Startup-Aware Dependent Task Scheduling With Bandwidth Constraints in Edge Computing

Jiong Lou, Zhiqing Tang, Weijia Jia, *Fellow, IEEE*, Wei Zhao, *Fellow, IEEE*, and Jie Li, *Senior Member, IEEE*

***Abstract*—In edge computing, applications can be scheduled in the granularity of inter-dependent tasks to proximate edge servers to achieve high performance. Before execution, the edge server must initialize the corresponding runtime environment, named task startup. However, existing studies on dependent task scheduling severely ignore bandwidth constraints during task startups, which is impractical and incurs a long startup latency. To fill in this gap, we first model the task startup process with bandwidth constraints on edge servers. Then, we formulate the dependent task scheduling problem with startup latency in heterogeneous edge computing. To efficiently generate schedules and satisfy the real-time requirements in edge computing, a novel low-complexity list scheduling algorithm integrated with cloud clone, Startup-aware Dependent Task Scheduling (SDTS), is proposed. Constrained by bandwidth and computation resources, SDTS first coordinates task startup, dependent data transmission, and task execution to optimize each task's finish time. Then, a cloud clone for each task is deployed to utilize scalable resources and initialized runtime environments. Furthermore, task scheduling refinement is designed to release the bandwidth and computation resources consumed by redundant tasks and improve the schedule. Extensive simulations based on real-world datasets show that SDTS substantially reduces 30%-60% makespan compared with existing baselines.**

***Index Terms*—Dependent task, edge computing, startup latency, makespan optimization.**

Manuscript received 17 July 2022; revised 1 January 2023; accepted 15 January 2023. Date of publication 23 January 2023; date of current version 8 January 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFE0201400, in part by the Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC), Zhuhai under Grant 2020KSYS007, in part by the Chinese National Research Fund (NSFC) under Grants 61872239 and 61932014, in part by the Institute of Artificial Intelligence and Future Networks funded by Beijing Normal University (Zhuhai) Guangdong, China Zhuhai Science-Tech Innovation Bureau under Grants ZH22017001210119PWC and 28712217900001, in part by the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University Zhuhai, and in part by the Key R&D Program of Jiangsu, China under Grant BE2020026. *(Corresponding authors: Zhiqing Tang; Weijia Jia.)*

Jiong Lou is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China, and also with Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Guangdong 519087, China (e-mail: lj1994@sjtu.edu.cn).

Zhiqing Tang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Guangdong 519087, China (e-mail: zhiqingtang@bnu.edu.cn).

Weijia Jia is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University, Guangdong 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@uic.edu.cn).

Wei Zhao is with the CAS Shenzhen Institute of Advanced Technology, Shenzhen 518055, China (e-mail: wzhao@aus.edu).

Jie Li is with the Department Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: lijiecs@sjtu.edu.cn).

Digital Object Identifier 10.1109/TMC.2023.3238868

I. Introduction

COMPARED to the remote cloud, edge computing brings the processing capability to the edge of the network [1]. Mobile users can offload applications to the edge servers that are proximate to the users to leverage sufficient computation resources with low latency [2]. To reduce the makespan [3], improve the throughput [4] or minimize the costs [5], a mobile application can be further divided into multiple inter-dependent tasks, and each task can be individually offloaded to different edge or cloud servers.

However, before executing a task on an edge server, the corresponding runtime environment should be initialized in advance, named task startup, which delays the task execution and affects users' quality of experience [6], [7], [8]. The task startup consists of downloading the runtime environment (e.g., container images, Virtual Machine (VM) ISO files, software packages, function codes, etc.) from the cloud (if not exist) and preparing the runtime environment (i.e., booting the container and importing packages) [9], [10], [11]. The size of the runtime environment is considerable, which can reach tens of MB to several GB [12], [13]. Since the limited memory and storage capacity of edge servers, and the increasing number of task types, it is impossible to store or initialize runtime environments required by all tasks. Therefore, the task startup is inevitable, which incurs a significant delay, especially in edge computing with limited downloading bandwidths.

Most of the studies on dependent tasks scheduling ignore the task startup [3], [5], [14], [15] or directly add a constant server/VM booting time [16], [17]. Only a few studies [6], [7] coarsely consider the task startup when making task scheduling decisions. In [7], the authors define a constraint that a task can only be offloaded to the edge servers where its runtime environment is already initialized, without considering the task startup process before execution. In [6], GenDoc is proposed to schedule dependent tasks with on-demand function configuration that is similar to task startup. They assume that multiple functions can be configured simultaneously on an edge server, and each function's configuration time is constant regardless of the bandwidth constraint. However, since the size of a runtime environment can be up to hundreds of MB, the environment downloading of multiple tasks on the same edge server can significantly prolong the startup latency (compared with the single

environment downloading) since the environment downloading process is constrained by the limited bandwidth. Therefore, edge server bandwidth is an important constraint that cannot be ignored in task startup.

Given the task startup latency with bandwidth constraints in edge computing, the following new challenges appear in the area of dependent task scheduling: (1) How to comprehensively model the dependent task scheduling with the task startup process. The task startups with bandwidth constraints on the same edge server should be carefully considered. (2) How to properly coordinate task startup, task execution, and dependent data transmission in heterogeneous edge computing. The task startups are constrained by the bandwidth, while the task executions are limited by the computation resources. (3) How to efficiently utilize the initialized runtime environments in the powerful cloud to release bandwidth and computation resources in edge servers and reduce the makespan further. All these challenges make the dependent task scheduling problem more intractable.

In this paper, we first model the task startup process with bandwidth constraints on edge servers to address the first challenge. Based on the task startup modeling that reflects the realistic scenario, the dependent task scheduling problem with startup latency in heterogeneous edge computing is formulated. Since the problem is NP-hard and the real-time requirements in edge computing, a low-complexity list scheduling algorithm integrated with cloud clone, Startup-aware Dependent Task Scheduling (SDTS), is proposed. To overcome the second challenge, task prioritization and edge server selection are particularly designed for dependent tasks with startup latency. SDTS judges each task's scheduling urgency according to both the runtime environment size and the longest path of its descendant tasks. Then, it optimizes each task's finish time by wisely overlapping the startup time, dependent data transmission time, and other tasks' execution time. To solve the third challenge, SDTS deploys a cloud clone[1] for each task to mitigate the impact of startup latency and leverage powerful cloud resources. SDTS adaptively decides to receive the dependent data from precedent tasks on edge servers or in the cloud. Moreover, to release the bandwidth and computation resources consumed by useless task executions, task scheduling refinement is designed to identify and remove redundant tasks that have no contribution to reducing the makespan during the scheduling process. After the scheduling process, it tightens the schedule to reduce the makespan by utilizing the released resources.

In the case of the unary processing model (i.e., an edge server can execute a task at one time), the time complexity of SDTS is $O(|\mathbf{V}|^2|\mathbf{S}|)$, where $|\mathbf{V}|$ and $|\mathbf{S}|$ are the number of tasks and edge servers, respectively. Its time complexity is as low as HEFT [19], a famous list algorithm for scheduling dependent tasks. Extensive simulations based on real-world datasets from Alibaba [20] prove that SDTS substantially outperforms baselines in terms of makespan.

To the best of our knowledge, we are the first team to study the scheduling problem of dependent tasks with startup latency under bandwidth constraints in heterogeneous edge computing. The contribution of this paper is summarized as follows:

1) The task startup process with bandwidth constraints on edge servers is modeled. Based on the task startup modeling, the scheduling problem of dependent tasks with startup latency in heterogeneous edge computing is formulated.
2) A novel low-complexity list scheduling algorithm is proposed to solve the problem. In edge server selection, the task startup time, dependent task scheduling time, and task execution time are wisely coordinated to reduce each task's finish time.
3) For each task, a cloud clone is deployed to leverage the powerful computation resources and initialized runtime environments in the cloud. Task scheduling refinement is particularly designed to efficiently remove redundant tasks to release bandwidth and computation resources, and tighten the schedule after the scheduling process.
4) Finally, extensive experiments with real-world data from Alibaba [20] are performed to demonstrate that SDTS substantially reduces 30%-60% makespan compared with existing baselines.

II. Related Work

In this section, the related work is classified in following aspects: (1) Dependent task scheduling in edge computing. (2) Dependent task scheduling with startup latency.

A. Dependent Task Scheduling in Edge Computing

There are several studies on dependent task scheduling in edge computing to minimize the makespan [3], [14], [21], [22], reduce the costs [5], [23], or maximize the throughput [4], [24]. MAUI [25] and CloneCloud [21] rely on a standard ILP solver to partition the entire application and decide which methods should be remotely executed to save energy. ThinkAir [14] is proposed to enhance the power of cloud computing by parallelizing method executions using multiple VM images. In [3], Mahmoodi et al. first study joint scheduling and offloading for mobile applications with arbitrary component dependencies. Since the problem is NP-hard, the authors find the optimal solution by using IBM CPLEX optimizer [26].

Recent studies [5], [6], [7], [27] consider the more complex scenarios consisting of edge servers and a remote cloud. Kao et al. [27] design a polynomial-time approximation algorithm (Hermes) to minimize the makespan under resource cost constraints when assigning task graphs that can be described as serial trees to multiple devices. However, in Hermes, the edge devices are assumed to execute an infinite number of tasks simultaneously. ITAGS [5] identifies each task's scheduling decision to minimize the total cost of an application under the deadline. It first uses a binary-relaxed version of the original problem to assign each task a sub-deadline and then greedily optimizes the scheduling of each task subject to its sub-deadline. In [28], the authors propose a heuristic algorithm to make scheduling decisions for dependent tasks to minimize the average makespan of applications, subject to their respective deadline constraints.

[1] A cloud clone is a task replication [18] that is executed in the cloud.

However, these studies assume that all runtime environments are already initialized on each server and any task can be executed on any edge server, which is impractical in resource-limited edge computing.

B. Dependent Task Scheduling With Startup Latency

Task startup is close to some other concepts, e.g., serverless function cold start [8], container startup [10] and VM booting [29]. These latencies can be up to tens of seconds [8], [30], [31], so several approaches,e.g., lightweight environment [32], [33] and environment cache [34], [35] are proposed to accelerate it.

There are also a few studies on scheduling dependent tasks with startup latency. In [7], Zhao et al. define a constraint that a task can only be offloaded to the edge servers configured with the corresponding required environment, but they do not consider the task startup process. In [36], Lin et al. propose a strategy to speed up microservice startups and lower image storage consumption by exploring the advantage of layer sharing. However, they simply start executing tasks after finishing the startup process of all tasks. In [37], the authors estimate the invocation time of each function in the workflow and speculatively deploy resources ahead of the estimated invocation time to reduce cold starts, but they do not consider the task scheduling problem on heterogeneous edge servers.

In [6], GenDoc is proposed to schedule dependent tasks with on-demand function configuration. They first make the function configuration decisions and then decide the start time for each task. However, they assume that multiple functions can be configured concurrently without bandwidth constraints on each other, which is an impractical and irrational model. In this article, the task startup process with bandwidth constraints is comprehensively modeled. Then, some tasks' execution time can be overlapped with other tasks' startup time, which can further reduce the makespan of dependent tasks. Finally, the makespan of dependent tasks is minimized by properly determining each task's execution server, start time, and the environment downloading order on each edge server.

III. Motivation

In this section, the importance of considering the bandwidth constraint during the task startup is shown. Fig. 1 illustrates the optimal schedules for a task graph ($v_1 \rightarrow v_2$) in different models. The task graph is shown in Fig. 1(a). In this illustrative example, two dependent tasks with different sizes of environments are assumed to have identical execution times on homogeneous edge servers.

Fig. 1(b) shows the optimal schedule for the model where the task startup is neglected. In this optimal schedule, two tasks are executed on the same edge server sequentially. In the model of Fig. 1(c), multiple tasks' environments can be initialized independently and are not constrained by the bandwidth, which has been adopted in previous studies [6], [16]. In the optimal schedule for this model, two tasks are still scheduled on the same edge server, and both environments are initialized at the beginning. However, in the realistic scenario, the downloading

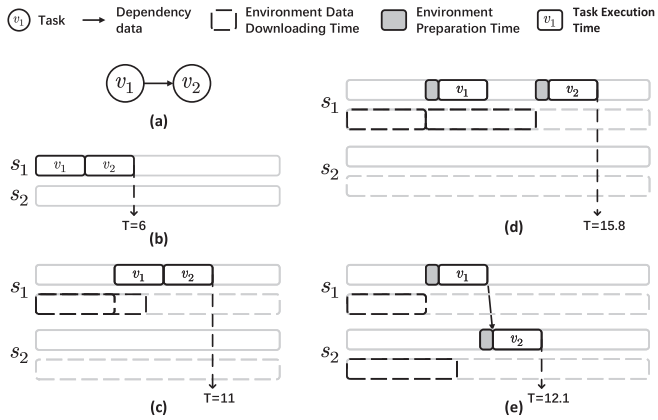


Fig. 1. (a) The simple example of dependent tasks. (b) The optimal schedule for the model where the task startup is neglected. (c) The optimal schedule for the model where multiple tasks' environments can be initialized concurrently without any bandwidth contention. (d) The schedule of (c) is adopted to the dependent tasks scheduling with startup latency under bandwidth constraints. (e) The optimal schedule for dependent tasks with startup latency under bandwidth constraints.

process of multiple tasks' environments at the same time will affect each other due to the constraints of the limited bandwidth. Therefore, it is essential to consider the startup process of multiple tasks.

In the more realistic model of Fig. 1(d) and (e), the downloading process of multiple environments is constrained by the edge node's bandwidth. In this paper, it is assumed that the downloading of one task's environment can fully utilize an edge node's bandwidth [38], [39], [40]. With full utilization of the bandwidth, sequential downloading of environments is better than parallel downloading in terms of the individual downloading finish time of each task's environment. For example, given two environments identically sized p and an edge server with bandwidth b, parallel downloading them with fair bandwidth allocation results in downloading finish times $\frac{2p}{b}$ and $\frac{2p}{b}$, respectively; with sequential downloading, the downloading finish times are $\frac{p}{b}$ and $\frac{2p}{b}$, respectively.

In this model, environment preparation does not occupy the bandwidth but utilizes computation resources. On an edge server, a task's execution time can be overlapped with other tasks' startup time. In Fig. 1(d), the schedule of Fig. 1(c) is adopted, which is far from optimal. Since the execution of task v_2 is delayed by the considerable environment downloading time of v_1 and v_2, the schedule generates an idle time interval. The makespan of schedule in Fig. 1(d) is 15.8. The optimal schedule is shown in Fig. 1(e), where each task is scheduled to an individual edge server. In this optimal schedule, two tasks' environments are downloaded in parallel on two edge servers. Task v_2's environment downloading time is overlapped with task v_1's environment downloading time and execution time, so the execution of task v_2 will not be delayed. The makespan of schedule in Fig. 1(e) is 12.1, which is 23% shorter than Fig. 1(d).

The illustrative example shows that the scheduling algorithm should be specifically designed to solve the problem of dependent task scheduling with startup latency.

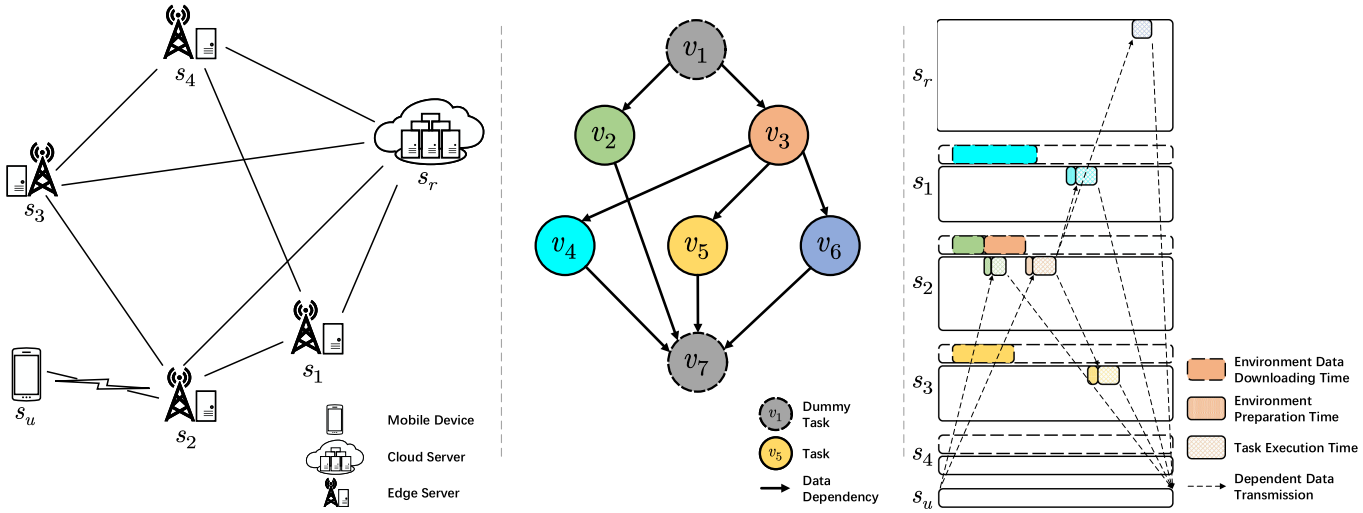


Fig. 2. An example of the system model. The application on the left is released by a mobile device, and dependent tasks are scheduled to different servers (including edge and cloud servers). An example of edge computing is in the middle, where edge servers and the cloud can communicate with each other by a fully connected network. On the right, dependent tasks are scheduled to edge servers or the cloud.

IV. System Model and Problem Formulation

A. System Model

Fig. 2 depicts an overview to illustrate the dependent task scheduling with startup latency. In edge computing, multiple resource-limited edge servers and the remote cloud are fully connected. A mobile device releases an application that can be scheduled at the level of dependent tasks. The server to which tasks are scheduled must initialize the corresponding runtime environments in advance. On each edge server, the environment downloading process included in the task startup is modeled as sequential downloading, which is introduced in the last section. For ease of reference, the key notations used in the paper are summarized in Table I.

Edge Server: The set of all edge servers in the edge computing network is denoted as $\mathbf{S} = \{s_1, s_2, \ldots, s_{|\mathbf{S}|}\}$. To model the heterogeneous processing capabilities of edge servers, the unrelated machine model where each task has machine-dependent execution time on each server is adopted [6], [41], [42]. Each edge server s_k has a limited capacity of computation resources that can execute c_k functions simultaneously. The links connecting edge servers have heterogeneous bandwidths. For ease of representation, the bandwidth between two edge servers s_k and s_l is defined as the latency per unit data transmission $d_{k,l}$, and specifically, $d_{k,l} = d_{l,k}$. The data transmission time on the same edge server is negligible, i.e., $d_{k,k} = 0$ [5].

Cloud Server: The cloud server is denoted as s_c. The cloud s_c has an infinite capacity of computation resources, which means that it can execute an infinite number of tasks simultaneously. The cloud has already initialized run environments for all tasks, which means zero task startup time in the cloud. The latency per unit data transmission of the edge-cloud link is denoted as d_c, which is much longer than $d_{k,l}$ between two edge servers s_k and s_l. $d_{c,c}$ is also assumed to be zero since the negligible transmission time inside the cloud [5].

TABLE I
Notations

Notation	Description
$\mathbf{S}$	Set of edge servers
$\|\mathbf{S}\|$	Total number of edge servers
s_c	Cloud server
s_u	Mobile device
c_k	Maximal number of tasks can run on edge server s_k simultaneously
$d_{k,l}$	Latency per unit data transmission from edge server s_k to edge server s_l
b_k	Downloading bandwidth of the edge server s_k
d_c	Latency per unit data transmission of the edge-cloud link
a	Application
G	Task graph of application a
T	Completion time of application a
$\mathbf{V}$	Set of tasks in application a
$\mathbf{E}$	Set of dependencies in application a
n	Total number of tasks in application a
$t_{i,k}$	Execution time of task v_i on server s_k
$e_{i,j}$	Dependency data from task v_i to task v_j
p_i	Environment size of task v_i
$t^s_{i,k}$	Start time of task v_i on server s_k
$t^f_{i,k}$	Finish time of task v_i on server s_k
$t^d_{i,k}$	Environment downloading time of task v_i on server s_k
$t^r_{i,k}$	Environment downloading finish time of task v_i on server s_k
$t^p_{i,k}$	Environment preparation time of task v_i on server s_k
w_i	Priority of task v_i

Application: The application is denoted as a and released by a mobile device s_u. The application structure is described by a Directed Acyclic Graph (DAG), named task graph $G = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is the set of nodes representing tasks, and $\mathbf{E}$ is the set of directed edges representing dependencies between tasks. The execution time of each task is defined by the execution time matrix T, where each element $t_{i,k}$ specifies the execution time of task $v_i \in \mathbf{V}$ on server $s_k \in \mathbf{S} \cup \{s_c\}$. Task replication [6],

[18] is allowed in this paper, which means that a task can be executed in more than one server to reduce the makespan. The directed edge $(i, j) \in \mathbf{E}$ specifies that there is some required data transmission, $e_{i,j}$, from task v_i to task v_j. If task v_i and task v_j are placed on s_k and s_l, respectively, the data transmission time is $e_{i,j}d_{k,l}$. For each edge $(i, j) \in \mathbf{E}$, task v_i is the predecessor of task v_j, and task v_j is the successor of task v_i. In practice, the task graph G can be obtained by applying a program profiler [14], [25]. In this paper, the corresponding task graph is given when the mobile device s_u releases the application [5], [6].

Before executing a task v_i on edge server s_k, the task startup should be finished. The task startup time mainly consists of two parts: (1) Environment downloading time $t^d_{i,k} = \frac{p_i}{b_k}$, where b_k is the downloading bandwidth of the edge server s_k and p_i is the environment size of task v_i. (2) Environment preparation time $t^p_{i,k}$ including data unzip time, package import time, etc., which also depends on the task and the server. The environment downloading time and environment preparation time of each task on the cloud are equal to zero, i.e., $t^d_{i,c} = 0$ and $t^p_{i,c} = 0$ [6].

Two dummy nodes with zero execution and startup time are inserted into the task graph. One dummy task named source task is inserted at the start to trigger the application, and another one named sink task is inserted at the end to receive all the results. The total number of tasks in application a is changed to $n = |\mathbf{V}| + 2$. Then, task graph G is relabeled by topological sorting so that for every directed edge (i, j), task v_i comes before v_j in the ordering. After insertion and sorting, both $\mathbf{V}$ and $\mathbf{E}$ are updated. The source task and the sink task are denoted as v_1 and v_n, respectively, which are executed on the release mobile device s_u. For a task v_i, tasks on the paths from v_i to the sink task v_n are its descendant tasks, and tasks on the paths from the source task v_1 to v_i are its ancestor tasks.

B. Problem Formulation

In this section, all constraints are introduced and the makespan optimization problem of dependent task scheduling with startup latency is formulated.

After an application a is released at the time of zero, the source task starts to run on the mobile device s_u, i.e., $t^s_{1,u} = 0$. The source task v_1 and sink task v_n run on the mobile device s_u, so that start times of these two tasks on other servers are set to a large enough number M:

$$t^s_{1,k} = t^s_{n,k} = M, \quad \forall s_k \in \mathbf{S} \cup \{s_c\}. \tag{1}$$

To satisfy the real-time requirements of applications in edge computing, once a task v_i starts to run on a server s_k, it cannot be paused or migrated until it completes. The constraints can be represented as:

$$t^f_{i,k} = t^s_{i,k} + t_{i,k}, \quad \forall v_i \in \mathbf{V}, \forall s_k \in \mathbf{S} \cup \{s_c\}. \tag{2}$$

Considering data transmission from task replications, the start time of a task v_i must be later than the earliest time when the dependency data transmission of its precedence task's replication is finished:

$$t^s_{i,k} \geq \min_{s_l \in \mathbf{S} \cup \{s_c\}} \{t^f_{j,l} + e_{j,i}d_{l,k}\}$$
$$\forall (j, i) \in \mathbf{E}, \forall s_k \in \mathbf{S} \cup \{s_c\}. \tag{3}$$

In this paper, the sequential downloading model of multiple tasks' environments is adopted, which means that each edge server only downloads one task's environment at one time. Therefore, the environment downloading time of different tasks on the same edge server are not overlapped:

$$\max\{t^r_{i,k} - t^d_{i,k}, t^r_{j,k} - t^d_{j,k}\} \geq \min\{t^r_{i,k}, t^r_{j,k}\},$$
$$\forall v_i, v_j \in \mathbf{V}, v_i \neq v_j, \forall s_k \in \mathbf{S}, \tag{4}$$

where $t^r_{i,k}$ and $t^r_{j,k}$ are environment downloading finish times of task v_i and task v_j on server s_k, respectively. Specifically, the environment downloading finish time of each task v_i in the cloud s_c is zero, i.e., $t^r_{i,c} = 0$.

A server s_k can start executing the task v_i after downloading and preparing the environment:

$$t^s_{i,k} \geq t^r_{i,k} + t^p_{i,k}, \quad \forall s_k \in \mathbf{S} \cup \{s_c\}, \forall v_i \in \mathbf{V}. \tag{5}$$

Due to the limited resources (e.g., memory and CPU), an edge server s_k can only execute c_k tasks simultaneously. For ease of representation, the capacity of edge server s_k is regarded as c_k virtual cores. The virtual execution core of task v_i on edge server s_k is donated as $h_{i,k} \in \{1, 2, \ldots, c_k\}$. The task occupies the virtual execution core since the server starts to prepare its environment [8]. Therefore, the time duration when the virtual core is occupied by task v_j includes the task's execution time and environment preparation time. The virtual core occupation time of any two tasks on the same virtual core of an edge server is not overlapped:

$$\max\{t^s_{i,k} - t^p_{i,k}, t^s_{j,k} - t^p_{j,k}\} \geq \min\{t^f_{i,k}, t^f_{j,k}\},$$
$$\forall v_i, v_j \in \mathbf{V}, v_i \neq v_j, \forall s_k \in \mathbf{S}, h_{i,k} = h_{j,k}. \tag{6}$$

Since the sink task receives the final execution results on the mobile device s_u and all tasks obey dependent constraints in (3), the completion time T of application a is equal to sink task v_n's finish time $t^f_{n,u}$:

$$T = t^f_{n,u} \tag{7}$$

The objective of the scheduling problem is to minimize the makespan of the application a:

Problem 1:

$$\min_{\{t^r_{i,k}, t^s_{i,k}, h_{i,k}\}} \quad T$$
$$\text{s.t.} \quad Eq.\ (1) - (7), \tag{8}$$

To minimize the makespan of the application a, a schedule that satisfies all the above constraints should be generated. The schedule of the application a is defined as $\{t^r_{i,k}, t^s_{i,k}, h_{i,k} | v_i \in$

Algorithm 1: SDTS.

Input: $\mathbf{S}$, G, $t_{i,k}$, $t^d_{i,k}$
Output: h_i,$h_{i,k}$,$t^s_{i,k}$,$t^s_{i,c}$,$t^r_{i,k}$

Initialize the data transmission graph $\widetilde{G}$ with a virtual end node v_e;
$L = [v_1]$;
for $s_k \in \mathbf{S}$ **do**
  $t^c_k = 0$
/* Task prioritization */
Compute the priority for each task via Eq.(9);
while L *is not empty* **do**
  $v_i = \text{argmax}_{v_i \in L} w_i$;
  if $v_i = v_1$ *or* $v_i = v_n$ **then**
    $h_i = u$;
    Compute $t^s_{i,u}$ via Eq.(15);
    $\widetilde{G}.add\_node(v_i)$, $\widetilde{G}.add\_edge(v_i, v_e)$;
  else
    /* Edge server selection */
    Call Algorithm 2 to select an edge server for v_i;
    /* Cloud clone deployment */
    Compute $t^i_{i,c}$ via Eq.(20);
    $t^s_{i,c} = t^i_{i,c}$;
    Update L;
    $\widetilde{G}.add\_node(v_i)$, $\widetilde{G}.add\_node(\widetilde{v_i})$;
    $\widetilde{G}.add\_edge(v_i, v_e)$, $\widetilde{G}.add\_edge(\widetilde{v_i}, v_e)$;
  /* Task scheduling refinement */
  Call Algorithm 3 to refine the schedule;
/* Task scheduling refinement */
Tighten the schedule based on $\widetilde{G}$;

Algorithm 2: Edge Server Selection.

Input: $\mathbf{S}$, G, v_i, $t^d_{i,k}$
Output: h_i, $h_{i,k}$, $t^s_{i,k}$, $t^r_{i,k}$, t^c_k, $l_{k,u}$

Compute $t^f_{i,k}$ and $h_{i,k}$ of the task v_i on each edge server s_k via (17) and (18);
$h_i = \arg\max_{s_k \in \mathbf{S}} \{t^f_{i,k}\}$;
Compute $t^s_{i,k}$ with $s_k = h_i$ via (15);
$t^r_{i,k} = t^c_k + t^d_{i,k}$;
$t^c_k = t^c_k + t^d_{i,k}$, $u = h_{i,k}$;
$l_{k,u}$ is updated by inserting v_i's virtual core occupation interval $[t^s_{i,k} - t^p_{i,k}, t^f_{i,k}]$;

$\mathbf{V}, s_k \in \mathbf{S} \cup \{s_c\}\}$, including the environment downloading finish time, each task's start time and the occupied virtual core.

Theorem 1: The dependent task scheduling problem with startup latency is NP-hard.

Proof: The dependent task scheduling problem with startup latency is NP-hard since it contains the dependent scheduling problem in [5] as a special case, which ignores the task startup time and assumes unary edge servers. □

V. Algorithm

A. SDTS

Due to the NP-hardness of the scheduling problem and the real-time requirements in edge computing, we propose SDTS, a low-complexity list scheduling algorithm integrated with cloud clone.

As shown in Algorithm 1, SDTS mainly consists of four steps: (1) Task prioritization. The priority of a task is computed according to the environment size and its descendant tasks' workload. (2) Edge server selection. Each task is scheduled to the edge server with the earliest finish time by considering the task startup, dependency data transmission, and task execution. (3) Cloud clone deployment. For each non-dummy task, a clone is deployed in the cloud to benefit from the powerful cloud resources and avoid task startup. (4) Task scheduling refinement. After each task's scheduling, redundant tasks are efficiently identified and removed. At last, the schedule is further improved by utilizing the free time slots.

Task Prioritization: A task v_i's priority w_i is defined as:

$$w_j = \overline{t^d_i} + r_i, \tag{9}$$

where $\overline{t^d_i}$ and r_i are the average environment downloading time and the upward rank, respectively.

The average environment downloading time of task v_i is defined as $\overline{t^d_i} = \frac{\sum_{s_k \in \mathbf{S}} t^d_{i,k}}{|\mathbf{S}|}$. Since the environment downloading of a task is only influenced by precedent environment downloading on the same edge server, the task's priority only considers its own average environment downloading time instead of the average environment downloading time of its descendant tasks. To reduce the makespan, tasks with heavy environments should be scheduled earlier.

The upward rank r_i represents the length of the longest path from task v_i to the sink task [19], recursively defined by:

$$r_j = \max_{(j,i) \in \mathbf{E}} \{r_i + e_{j,i}\overline{d}\} + \overline{t_j}, \tag{10}$$

where $\overline{t_j} = \frac{\sum_{s_k \in \mathbf{S}} t_{j,k}}{|\mathbf{S}|}$ is the average processing time of task v_j and $\overline{d} = \frac{\sum_{s_k \in \mathbf{S}} \sum_{s_l \in \mathbf{S}} d_{k,l}}{|\mathbf{S}|^2}$ is the average time per unit data transmission between edge servers. For the sink task v_n, $r_n = \overline{t_n} = 0$. Normally, tasks with higher upward rank have more descendant tasks and is more urgent to schedule.

The task priority defined in (9) additionally considers the average environment downloading time. A ready task list L is defined as the list of tasks whose predecessors' scheduling decisions are already made. At each time, a task with the highest priority is selected from the ready task list and scheduled to one of the edge servers.

Edge Server Selection: A proper edge server and the start time should be determined for executing the ready task with the highest priority. The prime idea is to greedily schedule each task to the edge server with the earliest finish time, which jointly considers task startup time, input data transmission time, and task processing time. Algorithm 2 describes the process of

selecting a proper edge server for the ready task v_i, and we illustrate it in the following.

Since the task startup time and the data transmission time can be overlapped, the start time of task v_i satisfies the following constraint:

$$t^s_{i,k} \geq \max\{t^e_{i,k}, t^i_{i,k}\}, \tag{11}$$

where $t^e_{i,k}$ and $t^i_{i,k}$ are the environment ready time and the input data ready time, respectively.

For environment ready time, to satisfy the constraints in (5), the download completion time t^c_k of each edge server s_k is defined and updated. The task scheduling sequence determines the environment download sequence. Then, the environment downloading finish time $t^r_{i,k}$ of task v_i on server s_k is computed by

$$t^r_{i,k} = t^c_k + t^d_{j,k}. \tag{12}$$

The environment ready time $t^e_{i,k}$ of task v_i on server s_k can be computed by

$$t^e_{i,k} = t^r_{i,k} + t^p_{i,k}. \tag{13}$$

For input data ready time, each dependency data $e_{j,i}$ is adaptively transferred from the task v_j on the edge server s_l or the cloud clone $\widetilde{v_j}$, which depends on the $e_{j,i}$'s transmission finish time. The input data ready time $t^i_{i,k}$ of task v_i on server s_k can be calculated by

$$t^i_{i,k} = \max_{(j,i)\in\mathbf{E}} \{\min\{t^f_{j,c} + e_{j,i}d_c, t^f_{j,l} + e_{j,i}d_{l,k}\}\}. \tag{14}$$

In order to improve the resource utilization of edge servers and make the start time earlier, the insertion-based policy $ESTfind$ [19] is applied to compute $t^s_{i,k}$. $ESTfind$ tries to insert a task at the earliest idle time between two already scheduled tasks on a server's virtual core if the idle time slot is large enough to accommodate the task. The start time $t^s_{i,k}$ of task v_i on server s_k is computed by

$$t^s_{i,k} = \begin{cases} 0, & i = 1, \\ ESTfind(s_k, \max\{t^e_{i,k}, t^i_{i,k}\}, t_{i,k}), & 1 < i < n, \\ t^i_{i,k}, & i = n, \end{cases} \tag{15}$$

where the insertion-based policy $ESTfind$ is implemented by searching for the earliest idle time slot that is capable of accommodating task v_i's execution time $t_{i,k}$ on server s_k's virtual cores. $ESTfind$ is defined as:

$$\begin{aligned} &ESTfind(s_k, t, t_{i,k}) \\ &\quad = \min_{u\leq c_k, o\in l_{k,u}} \{\max(t, x^s_{o,u,k} + t^p_{i,k}) | x^f_{o,u,k} \\ &\quad - \max(t, x^s_{o,u,k} + t^p_{i,k}) \geq t_{i,k}\}, \end{aligned} \tag{16}$$

where $l_{k,u}$ is the idle time slot list of virtual core u on server s_k. $x^s_{o,u,k}$ and $x^f_{o,u,k}$ are the start time and the finish time of the idle time slot o of virtual core u on server s_k, respectively. The virtual core that contains the selected idle time slot is obtained by

$$\begin{aligned} h_{i,k} = \ &\arg\min_{\{1,2,\ldots,c_k\}} \{ \min_{o\in l_{k,u}} \{\max(t, x^s_{o,u,k} + t^p_{i,k}) | x^f_{o,u,k} \\ &- \max(t, x^s_{o,u,k} + t^p_{i,k}) \geq t_{i,k}\}\}. \end{aligned} \tag{17}$$

The finish time of task v_i on server s_k is computed by

$$t^f_{i,k} = t^s_{i,k} + t_{i,k}. \tag{18}$$

Finally, the edge server s_k with the earliest finish time $t^f_{i,k}$ of task v_i is selected

$$h_i = \arg\max_{s_k\in\mathbf{S}} \{t^f_{i,k}\}, \tag{19}$$

where h_i is denoted as the edge server to execute task v_i.

Cloud Clone Deployment: At the same time as deploying the task on an edge server, a task clone is also deployed in the cloud. As mentioned in Section IV.A, task replication is allowed in this paper. Task replication [18], or task duplication [43] is an efficient technique that allows tasks to be executed multiple times on different servers within a schedule, providing benefits to makespan minimization. However, excessive task replications also occupy massive computation and communication resources [44], which delays the subsequent task execution and finally makes the makespan longer.

In SDTS, we heuristically set that each task can be executed only once on the selected edge server (i.e., task replication is not applied on edge servers). While deploying a task clone in the cloud will not have these shortcomings for the following reasons: First, in contract with the limited resources on edge servers, the computation resources in the cloud are scalable, which is able to execute a large number of tasks simultaneously. Second, the processing time of a task in the cloud is usually shorter than on edge servers for the powerful processing capability of the cloud, which may compensate for the longer transmission time and reduce the makespan. Third, the required run environment of each task in the cloud is already initialized, which will save the task startup time and not add the download overhead. Therefore, SDTS deploys a clone in the cloud for each task. The clone of task v_i is denoted as $\widetilde{v_i}$, and its start time and finish time are denoted as $t^s_{i,c}$ and $t^f_{i,c}$, respectively.

To determine the start time of a task clone $\widetilde{v_i}$ in the cloud, the input data ready time $t^i_{i,c}$ in the cloud is first computed:

$$t^i_{i,c} = \max_{(j,i)\in\mathbf{E}} \{\min\{t^f_{j,c}, t^f_{j,l} + e_{j,i}d_c\}\}, \tag{20}$$

where the task v_j is assumed to be executed on edge server s_l. In this way, each dependency data $e_{j,i}$ is adaptively transferred from the task v_j on the edge server s_k or the task clone $\widetilde{v_j}$ in the cloud, which is determined by their transmission finish times. Since the sufficient computation resources in the cloud, each task clone in the cloud can start immediately after receiving the input data, i.e., $t^s_{i,c} = t^i_{i,c}$.

Task scheduling refinement: After the above scheduling, each non-dummy task is deployed both on edge and cloud servers, and where the input data of each task is transferred from is adaptively decided according to the transmission finish time. There exist some redundant tasks whose output data is useless

Algorithm 3: Task Scheduling Refinement.

Input: $G, \widetilde{G}, v_i$
Output: $\widetilde{G}$

```
1  L_c = [];
2  for (j, i) ∈ E do
3      Add edges according to the rules described in
         task refinement scheduling;
4      if All successors of v_j is scheduled then
5          G̃.remove_edge(v_j, v_e);
6          G̃.remove_edge(ṽ_j, v_e);
7          L_c.add(v_j), L_c.add(ṽ_j);
8  while L_c is not empty do
9      v_k = L_c.pop();
10     if outdegree(v_k) = 0 then
11         G̃.remove_node(v_k);
12         for (v_j, v_k) ∈ Ẽ do
13             G̃.remove_edge(v_j, v_k);
14             L_c.add(v_j);
15 Release resources occupied by redundant tasks;
```

(i.e., its output data is not actually transferred to v_n through any path). These redundant tasks, including tasks on edge servers or task clones in the cloud, consume computation resources but have no contributions to minimizing the makespan. Furthermore, these consumed edge resources can be released for executing other urgent tasks instead, which reduces the makespan. To save resources and reduce the makespan, task scheduling refinement is designed to remove redundant tasks during the scheduling process and tighten the schedule after the scheduling process.

An data transmission graph $\widetilde{G} = (\widetilde{\mathbf{V}}, \widetilde{\mathbf{E}})$ is constructed and maintained during the scheduling process. Initially, the data transmission graph $\widetilde{G}$ only contains a virtual end node v_e. Algorithm 3 shows how to maintain $\widetilde{G}$ by adding tasks and removing redundant tasks after scheduling task v_i. When v_i is scheduled, both v_i and its cloud clone $\widetilde{v_i}$ (if v_i is a non-dummy task) are added to $\widetilde{G}$. Two edges $v_i \to v_e$, $\widetilde{v_i} \to v_e$ are added to $\widetilde{G}$. It ensures that before the scheduling decisions of all v_i's successors are made, v_i and $\widetilde{v_i}$ should be reserved. The actual data transmission between v_i and each of its predecessors v_j is decided by the following rules:

1) If $v_i = v_1$, there is no precedence of v_i.
2) If $v_i = v_n$, for each directed edge $(j, i) \in \mathbf{E}$, $v_j \to v_i$ is added to $\widetilde{G}$ if $t^f_{j,c} + e_{j,i}d_c \geq t^f_{j,l} + e_{j,i}d_{l,u}$; otherwise, $\widetilde{v_j} \to v_i$ is added to $\widetilde{G}$.
3) For each directed edge $(j, i) \in \mathbf{E}$ between two non-dummy tasks v_j and v_i, $v_j \to v_i$ is added to $\widetilde{G}$ if $t^f_{j,c} + e_{j,i}d_c \geq t^f_{j,l} + e_{j,i}d_{l,l}$; otherwise, $\widetilde{v_j} \to v_i$ is added to $\widetilde{G}$. $v_j \to \widetilde{v_i}$ is added to $\widetilde{G}$ if $t^f_{j,c} \geq t^f_{j,l} + e_{j,i}d_c$; otherwise, $\widetilde{v_j} \to \widetilde{v_i}$ is added to $\widetilde{G}$.

These rules ensure that there exists a path from v_1 to v_i and $\widetilde{v_i}$ so that v_i and $\widetilde{v_i}$ can receive the dependent data. Then, for each task v_i (execept v_n) whose successors are all already scheduled,

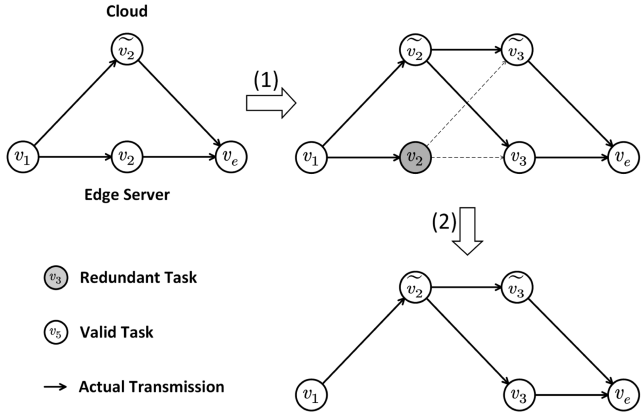
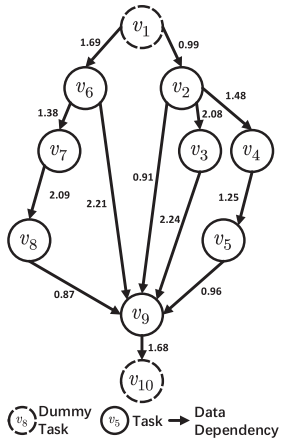


Fig. 3. (1) The data transmission graph $\widetilde{G}$ of a simple dependent task graph ($v_1 \to v_2 \to v_3 \to v_4 \to v_5$) is changed after scheduling v_3. The solid lines represent the actual data transmission adaptively selected according to the transmission finish time. (2) The redundant task v_2 is removed for it has no contribution to the final results, and the edge resources occupied by it are released.



Execution Time Matrix

	s_1	s_2	s_3	s_r
v_2	0.41	0.40	0.93	0.51
v_3	0.83	1.46	1.65	1.08
v_4	1.05	0.88	1.00	0.61
v_5	0.99	0.96	0.92	0.55
v_6	1.03	0.75	0.69	0.89
v_7	0.91	1.34	1.41	0.80
v_8	0.79	1.16	1.43	0.89
v_9	1.31	0.83	1.33	0.81

Transmission Time per Unit Data

	s_1	s_2	s_3	s_r
s_1	0	0.46	0.23	3.3
s_2	0.46	0	0.24	3.3
s_3	0.23	0.24	0	3.3
s_r	3.3	3.3	3.3	0

Fig. 4. Task graph example on the left, the table of execution time matrix on the right top and the table of transmission time per unit data on the right bottom.

two edges $v_i \to v_e$ and $\widetilde{v_i} \to v_e$ are removed from $\widetilde{G}$. It means that the v_i's and $\widetilde{v_i}$'s computation results are transferred to its successors, and whether v_i or $\widetilde{v_i}$ should be removed depends on their successors. Taking Fig. 3(1) as an example, the data transmission graph $\widetilde{G}$ for a simple dependent task graph ($v_1 \to v_2 \to v_3 \to v_4 \to v_5$) is changed after scheduling v_3.

In lines 8-15 of Algorithm 3, redundant tasks without any contribution to minimizing the makespan are efficiently identified and removed so that the occupied edge resources can be released. The tasks reserved in $\widetilde{G}$ are denoted as valid tasks. A valid task must have at least one valid successor task that contributes to minimizing the makespan. For each valid task, there must exist a path from it to the virtual end task v_e in $\widetilde{G}$. The tasks that cannot reach v_e are identified as redundant tasks and removed from $\widetilde{G}$. For example, in Fig. 3(2), tasks v_2 is removed for it cannot reach v_e and other tasks are reserved temporarily.

Since some redundant tasks are removed, there are some new idle time intervals of task execution and environment downloading on edge servers. Utilizing these idle time intervals can

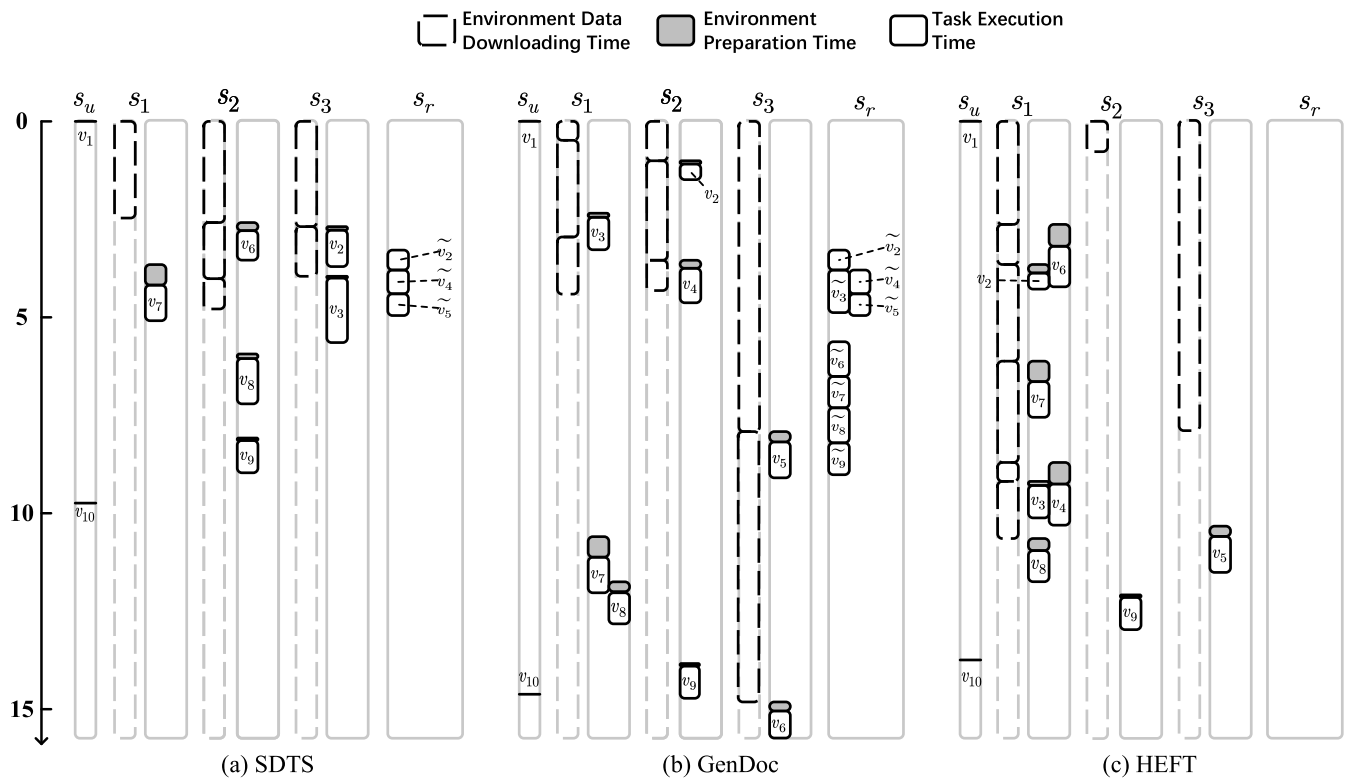


Fig. 5. Scheduling of task graph in Fig. 4 with SDTS, GenDoc and HEFT algorithms. (a) SDTS (makespan = 9.74) (b) GenDoc (makespan = 14.62) (c) HEFT (makespan = 13.74).

tighten the schedule and then reduce the makespan. In line 20 of Algorithm 1, the schedule tightening is achieved by changing the environment downloading finish time and task start time as early as possible without changing other variables (including the runtime environment downloading sequence, tasks' occupied virtual cores, selected edge servers, etc.). The detailed procedure is as follows: (1) Calculating the download finish time of each task according to the downloading sequence on each edge server. (2) Sorting tasks according to the dependent constraints in (3) and the task execution sequence on each virtual core. (3) Calculating the start times for these ordered tasks. Since the process of tightening the schedule brings additional computation overhead, it is only applied once after the entire application is already scheduled.

B. Scheduling Example

Fig. 5 illustrates an task graph example from Alibaba trace [20], which is scheduled by SDTS, GenDoc [6] and HEFT [19] in an edge computing network. The task execution time matrix and transmission time table are shown in Fig. 4. The number of tasks that can be executed simultaneously on each edge server is set to 2. The downloading bandwidth of servers s_1, s_2, and s_3 are 2.81, 2.86 and 1.07, respectively. It is noted that task v_1 and task v_{10} are dummy tasks. The environment sizes of tasks v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, and v_{10} are 0, 2.87, 1.36, 7.26, 8.46, 7.38, 6.92, 4.09, 2.21 and 0, respectively. In SDTS, the priorities of tasks are computed, and the scheduling ordering is $v_1, v_6, v_7, v_2, v_4, v_5, v_8, v_3, v_9, v_{10}$. The scheduling results of SDTS, GenDoc, and HEFT are shown in Fig. 5. SDTS produces a better schedule than the other baselines. Compared with the baselines, SDTS successfully finds a path $\widetilde{v_2} \rightarrow \widetilde{v_4} \rightarrow \widetilde{v_5}$ and offloads it to the cloud to achieve a shorter makespan. GenDos improperly schedules an important task v_6 to s_3 since it ignores the heterogeneous and constrained downloading bandwidth of edge servers, which generates a long startup latency. HEFT ignores the startup, so it schedules six tasks to s_1.

C. Time Complexity Analysis

Theorem 2: The time complexity of SDTS is $O(|\mathbf{V}|^2|\mathbf{S}| + |\mathbf{V}||\mathbf{C}|)$, where $|\mathbf{C}|$, $|\mathbf{V}|$, $|\mathbf{S}|$ are the total number of virtual cores, dependent tasks and edge servers, respectively. In the case of unary edge server model, the time complexity of SDTS is $O(|\mathbf{V}|^2|\mathbf{S}|)$.

Proof: The proposed algorithm consists of four steps: task prioritization, edge server selection, cloud clone deployment, and task scheduling refinement. In task prioritization, the time complexity of computing tasks' priorities is $O(|\mathbf{V}| + |\mathbf{E}|)$, where $|\mathbf{V}|$ and $|\mathbf{E}|$ are task number and edge number. In edge server selection, the time complexity of computing the input data ready time of each task, the environment ready time and the finish time are $O(|\mathbf{E}||\mathbf{S}|)$, $O(|\mathbf{V}||\mathbf{S}|)$ and $O(|\mathbf{V}||\mathbf{C}|)$, where $|\mathbf{S}|$ is the number of edge servers and $|\mathbf{C}| = \sum_{s_k \in \mathbf{S}} c_k$. In cloud clone deployment, the time complexity of computing the start times of task clones is $O(|\mathbf{E}| + |\mathbf{V}|)$. In task scheduling

refinement, the time complexity of maintaining data transmission graph $\widetilde{G}$ and tightening the schedule are $O(|\mathbf{E}|)$ and $O(|\mathbf{V}|^2)$. Therefore, the overall time complexity of the proposed algorithm is $O(|\mathbf{E}||\mathbf{S}| + |\mathbf{V}|^2 + |\mathbf{V}||\mathbf{C}|)$, and for dense DAGs, it becomes $O(|\mathbf{V}|^2|\mathbf{S}| + |\mathbf{V}||\mathbf{C}|)$. Specially, in the case of unary edge server model, the total number of virtual cores $|\mathbf{C}|$ is equal to $|\mathbf{S}|$, so the time complexity of SDTS is $O(|\mathbf{V}|^2|\mathbf{S}|)$.

VI. Evaluation

SDTS is evaluated via simulations. Simulation setup about the edge computing network, applications, and metrics to evaluate performance are introduced first, followed by the description of existing baselines. Simulation results and the corresponding analysis are then presented.

A. Simulation Setups

The SDTS and the simulation environment are implemented in Python 3.8 on a desktop with an Intel Core i9-10900K 3.70 GHz CPU and 32GB RAM. Each simulation result has been repeated ten times with different seeds to mitigate the influence of randomness. The details are introduced in the following.

Edge Computing Network: The edge computing network consists of 5 edge servers and a remote cloud. The average time per unit data transmission between edge servers is denoted as $\overline{d}$. For ease of representation, $\overline{d}$ is normalized to 1. Then $d_{k,l}$ between servers s_k and s_l is randomly chosen from $[\frac{1}{2}\overline{d}, \frac{3}{2}\overline{d}]$ to represent the heterogeneous bandwidths in edge computing. d_c is set to $15\overline{d}$ by default. The average of each row of T, $\overline{t_i}$, is chosen from $[\frac{1}{2}\overline{t}, \frac{3}{2}\overline{t}]$ to represent the workload of v_i, where $\overline{t}$ is the average task execution time and normalized to 1. Then, the execution time $t_{i,k}$ is randomly chosen from $[\frac{1}{2}\overline{t_i}, \frac{3}{2}\overline{t_i}]$. $t_{i,c}$ is set to $\frac{3}{4}\overline{t_i}$ by default. For each edge server s_k, the downloading bandwidth b_k is randomly chosen from $[\frac{1}{2}\overline{b}, \frac{3}{2}\overline{b}]$, where $\overline{b}$ is the average downloading bandwidth and normalized to 1. For each edge server, the number of tasks that can be executed simultaneously, c_k, is randomly selected from $\{1, 2, 3, 4\}$.

Application: To better compare against the most related baseline GenDoc, the simulation dataset is also generated based on Alibaba's trace of data analytics [20], which contains more than 2 million real applications with DAG dependency information. The average task number of applications in Alibaba's trace is 5.3. After filtering duplicated jobs with the same DAG structure, there are 16176 applications with unique DAG structures, each of which has 2 to 205 tasks. Specifically, more than 98% of the DAGs contain less than 50 tasks. The task graph structure of dependent task in Alibaba's dataset is applied in the following simulations.

Then, the weights of tasks and edges are generated. The communication to computation ratio (CCR) is used to represent the relation between dependencies' communication data and tasks' computation workload. CCR is defined as $CCR = \frac{\overline{e}\cdot\overline{d}}{\overline{t}}$ [19], where $\overline{e}$ is the average amount of communication data of dependencies. By default, $\overline{e} = 0.5$ and $CCR = 0.5$. The transmission data size of a dependency is randomly chosen from $[\frac{1}{2}\overline{e}, \frac{3}{2}\overline{e}]$. The downloading to computation ratio (DCR) is used to represent the relation between environment size and tasks' computation workload. DCR is defined as $DCR = \frac{\overline{p}}{\overline{b}\cdot\overline{t}}$, where $\overline{p}$ is the average size of tasks' environment. By default, $\overline{p} = 2$ and $DCR = 2$. Each task's environment size is randomly chosen from $[\frac{1}{2}\overline{p}, \frac{3}{2}\overline{p}]$. Each task's environment preparation time $t^p_{i,k}$ is randomly chosen from $[\frac{1}{2}\overline{t}^p, \frac{3}{2}\overline{t}^p]$, where $\overline{t}^p$ is the average environment preparation time. $\overline{t}^p$ is set to $\frac{1}{5}\overline{t}$ by default.

TABLE II
Key Parameters of Simulation Setups

Parameters	Default value
Average task execution time $\overline{t}$	1
$\overline{d}$	1
d_c	$15\overline{d}$
$\|\mathbf{S}\|$	5
$t_{i,c}$	$\frac{3}{4}\overline{t_i}$
c_k	$\{1, 2, 3, 4\}$
$\overline{b}$	1
$\overline{e}$	0.5
$\overline{p}$	2
CCR	0.5
DCR	2
Average environment preparation time $\overline{t}^p$	$\frac{1}{5}\overline{t}$

The key parameters of simulation setups are shown in Table II.

Metrics: In this paper, the application makespan is the metric used to evaluate each baseline, defined as

$$Makespan = \min\{t^f_{n,u}\} \tag{21}$$

Baselines: Given the real-time requirements in edge computing, the schedule should be generated in a short time. Therefore, SDTS is compared against the following representative and most cited heuristic algorithms.

1) *HEFT [19]:* It is a well-known heuristic that aims to minimize makespans of dependent tasks for heterogeneous computing without considering the task startup latency. The task placement and task order produced by HEFT are applied.
2) *GenDoc [6]:* It is proposed to address the placement and scheduling problem of dependent tasks with function configuration. GenDoc first determines which functions should be configured on each edge server and then design a dynamic programming method to schedule each task.
3) *Purely Cloud:* It places all the tasks in the cloud.
4) *Purely Local:* It places all the tasks on the edge server that can minimize the makespan.

B. Results and Analysis

In the experiment, the overall performance of SDTS and four baselines under the default simulation settings is first compared. Then, the scheduling results under different settings are comprehensively analyzed.

Fig. 6(a) depicts the Cumulative Distribution Function (CDF) of SDTS and four baselines. The overall performance of SDTS is consistently better than the four baselines. The average makespan of SDTS, HEFT, GenDoc, Purely Cloud, and Purely Local are 10.93, 18.22, 15.87, 21.17, and 28.29, respectively. SDTS achieves 40.02%, 31.12%, 48.37%, and 61.37% average makespan reduction compared with GenDoc, HEFT, Purely

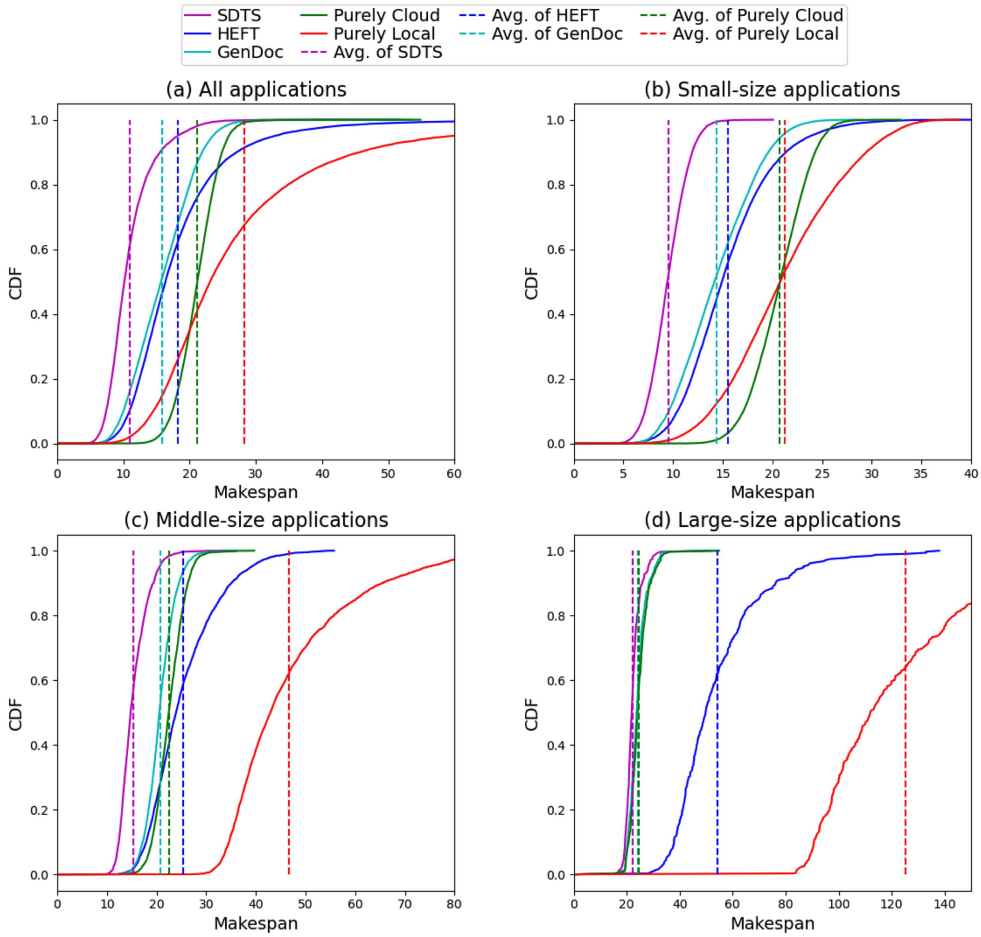


Fig. 6. CDF of makespan for all scheduling algorithms.

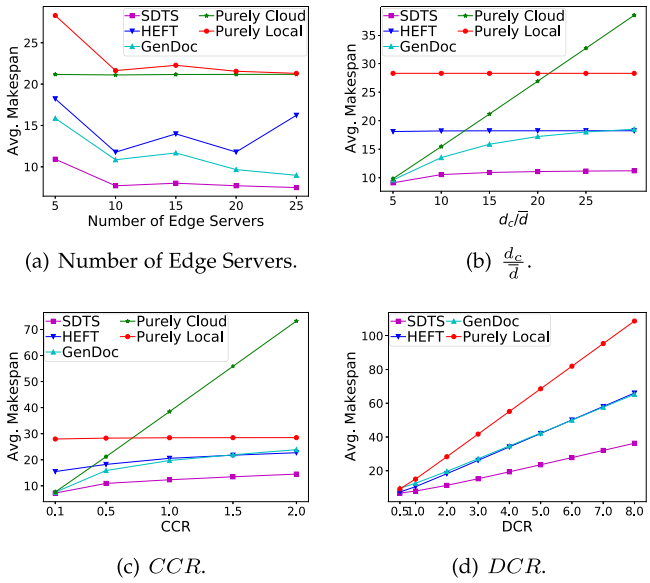
Cloud, and Purely Local, respectively. The reason is that SDTS makes full use of the parallelism nature of task startup on different edge servers and makes the startup time of each task overlap with the execution time of other tasks. GenDoc deploys multiple replications for a single task, so it performs better than HEFT with respect to the average makespan and the CDF of makespan. Compared to Purely Local, HEFT reduces the makespan by improving the parallelism of dependent task execution.

To further analyze the performance of all algorithms with different application sizes, applications are divided into three types according to their sizes: (1) Small-size applications with less than 20 tasks. (2) Middle-size applications with 20-50 tasks. (3) Large-size applications with more than 50 tasks. The results are shown in Fig. 6(b), (c) and (d), respectively. It can be observed that when the application size is smaller, the makespan reduction of SDTS is more. For small-size applications, the average makespan of SDTS is 34%-55% less than the four baselines. While for large-size applications, the average makespan of SDTS is close to GenDoc and Purely Cloud. The reason is that with more tasks, the computation and scalability advantages of the cloud are more obvious. Both SDTS and GenDoc offload nearly all dependent tasks to the cloud, leading to the close performance to Purely Cloud. The performance of Purely Local is much worse when scheduling large-size applications, for it exploits no task parallelism. HEFT tends to execute tasks on edge servers because of its greedy scheduling and the long latency of the edge-cloud link. Without leveraging the powerful cloud resources, HEFT also has poor performance when applications are large.

TABLE III
PAIRWISE MAKESPAN COMPARISON OF THE SCHEDULING ALGORITHMS

		SDTS	GenDoc	HEFT	Purely Cloud	Purely Local
SDTS	Better	*	**98.49%**	**99.70%**	**99.28%**	**100%**
	Equal		0.83%	0.07%	0.72%	0%
	Worse		0.68%	0.22%	0%	0%
GenDoc	Better	0.68%	*	63.85%	87.25%	97.56%
	Equal	0.83%		0.46%	12.75%	0.09%
	Worse	**98.49%**		35.69%	0%	2.35%
HEFT	Better	0.22%	35.69%	*	75.63%	92.13%
	Equal	0.07%	0.46%		0%	0.72%
	Worse	**99.70%**	63.85%		24.37%	7.15%
Purely Cloud	Better	0%	0%	24.37%	*	62.02%
	Equal	0.72%	12.75%	0%		0%
	Worse	**99.28%**	87.25%	75.63%		37.98%
Purely Local	Better	0%	2.35%	7.15%	37.98%	*
	Equal	0%	0.09%	0.72%	0%	
	Worse	**100%**	97.56%	92.13%	62.02%	

Table III lists the pairwise makespan comparison of SDTS and the four baselines for all applications from the Alibaba dataset [20]. Each cell in Table III indicates the comparison



(a) Number of Edge Servers. (b) $\frac{d_c}{\overline{d}}$. (c) CCR. (d) DCR.

Fig. 7. Average makespans of different algorithms with different numbers of edge servers, $\frac{d_c}{\overline{d}}$, CCR and DCR. (a) Number of Edge Servers. (b) $\frac{d_c}{\overline{d}}$. (c) CCR. (d) DCR.

results of the algorithm on the left with the algorithm on the top. SDTS produces better or comparable schedules for almost all applications compared to the baselines. For example, compared with the best baseline GenDoc, SDTS achieves better scheduling in 98.76% of applications, equivalent scheduling in 0.61% of applications, and worse scheduling in 0.63% of applications. It means that SDTS always captures the opportunities to optimize the schedule. There are two major reasons: (1) SDTS properly coordinates that task startup and task execution by fully considering the realistic downloading process of multiple environments. (2) SDTS deploys cloud clones and refines scheduling decisions to schedule some tasks to the cloud, which reduces the makespan.

Next, the impact of different factors on the scheduling results of all algorithms is analyzed. The default simulation parameters of the following experiments are set according to Table II. For each experiment, only one default parameter is varied to evaluate the performance of SDTS in different scenarios. The results are shown in Fig. 7.

Impact of the number of edge servers: Fig. 7(a) shows the scheduling results under different numbers of edge servers. SDTS consistently outperforms the four baselines with respect to makespans. With fewer edge servers, the improvement of SDTS is more obvious, for it is aware of the environment downloading burden on each edge server. Purely Cloud produces the same schedule that places all tasks in the cloud, so that it achieves the same makespan with different numbers of edge servers. When the number of edge servers increases, the results of GenDoc become better since it explores the parallelism of task executions. The makespan of Purely Local is reduced when the number of edge servers increases because it is more likely to find an edge server with less makespan when there are more edge server candidates. The performance of HEFT is unstable since it may schedule tasks to edge servers with heavy environment downloading overhead for unawareness of the startup time.

Impact of the transmission time of the edge-cloud link: Fig. 7(b) shows the average makespan of different algorithms under different d_c. With a short transmission time of the edge-cloud link, offloading all dependent tasks to the cloud achieves the minimal schedule for most applications, so the average makespans of SDTS, GenDoc, and Purely Cloud are close. The reason is that SDTS and GenDoc always deploy a task replication in the cloud, and the task clone adaptively receives input data from edge servers or the cloud. When d_c increases, SDTS still maintains good scheduling results, the average makespan of GenDoc increases for unawareness of environment downloading time on edge servers, and the average makespan of Purely Cloud increases linearly due to the long transmission time on the edge-cloud link. Purely Local schedules all tasks to an edge server. HEFT greedily schedules a task based on its finish time on each server, which makes it tend to place tasks on edge servers for less transmission time. As a result, Purely Local and HEFT are not influenced by varying d_c.

Different CCR and DCR can represent the characters of a wide range of applications. In the experiments of Fig. 7(c) and (d), we vary CCR and DCR to evaluate the performance of SDTS for different applications, respectively.

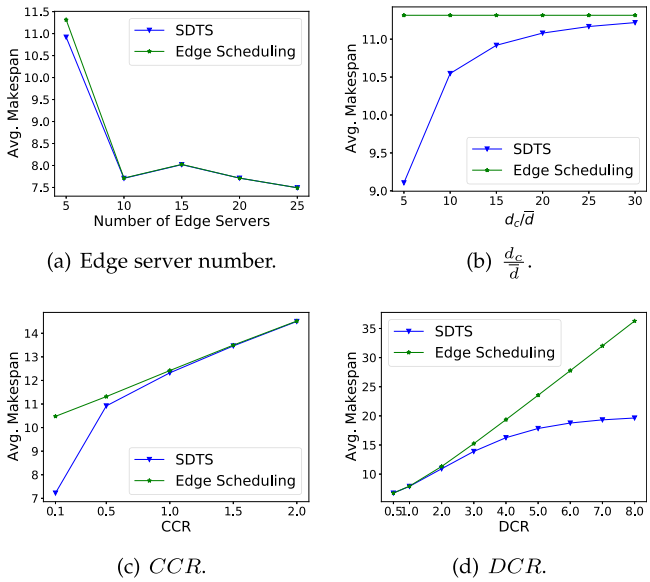
Impact of CCR: Fig. 7(c) depicts the average makespan of different algorithms under different CCRs. The overall performance of SDTS is the best of all scheduling algorithms. For computation-intensive applications with a low CCR, the communications are negligible, so SDTS, GenDoc, and Purely Cloud offload all tasks to the cloud to reduce the makespan. The average makespan of Purely Cloud increases linearly with the data transmission time of the edge-cloud link and Purely Local almost has the same makespan since it schedules all tasks on an edge server.

Impact of DCR: Fig. 7(d) depicts the average makespan of different algorithms under different DCRs. In this experiment, the cloud server is not considered. SDTS still achieves the best result under different DSRs. HEFT and GenDoc have close performance since they schedule tasks without considering the different task startup times on heterogeneous edge servers, which largely affects the makespan.

C. Effectiveness of Each Step

To evaluate the effectiveness of cloud clone deployment and task scheduling refinement, we conduct the following experiments. Same as the experiments of Fig. 7, the default simulation parameters of the following experiments are also set according to Table II. For each experiment, one parameter is changed to evaluate the performance of SDTS in different scenarios.

To evaluate the effectiveness of cloud clone deployment and task scheduling refinement, we remove these two steps in SDTS and name the remaining two steps Edge Scheduling. The comparison results of Edge Scheduling and SDTS under different simulation settings are in Fig. 8. In Fig. 8(a), as the number of edge servers increases, the environment downloading parallelism becomes enough and tasks can be finished earlier on



(a) Edge server number. (b) $\frac{d_c}{\overline{d}}$. (c) CCR. (d) DCR.

Fig. 8. Comparison of SDTS and Edge Scheduling. (a) Edge server number. (b) $\frac{d_c}{\overline{d}}$. (c) CCR. (d) DCR.

edge servers. As a result, the performance of SDTS is close to Edge Scheduling. Fig. 8(b) shows that cloud clone deployment and task scheduling refinement are more effective when the edge-cloud link has sufficient bandwidth (i.e., smaller d_c). With the shorter edge-cloud transmission time, the total makespan can benefit more from scheduling tasks to the powerful cloud. Cloud clone deployment and task scheduling refinement can efficiently find these optimizing chances. In Fig. 8(c), lower CCR means less required communication data between tasks, so execution in the cloud has more advantages. With higher CCR, SDTS tends to schedule tasks on edge servers since the transmission latency of the edge-cloud link is more considerable, and thus two algorithms have close performance. In Fig. 8(d), when the task environment size is larger, the environment downloading time has a greater impact on the makespan. Cloud clone deployment and task scheduling refinement bring more performance improvements since the already initialized environments in the cloud save much startup time.

Next, we conduct two experiments to further evaluate the effectiveness of task scheduling refinement. The number of edge servers and the transmission time of the edge-cloud link vary in the following experiments, respectively. The results are shown in Figs. 9 and 10.

Fig. 9 shows the average task execution number of scheduling algorithms under different numbers of edge servers, which reflects the resource consumption. HEFT, Purely Cloud, and Purely Local schedule each task to one of the edge servers or the cloud (i.e., they do not apply task replication), so the average task execution numbers of them are equal to the average task number of applications. GenDoc tries to fully use the computation resources of each edge server by deploying multiple replications of tasks on edge servers and always deploys a task clone in the cloud. The task execution number of GenDoc is the most of all algorithms. It increases linearly with the number of edge servers

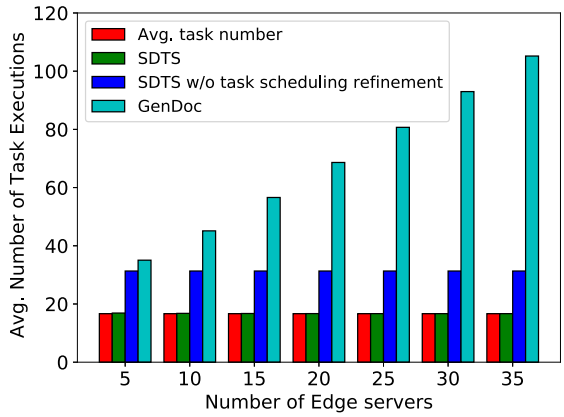


Fig. 9. The impact of task scheduling refinement on the average number of task executions.

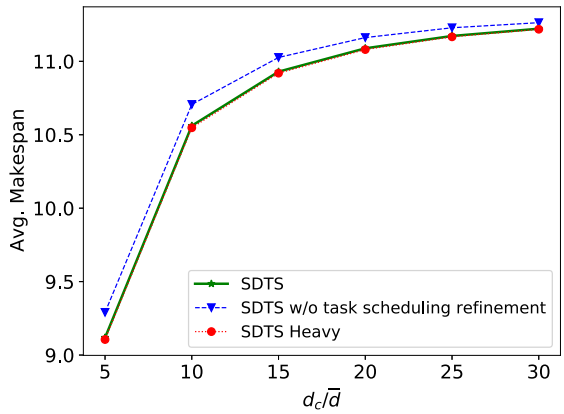


Fig. 10. The impact of task scheduling refinement on the average makespan.

for the total computation resources of edge servers also increase linearly with the number of edge servers. The task execution numbers of SDTS with and without task scheduling refinement are further compared. Since SDTS additionally deploys a task clone for each non-dummy task, without task scheduling refinement, the number of task executions is constantly equal to $2N-2$. Through task scheduling refinement, the task execution number is efficiently reduced and only larger than the average task number by 1%, which means that in most cases, SDTS executes a task only once and achieves good performance (i.e., short makespan) simultaneously.

In the experiment of Fig. 10, the impact of task scheduling refinement is evaluated in terms of the average makespan. As shown in line 20 of Algorithm 1, SDTS only tightens the schedule after all scheduling decisions are made. SDTS Heavy is designed to tighten the schedule after each time of removing redundant tasks. As a result, the time complexity of SDTS Heavy is higher than SDTS by $O(|\mathbf{V}|^3)$. In SDTS Heavy, after each time of removing redundant tasks, the idle time slots are utilized to change the scheduling decisions already generated and the following schedule decisions are made based on the tightened schedule. We conduct the experiment to find whether the SDTS Heavy outperforms SDTS. The result of Fig. 10 shows that the task scheduling refinement brings a slight makespan reduction,

which is about 1.5%. With larger d_c (i.e., the longer transmission time of the edge-cloud link), the improvements are smaller. The reason is that SDTS intends to schedule tasks on edge servers to avoid the long edge-cloud transmission time and fewer edge resources can be released by task scheduling refinement. Besides, it can be observed that SDTS and SDTS Heavy have very close performance under different settings, which means that multiple times of scheduling tightening can hardly improve the schedule further and scheduling tightening should be applied once to avoid high complexity.

VII Conclusion

In this paper, the environment initialization process of multiple dependent tasks is first modeled, particularly considering the significant startup latency caused by the limited bandwidth on edge servers. Next, we formulate the dependent task scheduling problem with startup latency. A novel list scheduling algorithm named SDTS is proposed to efficiently solve the problem. SDTS selects the edge server with the earliest finish time for each dependent task, considering the downloading workload, computation workload, and processing capability of edge servers. Besides, SDTS deployment a cloud clone for each task to utilize the scalable computation resources in the cloud. Extensive simulations based on real-world datasets prove that SDTS substantially outperforms existing baselines in terms of makespan. In the future, we will study the dependent task scheduling problem in a realistic edge computing network with dynamic bandwidth and computation resources.

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Gener. Comput. Syst.*, vol. 97, pp. 219–235, 2019.

[2] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surv. Tut.*, vol. 19, no. 3, pp. 1628–1656, Third Quarter 2017.

[3] S. E. Mahmoodi, R. Uma, and K. Subbalakshmi, "Optimal joint scheduling and cloud offloading for mobile applications," *IEEE Trans. Cloud Comput.*, vol. 7, no. 2, pp. 301–313, Second Quarter 2016.

[4] X. Ni, J. Li, M. Yu, W. Zhou, and K.-L. Wu, "Generalizable resource allocation in stream processing via deep reinforcement learning," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 1, 2020, pp. 857–864.

[5] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 37–45.

[6] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proc. IEEE/ACM 27th Int. Symp. Qual. Serv.*, 2019, pp. 1–10.

[7] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading dependent tasks in mobile edge computing with service caching," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 1997–2006.

[8] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 30–44.

[9] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3702–3712, Dec. 2016.

[10] 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. (HotEdge 20)*, Jun. 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/fu

[11] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the WAN," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 35–50.

[12] N. Zhao et al., "Large-scale analysis of the docker hub dataset," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2019, pp. 1–10.

[13] 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.

[14] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE Infocom*, 2012, pp. 945–953.

[15] Y. Gu and C. Q. Wu, "Performance analysis and optimization of distributed workflows in heterogeneous network environments," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1266–1282, Apr. 2016.

[16] V. Arabnejad, K. Bubendorfer, and B. Ng, "Budget and deadline aware e-science workflow scheduling in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 29–44, Jan. 2019.

[17] G. Yao, Y. Ding, and K. Hao, "Using imbalance characteristic for fault-tolerant workflow scheduling in cloud systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3671–3683, Dec. 2017.

[18] R. N. Calheiros and R. Buyya, "Meeting deadlines of scientific workflows in public clouds with tasks replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 7, pp. 1787–1796, Jul. 2014.

[19] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[20] Alibaba, "Alibaba trace," [EB/OL], 2022. [Online]. Available: https://github.com/alibaba/clusterdata

[21] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.

[22] M. Du, Y. Wang, K. Ye, and C. Xu, "Algorithmics of cost-driven computation offloading in the edge-cloud environment," *IEEE Trans. Comput.*, vol. 69, no. 10, pp. 1519–1532, Oct. 2020.

[23] J. Lou, Z. Tang, S. Zhang, W. Jia, W. Zhao, and J. Li, "Cost-effective scheduling for dependent tasks with tight deadline constraints in mobile edge computing," *IEEE Trans. Mobile Comput.*, early access, Jul. 6, 2022, doi: 10.1109/TMC.2022.3188770.

[24] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proc. 9th Int. Conf. Mobile Syst. Appl. Serv.*, 2011, pp. 43–56.

[25] E. Cuervo et al., "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Serv.*, 2010, pp. 49–62.

[26] IBM, "IBM CPLEX optimizer," [EB/OL], Aug. 24, 2021. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[27] 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.

[28] Y. Liu et al., "Dependency-aware task scheduling in vehicular edge computing," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 4961–4971, Jun. 2020.

[29] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 423–430.

[30] 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.

[31] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.

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

[33] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th Usenix Symp. Netw. Syst. Des. Implementation*, 2020, pp. 419–434.

[34] E. Oakes et al., "{SOCK }: Rapid task provisioning with serverless-optimized containers," in *Proc. {USENIX} Annu. Tech. Conf. {USENIX}*, 2018, pp. 57–70.

[35] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.

[36] 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.

[37] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 356–370.
[38] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2287–2295.
[39] Y. Zhao, C. Tian, J. Fan, T. Guan, X. Zhang, and C. Qiao, "Joint reducer placement and coflow bandwidth scheduling for computing clusters," *IEEE/ACM Trans. Netw.*, vol. 29, no. 1, pp. 438–451, Feb. 2021.
[40] S. K. Roy, R. Devaraj, and A. Sarkar, "Contention cognizant scheduling of task graphs on shared bus-based heterogeneous platforms," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 2, pp. 281–293, Feb. 2022.
[41] J. S. Chadha, N. Garg, A. Kumar, and V. Muralidhara, "A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 679–684.
[42] H. Tan, Z. Han, X.-Y. Li, and F. C. Lau, "Online job dispatching and scheduling in edge-clouds," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[43] M. Orr and O. Sinnen, "Integrating task duplication in optimal task scheduling with communication delays," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2277–2288, Oct. 2020.
[44] M. F. Aktaş and E. Soljanin, "Straggler mitigation at scale," *IEEE/ACM Trans. Netw.*, vol. 27, no. 6, pp. 2266–2279, Dec. 2019.

Jiong Lou received the BSE degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include edge computing, job scheduling, and serverless computing.

Zhiqing Tang received the BS degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015 and the PhD degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2022. He is currently an assistant professor with the Advanced Institute of Natural Sciences, Beijing Normal University, China. His current research interests include edge computing, resource scheduling, 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 State Kay Laboratory of Internet of Things for Smart City with the 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 Fellow the Distinguished Member of CCF.

Wei Zhao (Fellow, IEEE) received the graduation degree in physics from Shaanxi Normal University, China, in 1977, and the MSc and PhD degrees in computer and information sciences with the University of Massachusetts, Amherst, 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 with CAS Shenzhen Institute of Advanced Technology, the eighth Rector of the University of Macau, the Dean of Science with 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 with Texas A&M University. Professor Zhao 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.

Jie Li (Senior Member, IEEE) received the BE degree in computer science from Zhejiang University, Hangzhou, China, the ME degree in electronic engineering and communication systems from China Academy of Posts and Telecommunications, Beijing, China, and the Eng degree from the University of Electro-Communications, Tokyo, Japan. He is currently a chair professor in Department of Computer Science and Engineering, the director of Blockchain Research Centre, Shanghai Jiao Tong University, Shanghai, China. His research interests are in big data and AI, blockchain, network systems and security. He was a full professor in Department of Computer Science, University of Tsukuba, Japan. He was a visiting professor in Yale University, USA, Inria, France. He is the co-chair of IEEE Technical Community on Big Data and the founding Chair of IEEE ComSoc Technical Committee on Big Data and the Co-Chair of IEEE Big Data Community. He serves as an associated editor for many IEEE journals and transactions. He has also served on the program committees for several international conferences.