Cost-Effective Scheduling for Dependent Tasks with Tight Deadline Constraints in Mobile Edge Computing

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

Abstract—In Mobile Edge Computing (MEC), latency-sensitive mobile applications comprising dependent tasks can be scheduled to edge or cloud servers to reduce latency and execution costs. However, existing algorithms based on deadline distribution can hardly satisfy tight application deadlines in heterogeneous MEC due to lacking a global view of the future impacts on descendant tasks. To fill in this gap, we formulate the deadline-constrained cost optimization problem for dependent task scheduling in MEC and propose a low-complexity scheduling algorithm that considers a single task's future impacts in two stages. Specifically: (1) In the edge scheduling stage, each task is scheduled according to its successors' latest start times instead of its sub-deadline to alleviate the lateness of its successors. An edge-only schedule plan is generated by scheduling tasks only on edge servers to save execution costs. (2) In the cloud offloading stage, in order to utilize the powerful cloud resources to satisfy the deadline, the edge-only schedule plan missing the deadline is efficiently modified by properly offloading multiple successive tasks to the cloud. Simulation results show the substantial advantage of the proposed algorithm over baselines in both online and offline scenarios.

Index Terms—Mobile Edge Computing, Dependent Task, Task Scheduling, Tight Deadline, Constrained Optimization

1 Introduction

Mobile Edge Computing (MEC) [1] is a promising paradigm that provides computation resources at the edge of the Internet (e.g., wireless access points) in order that the resource-limited devices can offload mobile applications to nearby edge servers with low latency [2], [3]. Fast-growing mobile applications (e.g., augmented reality [4], cognitive assistance [5], and video processing [6]) are computation-intensive and latency-sensitive [7], so directly offloading the entire application to a proximate edge server can hardly satisfy the Quality of Service requirements (e.g., tight deadline) of these applications. To further improve the parallelism [8] and achieve higher performance [9], a mobile application can be divided into a set of inter-dependent tasks, and tasks can be scheduled to different edge servers.

In MEC, it is critical to satisfy the tight deadlines of mobile applications and reduce the total costs. For example, an augmented reality application [10] consisting of multiple dependent tasks (e.g., renderer, object recognizer, and tracker) typically requires a rapid response in tens of milliseconds. Most of the previous work on deadline constrained dependent tasks scheduling (e.g., scientific workflow [11]) is designed for cloud computing and can be classified into two categories: meta-heuristics [11], [12], [13] and deadline distribution based heuristics [14], [15], [16], [17]. Meta-heuristics are designed to schedule large-scale dependent tasks in the cloud since they are usually time-consuming and thus can hardly satisfy the real-time requirements of MEC. Deadline distribution based heuristics are suited to schedule the tight deadline constrained applications in MEC since the low complexity. They mainly consist of two steps [15], [16]: (1) Divide the application deadline into multiple sub-deadlines and distribute sub-deadlines to individual tasks. (2) Sort tasks and greedily select the resource with the lowest cost for each task to meet its sub-deadline.

However, these deadline distribution based heuristics only focus on scheduling each task to satisfy its sub-deadline but ignore the future impact of each decision on scheduling the following tasks, which has been pointed out as short-sighted [18], [19], [20]. Their short-sighted flaw is more severe in MEC than cloud computing due to the increasingly tight deadlines of mobile applications and the highly heterogeneous MEC network, making them easy to fall into local optima (i.e., unable to satisfy tight deadlines and reduce execution costs simultaneously). Specifically, the short-sighted flaw is mainly manifested in the following two issues:

Firstly, different transmission times of each task's output data incurred by heterogeneous bandwidths between edge servers affect the scheduling of the following tasks. Blindly scheduling a task with much output data to a server with low bandwidth will generate a long transmission time in the future, which squeezes the descendant tasks' execution

- *Corresponding author: Zhiqing Tang and Weijia Jia.*
- *J. Lou and S. Zhang are with 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, PR China. E-mails: {lj1994, zhang_sl}@sjtu.edu.cn*
- *Z. Tang and W. Jia are with Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai), and W. Jia is also with Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College Zhuhai, Guangdong, 519087, PR China. E-mails: domain@sjtu.edu.cn, jiawj@bnu.edu.cn*
- *W. Zhao is with CAS Shenzhen Institute of Advanced Technology, Shenzhen, 518055, PR China. E-mail: zhao.wei@siat.ac.cn*
- *J. Li is with the Department Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: lijiecs@sjtu.edu.cn*

time. Therefore, the short remaining time to execute the descendant tasks results in more application lateness or higher execution costs due to the tight deadlines.

Secondly, the highly heterogeneous processing and transmission capabilities of edge nodes and the cloud affect the scheduling of dependent tasks with tight deadlines. Offloading a single task to the cloud may suffer from a long transmission time. These short-sighted algorithms simply regard the cloud as powerful edge servers, making them tend to schedule a single task to a nearby edge server for the task's earlier finish time and lower execution cost. However, in this way, they severely neglect that the finish time of the entire application can benefit from executing multiple successive tasks in the cloud. As a result, more mobile applications will miss their deadlines.

In this paper, we first formulate the deadline-constrained cost optimization problem for dependent task scheduling in heterogeneous MEC. The inapproximability and NP-hardness of this problem are proved. Then, a novel low-complexity heuristic, named Deadline-constrained Cost-effective algorithm for Dependent task Scheduling (DCDS), is proposed to solve this problem. DCDS consists of two stages specifically designed to address the above two issues by considering the future impact of each task's scheduling decision. In the edge scheduling stage, an edge-only schedule plan is generated by deciding which edge server and when to execute each task. Firstly, the latest start time is assigned to each task based on the longest path of its descendant tasks. Then, the cheapest edge server is selected for each task to ensure the estimated start times of its immediate successor tasks satisfy the corresponding latest start times. In the cloud offloading stage, the edge-only schedule plan that misses the overall deadline is modified by particularly designed cloud offloading. Multiple successive tasks are offloaded to the powerful cloud, which offsets the long data transmission time of the edge-cloud link. By utilizing computation resources in the cloud, more applications' tight deadlines will be met, and more computation resources of edge servers will be reserved for future applications. The time complexity of DCDS is $O(|\mathbf{V}|^2|\mathbf{S}|)$, where $|\mathbf{V}|$ and $|\mathbf{S}|$ are the number of tasks and edge servers, respectively. Extensive simulation results based on real-world applications show the substantial performance advantage of DCDS over existing baselines under various simulation settings in both online and offline scenarios.

The main novelty of this paper is as follows. To the best of our knowledge, we are the first team to consider the dependent task scheduling problem with tight deadlines on the heterogeneous MEC network. We clearly point out the short-sighted flaw of deadline distribution based heuristics. As mentioned before, these short-sighted algorithms only schedule tasks to satisfy their sub-deadlines without considering the future impacts. When scheduling applications with tight deadlines on the heterogeneous MEC network, it results in more lateness, for these algorithms may schedule tasks to nearby edge servers with low bandwidth and also ignore the choice of utilizing the powerful computation resources in the cloud. We particularly design DCDS to solve these issues by (1) considering the transmission time of each task's output data and (2) offloading multiple successive tasks to the powerful cloud. Besides, the time complexity of DCDS is as low as the previous work.

The main contributions of this paper are summarized as follows:

1) Considering the latency-sensitive applications and the heterogeneous MEC network, we formulate the deadline-constrained cost optimization problem for dependent task scheduling in MEC. A novel low-complexity heuristic algorithm with two stages is proposed to schedule applications with tight deadlines in highly heterogeneous MEC.
2) In the edge scheduling stage, a single task's scheduling decision is made based on its successors' latest start times, which additionally considers its output data size and the selected server's bandwidth. In the cloud offloading stage, each task and its descendant tasks are scheduled as a whole to smartly leverage the powerful cloud resources and thus satisfy the tight deadlines of computation-intensive applications.
3) Extensive simulations based on representative real-world applications demonstrate the substantial performance advantage of DCDS compared to existing baselines in both online and offline scenarios.

2 Related Work

In this section, we classify the related work in two aspects: (1) Deadline-constrained Workflow scheduling. (2) Dependent task scheduling in MEC.

2.1 Deadline-constrained Workflow Scheduling

The workflow is also described by a Directed Acyclic Graph (DAG) in which each task is represented by a node, and each dependency is represented by a directed edge [21]. Several scheduling algorithms are proposed to reduce the total execution costs under the deadline constraint [22]. These algorithms are categorized into two classes: meta-heuristics and deadline distribution based heuristics.

Several meta-heuristics are proposed to address the workflow scheduling problem [11], [12], [13], [16], [23]. Rodriguez et al. [12] solve the optimization problem by using particle swarm optimization. In [16], the authors present an ant colony optimization based algorithm L-ACO, where the ant constructs an ordered task list according to the pheromone trail and probabilistic upward rank. In [23], Wu et al. propose a multiobjective evolutionary list scheduling (MOELS) algorithm, where a genome is represented by a scheduling sequence and a preference weight and is interpreted to a scheduling solution via a list scheduling heuristic. Nevertheless, these meta-heuristics are time-consuming and hardly meet the real-time requirements of MEC.

Deadline distribution based heuristics [14], [15], [16], [24] first set sub-deadlines to tasks and then select the cheapest resource for each task to meet its sub-deadline. Yu et al. [14] partition tasks into different levels and divide the deadline into partitions in proportion to their minimum execution time. Abrishami [15] et al. propose two algorithms called IC-PCP and IC-PCPD2 for scheduling workflows in the cloud based on Partial Critical Path [21]. IC-PCP distributes the sub-deadline to each PCP, and IC-PCPD2 further distributes

the sub-deadline to each task in proportion to its minimum execution time. Then, the cheapest resource which can meet the sub-deadline is selected. In [16], ProLiS is proposed to distribute the sub-deadline to each task proportionally to the probabilistic upward rank. The algorithm is designed to take into account the zero transmission time when tasks are co-located. In [24], Arabnejad et al. assign tasks to different levels and distribute the sub-deadline to each level according to both the task number and the total workload.

Compared with these algorithms, DCDS is designed for scheduling applications with tight deadlines in heterogeneous MEC. Specifically: (1) In the edge scheduling stage, DCDS selects an edge server for each task with considering the transmission time of its output data. (2) In the cloud offloading stage, the cloud is efficiently utilized by offloading multiple successive tasks.

2.2 Dependent Task Scheduling in MEC

Dependent task scheduling in MEC are divided into (a) Scheduling tasks on a single local device and the cloud. (b) Scheduling tasks on multiple edge servers and the cloud.

Some early work focused on scheduling dependent tasks on a single local device and the cloud. MAUI [25] enables fine-grained energy-aware offloading of mobile applications to the cloud. CloneCloud [26] makes offloading decisions flexible without modification to the application's source code and enables fine-grained partition on the thread level. ThinkAir [8] enhances the power of mobile cloud computing by parallelizing method execution using multiple virtual machine (VM) images. In [27], authors present an online task offloading algorithm to minimize the makespan of a single application. Mahmoodi et al. [28] first study joint scheduling and cloud offloading for mobile applications with arbitrary component dependencies and solve the problem by using IBM CPLEX optimizer [29].

Some recent work studied scheduling dependent tasks on multiple edge servers and the cloud [9], [17], [30], [31]. Kao [9] 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. GenDoc [30] is proposed to optimize the latency of dependent tasks in MEC, considering function configuration.

ITAGS [17] identifies each task's scheduling decision that minimizes the total cost of running an application under a deadline in MEC. It uses a binary-relaxed version of the original problem to set a sub-deadline for each task and then greedily optimizes the cost of each task subject to its sub-deadline. Compared with ITAGS, DCDS additionally considers the following points: Firstly, in the edge scheduling stage, considering the transmission time of each task's output data, DCDS selects the cheapest edge server for each task to ensure that the estimated start times of its immediate successor tasks satisfy the corresponding latest start times. Secondly, in the cloud offloading stage, the remote cloud with infinite capacity is explicitly modeled. Thirdly, DCDS is designed for scheduling dynamically released applications in online scenarios, which are common in MEC.

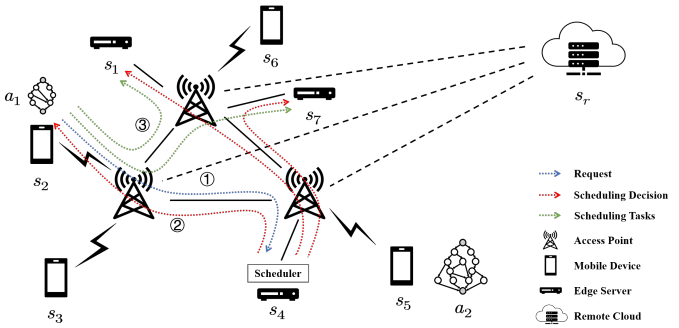


Fig. 1. An example of the MEC network consists of four mobile devices, three edge nodes, and a remote cloud. Servers in the MEC network are fully connected and can communicate with each other. A scheduler is deployed by a third-party service provider on one edge server. The scheduling process includes (1) The mobile device s_2 releases mobile application a_1 at the time of t_1^r. The scheduler collects the information of both the task graph and the MEC network. (2) Based on the information, the scheduler generates a schedule plan, returns it back to the mobile device, and sends commands to edge servers to reserve the corresponding execution time slots. (3) Dependent tasks are scheduled according to the schedule plan.

3 System Model and Problem Formulation

3.1 MEC network model

In this paper, the heterogeneous MEC network is considered with multiple mobile devices, several edge nodes, and a remote cloud. Fig. 1 depicts an illustrative example of an MEC network including seven heterogeneous edge servers (including mobile devices and edge nodes) and a remote cloud. A more general MEC is considered, where dependent tasks can be executed on mobile devices, edge servers, and the cloud [17], [32]. Incentive mechanisms can make mobile devices willing to offer processing capabilities for some returns [33]. Therefore, mobile devices are regarded as edge servers with fewer resources and lower costs. For ease of description, both mobile devices and edge servers are referred to as edge servers in the following.

As shown in Fig. 1, a centralized scheduler is deployed by a third-party service provider on one edge server [34]. When a mobile application is released from a mobile device, the application request, including the task graph information and the deadline, is forwarded to the scheduler through SDN [35]. Then, the scheduler collects the information of the MEC network. For example, the network status is measured by *iperf* tool, and the processing capability is measured by *perf-stat* tool. Since mobile application typically has a relatively short duration, we assume that the network status and processing capability is stable during each application's execution duration and can vary over a longer period (i.e., between multiple applications' execution). By maintaining the daemon process and a long-lived TCP connection with each edge server, the scheduler can obtain this real-time information before making the schedule plan. Based on the information, the scheduler calls DCDS to generate a schedule plan. If the deadline can be met, the scheduler returns the schedule plan back to the mobile device and sends commands to edge servers to reserve the corresponding execution time slots; otherwise, the scheduler rejects the mobile application request. The transmission time of request and scheduling decisions is typically much shorter

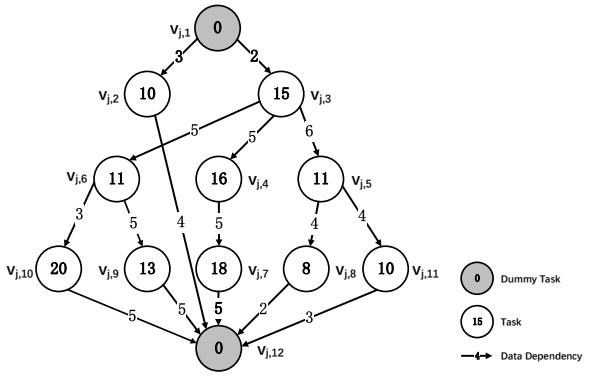


Fig. 2. An example of task graph G_j, where each node $v_{j,i}$ specifies a task labeled with the amount of its workload $w_{j,i}$, and each directed edge implies data dependency labeled with the amount of required data transmission $e^j_{i,i'}$. Two grey nodes represent the dummy tasks with zero workloads.

than the application's makespan, so the latency caused by invoking the centralized scheduler is ignored. Finally, dependent tasks are executed on the selected servers during the reserved time slots according to the schedule plan.

The key notations used in the paper are summarized in TABLE 1.

Edge server. Let the set of all edge servers be $\mathbf{S} = [s_1, \ldots, s_{|\mathbf{S}|}]$ and its size be $|\mathbf{S}|$. p_u and c_u are denoted as server s_u's execution time per unit workload and execution cost per unit workload, respectively. In general, faster servers are more expensive than slower ones [22]:

$$p_u \leq p_{u'}, \quad \text{if } c_u \geq c_{u'}. \tag{1}$$

The edge servers are assumed to be unary, i.e., each server executes one task at a time [17].

Cloud server. The remote cloud is denoted as s_r. The execution time per unit workload of the cloud p_r is lower than edge servers, and the execution cost per unit workload c_r is higher than edge servers. The remote cloud can process an infinite number of tasks simultaneously.

MEC network. The MEC network is fully connected. Let $d_{u,v}$ be the delay per unit data transmission from server s_u to server s_v. The bandwidth of the edge-cloud link is limited, so the corresponding delay per unit data transmission (i.e., $d_{r,v}$ and $d_{v,r}$) is much higher than transmission time between edge servers. Specially, $d_{r,v}$ and $d_{v,r}$ are equal to d_r. The communication delay between two tasks executed in the cloud or on the same edge server is negligible, i.e., $d_{r,r} = 0$ and $d_{u,u} = 0$ [17].

3.2 Application Model

Application. The set of all applications is defined as $\mathbf{A} = [a_1, \ldots, a_{|\mathbf{A}|}]$. The release time of application a_j is denoted as t^r_j. The application deadline t^d_j is set by the mobile user. The mobile device s_u that releases application a_j is denoted as a_j's release server h^r_j, i.e., $h^r_j = u$. The structure of application a_j is described by a DAG, named task graph, $G_j = (\mathbf{V}_j, \mathbf{E}_j)$. Fig. 2 depicts a task graph example, which consists of twelve dependent tasks. $\mathbf{V}_j$ is the set of nodes that represent tasks, and $\mathbf{E}_j$ is the set of directed edges

TABLE 1
Notations

Notation	Description
$\mathbf{S}$	Set of edge servers
$\|\mathbf{S}\|$	Total number of edge servers
$\mathbf{A}$	Set of applications
$\|\mathbf{A}\|$	Total number of applications
G_j	Task graph of application a_j
$\mathbf{V}_j$	Set of tasks in application a_j
$\mathbf{E}_j$	Set of dependencies in application a_j
p_u	Execution time per unit workload on server s_u
c_u	Execution cost per unit workload on server s_u
p_r	Execution time per unit workload in the cloud
c_r	Execution cost per unit workload in the cloud
$d_{u,v}$	Delay per unit data transmission from server s_u to server s_v
d_r	Delay per unit data transmission of the edge-cloud link
t^r_j	Release time of application a_j
t^d_j	Deadline of application a_j
h^r_j	Release server of application a_j
$w_{j,i}$	computation workload of task $v_{j,i}$
$e^j_{i,i'}$	Dependency data from task $v_{j,i}$ to task $v_{j,i'}$
n_j	Total number of tasks in application a_j
$v_{j,1}$	Source task of application a_j
v_{j,n_j}	Sink task of application a_j
$t^s_{j,i}$	Start time of task $v_{j,i}$
$h_{j,i}$	Execution server of task $v_{j,i}$
$t^f_{j,i}$	Finish time of task $v_{j,i}$
$EST(v_{j,i}, s_u)$	Earliest start time of task $v_{j,i}$ on server s_u
$EFT(v_{j,i}, s_u)$	Earliest finish time of task $v_{j,i}$ on server s_u
$EEST(v_{j,i})$	Estimated earliest start time of task $v_{j,i}$
$LT(v_{j,i})$	Latest start time of task $v_{j,i}$
$RTC(v_{j,i})$	Remaining execution time of task $v_{j,i}$ in the cloud

which represent dependencies between tasks. In practice, the task graph G_j, also named application profile, can be obtained by applying a program profiler [8], [25]. In this paper, we assume that the corresponding application profile is given when the mobile user releases the application.

Task. A task of the application a_j is denoted as $v_{j,i} \in \mathbf{V}_j$. Each task $v_{j,i}$ has a weight $w_{j,i}$, which represents the task's computation workload, i.e., the number of CPU cycles required to complete the task. The execution server of task $v_{j,i}$ is denoted as $h_{j,i}$. If task $v_{j,i}$ is scheduled on server s_u, then $h_{j,i} = u$ and the execution time and the execution cost of task $v_{j,i}$ are calculated as $w_{j,i}p_u$ and $w_{j,i}c_u$, respectively.

The directed edge $(i, i') \in \mathbf{E}_j$ specifies that there is some required data transmission $e^j_{i,i'}$ from task $v_{j,i}$ to task $v_{j,i'}$. For each edge $(i, i') \in \mathbf{E}_j$, task $v_{j,i}$ is the predecessor of task $v_{j,i'}$, and task $v_{j,i'}$ is the successor of task $v_{j,i}$. If two tasks $v_{j,i}$ and $v_{j,i'}$ with dependency data $e^j_{i,i'}$ are scheduled on servers s_u and $s_{u'}$, the transmission time is $e^j_{i,i'}d_{u,u'}$.

Two dummy nodes with zero computation workload (e.g., the grey nodes shown in Fig. 2) are inserted into the task graph. One dummy task named source task is inserted at the start to trigger the application, and another task named sink task is inserted at the end to receive all the results back. Thus, the total number of tasks in application a_j is changed:

$$n_j = |\mathbf{V}_j| + 2. \tag{2}$$

Then, task graph G_j are relabeled by topological sorting so that for every directed edge (i, i'), task $v_{j,i}$ comes before $v_{j,i'}$ in the ordering. After insertion and sorting, $\mathbf{V}_j$ and $\mathbf{E}_j$

are updated. The source task and the sink task are denoted as $v_{j,1}$ and v_{j,n_j}, respectively, and they are scheduled to application a_j's release server h_j^r by default. For a task $v_{j,i}$, all tasks that on the paths from $v_{j,i}$ to the sink task v_{j,n_j} are its descendant tasks, and all tasks that on the paths from the source task $v_{j,1}$ to $v_{j,i}$ are its ancestor tasks.

Generally, the MEC scenario can be simplified as follows. Firstly, the edge server is modeled as unary [17], which executes one task at a time. Each server has enough resources and the initialized runtime environment to execute each dependent task, only with different execution times. Secondly, faster edge servers are assumed to be more expensive than slower ones [22]. Thirdly, since this paper focuses on scheduling mobile applications, mobile devices are assumed to send the initial input data and receive the final results [17]. Finally, the status of servers (including the execution cost, the processing capability, the bandwidth, etc.) is stable during the execution of each mobile application but can vary over a longer period [17], [30].

3.3 Problem Formulation

In this subsection, we introduce constraints and formulate the dependent task scheduling problem.

The actual start time and actual finish time of a task $v_{j,i}$ are denoted as $t_{j,i}^s$ and $t_{j,i}^f$, respectively. Tasks cannot start before the application is released:

$$t_{j,i}^s \geq t_j^r, \quad \forall v_{j,i} \in \mathbf{V}_j, \forall a_j \in \mathbf{A}. \tag{3}$$

To satisfy the real-time requirements of mobile applications, once a task starts to run, it cannot be stopped or migrated until it completes. The constraints are represented as:

$$t_{j,i}^f = t_{j,i}^s + w_{j,i} p_{h_{j,i}}, \quad \forall v_{j,i} \in \mathbf{V}_j, \forall a_j \in \mathbf{A}. \tag{4}$$

Since edge servers in this paper are assumed to be unary, the execution time of any two non-dummy tasks (e.g., $t_{j,i}^s$ and $t_{j',i'}^s$) on the same edge server can not be overlapped:

$$\begin{aligned} &\max\{t_{j,i}^s, t_{j',i'}^s\} \geq \min\{t_{j,i}^f, t_{j',i'}^f\}, \\ &\forall\, v_{j,i} \neq v_{j',i'},\ h_{j,i} = h_{j',i'} \neq r. \end{aligned} \tag{5}$$

For dummy tasks, servers can execute them without considering overlap constraints in Eq. (5). An infinite number of tasks can run simultaneously in the remote cloud, so there is also no overlap constraint in s_r.

The two dummy tasks are required to be executed on the release server,

$$h_{j,1} = h_{j,n_j} = h_j^r, \quad \forall a_j \in \mathbf{A}. \tag{6}$$

The start time $t_{j,1}^s$ of the source task $v_{j,1}$ is equal to t_j^r.

The start time of a task must be larger than the finish time of its predecessors plus the required data transmission time. Therefore, dependency constraints are formulated:

$$t_{j,i'}^s \geq t_{j,i}^f + e_{i,i'}^j d_{h_{j,i},h_{j,i'}}, \quad \forall (i,i') \in \mathbf{E}_j, \quad \forall a_j \in \mathbf{A}. \tag{7}$$

The completion time of the application a_j is equal to the finish time of the sink task t_{j,n_j}^f.

$$\max_{v_{j,i} \in \mathbf{V}_j} t_{j,i}^f = t_{j,n_j}^f, \quad \forall a_j \in \mathbf{A}. \tag{8}$$

TABLE 2
Configurations of Edge Servers

Server	p_u	c_u	d_u	Server	p_u	c_u	d_u
s_1	6	11.6	8.6	s_4	7	9.3	19.6
s_2	7	9.3	7.6	s_5	5	15	17.0
s_3	5	15	8.0				

The scheduling problem has two coupled objectives. The first objective is to maximize the number of accepted applications that can complete before their deadlines, and the second objective is to minimize the total execution costs of accepted applications:

Problem 1.

$$\max_{\{h_{j,i}, t_{j,i}^s\}} \quad \sum_{a_j \in \mathbf{A}} 1\{t_{j,n_j}^f \leq t_j^d\} \tag{9}$$

$$\min_{\{h_{j,i}, t_{j,i}^s\}} \quad \sum_{a_j \in \mathbf{A}} \sum_{i=1}^{n_j} 1\{t_{j,n_j}^f \leq t_j^d\} w_{j,i} c_{h_{j,i}} \tag{10}$$

$$s.t. \quad Eq.\ (3) - (7),$$

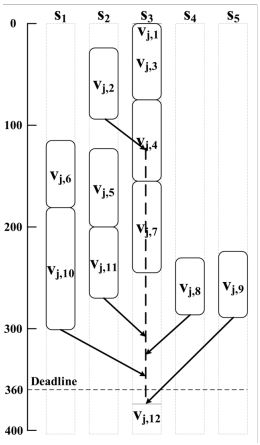
$1\{\cdot\}$ is an indicator function that equals to 1 if the condition inside holds, and 0 otherwise. To optimize the two objectives, a schedule plan is generated for each application. The schedule plan of application a_j is defined as $\{h_{j,i}, t_{j,i}^s | v_{j,i} \in \mathbf{V}_j\}$, including each task's execution server $h_{j,i}$ and start time $t_{j,i}^s$. A schedule plan that satisfies all constraints in Eq. (3)-(7) and meets the application deadline is denoted as a successful schedule plan.

3.4 Problem Analysis

Theorem 1. *Problem 1 is NP-hard and its accepted application maximization problem cannot be approximated within any factor unless NP = P.*

Proof. To prove the NP-hardness of Problem 1, the famous precedence-constrained job scheduling problem is considered. It has been proved to be NP-hard in [36]. Meanwhile, the precedence-constrained job scheduling problem is also a special case of Problem 1: Scheduling a single application with zero size of dependent data and only on identical edge servers. Thus, it can be reduced to Problem 1, and Problem 1 is also NP-hard.

Then, we further prove that the accepted application maximization problem of Problem 1 cannot be approximated within any factor unless NP=P. A special case of Problem 1 is considered, where multiple identical applications described in the last paragraph are scheduled. The release interval of any two consecutive applications is set to be larger than the deadline so that any two applications will not compete for computation resources on edge servers. Assuming that Problem 1 can be approximated within a factor larger than zero, there must exist successful schedule plans for applications in Problem 1. These successful schedule plans can also be applied to answer the precedence-constrained job scheduling problem, which has been proved to be NP-hard. Therefore, the accepted application maximization problem of Problem 1 cannot be approximated within any factor unless NP=P. □



Different Execution Servers for $v_{j,9}$

Server	Finish Time (s)	Transmission Time (s)	Transmission Finish Time (s)
s_1	379	43	422 ✗
s_2	315	38	353 ✓
s_3	310	0	310 ✓
s_4	321	98	419 ✗
s_5 (ProLis)	**289**	85	374 ✗

Scheduling Results of $v_{j,12}$'s Predecessors

Task	Finish Time	Transmission Finish Time
$v_{j,2}$	94	124
$v_{j,7}$	235	235
$v_{j,8}$	286	326
$v_{j,9}$	289	374 ✗
$v_{j,10}$	301	344
$v_{j,11}$	270	308

Fig. 3. An example of scheduling the application in Fig. 2 on edge servers of TABLE 2. The deadline is set to 360. The time values are rounded. In this example, the deadline distribution based heuristic, ProLis [16], generates an unsuccessful schedule plan since the transmission of $v_{j,9}$'s output data cannot be transferred before the deadline (shown in the table of scheduling results). When scheduling $v_{j,9}$, ProLis ignores the long transmission time of its output data but only focuses on its finish time. In the table of different execution servers for task $v_{j,9}$ on the bottom right, despite task $v_{j,9}$ finishing before its sub-deadline (i.e., 309) on s_5, the sink task $v_{j,12}$ is eventually late due to the aforementioned long transmission time. Whereas considering the output data transmission and selecting s_2 or s_3 instead of s_5 can meet the deadline, even though they do not satisfy the sub-deadline of task $v_{j,9}$. The simple example shows the importance of considering the future impacts when scheduling dependent tasks in MEC.

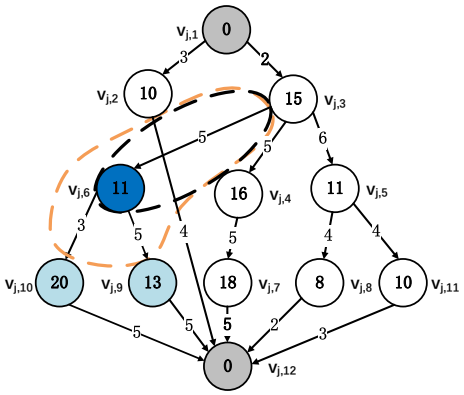


Fig. 4. An example of task graph G_j that illustrates the differences between DCDS and existing heuristics when scheduling the task $v_{j,6}$. The existing heuristics only consider the input data and the task $v_{j,6}$'s workload inside the dotted black circle. In DCDS, the future impact of the task $v_{j,6}$, i.e., the output data inside the orange circle, are additionally considered in the edge scheduling stage. In the cloud offloading stage, the blue node and its descendants (i.e., light blue nodes) are offloaded to the cloud together.

Due to the hardness of this problem and the tight application deadlines, low-complexity deadline distribution based heuristics are more suitable than the meta-heuristics with heavy computational overhead. However, these heuristics that focus on local scheduling decisions for a single task's deadline often fail to minimize the execution costs under tight deadlines.

Fig. 3 depicts an example where a representative deadline distribution based algorithm ProLis [16] is applied to schedule the application of Fig. 2 in an MEC network consisting of five edge servers configured in TABLE 2. p_u, c_u, and d_u represent each server's processing capability, execution costs, and bandwidths, respectively. In this example, each server has an identical transmission time with other servers. The server configurations are generated according to TABLE 3, and the detailed description is in Section 5. The deadline is set to 360. In Fig. 3, ProLis [16] generates a unsuccessful schedule plan. As shown in the Gantt chart on the left and the table of scheduling results on the bottom right, the task $v_{j,9}$'s output data transmission finish time is later than other predecessors of task $v_{j,12}$, which causes the lateness of the application. From the table of different execution servers for task $v_{j,9}$ on the upper right, it can be observed that when making the scheduling decision for task $v_{j,9}$, ProLis only considers the finish time of task $v_{j,9}$ but ignores the long transmission time produced by the considerable output data of $v_{j,9}$ and the limited bandwidth of s_5. Though task $v_{j,9}$ finishes before its sub-deadline (i.e., 309) on s_5, the sink task $v_{j,12}$ is eventually late due to the aforementioned long transmission time. Whereas considering the output data transmission and selecting s_2 or s_3 instead of s_5 can make the application complete before the deadline, even though they do not satisfy the sub-deadline of task $v_{j,9}$. The simple example shows the importance of considering the future impacts when scheduling dependent tasks in MEC.

4 Algorithm

To address the issues of previous studies, we propose DCDS based on the following two rationales: Firstly, latency-sensitive applications typically have tight deadlines, so the algorithm design should carefully take the future impacts into consideration to avoid improper scheduling. Taking Fig. 4 as an example, when scheduling a single task $v_{j,6}$, the transmission time of its output data $e^j_{6,9}$ and $e^j_{6,10}$ inside the orange circle should be considered at the same time. Secondly, servers in the cloud have several unique characteristics: nearly infinite capacity of computation resources, negligible inner transmission time, and limited bandwidth of the edge-cloud link. Therefore, offloading multiple successive tasks should be particularly and explicitly considered in the cloud offloading stage instead of only modeling the cloud as powerful edge servers like previous work.

DCDS has two stages to generate schedule plans for mobile applications: (1) In the edge scheduling stage, the latest start time is assigned to every task according to the longest path from it to the sink task. Then, the cheapest edge server for each task is selected to make the estimated start times of its immediate successor tasks satisfy the corresponding latest start times. (2) In the cloud offloading stage, DCDS tries to improve the edge-only schedule plan by incrementally changing tasks to execute in the cloud and keeping other tasks' scheduling results on edge servers unchanged.

Algorithm 1 shows the pseudocode of DCDS in MEC. DCDS is applied to generate a schedule plan when a mobile application a_j is released to the MEC network. Same as the settings of most of the related studies, the information of

Algorithm 1: DCDS

Input: G_j, t_j^r, t_j^d
Output: $h_{j,i}$,$t_{j,i}^s$
/* Edge scheduling stage */
1 Call Algorithm 2 and get the edge-only schedule plan $\{h_{j,i}, t_{j,i}^s | v_{j,i}\}$;
/* Cloud offloading stage */
2 **if** $t_{j,n_j}^f > t_j^d$ **then**
3 Call Algorithm 3 and modify the schedule plan $\{h_{j,i}, t_{j,i}^s | v_{j,i} \in \mathbf{V}_j\}$;
/* Apply the schedule plan */
4 **if** $t_{j,n_j}^f \leq t_j^d$ **then**
5 Accept a_j and apply $\{h_{j,i}, t_{j,i}^s | v_{j,i} \in \mathbf{V}_j\}$;
6 **else**
7 Reject a_j.;
8 Update idle time slots on edge servers;
9 **end**;

both the task graph and the MEC network is assumed to be known at the time of generating schedule plans. To reduce the algorithm complexity, for each application, DCDS is performed only once, instead of repetitively applying re-arrangement [37] or rescheduling methods [38]. In line 1, an edge-only schedule plan on edge servers for the application is generated in the edge scheduling stage. If the overall deadline cannot be satisfied, the cloud offloading stage is called to modify the schedule plan in line 3. In lines 4 - 7, if the overall deadline is satisfied, application a_j will be accepted, and the schedule plan will be applied; otherwise, application a_j will be rejected. In line 8, the idle time slots on each edge server are updated accordingly. The details of the edge scheduling stage and the cloud offloading stage are introduced in the following subsections.

4.1 Edge Scheduling

The edge scheduling stage has three steps: (1) Latest start time setting. (2) Task ranking. (3) Server selection.

Before introducing the latest start time, we give a brief introduction of the upward rank, $rank$. $rank(v_{j,i})$ represents the longest path from task $v_{j,i}$ to sink task v_{j,n_j} [39], which is recursively defined as:

$$rank(v_{j,i}) = \max_{(i,i') \in \mathbf{E}_j} \{rank(v_{j,i'}) + e_{i,i'}^j \overline{d}\} + w_{j,i}\overline{p}, \quad (11)$$

where $\overline{p}$ and $\overline{d}$ are the average execution time per unit workload and the average transmission delay per unit data, respectively. The upward rank is computed by traversing the graph upward from the sink task v_{j,n_j} to the source task $v_{j,1}$. The $rank(v_{j,n_j})$ of the sink task v_{j,n_j} is

$$rank(v_{j,n_j}) = w_{j,n_j}\overline{p} = 0. \quad (12)$$

The $rank(v_{j,1})$ of the source task $v_{j,1}$ represents the longest path of G_j.

Latest start time setting. Firstly, we assign each task $v_{j,i}$ the latest start time in proportion to the longest path of the task graph G_j minus the longest path from $v_{j,i}$ to the sink task, i.e., $rank(v_{j,1}) - rank(v_{j,i})$. The latest start time LT of task $v_{j,i}$ is computed via

$$LT(v_{j,i}) = t_j^r + (t_j^d - t_j^r) \times \frac{rank(v_{j,1}) - rank(v_{j,i})}{rank(v_{j,1})}. \quad (13)$$

Specially, $LT(v_{j,1})$ and $LT(v_{j,n_j})$ are equal to the release time t_j^r and the overall deadline t_j^d, respectively. We note that the latest start time can be modified according to different deadline distribution methods.

Task ranking. Next, tasks are sorted in ascending order of upward ranks. Since the rank upward of a task is always lower than its predecessors according to Eq. (11), this ranking method ensures the topological order of G_j and preserves the dependencies. Therefore, when scheduling a task, the scheduling decisions (i.e., execution servers and start times) of its predecessors are already determined.

Server selection. Finally, the execution server is selected for each task. When it is task $v_{j,i}$'s turn, scheduling decisions of its predecessors are already determined. The scheduling decision of task $v_{j,i}$ must satisfy dependency constraints in Eq. (7) and overlap constraints in Eq. (5). It is noted that the scheduling decisions of the source task $v_{j,1}$ are determined (i.e., $t_{j,1}^s = t_j^r$ and $h_{j,1} = h_j^r$), and the start time of the sink task v_{j,n_j} is determined regardless of overlap constraints in Eq. (5). For each task $v_{j,i}$ other than two dummy tasks, the earliest start time (EST) of $v_{j,i}$ on every server is computed before selecting the server.

In order to improve the resource utilization and make the start time earlier, the insertion-based policy $ESTfind$ widely used in workflow scheduling [19], [39] is applied to compute EST. $ESTfind$ tries to insert a task at the earliest idle time between two already scheduled tasks on a server if the time slot is large enough to accommodate the task. $EST(v_{j,i}, s_u)$ is computed by

$$EST(v_{j,i}, s_u) = \begin{cases} t_j^r, & i = 1, \\ ESTfind(s_u, TF(v_{j,i}, s_u), w_{j,i}p_u), & 1 < i < n_j, \\ TF(v_{j,i}), & i = n_j, \end{cases} \quad (14)$$

where $TF(v_{j,i}, s_u)$ is the input data transmission finish time of task $v_{j,i}$ on s_u. $TF(v_{j,i}, s_u)$ is computed by

$$TF(v_{j,i}, s_u) = \max_{(i',i) \in \mathbf{E}_j} \{t_{j,i'}^s + w_{j,i'}p_{h_{j,i'}} + e_{i',i}^j d_{h_{j,i'},u}\}. \quad (15)$$

The insertion-based policy $ESTfind$ is implemented by searching for the earliest idle time slot (after $TF(v_{j,i}, s_u)$) that is capable of accommodating task $v_{j,i}$'s execution time $w_{j,i}p_u$ on server s_u:

$$ESTfind(s_u, t, w_{j,i}p_u) = \min_{k \in s_u.l} \{max(t, x_{k,u}^s) | x_{k,u}^f - max(t, x_{k,u}^s) \geq w_{j,i}p_u\}, \quad (16)$$

where $s_u.l$ is the idle time slot list of server s_u, and $x_{k,u}^s$ and $x_{k,u}^f$ are the start time and the finish time of time slot k on server s_u, respectively. The earliest finish time (EFT) of task $v_{j,i}$ in server s_u is calculated by

$$EFT(v_{j,i}, s_u) = EST(v_{j,i}, s_u) + w_{j,i}p_u. \quad (17)$$

The overall deadline will be satisfied if a schedule plan meets all tasks' latest start time for $LT(v_{j,n_j}) = t_j^d$ and

Algorithm 2: Edge Scheduling

Input: $\mathbf{S}$, G_j, t_j^r, t_j^d
Output: $h_{j,i}$, $t_{j,i}^s$

1 Compute the upward rank for each task via Eq. (11);
2 Compute the latest start time for each task via Eq. (13);
3 Sort tasks of application a_j in a scheduling list in ascending order of upward ranks;
4 **for** $v_{j,i}$ *in the scheduling list* **do**
5 $h_{j,i} = -1$;
6 $c = \max_{s_u \in \mathbf{S}}\{c_u\} + 1$;
7 **if** $v_{j,i} = v_{j,1}$ *or* $v_{j,i} = v_{j,n_j}$ **then**
8 $h_{j,i} = h_j^r$;
9 **else**
10 **for** $s_u \in \mathbf{S}$ **do**
11 Compute $EEST(v_{j,i'})$ via Eq. (18);
12 **if** $\max_{(i,i') \in \mathbf{E}_j}\{EEST(v_{j,i'}) - LT(v_{j,i'})\} < 0$ *and* $c > c_u$ **then**
13 $h_{j,i} = u$;
14 $c = c_u$;
15 **if** $h_{j,i} = -1$ **then**
16 Compute $h_{j,i}$ via Eq. (20);
17 $t_{j,i}^s = EST(v_{j,i}, s_u)$, where $u = h_{j,i}$;
18 Update idle time slots on edge server $s_{h_{j,i}}$;
19 Recover idle time slots on edge servers to the state before scheduling a_j;
20 **end**;

$w_{j,n_j} = 0$. When scheduling a task $v_{j,i}$, the objective is to make every successor of $v_{j,i}$ start earlier than its latest start time. The estimated earliest start time ($EEST$) of $v_{j,i}$'s successors is:

$$EEST(v_{j,i'}) = \begin{cases} EFT(v_{j,i}, s_u) + e_{i,i'}^j \overline{d_u}, & i' \neq n_j, \\ EFT(v_{j,i}, s_u) + e_{i,i'}^j d_{u,h_j^r}, & i' = n_j, \end{cases} \tag{18}$$

where $\overline{d_u}$ is the average delay per unit data transmission from server s_u to other edge servers. $e_{i,i'}^j \overline{d_u}$ is used to estimate the required transmission time from task $v_{j,i}$ to task $v_{j,i'}$ before determining the execution server of the successor task $v_{j,i'}$. In this way, the size of the transmission data to successor tasks and the bandwidth of edge servers are considered when scheduling the current task. A server s_u is selected to meet the latest start times of its successors:

$$EEST(v_{j,i'}) \leq LT(v_{j,i'}), \quad \forall (i,i') \in \mathbf{E}_j, \tag{19}$$

If there exist edge servers that can satisfy Eq. (19) for every successor of task $v_{j,i}$, DCDS chooses the cheapest one; otherwise, it chooses the edge server that causes the minimal maximum lateness of the current task's successors:

$$h_{j,i} = \underset{s_u \in \mathbf{S}}{\operatorname{argmin}}\{\max_{(i,i') \in \mathbf{E}_j}\{EEST(v_{j,i'}) - LT(v_{j,i'})\}\}. \tag{20}$$

The detailed algorithm is shown in Algorithm 2. In lines 1-2, the upward ranks and latest start times of tasks are computed. In lines 5-14, if there exist edge servers that can satisfy Eq. (19), the cheapest one will be selected; otherwise, the server that causes the minimal maximum lateness of the successors is selected in line 16. In line 17, the start time of task $v_{j,i}$ is set. In line 18, task $v_{j,i}$ is scheduled to server $s_{h_{j,i}}$ and idle time slots in server $s_{h_{j,i}}$ are temporally updated.

Theorem 2. *The schedule plan generated in the edge scheduling stage satisfies constraints in Eq.* (3)-(7).

Proof. Each task's start time is later than the source task's start time since the source task is the root of the task graph. Based on $t_{j,i}^s = t_j^r$ in Eq. (14), all tasks' start times satisfy constraints in Eq. (3). Eq. (4) and Eq. (6) are satisfied for the algorithm settings. For overlap constraints in Eq. (5), the insertion-based policy $EST find$ makes sure that each task runs in an idle time slot of the selected edge server. Besides, according to $EST find$ in Eq. (14), each task's scheduling decisions meet dependency constraints in Eq. (7). □

4.2 Cloud Offloading

When the edge-only schedule plan generated by Algorithm 2 fails to meet the application's deadline, DCDS tries to utilize powerful cloud resources and improve the schedule plan instead of directly discarding it. However, changing any task from edge servers to the cloud will affect other tasks' scheduling, and cost-effectively rescheduling these affected tasks under deadline constraints is also NP-hard. For instance, in Fig. 4, if only task $v_{j,3}$ is offloaded to the cloud, rescheduling the sub task graph consisting of $v_{j,6}$, $v_{j,9}$, $v_{j,10}$ and $v_{j,12}$ on edge servers to meet the deadline is also NP-hard.

Therefore, in the cloud offloading stage, the problem is heuristically simplified by offloading some tasks to the cloud and keeping other tasks' scheduling decisions on edge servers unchanged. Specifically: (1) Applying one-climb offloading [40]. One-climb offloading means that in any task path from the source task to the sink task, there exists at most one data transmission from edge servers to the cloud [40]. Take Fig. 4 as an example, in a path $v_{j,1} \rightarrow v_{j,3} \rightarrow v_{j,6} \rightarrow v_{j,9} \rightarrow v_{j,12}$, there is only one climb at $v_{j,3} \rightarrow v_{j,6}$. (2) Setting the sink task to receive the data sent back from the cloud. If a task is scheduled to the cloud, its descendant tasks, except for the sink task, should also be scheduled to the cloud. In Fig. 4, if task $v_{j,6}$ is scheduled to the cloud, its descendants (light blue nodes) except for $v_{j,12}$ are also scheduled to the cloud. The heuristic simplification is reasonable since (a) frequent offloading can bring too much transmission latency, and (b) the size of the final result is relatively small for mobile applications [41], [42]. We leave the scheduling problem of offloading arbitrary tasks to the cloud as future work.

Through the above simplification, the scheduling problem is transformed to **determining which tasks and their descendants (except for the sink node) should run in the cloud to meet the overall deadline**.

Before describing the algorithm, some definitions are first introduced:

(1) Edge task: the task executed on edge servers.
(2) Cloud task: the task executed in the remote cloud.
(3) Offloading task: the cloud task, one of whose predecessors is executed on edge servers. On the one hand, offloading tasks belong to cloud tasks; On

Algorithm 3: Cloud Offloading

Input: $\mathbf{S}$, G_j, $h_{j,i}$, $t^s_{j,i}$, t^d_j
Output: $h_{j,i}$, $t^s_{j,i}$

1 Compute $RTC(v_{j,i})$ for each task via Eq. (21) and Eq. (22);
2 Initialize the edge task set $ETS = \mathbf{V}_j$;
3 Initialize the offloading task queue OTQ;
4 **for** $(i, n_j) \in \mathbf{E}_j$ **do**
5 **if** $t^s_{j,i} + w_{j,i}p_{h_{j,i}} + e^j_{i,n_j}d_{h_{j,i},h_{j,n_j}} > t^d_j$ **then**
6 $OTQ.push(v_{j,i})$, $ETS \leftarrow ETS \setminus \{v_{j,i}\}$;
7 **while** $len(OTQ) > 0$ **do**
8 $v_{j,i} = OTQ.pop()$;
9 **if** $v_{j,i} = v_{j,1}$ **then**
10 Return $h_{j,i}$,$t^s_{j,i}$;
11 **for** $(i', i) \in \mathbf{E}_j$ **do**
12 **if** $v_{j,i'} \in ETS$ *and* $t^s_{j,i'} + w_{j,i'}p_{h_{j,i'}} + e^j_{i',i}d_r + RTC(v_{j,i}) > t^d_j$ **then**
13 $OTQ.push(v_{j,i'})$, $ETS \leftarrow ETS \setminus \{v_{j,i'}\}$;
14 **for** $(i, i') \in \mathbf{E}_j$ **do**
15 **if** $v_{j,i'} \in ETS$ *and* $i' \neq n_j$ **then**
16 $OTQ.push(v_{j,i'})$, $ETS \leftarrow ETS \setminus \{v_{j,i'}\}$;
17 $t^s_{j,n_j} = t^d_j$;
18 **for** $v_{j,i} \in \mathbf{V}_j \setminus ETS$ **do**
19 Update idle time slots on edge server $s_{h_{j,i}}$;
20 $h_{j,i} = r$;
21 $t^s_{j,i} = t^d_j - RTC(v_{j,i})$;
22 Call RefineScheduling($\mathbf{S}$, G_j, $t^s_{j,i}$, $v_{j,i}$);
23 **end**;

Algorithm 4: RefineScheduling

Input: $\mathbf{S}$, G_j, $h_{j,i}$, $t^s_{j,i}$
Output: $h_{j,i}$, $t^s_{j,i}$

1 **for** $v_{j,i} = v_{j,2}$ *to* $v_{j,(n_j-1)}$ **do**
2 $c = c_{h_{j,i}}$;
3 **for** $s_u \in \mathbf{S} \setminus \{s_{h_{j,i}}\}$ **do**
4 Compute $EFT(v_{j,i}, s_u)$ via Eq. (14) and Eq. (17);
5 $LFT(v_{j,i}, s_u) = \min_{(i,i')\in\mathbf{E}_j} \{t^s_{j,i'} - e^j_{i,i'}d_{u,h_{j,i'}}\}$;
6 **if** $EFT(v_{j,i}, s_u) \leq LFT(v_{j,i}, s_u)$ *and* $c_u < c$ **then**
7 $c = c_u$;
8 $h_{j,i} = u$;
9 $t^s_{j,i} = EFT(v_{j,i}, s_u) - w_{j,i}p_u$;
10 **end**;

the other hand, cloud tasks are offloading tasks or descendants of offloading tasks.

(4) Remaining execution time in the cloud (RTC): for each task, the RTC is defined as the longest path from it to the sink task, assuming that the task and its descendant tasks are scheduled to the cloud.

RTC is used to identify the edge tasks, cloud tasks, and offloading tasks in a low time complexity way. Since the transmission time between two tasks in the cloud is negligible, RTC is defined as

$$RTC(v_{j,i}) = \max_{(i,i')\in\mathbf{E}_j} \{RTCE(e^j_{i,i'}) + w_{j,i}p_{j_r}\}, \quad (21)$$

where $RTCE(e^j_{i,i'})$ is the remaining execution time in the cloud on edge $e^j_{i,i'}$ and defined as

$$RTCE(e^j_{i,i'}) = \begin{cases} RTC(v_{j,i'}), & i' \neq n_j, \\ RTC(v_{j,i'}) + e^j_{i,n_j}d_r, & i' = n_j. \end{cases} \quad (22)$$

Specifically, $RTC(v_{j,n_j})$ is equal to 0 for sink task v_{j,n_j}.

A successful schedule plan obeys that: For each offloading task $v_{j,i}$ and anypredecessor of $v_{j,i}$ on edge servers (e.g., $v_{j,i'}$), $v_{j,i'}$'s finish time plus transmission time of the edge-cloud link plus $RTC(v_{j,i})$ satisfies the deadline t^d_j:

$$\underbrace{t^s_{j,i'} + w_{j,i'}p_{h_{j,i'}}}_{v_{j,i'}'s\ finish\ time} + \underbrace{e^j_{i',i}d_r}_{transmission\ time} + RTC(v_{j,i}) \leq t^d_j \quad (23)$$

The cloud offloading stage is designed to find all offloading tasks, set them and their descendants except the sink task as cloud tasks, and remain other tasks' scheduling decisions. It mainly includes: Firstly, any predecessor task $v_{j,i}$ of the sink task v_{j,n_j} that causes the application's lateness (i.e., $t^s_{j,i} + w_{j,i}p_{h_{j,i}} + e^j_{i,n_j}d_{h_{j,i},h_{j,n_j}} > t^d_j$) will be identified and changed to be an offloading task. Then, traversing the task graph upward from these offloading tasks, tasks are checked one by one whether they should be offloaded to the cloud.

Algorithm 3 shows the details of the cloud offloading stage. In line 1, the RTC of each task is computed by traversing the task graph upward from the sink task v_{j,n_j}. In lines 2-3, the edge task set ETS and the offloading task queue OTQ are initialized. Since all tasks are executed on edge servers by default, ETS initially includes all tasks of the application, and OTQ is empty. In lines 4-6, any predecessor task of the sink task v_{j,n_j} that causes the application's lateness will be removed from ETS and pushed to OTQ. In the loops of lines 7-16, the predecessors and successors of the task $v_{j,i}$ popped from OTQ will be verified whether they should be offloaded to the cloud. In lines 9-10, the original unsuccessful schedule plan is returned since the source task $v_{j,1}$ cannot be executed in the cloud. In lines 11-13, if any predecessor (e.g., $v_{j,i'}$) of $v_{j,i}$ is an edge task and unsatisfied Eq. (23), $v_{j,i'}$ is pushed to OTQ and removed from ETS. In lines 14-16, if any successor (except the sink task) of cloud tasks is an edge task, it is also pushed to OTQ and changed to be a cloud task. In line 17, the start time of the sink task v_{j,n_j} is set to t^d_j. In lines 20-21, the execution servers and the start times of cloud tasks are updated. In line 22, the function RefineScheduling is called to reduce the execution cost further while still satisfying all constraints.

The function RefineScheduling shown in Algorithm 4 is designed to refine tasks' schedule decisions. For each task $v_{j,i}$, the function modifies its scheduling decision to

gradually reduce the total cost without affecting other tasks. In detail, for tasks other than the sink task $v_{j,1}$ and the source task v_{j,n_j}, RefineScheduling computes the earliest finish time EFT and the latest finish time LFT on each server. If a task's output is transferred to its successor tasks in time, i.e., $EFT(v_{j,i}, s_u) \leq LFT(v_{j,i}, s_u)$, then server s_u is a feasible server for task $v_{j,i}$. In lines 6-9, the cheapest feasible server is selected as the execution server, and the start time is updated correspondingly.

Theorem 3. *The schedule plan generated in the cloud offloading stage satisfies constraints in Eq.* (3)-(7)*.*

Proof. In Sec. 4.1, we show that the schedule plan produced in the edge scheduling stage satisfies constraints in Eq. (3)-(7). The schedule plan returned in line 10 of Algorithm 3 remains unchanged, so it also satisfies all constraints. Then, the modified schedule plan in lines 20-21 of Algorithm 3 is considered. There is no overlap constraint Eq. (5) in the cloud since its infinite computation resources. We now prove that the execution server and start time changes in lines 20-21 of Algorithm 3 still obey with constraints in Eq. (3), (4), (6) and (7). The constraints in Eq. (4) are satisfied since $t^f_{j,i} = t^s_{j,i} + w_{j,i}p_{h_{j,i}}$. Two dummy tasks are scheduled to the release server, so constraints in Eq. (6) are met.

For dependency constraints in Eq. (7), dependencies are classified into: (1) Dependency between edge tasks, which is proved in Sec. 4.1. (2) Dependency between an edge task $v_{j,i}$ and an offloading task $v_{j,i'}$. Suppose edge task $v_{j,i}$ is scheduled to server s_u. From Eq. (23), we obtain that such dependency satisfies that

$$t^s_{j,i} + w_{j,i}p_u + e^j_{i,i'}d_r + RTC(v_{j,i'}) \leq t^d_j. \tag{24}$$

And based on line 21 of Algorithm 3, we obtain:

$$\begin{aligned} t^f_{j,i} + e^j_{i,i'}d_{u,r} &= t^s_{j,i} + w_{j,i}p_u + e^j_{i,i'}d_r \\ &\leq t^d_j - RTC(v_{j,i'}) = t^s_{j,i'}. \end{aligned} \tag{25}$$

(3) Dependency between two cloud tasks $v_{j,i}$ and $v_{j,i'}$. Firstly, the transmission time between servers in the remote cloud is negligible, i.e., $e^j_{i,i'}d_{r,r} = 0$. From Eq. (21) and Eq. (22), we get $RTC(v_{j,i}) \geq RTC(v_{j,i'}) + w_{j,i}p_r$. In addition, based on line 21 in Algorithm 3, we have:

$$\begin{aligned} t^f_{j,i} + e^j_{i,i'}d_{r,r} &= t^s_{j,i} + w_{j,i}p_r + e^j_{i,i'}d_{r,r} \\ &= t^s_{j,i} + w_{j,i}p_r \\ &= t^d_j - RTC(v_{j,i}) + w_{j,i}p_r \\ &\leq t^d_j - RTC(v_{j,i'}) = t^s_{j,i'}. \end{aligned} \tag{26}$$

(4) Dependency between a cloud task $v_{j,i}$ and the sink task v_{j,n_j}. Based on Eq. (21) and Eq. (22), we have $RTC(v_{j,i}) \geq w_{j,i}p_r + e^j_{i,n_j}d_r$. According to line 21 in Algorithm 3, we have:

$$\begin{aligned} t^f_{j,i} + e^j_{i,n_j}d_{h_{j,i},h_{j,n_j}} &= t^s_{j,i} + w_{j,i}p_r + e^j_{i,n_j}d_r \\ &= t^d_j - RTC(v_{j,i}) + w_{j,i}p_r + e^j_{i,n_j}d_r \\ &\leq t^d_j = t^s_{j,n_j}. \end{aligned} \tag{27}$$

As a result, all dependencies satisfy dependency constraints in Eq. (7). Furthermore, based on the dependency constraints and the fact that the source task is the root of the task graph, it is obvious that for every task $v_{j,i}$, $t^s_{j,i} \geq t^f_{j,1} = t^r_j$, which is equal to Eq. (3).

Besides, RefineScheduling modifies each task's scheduling decisions without violating any constraints. □

4.3 Time Complexity Analysis

In this subsection, we analyze the time complexity of DCDS. The proposed algorithm consists of the edge scheduling stage and the cloud offloading stage. In the edge scheduling stage, the time complexity of computing the upward rank and the latest start time for tasks is $O(|\mathbf{V}| + |\mathbf{E}|)$, where $|\mathbf{V}|$ and $|\mathbf{E}|$ are task number and edge number, respectively. The time complexity of selecting servers for tasks is $O((|\mathbf{V}| + |\mathbf{E}|)|\mathbf{S}|)$. In the cloud offloading stage, the time complexities of computing RTC for each task, changing edge tasks to cloud tasks, and RefineScheduling are $O(|\mathbf{E}|)$, $O(|\mathbf{V}| + |\mathbf{E}|)$ and $O((|\mathbf{V}| + |\mathbf{E}|)|\mathbf{S}|)$, respectively. Therefore, the overall time complexity of the proposed algorithm is $O((|\mathbf{E}| + |\mathbf{V}|)|\mathbf{S}|) = O(|\mathbf{E}||\mathbf{S}|)$, and for dense DAGs, it becomes $O(|\mathbf{V}|^2|\mathbf{S}|)$.

5 Evaluation

5.1 Simulation Setup

DCDS is evaluated via simulations. Simulation setup about the MEC network, applications, and metrics are introduced first, followed by the description of existing baselines. Simulation results and corresponding analysis are then presented. The DCDS and the simulation environment are implemented in Python 3.6 on a desktop with an Intel Core i9-10900K 3.70 GHz CPU and 32GB RAM. Each simulation result has been repeated ten times to mitigate the influence of randomness.

MEC network. The MEC network consists of $|\mathbf{S}| = 20$ edge servers and a remote cloud. The configurations of different edge server types are listed in TABLE 3. The server configurations in TABLE 3 are set by referring to [16], [23]. p is the execution time per unit workload and c is the execution cost per unit workload. The following adjustments are made to edge servers based on the configurations generated for cloud servers [16], [23]: (1) Since edge servers are resource-constrained, the execution time per unit workload of different edge servers in TABLE 3 is set to a narrower range (i.e., 5 - 10). (2) The edge servers with more powerful processing capabilities (i.e., lower execution time per unit workload) have higher execution costs, and the execution cost grows faster than the processing capability. Thus, the edge server with $p = 5$ is set to three times more expensive than the edge server with $p = 10$. To investigate the influence of different server configurations, additional experiments with server configurations of [16] are conducted in Section 5.2.3. The execution cost and the execution time per unit workload of each edge server in $\mathbf{S}$ are randomly chosen from TABLE 3. The execution cost and the execution time per unit workload of the cloud are 50 and 2, respectively. In this paper, to represent each edge server's network state, the average time $\overline{d_u}$ per unit data transmission between server s_u and other edge servers is randomly chosen from $[\frac{1}{2}\overline{d}, \frac{3}{2}\overline{d}]$. Then, $d_{u,v}$ of each pair of

TABLE 3
Capabilities and Costs of Server Types

Server type	p^*	c^{**}	Server type	p^*	c^{**}
Edge Server 1	10	5.0	*Edge Server* 5	6	11.6
Edge Server 2	9	6.1	*Edge Server* 6	5	15.0
Edge Server 3	8	7.5	*Cloud Server*	2	50.0
Edge Server 4	7	9.3			

* p is the execution time per unit workload.
** c is the execution cost per unit workload.

servers s_u and s_v is chosen from $[\frac{3}{4}\overline{d_u}, \frac{5}{4}\overline{d_u}]$. By default, d_r is set to $5\overline{d}$.

Application. The experiment application dataset is generated by randomly choosing different structures of real-world applications. For real-world applications, Workflow Generator [43] is used to generate structures of five well-known workflows [44] with different characteristics.

(1) Montage is an astronomy application used to generate custom mosaics of the sky, and most of its tasks are I/O intensive but do not require much CPU capacity.
(2) LIGO is used in physics for detecting gravitational waves and has CPU-intensive tasks that consume large amounts of memory.
(3) Epigenomics is a data processing pipeline to automate the execution of various genome sequencing operations.
(4) SIPHT is used in bioinformatics to automate the search for sRNA encoding-genes.
(5) CyberShake is used to characterize earthquake hazards by generating synthetic seismograms and is a data-intensive workflow with large memory and CPU requirements.

Then, the weights of tasks and edges are generated. The communication to computation ratio (CCR) is used to represent the relation between dependency transmission data and tasks' computation workload. CCR is defined as $CCR = \frac{\overline{e} \times \overline{d}}{\overline{w} \times \overline{p}}$, where $\overline{e}$ and $\overline{w}$ are the average amount of transmission data of dependencies and the average amount of task workload, respectively. The transmission data of a dependency is randomly chosen from $[\frac{1}{2}\overline{e}, \frac{3}{2}\overline{e}]$, and the workload of a task is randomly chosen from $[\frac{1}{2}\overline{w}, \frac{3}{2}\overline{w}]$.

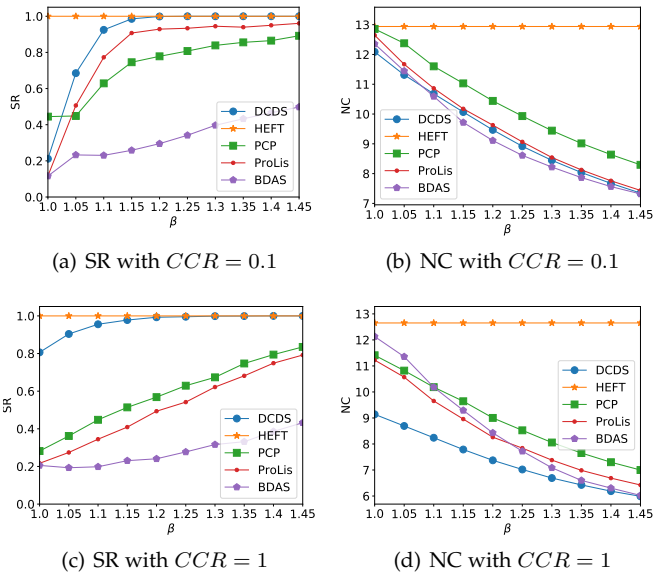
In offline and online scenarios, for every experiment, a trace containing 1000 applications is generated with different structures, workloads, and dependencies. Applications are released stochastically and associated with a deadline. By default, the release interval follows a Poisson distribution with $\lambda = \frac{1}{100}$ in online scenarios.

Metrics. Success Rate (SR) is defined as the ratio between the number of accepted applications and the total number of applications.

$$SR = \frac{\sum_{a_j \in \mathbf{A}} 1\{t^f_{j,n_j} \leq t^d_j\}}{|\mathbf{A}|}. \tag{28}$$

Normalized Cost (NC) is the total execution costs of accepted applications divided by the total workload of them.

$$NC = \frac{\sum_{a_j \in \mathbf{A}} \sum_{i=1}^{n_j} 1\{t^f_{j,n_j} \leq t^d_j\} w_{j,i} c_{h_{j,i}}}{\sum_{a_j \in \mathbf{A}} \sum_{i=1}^{n_j} 1\{t^f_{j,n_j} \leq t^d_j\} w_{j,i}}. \tag{29}$$



(a) SR with $CCR = 0.1$ (b) NC with $CCR = 0.1$

(c) SR with $CCR = 1$ (d) NC with $CCR = 1$

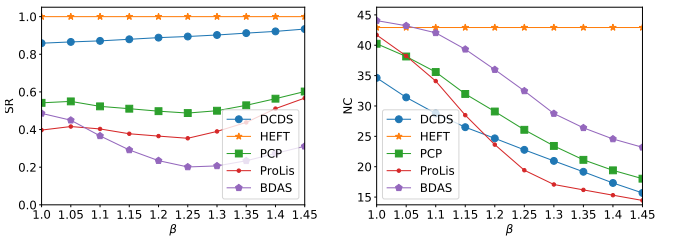
Fig. 5. SR and NC of different algorithms for applications with $CCR = 0.1, 1$ in an edge-only MEC network.

For an application a_j, the makespan of the schedule plan of HEFT [39] is denoted as M_j and is used as the basic deadline [16]. The deadline factor β is defined to represent the looseness degree of application deadlines. The deadline of an application a_j is

$$t^d_j = t^r_j + M_j \times \beta. \tag{30}$$

Baselines. DCDS is compared against the representative and most cited algorithms of cost-effective scheduling for deadline constrained dependent tasks. These baselines are all selected from deadline distribution based heuristics since the low complexity satisfies the real-time requirements of edge computing:

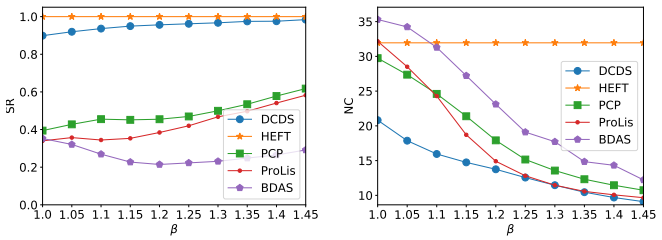
(1) HEFT [39] is a well-known heuristic that aims to minimize makespans of dependent tasks for heterogeneous computing. It schedules each task to the server with the earliest finish time. HEFT can be seen as a special case of deadline distribution based algorithm where each task's sub-deadline is equal to the application's release time.
(2) PCP [21] finds Partial Critical Path (PCP), distributes sub-deadlines to tasks on each PCP, and selects the cheapest resource to meet each task's sub-deadline.
(3) ProLis [16] is a stochastic scheduling algorithm. It sets a sub-deadline for each task proportionally to the longest path to the entry node. The probabilistic upward rank is used to represent the longest path in ProLis [16].
(4) BDAS [24], which distributes sub-deadlines based on a deadline proportion and the number of tasks in each level. In this paper, the total execution cost is the objective and the deadline is the constraint, so the deadline distribution method of BDAS is chosen as a baseline.



(a) SR with $CCR = 0.1$ (b) NC with $CCR = 0.1$

(c) SR with $CCR = 1$ (d) NC with $CCR = 1$

Fig. 6. SR and NC of different algorithms for applications with $CCR = 0.1, 1$ in a generic MEC network.



(a) SR with $d_r = 2\overline{d}$ (b) NC with $d_r = 2\overline{d}$

(c) SR with $d_r = 8\overline{d}$ (d) NC with $d_r = 8\overline{d}$

Fig. 7. SR and NC of different algorithms with different transmission times of the edge-cloud link.

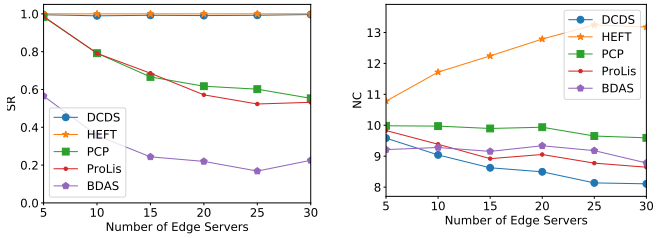
5.2 Results and Analysis

In the simulations, the performance of DCDS is compared with baselines in terms of SR and NC. Then, the time efficiency of DCDS is evaluated.

5.2.1 Offline Scenarios

In offline scenarios, there is no resource competition between tasks of different applications, and the computation resources are all available to the current application.

Fig. 5 shows the results of DCDS and four baselines in the edge-only MEC network (i.e., an MEC network only has edge servers). In this experiment, DCDS only applies the edge scheduling stage. The overall performance of DCDS is much better than baselines with different CCRs and deadline factors β. When scheduling a single task, DCDS takes into account its output data and servers' bandwidth, which alleviates the lateness of the following tasks and thus achieves a higher success rate. In addition, by scheduling the current task and the following tasks in a balanced way, DCDS preserves more time than other baselines to select cheaper servers for the following tasks. Thus, the normalized execution cost is also reduced. The improvement of DCDS becomes smaller with smaller CCR since the shorter transmission time causes less lateness of baselines. In particular, when $CCR = 0.1$ and $\beta = 1$, DCDS's SR is lower than PCP. The reason is that compared with other deadline distribution based baselines, PCP tends to set tighter sub-deadlines for individual tasks, which also results in higher execution costs. The HEFT schedules tasks to minimize the makespan without considering execution costs, and the basic deadline is set to the makespan of HEFT (i.e., in Eq. (30)). Therefore, HEFT's NC is the highest, and its SR is 1. PCP and ProLis have close performance, for they only have a slight difference in the concrete implementation of deadline distribution. ProLis tends to set as late sub-deadlines to tasks as possible and PCP sets tighter sub-deadlines, so the SR of PCP is slightly higher than ProLis in most cases, and

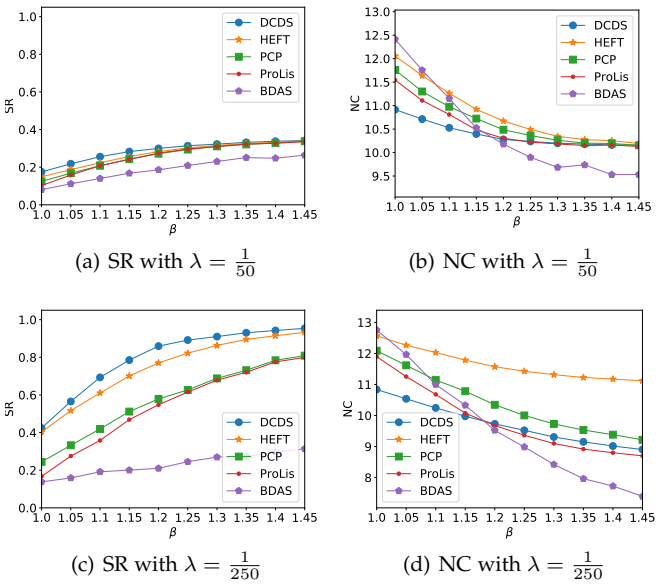


(a) SR under different numbers of edge servers (b) NC under different numbers of edge servers

Fig. 8. SR and NC of different algorithms with different numbers of edge servers.

the NC of PCP is higher than ProLis. BDAS has the worst performance due to the level-based deadline distribution method, which is specifically designed for looser deadlines and scalable resources in cloud computing instead of the tight application deadlines and the resource-limited MEC network.

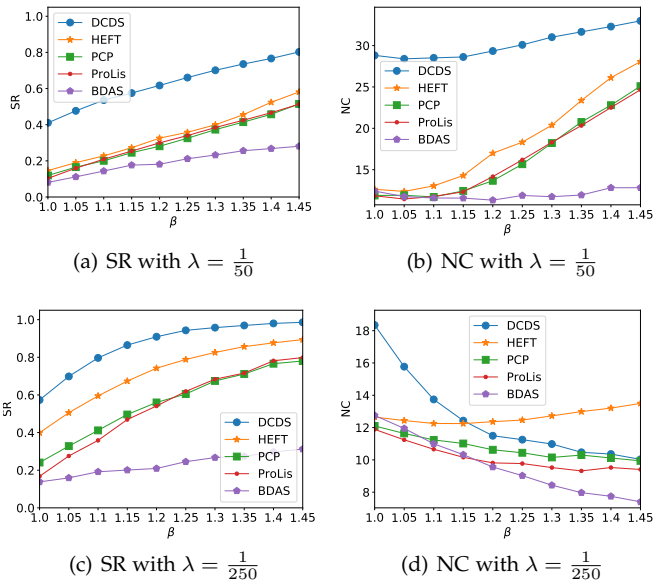
In the experiment of Fig. 6, applications are scheduled in a generic MEC network (i.e., an MEC network including edge servers and the remote cloud). In this scenario, both the edge scheduling stage and the cloud offloading stage are called. Similar to the above experiment, the overall performance of DCDS is still much better than baselines. When $CCR = 0.1$, the performance of three deadline distribution based algorithms is unstable, for they only focus on the current task's finish time when making the scheduling decision. In this way, offloading tasks to the cloud brings additional latency when transferring data back to the edge, which results in the lateness of the entire application. When $CCR = 1$, since the long transmission time of the edge-cloud link, and the high execution costs of the cloud, all algorithms prefer to select edge servers (i.e., the NC of $CCR = 1$ is much lower than $CCR = 0.1$). Therefore, the simulation results are close to Fig. 5(c) and Fig. 5(d), and



(a) SR with $\lambda = \frac{1}{50}$ (b) NC with $\lambda = \frac{1}{50}$

(c) SR with $\lambda = \frac{1}{250}$ (d) NC with $\lambda = \frac{1}{250}$

Fig. 9. SR and NC in online and edge-only scenarios with $\lambda = \frac{1}{50}, \frac{1}{250}$.



(a) SR with $\lambda = \frac{1}{50}$ (b) NC with $\lambda = \frac{1}{50}$

(c) SR with $\lambda = \frac{1}{250}$ (d) NC with $\lambda = \frac{1}{250}$

Fig. 10. SR and NC in online and generic scenarios with $\lambda = \frac{1}{50}, \frac{1}{250}$.

cloud resources are underutilized.

Impact of the transmission time between edge and cloud. The performance of different algorithms is compared under different transmission times per unit data of the edge-cloud link, as shown in Fig. 7. In this experiment, we first fix $CCR = 0.5$ and set the transmission time $d_r = \{2\overline{d}, 8\overline{d}\}$, where $\overline{d}$ is the average time per unit data transmission between edge servers. With shorter transmission time, i.e., $d_r = 2\overline{d}$, algorithms tend to schedule tasks to the cloud, resulting in a higher NC. However, deadline distribution based algorithms do not explicitly consider the significant data transmission time from the cloud back to edge servers. The completion times of applications are more likely later than deadlines, and thus the SR of these algorithms is unstable. With longer transmission time d_r, i.e., $d_r = 8\overline{d}$, the results are close to Fig. 5(c) and Fig. 5(d) for algorithms prefer to select edge servers.

Impact of the number of edge servers. The performance of different algorithms is compared under different numbers of edge servers, as shown in Fig. 8. In this experiment, we fix $CCR = 0.5$ and $\beta = 1.2$. It can be observed that DCDS consistently outperforms the four baselines in terms of SR and NC. The performance of PCP, ProLis, and BDAS is unstable with varying numbers of edge servers since the base deadline M_j is determined by HEFT and different configurations of the MEC network. These baselines schedule a single task only to meet the sub-deadline, so they can not guarantee stable performance.

5.2.2 Online Scenarios

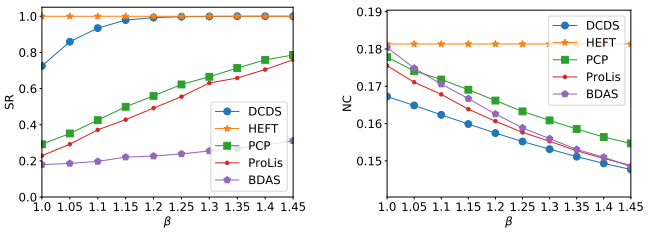
In online scenarios, applications released at different times share the computation resources of edge servers and the cloud. Therefore, the SR of all algorithms in online scenarios is lower than in offline scenarios. In Fig. 9 and Fig. 10, DCDS still outperforms other baselines in terms of SR. We detailly analyze the simulation results in the following.

In Fig. 9, applications are scheduled in an edge-only MEC network. We observe that the performance advance of DCDS is more significant with tighter deadlines. The reason is that DCDS selects suitable servers for application tasks with different importance: fast servers for urgent tasks and slow servers for other tasks. Thus, the edge servers' resource utilization of DCDS is higher than baselines under tight deadlines. When deadlines are looser, the limited computation resources of edge servers gradually become the bottleneck. Comparing the experiment results of $\lambda = \frac{1}{50}$ and $\lambda = \frac{1}{250}$, the SR of scheduling algorithms is lower when the workload is more intensive (i.e., larger λ means smaller application release intervals). When edge servers cannot afford such an intensive workload, scheduling results gradually converge to the limitation of computation resources.

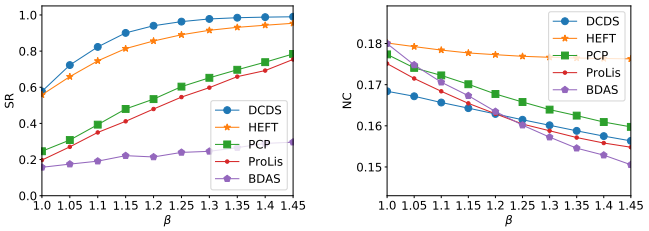
In Fig. 10, applications are scheduled in a generic MEC network. When $\lambda = \frac{1}{50}$, both SR and NC of DCDS are much higher than baselines since DCDS efficiently utilizes the expensive computation resources in the cloud to complete more applications before their deadlines. The reason is that DCDS computes each task's remaining execution time in the cloud, while other algorithms only focus on a single task's finish time. This short-sighted feature makes these baselines tend to schedule tasks on edge servers to avoid the considerable transmission latency of the current task, which leads to underutilized cloud resources. Naturally, using more cloud resources results in a higher NC of DCDS than baselines since its higher cost. When $\lambda = \frac{1}{250}$ (i.e., less intensive workload), the NC of DCDS becomes much lower than $\lambda = \frac{1}{50}$ since a larger proportion of tasks can be executed on cheap edge servers.

5.2.3 Different Server Configurations

In this section, the server configuration of [16] is applied to further evaluate the performance of DCDS. Different types of servers are listed in TABLE 4. In this configuration, edge servers have a wide range of processing capabilities and relatively close execution costs. Since the execution time per unit workload of servers in this experiment are shorter than the TABLE 3, the λ of online scenarios is set to $\frac{1}{20}$ (i.e.,



(a) SR of offline scheduling (b) NC of offline scheduling



(c) SR of online scheduling (d) NC of online scheduling

Fig. 11. SR and NC of offline and online scheduling.

TABLE 4
Server Configurations of [16]

Server type	p^*	c^{**}	Server type	p^*	c^{**}
Edge Server 1	1	0.12	*Edge Server* 6	0.29	0.17
Edge Server 2	0.67	0.13	*Edge Server* 7	0.25	0.18
Edge Server 3	0.5	0.14	*Edge Server* 8	0.22	0.19
Edge Server 4	0.4	0.15	*Cloud Server*	0.2	0.2
Edge Server 5	0.33	0.16			

* p is the execution time per unit workload.
** c is the execution cost per unit workload.

shorter release interval). The experiment results of offline and online scheduling are shown in Fig. 11. Similar to the results of the above experiments, DCDS consistently outperforms four baselines under the edge server configurations of [16]. In the offline scenario, the NC of DCDS is lower than other baselines with different β. Besides, the reduction of NC under the configurations of TABLE 4 is less than TABLE 3 for the execution costs are closer between servers with different processing capabilities in TABLE 4.

5.2.4 Runtime Comparison

The scheduling algorithm runtime is critical due to the tight deadline requirements of mobile applications. The time efficiency of DCDS and baselines is compared.

Montage workflows with the three different sizes are generated: Montage_25 (i.e., a Montage workflow with 25 tasks), Montage_50, Montage_100. The experiment is repeated 500 times under the default settings of the offline scenario to get the average runtime, and the results are shown in TABLE 5. Since the task graphs of mobile applications are usually sparse, i.e., the number of edges is close to the number of nodes, the runtime of all algorithms is nearly in proportion to the task number. The runtimes of the four baselines are at the same level, for they all focus on the finish time computation of each task on each server. The runtime of DCDS is longer than other algorithms since the estimated earliest start times of each task's immediate successors are computed in the edge scheduling stage. Besides, in the cloud offloading stage, whether every task can run on every server is tested, and in each test, the task's predecessors and successors are taken into consideration. Though, the runtime of DCDS is still at an affordable level.

TABLE 5
Runtime (s) comparison with different sizes of montage workflows for DCDS and baselines

Algorithms	Montage_25	Montage_50	Montage_100
DCDS	0.021	0.045	0.096
HEFT	0.007	0.015	0.032
PCP	0.007	0.016	0.033
ProLis	0.007	0.016	0.032
BDAS	0.008	0.016	0.032

TABLE 6
Runtime (s) comparison with different numbers of edge servers for DCDS and baselines

Number of Edge Servers	10	20	30	40	50
DCDS	0.022	0.041	0.059	0.078	0.097
HEFT	0.008	0.015	0.022	0.028	0.035
PCP	0.007	0.015	0.021	0.028	0.034
ProLis	0.008	0.014	0.021	0.028	0.034
BDAS	0.008	0.014	0.022	0.028	0.035

In TABLE 6, the runtimes for Montage_25 of different algorithms are compared over different edge server numbers, $|\mathbf{S}| = \{10, 20, 30, 40, 50\}$. Similarly, the runtimes of the four baselines are at the same level, and DCDS has the longest runtime. The runtime increases linearly with the number of edge servers, which is in line with the time complexity of DCDS analyzed in Section 4.3.

6 Discussion

6.1 Fault Tolerance

Fault tolerance is another critical feature of the dependent task scheduling problem [45], [46]. Hardware and software failures may increase the possibility of mobile application lateness. The unexpected delay of data communication and task execution due to the dynamics of MEC is thought of as a type of failure (i.e., the execution result is not available in time) [45]. Fault tolerance can be considered in DCDS by adding task replications or choosing faster servers for tasks to leave more slacks for rescheduling [46]. However, both of these solutions will encounter the following challenging issues: (1) They inevitably lead to higher execution costs [45]. (2) They will occupy more computation resources [46], which may result in the lateness of the following tasks. (3) The heterogeneity, dynamics, and resource limitation of edge nodes in MEC need to be comprehensively considered, resulting in a much more complex system model. These challenges make the fault-tolerant scheduling of dependent tasks with tight deadlines much intractable. In this paper, we focus on latency-sensitive dependent task scheduling without considering server failures and leave further studies about fault tolerance as future work.

6.2 Link Capacity

Some studies [47], [48] on dependent task scheduling additionally consider the link/path capacity constraint or

network resource contention. In their network models, the data transmission of different tasks must share the network bandwidth [48] or the link bandwidth [47], which means that a task's transmission must wait for other tasks' on the same link. In this paper, to incorporate the link/path capacity constraint, we can directly modify the EST computation in Eq. (14) and EST estimation in Eq. (18) by allocating idle time intervals of the link resource on dependent data transmissions. After scheduling each task, the corresponding time intervals of the link resource should be labeled as "busy".

However, the problems of allocating link resources and scheduling dependent tasks are coupled with each other. In addition, the network routing and background traffic should also be taken into consideration. To achieve higher performance, the scheduling decisions should be made by jointly considering these aspects. As a result, the scheduling problem will be more complicated. In this paper, we focus on dependent task scheduling with tight deadline constraints and leave the incorporation of link resource scheduling and network routing as future work.

6.3 AI Applications

AI applications are emerging mobile applications that can provide powerful capabilities. They are divided into inference applications and training applications. For inference applications based on neural networks, the entire neural network can be partitioned into multiple parts executed on different devices to achieve lower latency [49]. The neural network can also be modeled as a DAG [50], where each vertice represents a layer of the neural network. The modeling is akin to the task graph in this paper, so DCDS can still be applied to schedule these applications. Moreover, these applications typically need more Python libraries and large amounts of trained neural network parameters, whose size is large. The runtime initialization process and storage capacity should be carefully considered. A neural network layer requires much more computation resources than traditional dependent tasks, so the set of candidate servers is different for diverse layers. For training applications based on neural networks, e.g., Federated Learning [51], the communication mechanism and the training algorithm should be further improved for heterogeneous MEC networks. We will take these unique features into consideration and design more suitable algorithms in the future.

6.4 Server Sharing

Generally, MEC is multi-tenant, where edge servers are shared by dependent tasks and other applications. In this paper, edge servers are assumed to be unary, so each server executes one task at a time [17]. Edge servers are shared in a time-multiplexing manner and other applications can occupy computation resources in some time slots. To deal with the problem of server sharing, the following modifications can be applied to this paper: (1) Before scheduling a task graph, the scheduler collects the information of the time slots reserved for other applications. (2) The idle time slot list of each server in Eq. (16) is updated by removing these reserved time slots. (3) After the schedule plan is generated, the execution time slot of each dependent task is reserved and not allowed to be used for other applications.

However, when multiple tasks are allowed to be executed simultaneously on an edge server, these tasks compete for shared resources such as CPU cache, network and memory bandwidth. Thus, a complicated performance interference model of multiple running tasks should be built [52]. In the future, we will study the problem of dependent task scheduling incorporated with performance interference.

7 Conclusion

In this paper, we formulate the deadline-constrained cost optimization problem for dependent task scheduling in heterogeneous MEC. We design DCDS by considering the future impacts of current scheduling decisions. DCDS has two stages: (1) In the edge scheduling stage, the latest start time is assigned to each task, and the current task is scheduled to the cheapest edge server to satisfy the latest start times of its immediate successor tasks. (2) In the cloud offloading stage, to deal with the long transmission time of the edge-cloud link, the remaining execution time in the cloud is computed for each task, and schedule plans are efficiently modified by offloading multiple tasks to cloud servers. Simulation results based on well-known real-world applications show the substantial performance advantage of DCDS over baselines in both online and offline scenarios.

Acknowledgments

This work is supported by Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC), Zhuhai, Project No. 2020KSYS007, Chinese National Research Fund (NSFC) Project No. 61872239; The Institute of Artificial Intelligence and Future Networks funded by Beijing Normal University (Zhuhai) Guangdong, China; Zhuhai Science-Tech Innovation Bureau, Nos. ZH22017001210119PWC and 28712217900001 and Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University Zhuhai.

References

[1] "Mobile-edge computing – introductory technical white paper," [EB/OL], https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf Accessed Sep. 2, 2021.

[2] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[3] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[4] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein, "Will serverless computing revolutionize nfv?" *Proceedings of the IEEE*, vol. 107, no. 4, pp. 667–678, 2019.

[5] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 68–81.

[6] "Openalpr automatic license plate recognition," [EB/OL], https://www.openalpr.com/ Accessed Spe. 6, 2021.

[7] B. Sonkoly, J. Czentye, M. Szalay, B. Németh, and L. Toka, "Survey on placement methods in the edge and beyond," *IEEE Communications Surveys & Tutorials*, 2021.

[8] 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 *2012 Proceedings IEEE Infocom*. IEEE, 2012, pp. 945–953.

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

[10] A. Al-Shuwaili and O. Simeone, "Energy-efficient resource allocation for mobile edge computing-based augmented reality applications," *IEEE Wireless Communications Letters*, vol. 6, no. 3, pp. 398–401, 2017.

[11] Z. Zhu, G. Zhang, M. Li, and X. Liu, "Evolutionary multi-objective workflow scheduling in cloud," *IEEE Transactions on parallel and distributed Systems*, vol. 27, no. 5, pp. 1344–1357, 2015.

[12] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioningand scheduling algorithm for scientific workflows on clouds," *IEEE transactions on cloud computing*, vol. 2, no. 2, pp. 222–235, 2014.

[13] Z.-G. Chen, Z.-H. Zhan, H.-H. Li, K.-J. Du, J.-H. Zhong, Y. W. Foo, Y. Li, and J. Zhang, "Deadline constrained cloud computing resources scheduling through an ant colony system approach," in *2015 international conference on cloud computing research and innovation (ICCCRI)*. IEEE, 2015, pp. 112–119.

[14] J. Yu, R. Buyya, and C. K. Tham, "Cost-based scheduling of scientific workflow applications on utility grids," in *First International Conference on e-Science and Grid Computing (e-Science'05)*. Ieee, 2005, pp. 8–pp.

[15] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[16] Q. Wu, F. Ishikawa, Q. Zhu, Y. Xia, and J. Wen, "Deadline-constrained cost optimization approaches for workflow scheduling in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3401–3412, 2017.

[17] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 37–45.

[18] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira, "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *2010 18th Euromicro conference on parallel, distributed and network-based processing*. IEEE, 2010, pp. 27–34.

[19] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2013.

[20] H. Djigal, J. Feng, J. Lu, and J. Ge, "Ippts: an efficient algorithm for scientific workflow scheduling in heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1057–1071, 2020.

[21] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Cost-driven scheduling of grid workflows using partial critical paths," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1400–1414, 2011.

[22] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in cloud: a survey," *The Journal of Supercomputing*, vol. 71, no. 9, pp. 3373–3418, 2015.

[23] Q. Wu, M. Zhou, Q. Zhu, Y. Xia, and J. Wen, "Moels: Multiobjective evolutionary list scheduling for cloud workflows," *IEEE Transactions on Automation Science and Engineering*, vol. 17, no. 1, pp. 166–176, 2019.

[24] V. Arabnejad, K. Bubendorfer, and B. Ng, "Budget and deadline aware e-science workflow scheduling in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 29–44, 2018.

[25] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010, pp. 49–62.

[26] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 301–314.

[27] M. Jia, J. Cao, and L. Yang, "Heuristic offloading of concurrent tasks for computation-intensive applications in mobile cloud computing," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 352–357.

[28] S. E. Mahmoodi, R. Uma, and K. Subbalakshmi, "Optimal joint scheduling and cloud offloading for mobile applications," *IEEE Transactions on Cloud Computing*, vol. 7, no. 2, pp. 301–313, 2016.

[29] IBM, "Ibm cplex optimizer," [EB/OL], https://www.ibm.com/analytics/cplex-optimizer Accessed Aug 24, 2021.

[30] 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 *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE, 2019, pp. 1–10.

[31] Z. Tang, J. Lou, F. Zhang, and W. Jia, "Dependent task offloading for multiple jobs in edge computing," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–9.

[32] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," *IEEE Communications Surveys & Tutorials*, 2021.

[33] L. Li, T. Q. Quek, J. Ren, H. H. Yang, Z. Chen, and Y. Zhang, "An incentive-aware job offloading control framework for multi-access edge computing," *IEEE Transactions on Mobile Computing*, vol. 20, no. 1, pp. 63–75, 2019.

[34] 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 *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2287–2295.

[35] A. C. Baktir, A. Ozgovde, and C. Ersoy, "How can edge computing benefit from software-defined networking: A survey, use cases, and future directions," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2359–2391, 2017.

[36] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[37] C.-C. Hsu, K.-C. Huang, and F.-J. Wang, "Online scheduling of workflow applications in grid environments," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 860–870, 2011.

[38] W. Chen, Y. C. Lee, A. Fekete, and A. Y. Zomaya, "Adaptive multiple-workflow scheduling with task rearrangement," *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1297–1317, 2015.

[39] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[40] W. Zhang, Y. Wen, and D. O. Wu, "Collaborative task execution in mobile cloud computing under a stochastic wireless channel," *IEEE Transactions on Wireless Communications*, vol. 14, no. 1, pp. 81–93, 2014.

[41] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[42] J. Yi, S. Choi, and Y. Lee, "Eagleeye: Wearable camera-based person identification in crowded urban spaces," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.

[43] "Workflow generator," [EB/OL], https://confluence.pegasus.isi.edu/display/pegasus/Deprecated+Workflow+Generator Accessed Spe. 6, 2021.

[44] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future generation computer systems*, vol. 29, no. 3, pp. 682–692, 2013.

[45] G. Yao, Y. Ding, and K. Hao, "Using imbalance characteristic for fault-tolerant workflow scheduling in cloud systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3671–3683, 2017.

[46] G. Yao, Q. Ren, X. Li, S. Zhao, and R. Ruiz, "A hybrid fault-tolerant scheduling for deadline-constrained tasks in cloud systems," *IEEE Transactions on Services Computing*, 2020.

[47] S. Wang, W. Chen, X. Zhou, L. Zhang, and Y. Wang, "Dependency-aware network adaptive scheduling of data-intensive parallel jobs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 515–529, 2018.

[48] F. Giroire, N. Huin, A. Tomassilli, and S. Pérennes, "When network matters: Data center scheduling with network tasks," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2278–2286.

[49] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the

cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[50] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Di Francesco, "Distributed inference acceleration with adaptive dnn partitioning and offloading," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 854–863.

[51] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "Adaptive federated learning in resource constrained edge computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[52] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to schedule long-running applications in shared container clusters at scale," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–17.

Jiong Lou received the B.S. degree from Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, in 2016 and is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include edge computing, resource allocation, and dependent task scheduling.

Zhiqing Tang received the B.S. degree from School of Communication and Information Engineering, University of Electronic Science and Technology of China, China, in 2015 and is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include edge computing, resource allocation, and reinforcement learning.

Songli Zhang received the B.S. degree from Department of Electrical Engineering, Shandong University, China, in 2019 and is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His current research interests include network function virtualization, mobile 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 State Kay Laboratory of Internet of Things for Smart City at 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 over 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 of IEEE and the Distinguished Member of CCF.

Wei Zhao (Fellow, IEEE) completed his undergraduate studies in physics at Shaanxi Normal University, China, in 1977, and received his MSc and PhD degrees in Computer and Information Sciences at the University of Massachusetts at Amherst in 1983 and 1986, respectively. Professor Zhao has served important leadership roles in academic including the Chief Research Officer at the American University of Sharjah, the Chair of Academic Council at 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. 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. Professor Zhao was awarded the Lifelong Achievement Award by the Chinese Association of Science and Technology in 2005.

Jie Li (M'95, SM'01) received the B.E. degree in computer science from Zhejiang University, Hangzhou, China, the M.E. degree in electronic engineering and communication systems from China Academy of Posts and Telecommunications, Beijing, China. He received the Dr. 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 SJTU Blockchain Research Centre,Shanghai Jiao Tong University (SJTU), 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.