Efficient Container Assignment and Layer Sequencing in Edge Computing

Jiong Lou, Hao Luo, Zhiqing Tang, Weijia Jia, *Fellow, IEEE*, and Wei Zhao, *Fellow, IEEE*

Abstract—Containers are becoming a popular way of running applications in edge computing. Before running the application, the edge node must download the application's container image consisting of multiple layers. However, given the limited bandwidth in edge computing, the container startup latency due to long image download time seriously affects the real-time performance. In this article, we jointly determine the container assignment and the layer download sequence to reduce the total startup latency. We formulate the Container Assignment and Layer Sequencing (CALS) problem and prove its NP-hardness. A Layer-Aware Scheduling Algorithm (LASA) is proposed, fully considering layer sharing among images. First, layers shared by the same set of images are grouped to reduce CALS's problem scale without affecting the optimal result. Second, considering both layer sharing and existing layer size on edge nodes, a layer-aware algorithm is designed to assign containers to appropriate edge nodes. Finally, to determine the layer download sequence on each edge node, an approximation algorithm is proposed. We further analyze the approximation ratio of LASA in the case of identical edge nodes with sufficient capacity. Extensive experiments based on real-world data show the effectiveness of LASA, which reduces the total startup latency by 40% to 60%.

Index Terms—Container scheduling, container startup, edge computing, layer sharing

1 Introduction

With the increasing demand for low-latency and highly flexible applications, cloudlets [1], fog [2] and edge computing [3] that are in closer proximity to mobile devices provide attractive ways to deploy applications. Ultra-low latency and ultra-high bandwidth 5G technology further facilitates the development of edge computing [4], [5]. Virtualization can provide isolated environments for applications to avoid software-dependency conflicts and enhance system robustness [6]. However, in edge computing, the computation resources and communication resources are limited compared with the cloud, and the edge environment changes are rapid [7], [8]. Traditional virtualization techniques, i.e., heavy virtual machine (VM), cannot resolve these issues. The emerging technique, container, is believed to be a promising way to deploy applications in edge computing [9], [10]. Multiple containers on the same node share the machine's OS system kernel and thereby do not require an OS per container, driving higher server efficiencies, suitable for resource-limited edge nodes.

Though the container is lightweight, its startup latency can significantly affect users' quality of experience, especially for applications that have short execution times (e.g., processing periodic updates from sensors) or need rapid response times (e.g., robot motion [11]) in edge computing. The container startup latency consists of fetching (if not exist) the container image from the remote registry to its host machine and installing the image. It is reported that, on average, a median container startup needs 25 seconds in Google clouds [12]. In edge computing, the startup latency is much higher for longer image download time due to the limited bandwidth. For example, the download time of an image sized 300 MB is at least 240 seconds with a 10 Mbps link. The image download time occupies the most proportion of the startup time since the image installation latency is lower (about one second) and more stable [13] on heterogeneous devices. Besides, due to limited storage resources, dynamic user mobility, and a huge number of container images, it is impossible to store all images on every edge node in advance. Thus, the considerable container startup time becomes an urgent problem to be optimized.

In the literature, some studies are proposed to reduce container startup latency by fetching image files on demand [14], extracting common parts of multiple containers [15], [16], or reorganizing images [17]. However, these studies

- *Jiong Lou is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China, and also with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Guangdong 519087, China. E-mail: lj1994@sjtu.edu.cn.*
- *Hao Luo is with the Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College Zhuhai, Guangdong 519087, China. E-mail: r130201705@mail.uic.edu.cn.*
- *Zhiqing Tang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai), Zhuhai 519087, China. E-mail: domain@sjtu.edu.cn.*
- *Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai), Zhuhai 519087, China, and also with the Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College Zhuhai, Guangdong, 519087, China. E-mail: jiawj@bnu.edu.cn.*
- *Wei Zhao is with the CAS Shenzhen Institute of Advanced Technology, Shenzhen 518055, China. E-mail: zhao.wei@siat.ac.cn.*

Manuscript received 29 Apr. 2021; revised 12 Feb. 2022; accepted 9 Mar. 2022. Date of publication 16 Mar. 2022; date of current version 10 Apr. 2023.
This work was supported in part by Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College (UIC), Zhuhai, under Grant 2020KSYS007; in part by the Chinese National Research Fund (NSFC), under Grant 61872239; in part by Zhuhai Science-Tech Innovation Bureau, under Grants ZH22017001210119PWC and 28712217900001 and in part by the Guangdong Engineering Center for Artificial Intelligence and Future Education, Beijing Normal University, Zhuhai, Guangdong, China. (Corresponding authors: Zhiqing Tang and Weijia Jia.)
Digital Object Identifier no. 10.1109/TSC.2022.3159728

either modify the container system architectures [14], [17] or reduce container isolation [15], [16]. In fact, a container image consists of multiple individual layers and different container images share common layers [14], which efficiently saves storage by reducing redundant files. Moreover, the layer sharing mechanism can also be utilized to reduce startup latency without modifying containers. Fu *et al.* [13] design a layer-match scheduling policy that assigns a container to the edge node that stores the most layers required by the container to reduce the remaining download time.

Nevertheless, the previous studies still ignore the following essential issues: 1) Joint scheduling of multiple containers. Edge computing can serve multiple users simultaneously [18]. Making scheduling decisions for a set of containers can achieve a lower total startup latency than scheduling each container independently, like layer-match scheduling in [13] (called Container Assignment problem). 2) The impact of layer download sequence (called Layer Sequencing problem). Since a container can run right after fetching all of its layers, a proper download sequence of layers belonging to different containers can further reduce the total startup latency. Furthermore, heterogeneous edge environments and layer sharing among images make the problem more challenging.

In this paper, to address these issues, we consider assigning multiple containers to heterogeneous edge nodes and sequencing layers on each edge node. The Container Assignment and Layer Sequencing (CALS) problem is formulated as a Mixed Integer Quadratic Programming (MIQP) problem. Since the CALS problem involves assigning multiple containers and sequencing layers shared by different container images, it is proved to be NP-hard.

A three-step Layer-Aware Scheduling Algorithm (LASA) is proposed to make scheduling decisions efficiently: First, the layers shared by the same set of containers are grouped. The optimal result of scheduling a group of layers as a whole is proved to be equal to scheduling individual layers. The problem scale is drastically reduced. Second, a layer-aware algorithm is designed to assign containers to edge nodes properly. At each time, a container-node pair is selected by considering two important factors, layer sharing among containers and the size of existing layers already on edge nodes. After determining every container's assignment, the CALS problem is decomposed into multiple independent sub-problems that optimize each edge node's layer download sequence. Finally, to determine the layer download sequence, a greedy layer sequencing algorithm with an approximate ratio of 2 is proposed, efficiently running on each edge node in parallel. The approximation ratio of LASA is proved to be $2|E|$ in the case of identical edge nodes with infinite storage capacity and unlimited running container number, where $|E|$ is the number of edge nodes. Extensive simulations are performed to compare the performance of LASA with existing baselines. The simulation results show that LASA substantially reduces the total startup time by 40% to 60%.

To the best of our knowledge, we are the first team to tackle the CALS problem in edge computing. The main contributions of this paper are summarized as:

1) To minimize container startup latency, we jointly consider the container assignment and the layer download sequence, and formulate the CALS problem, which is proved to be NP-hard.
2) To solve the CALS problem, LASA is proposed to make scheduling decisions efficiently. To determine the layer download sequence on each edge node, a greedy layer sequencing algorithm with an approximate ratio of 2 is designed, which can run for each edge node in parallel. Further, We analyze the computational complexity of LASA and prove the approximation ratio of LASA in the case of identical edge nodes with sufficient storage capacity and unlimited running container number.
3) Finally, we perform extensive experiments with real-world data collected from DockerHub [19], and demonstrate the efficiency of the proposed algorithm in comparison with existing baselines.

2 Related Work

2.1 Container Scheduling

In edge computing, tasks are typically running on specific containers. Related work about container scheduling and task scheduling is discussed [20], [21], [22]. Chen *et al.* [20] first study the multi-user computation offloading problem for edge computing and design a distributed task offloading algorithm based on game theory. An approximation algorithm, called OnDisc [21], is derived to optimize the response time of online multi-task dispatching and scheduling. A Logic-Based Benders Decomposition algorithm [22] is designed to maximize the admitted task number, jointly considering task assignment, resource allocation, and task execution order. However, these studies do not consider optimizing the significant container startup latency.

2.2 Dependent Task Scheduling and Network Function Virtualization Placement

The layer sequencing on an edge node is a special case of the dependent task scheduling problem that has been further studied [23], [24]. In [23], the authors propose heuristics to maximize the overall computation of co-located edge devices. The authors of [24] propose an approximation scheme to minimize the delay. However, dependent tasks can be assigned to different nodes, whereas layers of an image must be downloaded on the same edge node to form the complete image.

Virtual Network Function (VNF) placement [25], [26], [27] is also related to this paper. VNF placement is to place VNFs to minimize the latency [26], reduce costs [28], or maximize the throughput [29]. However, there are major differences between the CALS problem and the VNF placement problem: 1) The VNF placement problem ignores the startup time or defines a deployment cost. It neglects the layer sharing feature and the layer download sequence, making it far from optimal. 2) VNFs can be scheduled to different servers, while layers of a container must be downloaded on the same edge node to which the container is assigned. It is not appropriate to regard each image layer as a VNF.

2.3 Overlapping Job Scheduling

Some studies focus on the parallel machine overlapping job scheduling problem [30], [31]. Overlapping jobs are jobs

Fig. 1. Layers of three popular images named geonetwork, tomcat and xwiki. Circles represent layers that constitute images. The circles on the left with bold lines represent the layers shared by three images.

with duplicate contents [31], analogous to layer sharing among images. Since the overlapping job scheduling problem is proved to be NP-hard [30], greedy algorithms [30] and a branch-and-bound algorithm [31] are proposed to solve it. However, these studies only aim to optimize makespans of the scheduling problem. In this way, these algorithms optimize the maximum makespan of each machine without considering the job sequence. However, the job sequence can significantly influence each job's completion time, and the layer sequence problem on a single edge node is proved to be NP-hard in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSC.2022.3159728.

2.4 Containerized Application Startup Acceleration

Much effort has been made to accelerate the containerized application startup. Slacker [14] lazily pulls files during runtime. Ma *et al.* [32] propose an edge computing platform architecture to support seamless application migration, which reduces the transferred file volumes by leveraging images' layered storage. Skourtis *et al.* [17] find that many layers only differ in a small number of files, so they reorganize image layers to reduce storage and network consumption. Cntr [15] and Pocket [16] move common parts of multiple containers to a daemon process. However, these approaches are not transparent, for they require substantial changes to containers or reduce the isolation of containers. Layer-aware container scheduling is orthogonal and complementary to these techniques, and neither container images nor the container system architecture is modified.

2.5 Layer Match Container Scheduling

A layer-match scheduler [13] is proposed to reduce the image download time in container management systems by taking layer sharing into account. The scheduler tries to place a container at a node storing most image layers required by this container. The main differences between this work and [13] are as follows: 1) Joint scheduling of multiple containers is considered in this work, which can further reduce the total startup latency. 2) The impact of layer download sequence is studied to reduce the total startup latency, which is ignored in [13]. 3) The CALS problem in the heterogeneous edge computing environment is investigated in this work.

3 System Model and Problem Formulation

3.1 Background

3.1.1 Containers, Images and Layers

A container is a standard unit of software that packages up code and all its dependencies to be deployed quickly and reliably to various computing environments. Each container has its own environment called namespace, where specific processes are running and isolated from the rest of the system.

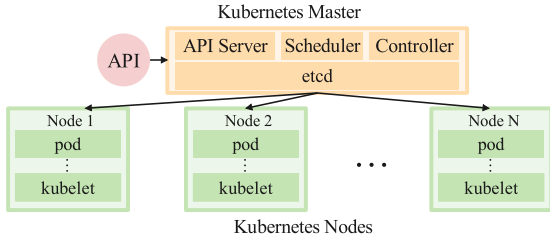


Fig. 2. The Kubernetes cluster of one master and multiple nodes.

On a physical machine, containers share an operating system (OS) kernel, using fewer resources than VMs.

An image is a lightweight, standalone, executable software package that includes everything needed to run a container: a Linux distribution, application binaries, configuration files, etc. The image is read-only, copy-on-write, and thus can be shared by multiple containers.

As container images are self-contained, different images frequently include common files [33]. The layer sharing mechanism is applied to reduce redundant files. Each image consists of a list of read-only layers, which can be shared among images. Each layer has a hash digest taken over its content so that it can be uniquely identified [13]. Fig. 1 shows the relationships between layers and images. Layers are stacked, and union mounted to the container's root file system at runtime [32]. The lines between circles represent the layer stack order in an image but not the download order, and layers can be downloaded in arbitrary order.

3.1.2 Container Startup Time

The container startup time can dominate the latency of real-time applications in edge computing [34]. The container startup time consists of the image download time and installation time. Since containers share the OS kernel, the image installation time is stable (around one second) irrespective of the image size [13]. Compared with the image installation time, the download time is considerable. According to [17], for 10,000 most popular images in DockerHub [19], the average image size achieves 500 MB. On an edge node connected to the cloud by a 10 Mbps link, it costs 400 seconds to download a middle-size image sized 500 MB.

3.1.3 Kubernetes

Kubernetes [35] is one of the most successful open-source systems for automating deployment, scaling, and management of containerized applications. A Kubernetes cluster consists of at least one master and multiple compute nodes. Fig. 2 shows the architecture of Kubernetes. The master includes a highly-available database named etcd, an API Server for exposing APIs, a Scheduler for scheduling deployments, and a Controller for managing the overall cluster. Each node is a workhorse of a Kubernetes cluster, consisting of many pods and a management component named kubelet. A pod is a collection of containers and serves as Kubernetes' core unit of management.

3.2 System Model

Fig. 3 illustrates an edge computing system comprising multiple user equipments (UEs), a set of heterogeneous edge nodes $E = \{e_1, e_2, \ldots, e_{|E|}\}$, a scheduler and a container

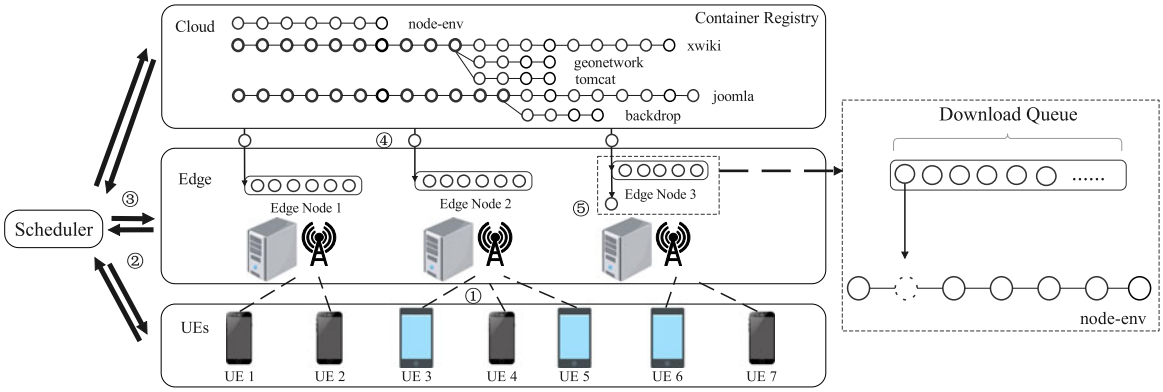


Fig. 3. Architecture of edge computing system. Each edge node is associated with a download queue, and each circle in the download queue represents a layer. An edge node downloads layers from the container registry according to the layer sequence in its download queue. As shown on the right, the layer at the left of the sequence will be downloaded first, constituting a complete image node-env with local layers. The scheduling process is: (1) Offloading tasks, (2) Collecting information, (3) Making scheduling decisions, (4) Downloading layers, and (5) Running containers.

registry. The scheduler collects information and then decides the container assignment and the layer download sequence. The container registry in the cloud is a repository for storing images. The container scheduling process is: (1) Multiple UEs offload multiple tasks. (2) The scheduler collects information of tasks and edge nodes. (3) Based on the collected information, the scheduler makes decisions of container assignment and layer sequencing. (4) With the decisions and other prior information, each edge node constructs a download queue and downloads layers according to the sequence in its download queue. (5) Each container starts to run when all layers belonging to it are ready.

Each task runs on a specific container. Thus, a set of containers denoted as $C = \{c_1, c_2, \ldots, c_{|C|}\}$ are also used to represent the set of tasks. The binary variable a_{jk} is denoted whether the container c_j is assigned to the edge node e_k. If c_j is assigned to e_k, then $a_{jk} = 1$, otherwise, $a_{jk} = 0$. Each container c_j should be assigned to one edge node

$$\sum_{e_k \in E} a_{jk} = 1, \quad \forall c_j \in C. \tag{1}$$

The set of unique layers that constitute all images of containers in C is denoted as $L = \{l_1, l_2, \ldots, l_{|L|}\}$. The size of layer l_i is defined as p_i. $r_{ij} \in \{0, 1\}$ is defined to indicate whether a layer l_i belongs to container c_j's image. If container c_j requires layer l_i, then r_{ij} is set to 1, otherwise, set to 0. r_{ij} is obtained before scheduling. Layers are shared by different images so that the edge node only needs to download a layer once. Binary variable $d_{ik} \in \{0, 1\}$ is defined as whether layer l_i will be downloaded on edge node e_k. The time when the edge node e_k finishes downloading the layer l_i is defined as layer l_i's ready time t^l_{ik}. The container c_j cannot start to run on the edge node e_k until all layers belonging to its image are downloaded, so its startup time t^c_j is larger than its layers' ready time on the edge node e_k

$$t^c_j \geq t^l_{ik} r_{ij} a_{jk}, \quad \forall c_j \in C, \forall l_i \in L, \forall e_k \in E. \tag{2}$$

Edge nodes are computational devices deployed at access points and connect with UEs through low-latency wireless communication. The storage, bandwidth and running container number limitation of an edge node e_k are defined as s_k, b_k and m_k, respectively. The total size of layers stored on an edge node e_k cannot exceed its storage limitation

$$\sum_{l_i \in L} d_{ik} p_i \leq s_k, \quad \forall e_k \in E. \tag{3}$$

A limited number of containers can run concurrently on an edge node

$$\sum_{c_j \in C} a_{jk} \leq m_k, \quad \forall e_k \in E. \tag{4}$$

Each edge node is associated with a download queue and downloads one layer at the head of the queue (on the left side) at each time. The layer sequence in the download queue should be determined. The download precedence of layers l_i, $l_{i'}$ on edge node e_k is defined as a binary variable $x^k_{ii'} \in \{0, 1\}$. If $x^k_{ii'}$ is set to 1, the layer l_i should be downloaded prior to layer $l_{i'}$, otherwise, $l_{i'}$ is downloaded prior to l_i. Especially, x^k_{ii} is equal to 1, making sure that every layer's ready time includes its own download time. Besides, the precedence between two layers obeys that

$$x^k_{ii'} + x^k_{i'i} = 1, \quad \forall l_i, l_{i'} \in L, i \neq i', \forall e_k \in E. \tag{5}$$

The precedence relation between layers is transitive

$$x^k_{ij} + x^k_{jl} + x^k_{li} \leq 2, \quad \forall l_i, l_j, l_l \in L, i \neq j \neq l. \tag{6}$$

The transitive constraint is briefly explained. For instance, three layers l_i, l_j, l_l are downloaded on an edge node e_k. If layer l_i is downloaded before layer l_j, i.e., $x^k_{ij} = 1$, and layer l_j is downloaded before layer l_l, i.e., $x^k_{jl} = 1$, then layer l_l cannot be downloaded before layer l_i, i.e., $x^k_{li} = 0$. Thus, the sum of x^k_{ij}, x^k_{jl} and x^k_{li} cannot be larger than 2. Considering both queuing time and download time, the ready time of the layer l_i on the edge node e_k can be calculated by:

$$t^l_{ik} = \sum_{l_j \in L} x^k_{ji} p_j / b_k, \quad \forall l_i \in L, \forall e_k \in E. \tag{7}$$

An edge node can download multiple layers at the same time in the real world (concurrent downloading), but the total startup time cannot be reduced since the downloading process is limited by the bandwidth. For any scheduling

result (i.e., each layer's ready time) of the concurrent downloading, a corresponding schedule of sequential downloading that is better than or equal to it can be constructed in polynomial time. First, the layers downloaded on an edge node are relabeled according to the ascending order of their ready times in concurrent downloading. After relabeling, layer l_1 has the earliest ready time, and layer l_n has the latest ready time. In sequential downloading, the layers are sequentially downloaded according to the labeling order. Thus, for layer l_i, its ready time t^l_{ik} is $\sum_1^i p_i/b_k$. In concurrent downloading, since the ready times of layers l_1 to l_{i-1} are earlier than layer l_i, the total size of downloaded layer data before the ready time of layer l_i is at least $\sum_1^i p_i$. There, layer l_i's ready time in concurrent downloading is $\sum_1^i p_i/b_k$ at least. If and only if the layers are downloaded sequentially according to the labeling order, the equality holds. Therefore, any layer's ready time and any container's startup time cannot be reduced by concurrent downloading.

In this work, the edge computing scenario is simplified. First, the task data transmission time is neglected for sufficient bandwidth between edge nodes and UEs. Second, we focus on optimizing container startup time, so the container runtime is assumed to be equal for all edge nodes. Third, the image installation time is more stable and shorter than the image download time. Thus, the image installation time can be taken as a constant and neglected for simplicity. Our objective is to optimize the sum of the startup time of all containers $\sum_{c_j \in C} t^c_j$.

3.3 Problem Formulation

The CALS problem is formulated as

$$P1: \min_{\{a_{jk}, x^k_{ii'}, d_{ik}, t^c_j, t^l_{ik}\}} \sum_{c_j \in C} t^c_j \tag{8}$$

$$s.t. \quad (1), (2), (3), (4), (5), (6), (7)$$

$$d_{ik} m_k \geq \sum_{c_j \in C} r_{ij} a_{jk}, \quad \forall l_i \in L, \forall e_k \in E. \tag{9}$$

The objective in Equation (8) is to find the optimal schedule $\Omega = [\Omega^a, \Omega^s]$ that minimizes the sum of all containers' startup times, where $\Omega^a = \{a_{jk} | c_j \in C, e_k \in E\}$ and $\Omega^s = \{x^k_{ii'} | l_i, l_{i'} \in L, e_k \in E\}$. d_{ik}, t^c_j and t^l_{ik} can be computed based on the schedule Ω. The constraints in (9) make sure that for each layer l_i, when any container c_j requiring it is assigned to an edge node e_k (i.e, $a_{jk} = 1$ and $\sum_{c_j \in C} r_{ij} a_{jk} > 0$), then the layer l_i should be downloaded to edge node e_k, i.e., $d_{ik} = 1$. Otherwise, $\sum_{c_j \in C} r_{ij} a_{jk} = 0$ and therefore d_{ik} is equal to 0 to minimize download size. Ω^a and Ω^s make the optimization problem be a Mixed Integer Quadratic Programming (MIQP) with high complexity. Next, the NP-hardness of the CALS problem is shown.

Theorem 1 *The CALS problem is NP-hard and cannot be approximated within any factor unless NP = P.*

The theorem is proved in Appendix A, available in the online supplemental material.

4 Layer-Aware Scheduling Algorithm

Due to the NP-hardness and inapproximability of CALS in general, we use the storage capacity and running container number constraints to generate edge node candidates and focus on the container assignment and layer scheduling problem. LASA is designed based on a special case of CALS (i.e., CALS-S, the layer sequencing problem on a single edge node with infinite storage capacity and unlimited running container number).

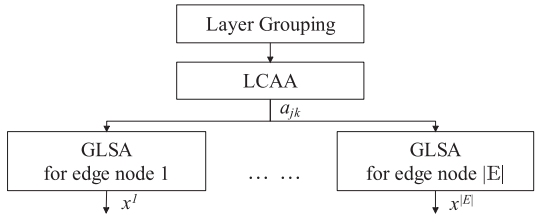


Fig. 4. The algorithm flow of LASA.

The algorithm flow of LASA is shown in Fig. 4. LASA has three steps. First, the common layers shared by the same set of containers are grouped. The optimal result of scheduling a group of layers as a whole is proved to be equal to scheduling individual layers. Thus, the same scheduling decisions are made for the entire group of layers. The problem scale is drastically reduced by layer grouping. Second, the Layer-aware Container Assignment Algorithm (LCAA) is designed to properly assign containers to edge nodes in sequence. At each time, one container-node pair is selected by considering both layer sharing among containers and the existing layer size of each edge node. After determining container assignment variables a_{jk}, the CALS problem is decomposed into independent sub-problems that optimize each edge node's layer download sequence. Finally, the Greedy Layer Sequencing Algorithm (GLSA) with an approximate ratio of 2 is proposed to determine the layer download sequence. In GLSA, each sub-problem is converted to a classic precedence-constrained single machine job scheduling problem, and the layers are divided into an ordered list of sets by Sidney Decomposition [36]. The layer order across sets is determined based on the ordered list, and the layer order within each set is determined greedily. The layer sequencing algorithm can efficiently produce layer sequence variables $x^k_{ij'}$ for each edge node e_k in parallel.

4.1 Layer Grouping

The total number of binary variables of a_{jk} and $x^k_{ii'}$ is $|E||C| + |E||L|^2$. Grouping layers can simplify the optimization problem before assigning containers and sequencing layers. Layer grouping is defined as follows:

Definition 1 *Any two layers l_i, $l_{i'}$ having the same relation with all containers*

$$r_{ij} = r_{i'j}, \quad \forall c_j \in C, \tag{10}$$

are added into the same group.

Fig. 5 is an example of layer grouping. Fifty-eight individual layers of six containers are converted into eight layer groups. The ten layers in the middle are simultaneously shared by three containers, so they are grouped. The twelve layers shared by joomla and backdrop are combined into one group. The seven layers on the top only belonging to node-env are also combined to a group.

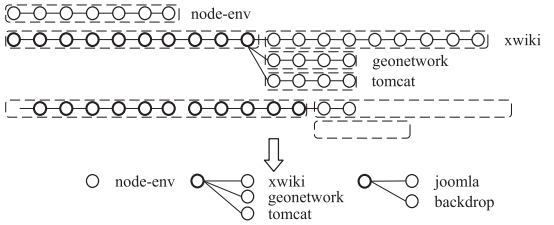


Fig. 5. Layer grouping example. In each dotted frame, layers that belonging to the same set containers are grouped.

The layer grouping is driven by the fact that layers shared by the same set of containers should be scheduled as a whole without affecting the optimal objective in (8).

Theorem 2 *The optimal objective in (8) of scheduling layer groups is equal to scheduling individual layers.*

Proof In Appendix B, available in the online supplemental material, the theorem is formally proved. □

The layer grouping has low computation complexity. The total number of r_{ij} is $|L||C|$ and layers are grouped by traversing the relation parameter r_{ij}. Consequently, the computation complexity is $O(|L||C|)$.

In container assignment and layer sequencing, layers of the same group are scheduled together, so the layer is used to represent the layer group in the following for simplicity.

4.2 Container Assignment

In this step, LCAA is designed to determine container assignment variables $a_{jk} \in \Omega^a$.

The LCAA considers two critical factors: (1) Layer sharing among containers. If containers are assigned to edge nodes without considering layer sharing, different edge nodes will download duplicated layers, which costs extra bandwidth. Downloading redundant layers also increases the queuing latency. Therefore, the total startup time of all containers increases. (2) The existing layer size of each edge node. Assigning containers to edge nodes barely according to layer sharing between the containers and edge nodes leads to unbalanced workloads among edge nodes. Though redundant layer downloading is eliminated, it will make layers be downloaded on a few edge nodes and leave other nodes underutilized, leading to high queuing latency.

To trade off the layer sharing and the existing layer size among edge nodes, a heuristic algorithm that greedily selects one container-node pair at each time is designed. The container assignment algorithm is shown in Algorithm 1.

In Algorithm 1, the inputs are the bandwidth set $\{b_k|e_k \in E\}$, storage set $\{s_k|e_k \in E\}$, running container number limitation $\{m_k|e_k \in E\}$, layer size $\{p_i|l_i \in L\}$ and relation information $\{r_{ij}|l_i \in L, c_j \in C\}$. The outputs are containers' assignment decisions a_{jk}. In lines 1 - 2, the existing layer set L^k of each edge node, the running container number of each edge node N_k, the remaining container set C^r, and the assignment decision a_{jk} of each container c_j on each edge node e_k are initialized by an empty set, 0, C and 0, respectively. Lines 3 - 18 describe loops of assigning containers to edge nodes. Within each loop, a container-node pair is selected by calculating the $score$ in line 10, which consists of both layer sharing and existing layer size. The term $\sum_{l_i \in \Delta_L} p_i$ in line 10 represents the layer size increment after the assignment. Less layer size increment means more layer sharing and fewer redundant layers. The term $\sum_{l_i \in L^k} p_i$ in line 10 represents the size of layers already existing on the edge node. Assigning containers to the edge nodes with the least existing layer size can alleviate unbalanced workloads. $\alpha \in [0, 1]$ is a hyperparameter used to trade off these two factors, and the result is normalized by each edge node's bandwidth b_k. In line 11, only edge nodes that do not exceed storage capacity and running container number limitation are allowed to be candidates. In line 12, the container-node pair with the least $score$ is selected. The assignment decision $a_{j_a k_a}$, the number of edge node's containers N_{k_a}, the container set C^r and the edge node's layer set L^{k_a} are updated in lines 16 - 17. After running $|C|$ loops, the assignment variable a_{jk} for each container c_j on each edge node e_k is produced.

Algorithm 1. LCAA

Input: $\{b_k|e_k \in E\}$, $\{s_k|e_k \in E\}$, $\{m_k|e_k \in E\}$, $\{p_i|l_i \in L\}$, $\{r_{ij}|l_i \in L, c_j \in C\}$
Output: a_{jk}

1: Initialize $L^k \leftarrow \{\}, N_k \leftarrow 0, a_{jk} \leftarrow 0,\ \forall c_j \in C, \forall e_k \in E$
2: Initialize $C^r \leftarrow C$
3: **while** $C^r \neq \emptyset$ **do**
4: $\quad j_a \leftarrow -1, k_a \leftarrow -1$
5: $\quad score_a \leftarrow \frac{2\sum_{l_i \in L} p_i}{\min(b_k|e_k \in E)}$
6: $\quad$ **for** $c_j \in C^r$ **do**
7: $\quad\quad$ **for** $e_k \in E$ **do**
8: $\quad\quad\quad L' \leftarrow L^k \cup \{l_i|l_i \in L, r_{ij} = 1\}$.
9: $\quad\quad\quad \Delta_L \leftarrow L' \setminus L^k$
10: $\quad\quad\quad score \leftarrow \frac{(1-\alpha)\sum_{l_i \in \Delta_L} p_i + \alpha \sum_{l_i \in L^k} p_i}{b_k}$
11: $\quad\quad\quad$ **if** $score < score_a$ and $\sum_{l_i \in L'} p_i \leq s_k$ and $N_k + 1 \leq m_k$ **then**
12: $\quad\quad\quad\quad j_a \leftarrow j, score_a \leftarrow score, k_a \leftarrow k$
13: $\quad\quad\quad$ **end if**
14: $\quad\quad$ **end for**
15: $\quad$ **end for**
16: $\quad a_{j_a k_a} \leftarrow 1, N_{k_a} \leftarrow N_{k_a} + 1, C^r \leftarrow C^r \setminus \{c_{j_a}\}$
17: $\quad L^{k_a} \leftarrow L^{k_a} \cup \{l_i|l_i \in L, r_{ij_a} = 1\}$
18: **end while**

4.3 Layer Sequencing

Binary variables d_{ik} are determined by assignment variables a_{jk} based on Equation (9). P1 is decomposed into $|E|$ independent sub-problems, which can be solved in parallel. The sub-problem for the edge node e_k is defined as follows:

$$P4: \min_{\{x^k_{ii'}, t^c_j, t^l_{ik}\}} \sum_{c_j \in C_k} t^c_j \tag{11}$$

$$s.t. \quad x^k_{ii'} \in \{0, 1\}, \quad \forall l_i, l_{i'} \in L_k, \tag{12}$$

$$x^k_{ii} = 1, \quad \forall l_i \in L_k, \tag{13}$$

$$x^k_{ii'} + x^k_{i'i} = 1, \quad \forall l_i, l_{i'} \in L_k, i \neq i', \tag{14}$$

$$x^k_{ij} + x^k_{jl} + x^k_{li} \leq 2, \quad \forall l_i, l_j, l_l \in L_k, i \neq j \neq l, \tag{15}$$

$$t^l_{ik} = \sum_{l_j \in L_k} x^k_{ji} p_j / b_k, \ \forall l_i \in L_k, \tag{16}$$

$$t^c_j \geq t^l_{ik} r_{ij}, \quad \forall l_i \in L_k, \forall c_j \in C_k, \tag{17}$$

where $C_k = \{c_j | c_j \in C, a_{jk} = 1\}$ and $L_k = \{l_i | l_i \in L, d_{ik} = 1\}$ are the container set and the layer set of the edge node e_k, respectively. Though with a single edge node, P4 is generally NP-hard and difficult to get an exact solution, as proved in Appendix A, available in the online supplemental material.

The sub-problem in (11) can be converted to a special case of $1|prec|\sum w_i c_i$. $1|prec|\sum w_i c_i$ is the problem of sequencing precedence-constrained jobs on a single machine to minimize the total weighted completion time. Sidney Decomposition [36] is an efficient algorithm to solve $1|prec|\sum w_i c_i$ with an approximate ratio of 2. Therefore, inspired by Sidney Decomposition [36], an efficient layer sequencing algorithm is designed.

First, how to convert a sub-problem to an instance of $1|prec|\sum w_i c_i$ is introduced. For a sub-problem k, both containers and layers are regarded as jobs, i.e., $J_k = L_k \cup C_k$. The difference between $1|prec|\sum w_i c_i$ and $1|prec|\sum c_i$ is that the former problem sets a weight for every job. In the converted problem, each layer l_i and each container c_j are regarded as the job j_i and the job j_j, respectively. The weights and sizes of jobs are defined as

$$w_i = \begin{cases} 0 \text{ if } j_i \in L_k \\ 1 \text{ if } j_i \in C_k \end{cases}, \quad p_i = \begin{cases} p_i \text{ if } j_i \in L_k \\ 0 \text{ if } j_i \in C_k \end{cases}.$$

The converted optimization problem of $1|prec|\sum w_i c_i$ is defined as

$$P5: \min_{\{x^k_{ii'}, t^j_j\}} \sum_{j_j \in J_k} w_j t^j_j = \min_{x^k_{ii'}} \sum_{j_j \in C_k} t^j_j \tag{18}$$

$$s.t. \quad x^k_{ii'} \in \{0,1\}, \quad \forall j_i, j_{i'} \in J_k, \tag{19}$$

$$x^k_{ii} = 1, \quad \forall j_i \in J_k, \tag{20}$$

$$x^k_{ii'} + x^k_{i'i} = 1, \quad \forall j_i, j_{i'} \in J_k, i \neq i', \tag{21}$$

$$x^k_{ij} + x^k_{jl} + x^k_{li} \leq 2, \quad \forall j_i, j_j, j_l \in J_k, i \neq j \neq l, \tag{22}$$

$$t^j_i = \sum_{j_j \in J_k} x^k_{ji} p_j / b_k = \sum_{j_j \in L_k} x^k_{ji} p_j / b_k, \ \forall j_i \in J_k, \tag{23}$$

$$x^k_{ij} = 1, \quad \forall j_i \in L_k, \forall j_j \in C_k, r_{ij} = 1, \tag{24}$$

where t^j_j is j_j's completion time and also equal to c_j's startup time or l_j's ready time. Constraints in (24) is equal to constraints in (17) since each container c_j is regarded as a job sized 0. Thus, the converted problem's objective and constraints are equal to the original problem.

Then, we give a brief introduction to Sidney Decomposition [36], which is a 2-approximation algorithm for general instances of $1|prec|\sum w_i c_i$. It decomposes jobs into a list of disjoint sets $Y_k = [S_1, S_2, \ldots | \bigcup S_o = J_k]$ ordered by the ratio $\rho(S_o) = \frac{\sum_{j_j \in S_o} w_j}{\sum_{j_j \in S_o} p_j}$. Each set comprises multiple jobs. Sidney proves that for an optimal sequence, jobs in different sets must run following the set order. In Sidney's algorithm, jobs in the same set can run in an arbitrary sequence under precedence constraints. More details can refer to [36]. In this paper, Sidney Decomposition is implemented by the pseudoflow algorithm for linear parametric minimum cut problem [37]. Sidney Decomposition is applied to get a list of disjoint sets Y_k, and then the layer sequence within each set is determined greedily.

Algorithm 2. GLSA

Input: $\{p_i | l_i \in L_k\}$, $\{r_{ij} | l_i \in L_k, c_j \in C_k\}$, b_k, C_k
Output: $x^k_{ii'}$, t^l_{ik}, t^c_j

1: $Y_k = SidneyDecomposition(\{p_i | l_i \in L_k\}, \{r_{ij} | l_i \in L_k, c_j \in C_k\})$
2: Initialize $L_{sequenced} \leftarrow \{\}$
3: **for** $S_o \in Y_k$ **do**
4: $S^c_o \leftarrow \{\}$
5: **for** $j_j \in S_o$ **do**
6: **if** $c_j \in C_k$ **then**
7: $S^c_o \leftarrow S^c_o \cup \{c_j\}$
8: **end if**
9: **end for**
10: **while** $S^c_o \neq \emptyset$ **do**
11: **for** $c_j \in S^c_o$ **do**
12: $L^j \leftarrow \{l_i | l_i \in L_k, r_{ij} = 1\} \setminus L_{sequenced}$
13: $p^j \leftarrow \sum_{l_i \in L^j} p_i$
14: **end for**
15: $j \leftarrow \text{argmin}_j p^j$
16: $S^c_o \leftarrow S^c_o \setminus \{c_j\}$
17: **for** $l_i \in L^j$ **do**
18: $x^k_{ii} \leftarrow 1$
19: **for** $l_{i'} \in L_{sequenced}$ **do**
20: $x^k_{i'i}, x^k_{ii'} \leftarrow 1, 0$
21: **end for**
22: $L_{sequenced} \leftarrow L_{sequenced} \cup \{l_i\}$
23: **end for**
24: $t^c_j \leftarrow \sum_{l_i \in L_{sequenced}} p_i / b_k$
25: **end while**
26: **end for**

As shown in Algorithm 2, GLSA inspired by Sidney Decomposition [36] is proposed to determine the layer sequence on each edge node e_k. The inputs of Algorithm 2 are the layer size set $\{p_i | l_i \in L_k\}$, the relation information $\{r_{ij} | l_i \in L, c_j \in C_k\}$, the bandwidth b_k of e_k and the entire container set C_k. The outputs are the layer sequence variables $x^k_{ii'}$, layer ready time t^l_{ik}, and container startup time t^c_j. In line 1, the list of sets Y_k is produced by Sidney Decomposition function. In line 2, the sequenced layer set $L_{sequenced}$ is initialized by an empty set. In lines 3 - 26, layers within each disjoint set are sorted. Since the weight w_i of every layer is equal to 0, a layer l_i in a set S_o must belong to one of container c_j in the same set S_o. Otherwise, the layer l_i should be removed from the set S_o since the redundant layer l_i with weighted 0 lower the $\rho(S_o)$. Thus, layers in a set are sequenced in the unit of containers. The container set S^c_o is extracted from each set S_o in lines 4 - 9. In lines 10 - 25, containers in S^c_o are sorted by their remaining layer size. In lines 11 - 14, each container's remaining layer size p^j in the container set S^c_o is calculated, and the container c_j with the least remaining layer size is selected in line 15. In lines 17 - 23, the layer sequence variables related to the remaining layers of the selected container c_j are determined, and the remaining layers of the selected container c_j are added into the sequenced layer set. In this algorithm, the adding sequence of the selected container's remaining layers is arbitrary. After selection, the startup time t^c_j for each container c_j is calculated in line 24.

Lemma 3 *The proposed layer sequencing algorithm has an approximation ratio of 2.*

Proof According to [36], any algorithm for $1|prec|\sum w_i c_i$ consistent with Sidney Decomposition has an approximation ratio of 2. GLSA only changes the layer order within each disjoint set without changing the layer order across different sets. Besides, the sub-problem is equal to the converted problem. Thus, the proposed layer sequencing algorithm has an approximation ratio of 2. □

4.4 Complexity Analysis

The computation complexity of LASA is analyzed. First, as mentioned in Section 4.1, the computation complexity of grouping layers is $O(|L||C|)$. Then, container assignment variables are computed by Algorithm 1 in $O(|C|^2|E|)$ time. Then, the CALS problem is decomposed into independent sub-problems. For sub-problems, the computation complexity of problem conversion is $O(|L_k| + |C_k|)$. The computation complexity of implementing Sidney Decomposition is $O(mn\log n)$ [37], where $n = |L_k| + |C_k|$ and $m = \sum_{l_i \in L_k, c_j \in C_k} r_{ij}$. The computation complexity of the layer sequencing within each set in lines 2 - 26 of Algorithm 2 is $O(|C_k|^2|L_k|)$. Therefore, the total computation complexity of LASA is $O(|C|^2|E| + (\sum_{l_i \in L, c_j \in C} r_{ij})(|L| + |C|)\log(|L| + |C|) + |C|^2|L|)$. Experiments are conducted to prove the LASA is quite efficient with a realistic trace. The result is depicted as Figs. 18 and 19 in Section 5.3.

4.5 Approximation Ratio of LASA

Theorem 4 *The LASA is a polynomial-time $2|E|$-approximation algorithm for CALS of identical edge nodes with infinite storage capacity and unlimited running container number, where $|E|$ is the number of edge nodes.*

In Appendix C, available in the online supplemental material, the theorem is formally proved.

5 Evaluation

The experiments are conducted in a simulation environment. The simulation environment and LASA are implemented in Python 3.6 on a desktop with an Intel Core i7-10750H 2.60GHz CPU and 16GB RAM. In the experiments, a real edge computing scenario with multiple edge nodes is considered. By default, the bandwidth is set to 10 Mbps, the number of edge nodes is set to 15, the running container number limitation is set to 50, the storage capacity limit is set to 20 GB, the total number of containers is set to 200, and α is set to 0.5.

Container data from [13] is used. They collected the latest versions of the 5K most popular images from DockerHub [19]. In the simulation, 155 most frequently used images are selected from their dataset, and the total size of 155 images is 60 GB. There are 810 unique layers in total, and the total size of unique layers is 30 GB. For each experiment, the container set is randomly chosen from the 155 images following the uniform and Zipf [38] distributions. The Zipf distribution fits to model file popularity [39], task request popularity [13], and user image request distribution [40]. In this paper, the Zipf distribution is used to model the cases that the container request distribution is skewed. The Zipf distribution is also applied in one of the baselines, Layer-match Scheduling [13], which generates more convincing comparison results. The shape factor of Zipf is set to 1.1 by default. Each experiment is repeated ten times.

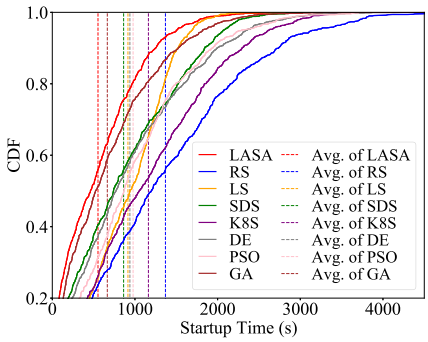
LASA is compared with seven baselines: (1) Random Scheduling (RS): Randomly select a container-node pair at each time and sequence layers according to the assignment order. (2): Layer-match Scheduling [13] (LS): For each container, select an edge node with the most amount of its image layers stored locally and sequence layers according to the assignment order. (3): Sidney Decomposition-based Scheduling (SDS): First sequence containers by Sidney Decomposition, then continuously assign containers to one node until achieving threshold, and sequence layers by GLSA. (4): Kubernetes Scheduling (K8S): Kubernetes default scheduling policy schedules containers to edge nodes with the required images stored locally, otherwise, to the edge node with the least total download size. (5) Differential Evolution (DE) [41]: In DE, the chromosome combines a container-node affinity vector and a container priority vector. A chromosome is interpreted by first sorting containers in the descending order of container priority and then selecting the edge node with the highest container-node affinity for each container. (6) Particle Swarm Optimization (PSO) [42]: The definition of particle and fitness is as same as chromosome and interpretation in DE. (7) Genetic Algorithm (GA) [43]: The definition of chromosome and interpretation is as same as DE. The main difference is that GA focuses on the crossover while DE focuses on the mutation. The population size and iteration number of three meta-heuristic baselines are 50 and 200. The mutation probability of DE and GA is 0.001. In PSO, inertia weight, cognitive parameter, and social parameter are 0.7, 0.5, and 0.5.

5.1 Comparison With Baselines

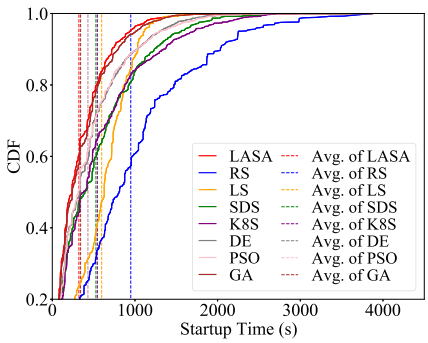
In this subsection, extensive experiments are conducted to compare the performance of LASA against the baselines.

The cumulative distribution function (CDF) of the container startup time of LASA and baselines is shown in Fig. 6. Fig. 6a shows the overall performance of different algorithms when containers follow the uniform distribution. Compared with the baselines, the CDF curve of LASA is always closer to the left, which means that the startup time of LASA is consistently shorter. The dotted line represents the average container startup time. Compared with RS, LS, SDS, K8S, DE, PSO, and GA, LASA reduces the average startup time by 60%, 40%, 36%, 52%, 41%, 43%, and 17%, respectively. Similarly, Fig. 6b shows that compared with the baselines, the average startup time of LASA is reduced by 6% to 66% under the Zipf distribution. The container startup time of the Zipf distribution is shorter than the uniform distribution since more containers of the same type share a few images under the Zipf distribution and thereby unique layers are fewer. The performance of K8S is much better under the Zipf distribution, for it tends to assign containers of the same type to one edge node.

The performance of LASA is compared against meta-heuristic baselines with different iteration numbers. The results are shown in Fig. 7. In Fig. 7a, the best schedule of each meta-heuristic algorithm until each iteration is recorded. PSO quickly falls into the local minimum in a few

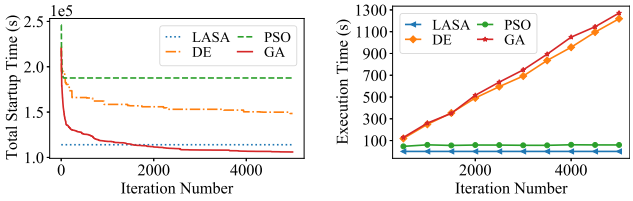


(a) Uniform distribution



(b) Zipf distribution

Fig. 6. CDF of container startup time for LASA and baselines under uniform and Zipf distributions.

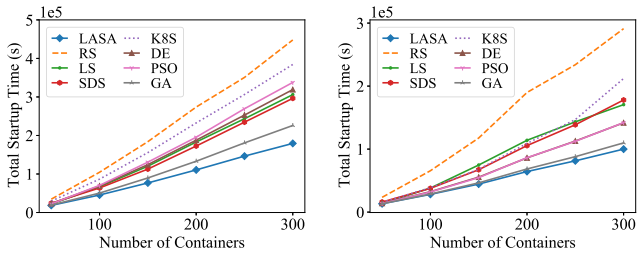


(a) Total startup time (b) Execution time

Fig. 7. Results for LASA and meta-heuristic baselines with different number of iterations under uniform distribution.

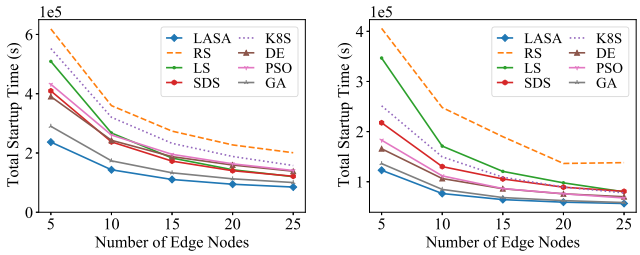
iterations, so the execution is terminated early. The total startup times of GA and DE continue to decrease as the iteration number increases. Since the elements of the container-node affinity vector and the container priority vector are ordered to generate the final schedule, the crossover operation is more efficient than the mutation. Therefore, the convergence speed of GA that focuses on the crossover is faster than DE. The total startup time of GA is lower than LASA after 1400 iterations and 7% lower than LASA in 5000 iterations. To compare the execution time of different algorithms fairly, multiprocessing and multithreading are not applied in this experiment. Compared with LASA that generates a schedule in 0.33 seconds, the meta-heuristic baselines require up to thousands of seconds to find a satisfactory schedule.

Then, the total startup times of LASA and baselines are evaluated with identical edge nodes. Fig. 8 shows the evaluation results. It suggests that with sufficient resources, the total startup time of LASA has an approximately linear relation with the number of containers. As the container number



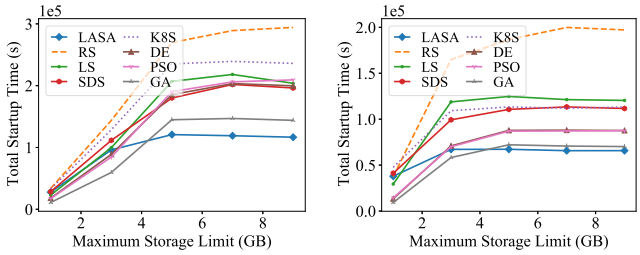
(a) Uniform distribution (b) Zipf distribution

Fig. 8. Total startup time comparison with different container numbers.



(a) Uniform distribution (b) Zipf distribution

Fig. 9. Total startup time comparison with different edge node numbers.

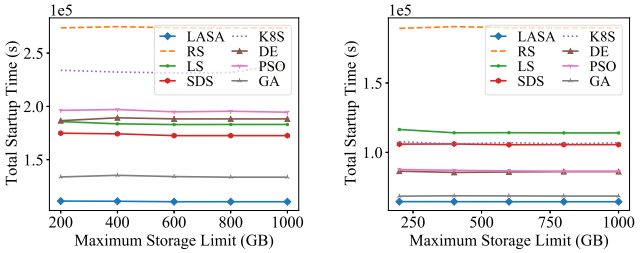


(a) Uniform distribution (b) Zipf distribution

Fig. 10. Total startup time comparison with different maximum storage capacity.

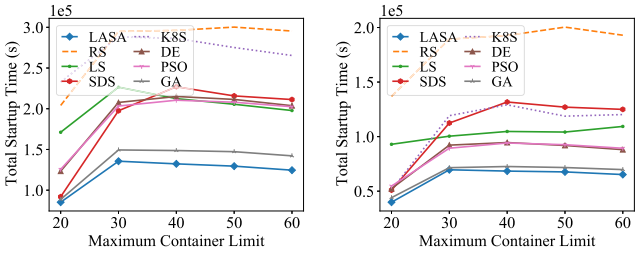
increases, LASA consistently achieves the best performance. The improvement of LASA is even more as the number of containers increases since more containers indicate a longer queuing time to be optimized. Fig. 9 shows the evaluation results of different edge node numbers. As the edge node number increases, the average size of layers downloaded on each edge node decreases, and the queuing time and the total startup time decrease. RS achieves the worst results since it randomly schedules containers.

Figs. 10, 12 and 13 show the performance comparison under heterogeneous storage capacity, running container number limitation and bandwidth. As shown in Fig. 10, each edge node's storage is randomly distributed between 0 and the maximum storage, and the maximum storage capacity of edge nodes is set from 1 to 9 GB. Fig. 10 shows that when the storage is insufficient, e.g., the maximum storage capacity is 1 GB, the result of LASA is worse than PSO and GA. In this case, LASA downloads layers of 164 containers, but PSO and GA only download layers of 82 and 62 containers, respectively. This result shows that PSO and GA search for schedules that reduce the total startup time by downloading fewer larger containers. As the storage capacity increases, the



(a) Uniform distribution (b) Zipf distribution

Fig. 11. Total startup time comparison with scaled maximum storage capacity.



(a) Uniform distribution (b) Zipf distribution

Fig. 12. Total startup time comparison between LASA and baselines with different maximum running container number limitation.

performance of LASA is obviously better than other algorithms. To further compare different algorithms in common cases of sufficient storage capacity on edge nodes, the maximum storage capacity is scaled to 1000 GB. It can be observed in Fig. 11 that all algorithms' performance is stable as the maximum storage capacity increases and LASA still outperforms other baselines.

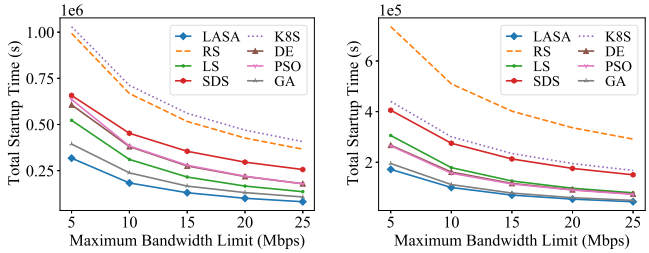
In Fig. 12, the running container number limitation is randomly distributed between 0 and the maximum running container number limit. When the running container number limit is the bottleneck (i.e., less than 30), as the running container number increases, more containers can be assigned, and thus the total startup time increases. When edge nodes can hold all containers, LASA's performance becomes stable and better than the baselines. In Fig. 13, each edge node's bandwidth is randomly distributed between 0 and the maximum bandwidth. With the increase of the maximum bandwidth, the total startup time of all algorithms is reduced, and LASA still outperforms the baselines.

5.2 Effect of LCAA and GLSA

In this subsection, the effect of LCAA and GLSA is analyzed separately.

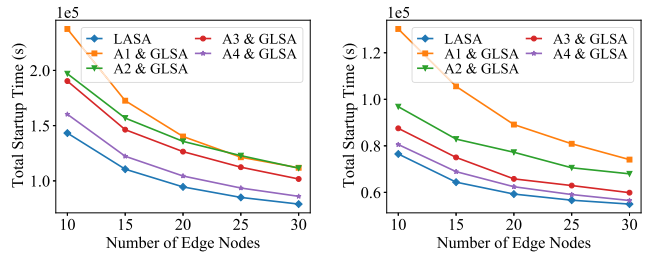
First, the layer sequencing algorithm is set to GLSA, and LCAA is compared with four baselines: (**A1**) Continuously assign containers according to the order of Sidney Decomposition to one edge node until achieving the threshold (i.e., the capacity constraints). (**A2**) Randomly select a container-node pair at each time. (**A3**) Assign containers to each edge node fairly. (**A4**) Assign the container to one edge node with the least estimated startup time (i.e., the image is placed at the end of the download queue by default).

Fig. 14 shows the total startup time of different container assignment algorithms. Compared with all baselines, LCAA consistently reduced the total startup time as the edge node



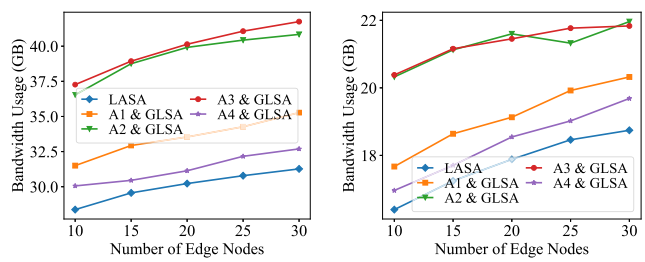
(a) Uniform distribution (b) Zipf distribution

Fig. 13. Total startup time comparison between LASA and baselines with different maximum bandwidth capacity.



(a) Uniform distribution (b) Zipf distribution

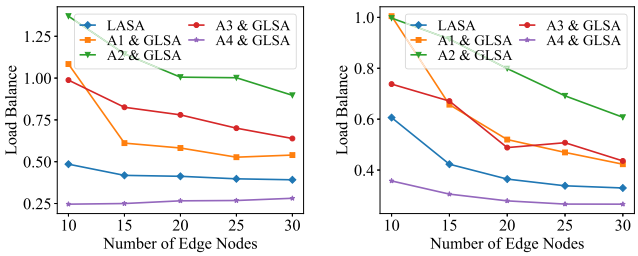
Fig. 14. Total startup time comparison between LCAA and four baselines with different numbers of edge nodes.



(a) Uniform distribution (b) Zipf distribution

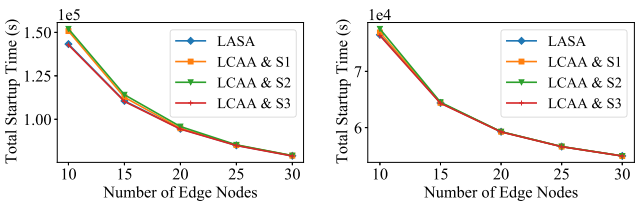
Fig. 15. Bandwidth usage comparison between LCAA and four baselines with different numbers of edge nodes.

number increased. Moreover, the bandwidth usage and the load balance of different container assignment algorithms are further analyzed. Fig. 15 shows the bandwidth usage of different container assignment algorithms. Less bandwidth usage means less redundant downloading. The total downloading size, $\sum_{e_k \in E} \sum_{l_i \in L} d_{ik} p_i$, is used to represent bandwidth usage. As the edge node number increases, the container assignment is more scattered, which incurs less layer sharing and more redundant downloading. Compared with the best baseline, **A1**, the bandwidth usage of LCAA is reduced by 11% and 8% under the uniform and Zipf distribution. Fig. 16 shows the load balance of different container assignment algorithms. Unbalanced workloads make more layers downloaded on a few edge nodes, which leads to longer queuing times. The standard deviation of edge nodes' download overhead, $\sigma = \sqrt{\frac{1}{|E|}\sum_{e_k \in E}(x_k - \bar{x})^2}$, where $x_k = \sum_{l_i \in L} d_{ik} p_i$, is used to represent the degree of load balance. The smaller the standard deviation, the better the load balance. **A1**, **A2**, and **A3** rely on the random edge node permutation or random edge nodes selections, so their results are relatively unstable. The standard deviation of LCAA is lower



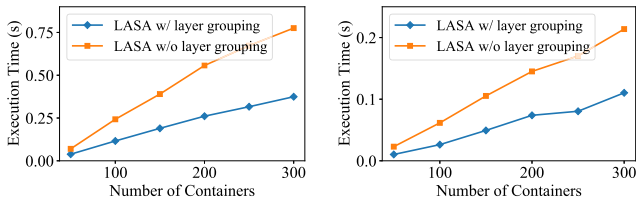
(a) Uniform distribution (b) Zipf distribution

Fig. 16. Load balance comparison between LCAA and four baselines with different numbers of edge nodes.



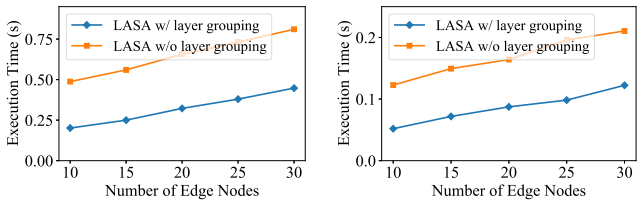
(a) Uniform distribution (b) Zipf distribution

Fig. 17. Total startup time comparison between LCAA and three baselines with different numbers of edge nodes.



(a) Uniform distribution (b) Zipf distribution

Fig. 18. The impact of layer grouping on execution time with different numbers of containers.



(a) Uniform distribution (b) Zipf distribution

Fig. 19. The impact of layer grouping on execution time with different numbers of edge nodes.

than **A1**, **A2** and **A3**. The standard deviation of **A4** is lower than LCAA since **A4** tends to schedule the container to the edge node with the least download size at each time, which leads to more balanced workloads. These results show that LCAA achieves a better tradeoff between bandwidth usage and load balance than other baselines.

In general, for random assignment **A2**, the distribution of containers is more dispersed with more edge nodes, which incurs more downloading. However, in Fig. 15b, the downloading size of **A2** with 25 edge nodes is less than with 20 edge nodes. To find out the reason, we fix the edge computing network and the container dataset, and use different random seeds to repeat the experiment with 25 edge nodes. It can be observed in Fig. 20 that the performance of three baselines is largely influenced by the random seed since they randomly select edge nodes (**A1**), containers (**A4**), or container-node pairs (**A2**). The performance of **A2** is unstable with different random seeds, so the total downloading size may be less with more edge nodes in some cases. More experiment results of different Zipf factors are in Appendix D, available in the online supplemental material.

Then, the container assignment algorithm is set to LCAA, and GLSA is compared with three baselines: (**S1**) Sequence layers according to the container assignment order. (**S2**) Sequence layers by container size in descending order. (**S3**) The optimal sequence obtained by using IBM CPLEX optimizer [44]. Fig. 17 shows the startup time of different layer sequencing algorithms when containers follow the uniform and Zipf distribution. Compared with **S1**, **S2**, the startup time of GLSA is reduced by up to 4%, 6% under uniform distribution and 0.5%, 1.3% under the Zipf distribution. Since the Zipf distribution is long-tailed, most containers share a few images, and unique layers are fewer. Thus, the improvement of GLSA under the uniform distribution is more than the Zipf distribution. The gap between the optimal result and GLSA on startup time is only 0.3% and 0.2% under the uniform and Zipf distribution, respectively. The reasons why the improvement of layer sequencing is less than the container assignment are as follows: (1) GLSA offers a proper container assignment order to the **S1**. (2) The amounts of shared layers on each edge node decrease as the edge node number increases.

5.3 Execution Time

The impact of layer grouping on the algorithm execution time is evaluated and the results are shown in Figs. 18 and 19. In Fig. 18, as the number of containers increases from 50 to 300, layer grouping reduces execution time by around 50% than without layer grouping. Additionally, the comparison of the layer number and the group number shown in Table 1 illustrates the reason why layer grouping reduces the execution time. After grouping, the group number is reduced by up to 70% and 73% under uniform and Zipf distributions. Therefore, the problem scale is significantly reduced, and the algorithm execution time is reduced. In Fig. 19, as the edge node number increases from 10 to 30, layer grouping reduces execution time by up to 59%.

5.4 Impact of the Hyperparameter

In this subsection, the impact of the hyperparameter α on the total startup time, bandwidth usage, and load balance is evaluated. The hyperparameter α trades off the layer sharing and the existing layer size. When α is 1, LASA assigns containers to the edge node with the smallest existing layer size, which represents less queuing time to some extent. When α is 0, LASA chooses the container-node pair with the smallest downloading increment. When α takes the value between 0 and 1, LASA trades off existing layer size and layer sharing. Fig. 21 shows the impact of α under the default setting. As α increases, the total download size gradually increases, the standard deviation of the download size gradually decreases (i.e., unbalanced workloads), and the

TABLE 1
Problem Scale Reduction by Layer Grouping

Container Number		50	100	150	200	250	300
Uniform	w/ grouping	94	188	248	296	328	349
	w/o grouping	307	552	709	827	911	965
Zipf	w/ grouping	44	78	111	141	152	171
	w/o grouping	166	275	362	434	462	516

total startup time first decreases and then increases. According to the evaluation results, α is heuristically set to 0.5 to trade off bandwidth usage and load balance properly.

6 Discussion

6.1 User Mobility

In edge computing, user mobility management is a big issue [45]. Mobile users can offload different tasks at any place and any time, so the total number of images can be huge. It is impossible to download and store all images on each resource-limited edge node in advance.

To address this issue, user mobility prediction and container caching can be applied. User mobility prediction estimates the distribution of each user's future positions [46]. With this information, the corresponding candidate edge servers can be selected to download and cache the required images in advance. Specifically, the candidate edge server selection can be integrated with line 11 of Algorithm 1.

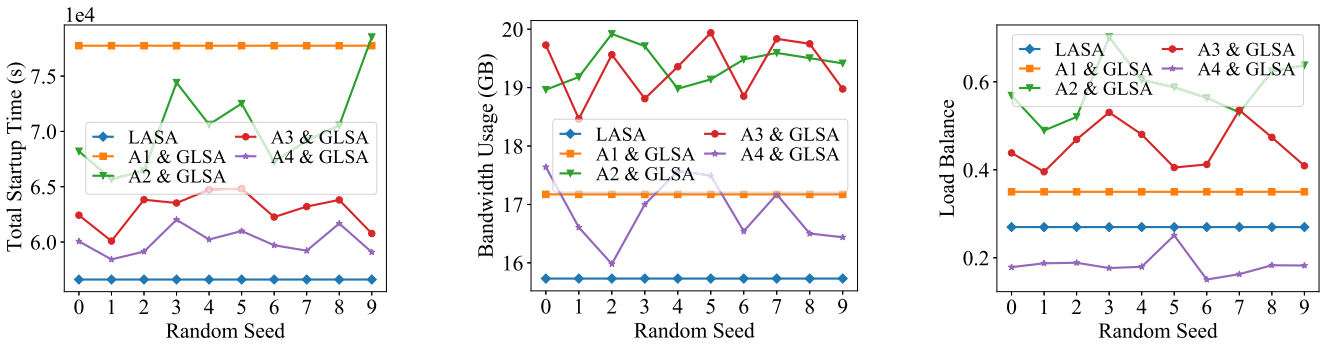
6.2 Online Container Scheduling

Online container scheduling is a practical scenario where containers can be scheduled once offloaded by users without waiting for joint scheduling. More issues should be considered in the online scheduling problem. First, container assignment decisions should be made to reduce the accumulated task latency in the long term. The correlations of decisions at different time points should be further studied. Second, some image layers can be selectively evicted when the storage of edge nodes is in short. The layer eviction policy is also very hard to design for the layer sharing feature and the varying distribution of container requests. Third, the task execution time should also be considered. After finishing the task execution, the occupied computation and storage resources can be released. The problem objective will be turned to minimizing the total task latency.

The online scheduling problem can be modeled as a Markov decision process. A centralized reinforcement learning agent is trained to replace the LCAA. The agent makes container assignment decisions to optimize the accumulated task latency in the long term. GLSA can still be applied on each edge server to resequence the layers triggered by a new container assignment. For layer eviction, the layer size, invoke frequency, and edge node capacity are jointly considered to design a customized layer scoring algorithm.

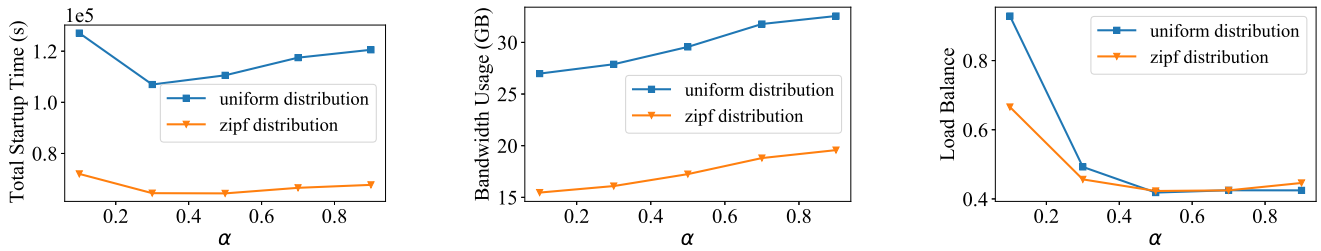
6.3 Implementation in Kubernetes

In this paper, we focus on modeling the system, formulating the CALS problem, and designing LASA. Extensive simulations are conducted on real-world data collected from DockerHub [19]. The prototype implementation in Kubernetes is left as future work. Besides, some new issues like layer eviction and online scheduling also appear during our ongoing implementation in Kubernetes, which includes container assignment and layer sequencing. In container assignment, task requests that arrived in Kubernetes are scheduled by



(a) Total startup time (b) Bandwidth usage (c) Load balance

Fig. 20. Impact of the random seed.



(a) Total startup time (b) Bandwidth usage (c) Load balance

Fig. 21. Impact of the hyperparameter α.

Kubernetes Scheduler. Kubernetes Scheduler selects a suitable edge node for the task in a 2-step operation: First, it finds the set of edge nodes where it is feasible to schedule the task as in line 11 of Algorithm 1. Second, the scheduler collects the information of the remaining edge nodes, computes a score for each edge node based on Algorithm 1, and selects the edge nodes. In layer sequencing, according to the result of Algorithm 2, the layers of one image are actually downloaded continuously on each edge node. Images can be downloaded according to the download queue of layers by using the docker pull command.

7 Conclusion

In this paper, we jointly schedule multiple containers to reduce the total startup time by considering the layer sharing feature. LASA is designed to make scheduling decisions efficiently. First, layers shared by the same set of containers are grouped to reduce the problem scale of CALS. Second, considering both the layer sharing and existing layer size on edge nodes, a heuristic algorithm is proposed to schedule containers to appropriate edge nodes. The CALS problem is decomposed into multiple independent sub-problems. Finally, a layer sequencing algorithm with an approximate ratio of 2 is designed to determine the layer download sequencing on each edge node. We use a real-world trace to conduct extensive experiments. The experimental results prove the effectiveness of the LASA algorithm, which reduces the total startup time by 40% to 60%.

References

[1] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Fourth Quarter 2009.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st ed. MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[4] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Commun. Surv. Tuts.*, vol. 19, no. 3, pp. 1657–1681, Third Quarter 2017.

[5] K. Zhang, S. Leng, Y. He, S. Maharjan, and Y. Zhang, "Cooperative content caching in 5G networks with mobile edge computing," *IEEE Wireless Commun.*, vol. 25, no. 3, pp. 80–87, Jun. 2018.

[6] J. Zhang, X. Zhou, T. Ge, X. Wang, and T. Hwang, "Joint task scheduling and containerizing for efficient edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 2086–2100, Aug. 2021.

[7] Q.-V. Pham *et al.*, "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art," *IEEE Access*, vol. 8, pp. 116 974–117 017, 2020.

[8] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Commun. Surv. Tuts.*, vol. 22, no. 2, pp. 869–904, Apr.-jun. 2020.

[9] L. Wang, L. Jiao, T. He, J. Li, and M. Mühlhäuser, "Service entity placement for social virtual reality applications in edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 468–476.

[10] Q. Qu, R. Xu, S. Y. Nikouei, and Y. Chen, "An experimental study on microservices based edge computing platforms," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2020, pp. 836–841.

[11] iRobot ready to unlock the next generation of smart homes using the AWS cloud, 2019. [Online]. Available: https://aws.amazon.com/solutions/case-studies/irobot/9

[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–17.

[13] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proc. 3rd USENIX Workshop Hot Top. Edge Comput.*, 2020.

[14] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau , and R. H. Arpaci-Dusseau , "Slacker: Fast distribution with lazy docker containers," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 181–195.

[15] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "CNTR: Lightweight OS containers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 199–212.

[16] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *Proc. 3rd USENIX Workshop Hot Top. Edge Comput.*, 2020.

[17] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving perfect layers out of Docker images," in *Proc. 11th USENIX Workshop Hot Top. Cloud Comput.*, 2019, Art. no. 17.

[18] M.-H. Chen, B. Liang, and M. Dong, "Joint offloading and resource allocation for computation and communication in mobile cloud with computing access point," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[19] Docker official images, 2021. [Online]. Available: https://github.com/docker-library/

[20] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[21] H. Tan, Z. Han, X.-Y. Li, and F. C. M. Lau, "Online job dispatching and scheduling in edge-clouds," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.

[22] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency IoT services in multi-access edge computing," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 668–682, Mar. 2019.

[23] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, 2015, pp. 9–16.

[24] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," *IEEE Trans. Mobile Comput.*, vol. 16, no. 11, pp. 3056–3069, Nov. 2017.

[25] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *IEEE Trans. Commun.*, vol. 64, no. 9, pp. 3746–3758, Sep. 2016.

[26] R. Cziva, C. Anagnostopoulos, and D. P. Pezaros, "Dynamic, latency-optimal VNF placement at the network edge," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 693–701.

[27] Z. Ning *et al.*, "Distributed and dynamic service placement in pervasive edge computing networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1277–1292, Jun. 2021.

[28] R. Solozabal, J. Ceberio, A. Sanchoyerto, L. Zabala, B. Blanco, and F. Liberal, "Virtual network function placement optimization with deep reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 2, pp. 292–303, Feb. 2020.

[29] G. Sallam and B. Ji, "Joint placement and allocation of virtual network functions with budget and capacity constraints," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 523–531.

[30] H. Zhu and O. H. Ibarra, "On some approximation algorithms for the set partition problem," in *Proc. 15th Triennial Conf. Int. Fed. Oper. Res. Soc.*, 1999.

[31] J.-Y. Wang, "Minimizing the total weighted tardiness of overlapping jobs on parallel machines with a learning effect," *J. Oper. Res. Soc.*, vol. 71, no. 6, pp. 910–927, 2020.

[32] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Trans. Mobile Comput.*, vol. 18, no. 9, pp. 2020–2033, Sep. 2019.

[33] N. Zhao *et al.*, "DupHunter: Flexible high-performance deduplication for docker registries," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 769–783.

[34] N. Zhao *et al.*, "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 918–930, Apr. 2021.

[35] Kubernetes, 2021. [Online]. Available: https://kubernetes.io

[36] J. B. Sidney, "Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs," *Operations Res.*, vol. 23, no. 2, pp. 283–298, 1975.

[37] D. S. Hochbaum, "The pseudoflow algorithm: A new algorithm for the maximum-flow problem," *Operations Res.*, vol. 56, no. 4, pp. 992–1009, 2008.
[38] D. M. Powers, "Applications and explanations of Zipf's law," in *Proc. Joint Conf. New Methods Lang. Process. Comput. Natural Lang. Learn.*, 1998, pp. 151–160.
[39] M. J. Siavoshani, F. Parvaresh, A. Pourmiri, and S. P. Shariatpanahi, "Coded load balancing in cache networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 347–358, Feb. 2020.
[40] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–9.
[41] R. Storn and K. Price, "Differential evolution–A simple and efficient heuristic for global optimization over continuous spaces," *J. Global Optim.*, vol. 11, no. 4, pp. 341–359, 1997.
[42] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. Int. Conf. Neural Netw.*, 1995, pp. 1942–1948.
[43] T. Weise, "Global optimization algorithms-theory and application," Free Softw. Foundation, 2011. [Online]. Available: http://www.it-weise.de/
[44] Cplex, IBM ILOG, "V12. 1: User's manual for CPLEX," *Int. Business Mach. Corporation*, vol. 46, no. 53, p. 157, 2009.
[45] N. Aljeri and A. Boukerche, "Mobility management in 5G-enabled vehicular networks: Models, protocols, and classification," *ACM Comput. Surv.*, vol. 53, no. 5, pp. 1–35, 2020.
[46] E. F. Maleki, L. Mashayekhy, and S. M. Nabavinejad, "Mobility-aware computation offloading in edge computing using machine learning," *IEEE Trans. Mobile Comput.*, to be published, doi: 10.1109/TMC.2021.3085527.

Jiong Lou received the BS degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China, in 2016. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His current research interests include edge computing, resource allocation, and reinforcement learning.

Hao Luo received the BSc degree in computer science and technology from the Civil Aviation University of China, Tianjin, China, in 2016, and the MSc degree in computer science and technology from Huaqiao University, Xiamen, China, in 2020. He is currently working as a research assistant with the Institute of Artificial Intelligence and Network, Beijing Normal University.

Zhiqing Tang received the BS degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, Chengdu, China, in 2015. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His current research interests include edge computing, resource allocation, and reinforcement learning.

Weijia Jia (Fellow, IEEE) is currently a chair professor, director of BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (Zhuhai) and VP for Research of BNU-HKBU United International College (UIC) and has been the Zhiyuan chair professor of Shanghai Jiao Tong University, China. He was the chair professor and the deputy director of the State Kay Laboratory of Internet of Things for Smart City, University of Macau. His contributions have been recognized as optimal network routing and deployment; anycast and QoS routing, sensors networking, AI (knowledge relation extractions; NLP etc.) and edge computing. He has more than 600 publications in the prestige international journals/conferences and research books and book chapters. He has received the best product awards from the International Science & Tech. Expo (Shenzhen) in 2011–2012 and the 1st Prize of Scientific Research awards from the Ministry of Education of China in 2017 (list 2). He is the distinguished member of CCF.

Wei Zhao (Fellow, IEEE) received the undergraduate degree in physics from Shaanxi Normal University, Xi'an, China, in 1977, and the MSc and PhD degrees in computer and information sciences from the University of Massachusetts at Amherst, Amherst, Massachusetts, in 1983 and 1986, respectively. He has served important leadership roles in academic including the chief research officer with the American University of Sharjah, the chair of Academic Council, CAS Shenzhen Institute of Advanced Technology, the eighth rector of the University of Macau, the dean of science at 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 at Texas A&M University. He has made significant contributions to cyber-physical systems, distributed computing, real-time systems, and computer networks. He led the effort to define the research agenda of and to create the very first funding program for cyber-physical systems in 2006. His research results have been adopted in the standard of Survivable Adaptable Fiber Optic Embedded Network. He was awarded the Lifelong Achievement Award by the Chinese Association of Science and Technology in 2005.