, & \text{if } \delta > x_n^*. \end{cases} \tag{19}$$

We define $f_k(\delta)$ as $f_k(\delta) = WORST(\delta)/OPT(\delta)$, thus the maximum $f_k(\delta)$ is the worst-case competitive ratio. The function of $f_k(\delta)$ is expressed as:

$$f_k(\delta) = \begin{cases} r_k^n / r_k^1, & \text{if } 1 \leq \delta \leq x_n^*, \\ (\alpha r_k^n x_n^* + d_k^1)/\alpha r_k^1 \delta, & \text{if } \delta > x_n^*. \end{cases} \tag{20}$$

If $1 \leq \delta \leq x_n^*$, $f_k(\delta)$ is a constant function. If $\delta > x_n^*$, $f_k(\delta)$ is the monotonically decreasing function, so we need to compare the value $f_k(\delta = x_n^* + 1)$ with the value r_k^n/r_k^1. When $\delta = x_n^* + 1$, $f_k(x_n^* + 1) = (\alpha r_k^n x_n^* + d_k^1)/\alpha r_k^1 (x_n^* + 1)$. Obviously, $(\alpha r_k^n x_n^* + d_k^1)/\alpha r_k^1 (x_n^* + 1) - r_k^n / r_k^1 = (d_k^1 - \alpha r_k^n)/\alpha r_k^1 (x_n^* + 1) > 0$. Therefore, the maximum $f_k(\delta)$ would be reached when $\delta = x_n^* + 1$ and its value could be computed by $\max\{(\alpha r_k^n x_n^* + d_k^1)/\alpha r_k^1 (x_n^* + 1)\}, n \in \mathcal{N}$. □

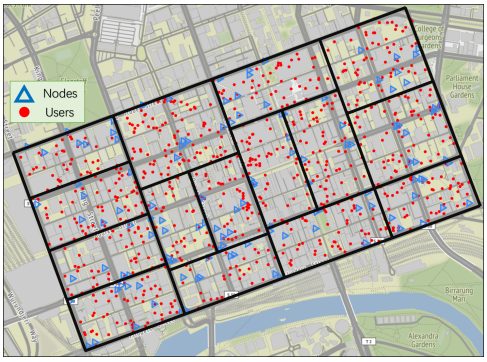


Fig. 3. Graphical representation of the edge nodes and users in Melbourne CBD area in the used dataset.

V. Performance Evaluation

In this section, we validate the performance of O-RDC with trace-driven simulations and OpenWhisk implementation.

A. Simulation Setup

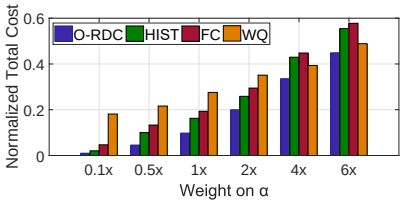
Dataset: We take advantage of the EUA dataset [34], which contains 125 edge nodes and 816 users distributed in the area, we divide this area into 15 zones and each zone contains 7 to 10 edge nodes, as shown in Figure 3. The network latency between two zones in our model is measured by the geographical distance which is computed by the zone's approximate center coordinates.

Request: We assume in our system there exist $K = 50$ types of requests. The corresponding container sizes are in $[20, 250]$ MB. The users within each zone will randomly generate the requests and the generated requests conform to the Zipf-β popularity law [29], [28]. Typically, the parameter β is between 0.5 and 1.5. Furthermore, We use the Azure dataset [12], which contains the invocations of functions across 14 days on Azure, to generate the request arrival pattern.

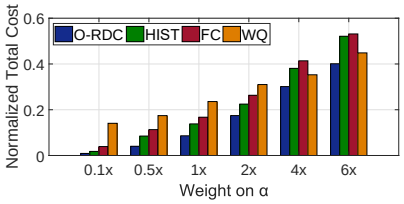
Edge node: We assume the system has 5 types of edge nodes, where the CPU frequencies are in $\{2.4, 2.7, 3.0, 3.3, 3.6\}$ GHz and the allocated memory resources are randomly distributed within $[16, 24]$ GB.

Instantiation and retention cost: According to [13], we assume the container instantiation cost d_k^n is inversely proportional to the CPU frequency of node n and the container retention cost r_k^n is proportional to the CPU frequency of node n. The d_k^n and r_k^n are all proportional to the container size u_k.

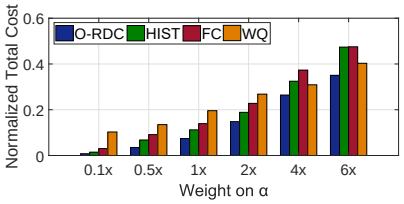
Number of users: To best simulate the resource constraints in edge nodes, we assume the total number of users in our simulation is $10\times$ or $20\times$ the users in the chosen dataset. This creates a scenario where a single node cannot handle all the distributed requests due to its limited memory capacity as the number of users increases.



(a) Zipf-$\beta = 0.5$

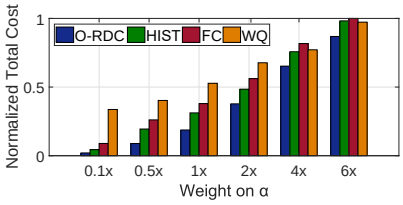


(b) Zipf-$\beta = 1.0$

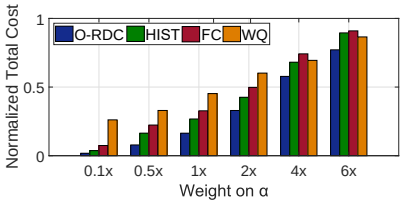


(c) Zipf-$\beta = 1.5$

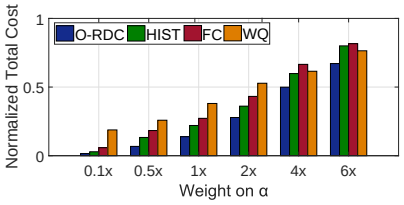
Fig. 4. Total cost with $10\times$ users.



(a) Zipf-$\beta = 0.5$



(b) Zipf-$\beta = 1.0$



(c) Zipf-$\beta = 1.5$

Fig. 5. Total cost with $20\times$ users.

B. Performance Benchmark

- **Histogram (HIST):** This is the strategy proposed by [12]. The history arrival information of requests is used to determine pre-warming and destroying containers.
- **Fixed Caching (FC):** This is the caching strategy widely used in AWS Lambda [8]. An instantiated container will be cached for a fixed long duration. Thus, using the solution of classic ski-rental problem, we denote the maximum $d_k^n/\alpha r_k^n$ as the Fixed Caching (**FC**) for type-k request, respectively.
- **Warm Queue (WQ):** This is the strategy proposed by [35]. A warm queue, whose capacity equals the maximum number of requests, is designed to cache containers. When replacing cached containers, it uses the First In and First Out strategy. This is a retention-unaware caching strategy only considering how to replace containers.

Without loss of generality, the above two strategies distribute requests evenly to each node. We compare the total cost accumulated across 1000 time slots.

C. Performance with Synthetic Trace

Overall performance summary: From Figure 4(a) to Figure 5(c), we can observe that O-RDC could receive obvious better performance for reducing the total cost by up to 94.5% compared with WQ. O-RDC could also outperform HIST and FC by up to 71.7% and 53.6%, respectively.

Impact of parameter α: We set α from 0.1 to 6. When α is greater than 6, the container retention cost αr_k^n becomes larger than the container instantiation cost d_k^n, which implies caching containers will always incur more cost than instantiating containers so the container caching cannot provide any benefit. From Figure 4 to Figure 5, we can observe that the value of α has a noticeable impact on the overall system cost. O-RDC and FC shorten their caching durations when α increases, leading to more times to instantiate containers and more retention cost. HIST pre-warm and cache containers based on recent request arrival pattern. WQ caches containers based only on the number of repeated requests between two consecutive time slots, which is not helpful in optimizing the total cost across multiple time slots. That is why we observe a dramatic increase in the total cost. The results under varying α prove the effectiveness and generality of O-RDC.

TABLE II
REQUEST DISTRIBUTION PROBABILITY

Node Type	1	2	3	4	5
Probability	23.8%	15.2%	11%	8.3%	41.7%

Impact of parameter β: We use the Zipf law to simulate the request popularity and vary the Zipf skewness parameter β from 0.5 to 1.5 [29], [28]. Observing from Figure 4 to Figure 5, increasing β leads to decreased total cost by up to 21.1% for all baselines. As β increases, higher probability requests will be generated more frequently and lower probability requests become more sparse. For O-RDC, FC, and HIST, the cached containers could serve more higher probability requests with shorter request arrival intervals so as to reduce the instantiation cost. For WQ, generating popular requests more frequently increases the number of repeated requests between two consecutive time slots, which correspondingly reduces the container replacement times. This is why the increase of β leads to similar declines in total cost for all baselines. Results prove the effectiveness of O-RDC in most scenarios where the requests conform to the Zipf law.

Impact of the number of users: We increase the number of users in the dataset from $10\times$ to $20\times$ to explore the impact of limited memory capacity on overall system cost. Results from Figure 4 to Figure 5 show that O-RDC performs stably and optimally even when doubling the number of users. The total cost of O-RDC generally increases from 193% to 205%. Although the significant increase of users could break the optimal request distribution on the node, as shown in Table II,

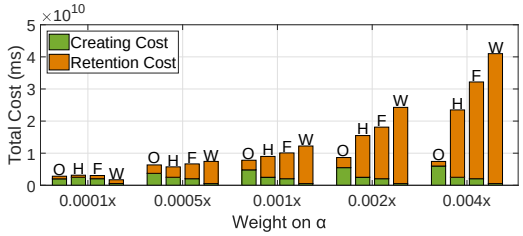


Fig. 6. Performance on Apache OpenWhisk.

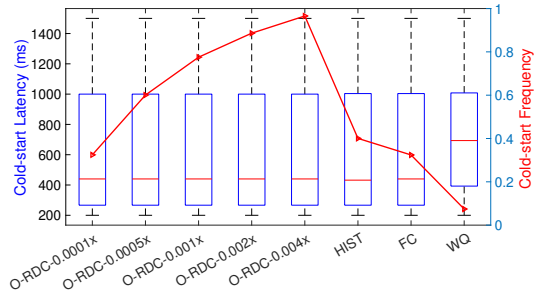


Fig. 7. The cold-start latency and frequecy on O-RDC and baselines.

due to the memory resource constraints, O-RDC can still cache the most used containers and timely destroy the idle containers. This confirms that the performance of O-RDC is still bounded when the optimal request distribution and container caching cannot be realized due to the memory resource constraints.

D. Performance with Implementation in Apache OpenWhisk

In OpenWhisk, Controller routes actions invocations and Invoker launches the containers to execute the actions [36]. We modify these components (i.e., Controller and Invoker) to implement O-RDC. We add the logic of request distribution into the Controller and modify the unloading time of containers in Invoker. We implement OpenWhisk on 3 servers with different CPU frequencies (2.4GHz, 3.2GHz, 4.0GHz). The request arrival pattern is simulated by Azure dataset [12]. We set the Fixed Caching (FC) as 10-minute fixed keep-alive policy which is used in several serverless computing platforms and set each time slot as 1 second which is mostly used in serverless computing platforms to figure out cost. We set instantiation cost d_k^n and retention cost r_k^n as instantiation latency and retention time, respectively. In this case, when $\alpha \leq 0.004$, the $\alpha r_k^n \leq d_k^n$ will always follow. Figure 6 shows the total time cost of our system under different retention parameter α, which has a same trend as simulation results. O-RDC outperforms HIST by up to 35.2%, fixed 10-min keep-alive policy by up to 51.4% and WQ by up to 36.7%, respectively. Figure 7 shows the cold-start latency and frequency on O-RDC and all baselines. We find that the median cold-start latency of WQ is higher than O-RDC, HIST, and FC. That is because WQ uses the First In and First Out strategy, which is different from others who make decision based on the caching duration. The results also show that when α grows, the cold-start frequency of O-RDC increases from 32.4% to 96.5% since it will destroy containers more frequently to suppress the retention cost. The cold-start frequency of all baselines keeps lower than 40%, which results in a huge retention cost (see Figure 6). The results further confirm the usability of our algorithms in the real system.

VI. Related Work

Request distribution and service placement in edge computing: A lot of works have been carried out on request distribution in edge computing. He et al. propose a joint service placement and request scheduling algorithm to maximize the number of served users in mobile edge computing under resource constraints [21]. Later, they investigate a similar problem by considering a two-time-scale framework to enable system stability [22]. Closely related to [21], Poularakis et al. propose an approximation algorithm based on the randomized rounding technique to maximize the number of served users [23]. However, all these works focus on optimizations in a single time slot. Our optimization problem spans across multiple time slots. Service placement in edge computing has also been extensively studied in [18], [19], [20], [30], [37], [38], [39], [40], but not with service caching considered.

Content caching in edge computing: In edge computing environments, content caching is of high importance as it can significantly reduce the delay in content delivery if done properly. Content caching has been widely studied for wireless/edge networks [27], [41], [42], [25], [26], mostly with the target of reducing the content fetching time. For example, Hou et al. present an online strategy for single node [25] and Dehghan et al. consider the joint optimization of content caching and request routing [26]. However, none of the above consider the retention cost of cached content. While [28] and [29] seem closely related to our problem as they also study the tradeoff between content fetching time and cache retention cost, content caching is fundamentally different from container caching since cached content can serve multiple requests simultaneously, while a cached container can serve only one request at a time. This renders all the existing solutions for content caching inapplicable. Furthermore, our work contains request distribution strategies based on the heterogeneous hardware specifications of edge nodes, which has not been covered in the existing works.

VII. Conclusion

In this paper, we study the retention-aware container caching problem in serverless edge computing, with the goal of optimizing system efficiency by container caching jointly with request distribution. We show by analysis that this problem is hard to solve even when being simplified. We reveal that the problem in special cases can be mapped to the classic ski-rental problem and we propose an online competitive algorithm for these special cases. We further propose an online algorithm for the general case considering resource constraints and network latency. Extensive trace-driven simulations demonstrate the effectiveness of our algorithm on the overall system cost.

APPENDIX

A. Proof of Theorem 1

Theorem 1. *The simplified C2RD problem is NP-hard.*

Proof. The proof can be conducted by a polynomial-time reduction from the generalized assignment problem (GAP), a classic combinatorial optimization problem which is known to be NP-hard [43]. GAP is described as assigning a set of K jobs to a set of N agents with the minimum cost. Define $c_{n,k}$ as the cost of assigning job k to agent n, $w_{n,k}$ as the resource required by agent n to perform job k, and W_n as the resource availability of agent n and binary variable $x_{n,k}$ indicates whether k is assigned to n (1 if agent n performs job k and 0 otherwise). GAP can be formalized as follows:

$$\begin{aligned}(\mathbf{P}_2)\quad &\min \textstyle\sum_{n=1}^{N}\sum_{k=1}^{K} c_{n,k}x_{n,k}\\ \text{s.t.}\quad &\textstyle\sum_{n=1}^{N} x_{n,k}=1, \forall k,\\ &\textstyle\sum_{k=1}^{K} w_{n,k}x_{n,k}\le W_n, \forall n,\\ &x_{n,k}\in\{0,1\}, \forall n, \forall k.\end{aligned} \tag{21}$$

Now we show that GAP is equivalent to a simplified version of C2RD. We assume in a zone each type of request is only generated once (i.e., $\sum_{n\in\mathcal{N}} m_{k,t}^n = 1, \forall k$). Also, we destroy the container immediately after serving a request. That is, the optimization goal is to minimize the instantiation cost $\sum_{n=1}^{N}\sum_{k=1}^{K} d_k^n m_{k,t}^n$ when assigning the requests k to nodes n while meeting the resource constraints $u_k m_{k,t}^n \le U_n$. We formulate this simplified version of C2RD as follows:

$$\begin{aligned}(\mathbf{P}_3)\quad &\min \textstyle\sum_{n=1}^{N}\sum_{k=1}^{K} d_k^n m_{k,t}^n\\ \text{s.t.}\quad &\textstyle\sum_{n=1}^{N} m_{k,t}^n=1, \forall k,\\ &\textstyle\sum_{k=1}^{K} u_k m_{k,t}^n\le U_n, \forall n,\\ &m_{k,t}^n\in\{0,1\}, \forall n, \forall k.\end{aligned} \tag{22}$$

The equivalence of $\mathbf{P}_2$ and $\mathbf{P}_3$ is achieved if we map agents, jobs, job assignment cost $c_{n,k}$, job required resource $w_{n,k}$, and resource availability W_n in GAP to edge nodes, requests, container instantiation cost d_k^n, container resource demand u_k, and node resource capacity U_n in C2RD, respectively. Clearly, the mapping can be done in polynomial time. □

B. Proof of Theorem 3

Theorem 3. *The optimal request distribution and container caching function $p_n^*(x)$ satisfies the following properties:*

$$p_n^*(x)=\begin{cases}\varphi_n e^{\frac{x}{\theta_k^n}}, & \text{if } x\in(b_{n+1},b_n),\\ 0, & \text{otherwise},\end{cases} \tag{23}$$

where θ_k^n and φ_n satisfy that

$$\begin{aligned}\theta_k^n &= d_k^n/\alpha r_k^n,\\ \varphi_n d_k^n e^{\frac{b_n}{\theta_k^n}} &= \varphi_{n-1} d_k^{n-1} e^{\frac{b_n}{\theta_k^{n-1}}}, \forall n\in[2,N].\end{aligned} \tag{24}$$

The range (b_{n+1}, b_n) represents the caching duration on node n, where caching breakpoints b_n satisfy that $d_k^N/\alpha r_k^1 = b_1 \ge b_2 \ge ... \ge b_N \ge b_{N+1} = 0$. Solving problem $\mathbf{P}_4$ is equivalent to computing caching breakpoints $\{b_1, b_2, ..., b_{N+1}\}$.

Proof. (sketch) The minimum γ is reached when it satisfies caching duration $x \in [0, d_k^N/\alpha r_k^1]$. With $x \in [0, d_k^N/\alpha r_k^1]$, the request arrival time interval δ can be reduced to $\delta \in (0, d_k^N/\alpha r_k^1]$ to solve $\mathbf{P}_4$. Thus the search space for $\mathbf{P}_4$ can be significantly reduced as follows:

$$\begin{aligned}(\mathbf{P}_5)\quad &\min \gamma\\ \text{s.t.}\quad &\textstyle\sum_{n\in\mathcal{N}} C_n(p_n(x),\delta)/\alpha r_k^1\delta\le\gamma,\\ &\textstyle\sum_{n\in\mathcal{N}}\int_0^{d_k^N/\alpha r_k^1} p_n(x)dx=1,\\ &x\in[0,d_k^N/\alpha r_k^1], \delta\in(0,d_k^N/\alpha r_k^1].\end{aligned} \tag{25}$$

For any probability function $p_n(x) = p_n^*(x)$, it holds that

$$\textstyle\sum_{n\in\mathcal{N}} C_n(p_n(x),\delta)/\alpha r_k^1\delta = \varphi_1\theta_k^1 e^{d_k^N/d_k^1}. \tag{26}$$

From the above equation we know that solving problem $\mathbf{P}_5$ is equivalent to minimizing φ_1, i.e.,

$$\begin{aligned}(\mathbf{P}_6)\quad &\min \varphi_1\\ \text{s.t.}\quad &\textstyle\sum_{n\in\mathcal{N}}\varphi_n\theta_k^n\left(e^{\frac{b_{n-1}}{\theta_k^n}}-e^{\frac{b_n}{\theta_k^n}}\right)=1,\\ &\varphi_n d_k^n e^{\frac{b_n}{\theta_k^n}} = \varphi_{n-1} d_k^{n-1} e^{\frac{b_n}{\theta_k^{n-1}}}, \forall n\in[2,N],\\ &d_k^N/\alpha r_k^1 = b_1\ge b_2\ge ...\ge b_N\ge b_{N+1}=0,\\ &\varphi_n>0, \forall n\in\mathcal{N}.\end{aligned} \tag{27}$$

If we already know the values for caching breakpoints $\{b_1, b_2, ..., b_{N+1}\}$, we know φ_n is proportional to φ_1. As a result, we can obtain constants ω_n where $\omega_n\varphi_1 = \int_{b_{n+1}}^{b_n} p_n(x)dx$ and a constant $\omega = \sum_{n\in\mathcal{N}}\omega_n$ where $\omega\varphi_1 = \sum_{n\in\mathcal{N}}\int_{b_{n+1}}^{b_n} p_n(x)dx$. Using the fact that $\sum_{n\in\mathcal{N}}\int_{b_{n+1}}^{b_n} p_n(x)dx = 1$, we can solve φ_1, all the φ_n and consequently the optimization problem $\mathbf{P}_6$. We first fix φ_1 to be 1 and compute the optimal caching breakpoints $\{b_1, b_2, ..., b_{N+1}\}$ that maximize $\sum_{n\in\mathcal{N}}\int_{b_{n+1}}^{b_n} p_n(x)dx$. With these values, we compute the values for φ_n and ω_n. Then, we add up all ω_n to be ω, i.e., $\omega = \sum_{n\in\mathcal{N}}\omega_n$, and normalize all the probability function $p_n(x)$ by setting φ_n to be φ_n/ω. The proof follows from [33] where more details can be found (see Lemma 1 to Lemma 6). □

C. Proof of Theorem 4

Theorem 4. *The time complexity of O-RDC is $O(KZ^2N^2)$.*

Proof. Firstly, O-RDC using Algorithm 1 to compute $p_n^*(x)$ for the type-k request needs to traverse N nodes three times. In Algorithm 1, each traversal will orderly execute the function in line 3-4, 7-8 and 11 respectively, which leads to $O(N)$ time complexity. Thus, the time complexity to compute $p_n^*(x)$ for K types of requests is $O(KN)$. Specifically, $p_n^*(x)$ for each type of request will be only computed once. After that, in Algorithm 3, O-RDC will traverse Z zones to handle requests. In each zone, O-RDC needs to handle K types of requests. Now we analyze the time complexity of handling one type of request in a zone. In line 4-5, O-RDC collecting the generated requests and cached containers requires $O(N)$ time complexity. In line 8-9 and 16-17, O-RDC distributing

requests and destroying containers also leads to $O(N)$ time complexity. The highest computation is in line 13-14 where O-RDC will traverse neighbor zones and their contained nodes again, which causes $O(ZN^2)$ time complexity. Therefore, the time complexity to handle K types of requests in a zone is $O(KZN^2)$. Consequently, the time complexity of traversing Z zones to handle requests is $O(KZ^2N^2)$. □

REFERENCES

[1] A. I. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] M. Satyanarayanan and N. Davies, "Augmenting cognition through edge computing," *IEEE Computer*, vol. 52, no. 7, pp. 37–46, 2019.

[3] L. L. Peterson, T. E. Anderson, S. Katti, N. McKeown, G. M. Parulkar, J. Rexford, M. Satyanarayanan, O. Sunay, and A. Vahdat, "Democratizing the network edge," *Computer Communication Review*, vol. 49, no. 2, pp. 31–36, 2019.

[4] Q. Chen, Z. Zheng, C. Hu, D. Wang, and F. Liu, "On-edge multi-task transfer learning: Model and practice with data-driven task allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1357–1371, 2020.

[5] M. Li, Q. Zhang, and F. Liu, "Finedge: A dynamic cost-efficient edge resource management platform for NFV network," in *IEEE/ACM IWQoS*. IEEE, 2020, pp. 1–10.

[6] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

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

[8] "AWS Lambda," https://aws.amazon.com/lambda/, 2019.

[9] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: extending serverless computing to the edge of the network," in *ACM International Systems and Storage Conference (SYSTOR)*, 2017, p. 28:1.

[10] "Aliyun EdgeRoutine," https://www.aliyun.com/activity/cdn/edgeroutine, 2020.

[11] "AWS Lambda@Edge," https://aws.amazon.com/lambda/edge/, 2019.

[12] M. Shahrad, R. Fonseca, I. 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 Annual Technical Conference (ATC)*, 2020, pp. 205–218.

[13] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 218–233.

[14] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: towards high-performance serverless computing," in *USENIX Annual Technical Conference (ATC)*, 2018, pp. 923–935.

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019.

[16] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, "WARM: improving NAND flash memory lifetime with write-hotness aware retention management," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–14.

[17] S. Lee, K.-D. Kang, H. Lee, H. Park, Y. Son, N. S. Kim, and D. Kim, "Greendimm: Os-assisted dram power management for dram with a sub-array granularity power-down state," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 131–142.

[18] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1002–1016, 2017.

[19] L. Wang, L. Jiao, T. He, J. Li, and M. Mühlhäuser, "Service entity placement for social virtual reality applications in edge computing," in *IEEE Conference on Computer Communications (INFOCOM)*, 2018, pp. 468–476.

[20] S. Pasteris, S. Wang, M. Herbster, and T. He, "Service placement with provable guarantees in heterogeneous edge computing systems," in *IEEE Conference on Computer Communications (INFOCOM)*, 2019, pp. 514–522.

[21] T. He, H. Khamfroush, S. Wang, T. L. Porta, and S. Stein, "It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 365–375.

[22] V. Farhadi, F. Mehmeti, T. He, T. L. Porta, H. Khamfroush, S. Wang, and K. S. Chan, "Service placement and request scheduling for data-intensive applications in edge clouds," in *IEEE Conference on Computer Communications (INFOCOM)*, 2019, pp. 1279–1287.

[23] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *IEEE Conference on Computer Communications (INFOCOM)*, 2019, pp. 10–18.

[24] L. Wang, L. Jiao, J. Li, J. Gedeon, and M. Mühlhäuser, "MOERA: mobility-agnostic online resource allocation for edge computing," *IEEE Trans. Mob. Comput.*, vol. 18, no. 8, pp. 1843–1856, 2019.

[25] I. Hou, T. Zhao, S. Wang, and K. Chan, "Asymptotically optimal algorithm for online reconfiguration of edge-clouds," in *ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2016, pp. 291–300.

[26] M. Dehghan, B. Jiang, A. Seetharam, T. He, T. Salonidis, J. Kurose, D. Towsley, and R. K. Sitaraman, "On the complexity of optimal request routing and content caching in heterogeneous cache networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1635–1648, 2017.

[27] J. Tadrous and A. Eryilmaz, "On optimal proactive caching for mobile networks with demand uncertainties," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2715–2727, 2016.

[28] S. Shukla and A. A. Abouzeid, "Proactive retention aware caching," in *IEEE Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9.

[29] S. Shukla, O. Bhardwaj, A. A. Abouzeid, T. Salonidis, and T. He, "Proactive retention-aware caching with multi-path routing for wireless edge networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 1286–1299, 2018.

[30] B. Gao, Z. Zhou, F. Liu, and F. Xu, "Winning at the starting line: Joint network selection and service placement for mobile edge computing," in *IEEE Conference on Computer Communications (INFOCOM)*, 2019, pp. 1459–1467.

[31] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *Annual Symposium on Foundations of Computer Science (FOCS)*, 1986, pp. 244–254.

[32] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. S. Owicki, "Competitive randomized algorithms for non-uniform problems," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1990, pp. 301–309.

[33] L. Ai, X. Wu, L. Huang, L. Huang, P. Tang, and J. Li, "The multi-shop ski rental problem," in *ACM SIGMETRICS*, 2014, pp. 463–475.

[34] P. Lai, Q. He, M. Abdelrazek, F. Chen, J. Hosking, J. Grundy, and Y. Yang, *Optimal Edge User Allocation in Edge Computing with Variable Sized Vector Bin Packing*, 11 2018, pp. 230–245.

[35] M. G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410.

[36] "Apache OpenWhisk," https://openwhisk.apache.org/, 2019.

[37] Q. Zhang, F. Liu, and C. Zeng, "Online adaptive interference-aware vnf deployment and migration for 5g network slice," *IEEE/ACM Transactions on Networking*, vol. 29, no. 5, pp. 2115–2128, 2021.

[38] P. Jin, X. Fei, Q. Zhang, F. Liu, and B. Li, "Latency-aware vnf chain deployment with efficient resource reuse at network edge," in *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 267–276.

[39] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "Nfvdeep: Adaptive online service function chain deployment with deep reinforcement learning," in *Proceedings of the International Symposium on Quality of Service (IWQoS)*, 2019, pp. 1–10.

[40] S. Chen, L. Jiao, F. Liu, and L. Wang, "Edgedr: An online mechanism design for demand response in edge clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 343–358, 2022.

[41] Y. Huang, X. Song, F. Ye, Y. Yang, and X. Li, "Fair caching algorithms for peer data sharing in pervasive edge computing environments," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 605–614.

[42] X. Cao, J. Zhang, and H. V. Poor, "An optimal auction mechanism for mobile edge caching," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 388–399.

[43] P. C. Chu and J. E. Beasley, "A genetic algorithm for the generalised assignment problem," *Computers & OR*, vol. 24, no. 1, pp. 17–23, 1997.