Dynamic Caching Dependency-Aware Task Offloading in Mobile Edge Computing

Liang Zhao, Zijia Zhao, Ammar Hawbani, Zhi Liu, Zhiyuan Tan, Keping Yu

Abstract—Mobile Edge Computing (MEC) is a distributed computing paradigm that provides computing capabilities at the periphery of mobile cellular networks. This architecture empowers Mobile Users (MUs) to offload computation-intensive applications to large-scale computing nodes near the edge side, reducing application latency for MUs. The resource allocation and task offloading in MEC has been widely studied. However, the burgeoning complexity inherent to modern applications, often represented as Directed Acyclic Graphs (DAGs) comprising a multitude of subtasks with interdependencies, poses huge challenges for application offloading and resource allocation. Meanwhile, previous work has neglected the impact of edge caching on the offloading execution of dependent tasks. Therefore, this paper introduces a novel dynamic caching dependency-aware task offloading (CachOf) scheme. First, to effectively enhance the rationality of cache and computing resource allocation, we develop a subtask priority computation scheme based on DAG dependencies. This scheme includes the execution sequence priority of subtasks on a single MU and the offloading sequence priority of subtasks from multiple MUs. Second, a dynamic caching scheme, designed to cater to dependent tasks, is proposed. This caching approach can not only assist offloading decisions, but also contribute to load balancing by harmonizing caching resources among edge servers. Finally, based on the task prioritization results and caching results, this paper presents a Deep Reinforcement Learning (DRL)-based offloading scheme to judiciously allocate resources and improve the execution efficiency of applications. Extensive simulation experiments demonstrate that CachOf outperforms other baseline schemes, achieving improved execution efficiency for applications.

Index Terms—Mobile Edge Computing, Dependency Application, Resource Allocation, Task Offloading, Dynamic Caching, Deep Reinforcement Learning.

1 Introduction

With the rapid evolution of the Internet of Things (IoT) and 6G mobile technologies, more and more computation-intensive and latency-sensitive applications emerge, such as Virtual Reality (VR), Augmented Reality (AR) and mobile gaming, posing a formidable challenge for IoT devices with limited computing and storage resources [1]. Traditional cloud computing struggles to meet the needs of these applications due to unpredictable transmission costs [2]. Mobile Edge Computing (MEC) pushes computing resources from cloud centers to the edge of the radio access network, allowing Mobile Users (MUs) to execute applications directly on nearby edge servers, improving Quality of Service (QoS) [3]. Although MEC presents a promising alternative to traditional cloud computing, it is imperative to acknowledge that MEC is complex and offloading operations do not necessarily lead to superior system performance [4]. The intricacies of the MEC environment are multi-faceted. First, the non-uniform distribution of MUs often leads to overloading of some edge servers, while others are underutilized [5]. Second, there is a discrepancy between the abundance of user service requests and the limited computational resources of edge servers. In addition, in the MEC framework, the poorer radio link states may result in higher offloading costs [6]. Therefore, designing rational offloading algorithms to cope with the complex MEC environment is a formidable undertaking.

Existing studies on application offloading in MEC system are currently categorized into two ways, coarse-grained scheduling and fine-grained scheduling [7]. The coarse-grained scheduling operates at the application level. However, modern applications are increasingly fragmented into interdependent subtasks, and traditional coarse-grained scheduling struggles to meet the evolving user requirements [8]. The fine-grained scheduling approach models applications as Directed Acyclic Graphs (DAGs) with interdependencies. For example, in the data analysis pipeline, one application can be divided into multiple dependent subtasks, such as data preprocessing, feature extraction, statistical analysis, and model training. Each subtask needs to be executed in dependency order, and its delay directly affects the real-time performance of the subsequent tasks [9]. Although some studies have addressed the order of subtasks execution, existing sequencing schemes are often simple, focusing primarily on the offloading priority of subtasks within a single application and then sequential execution [10]. However, they neglect parallel processing of subtasks within the same application, e.g., statistical analysis and model training are subtasks that can be processed in parallel in the above example because they do not

- *Liang Zhao, Zijia Zhao, and Ammar Hawbani are with the School of Computer Science, Shenyang Aerospace University, Shenyang 110136, China. (e-mail: lzhao@sau.edu.cn, zijiazhaostu@163.com, anmande@ustc.edu.cn).*
- *Zhi Liu is with the Department of Computer and Network Engineering, The University of Electro-Communications, Tokyo, Japan. (e-mail: liu@ieee.org).*
- *Zhiyuan Tan is with the School of Computing, Engineering and the Built Environment, Edinburgh Napier University, UK. (e-mail: z.tan@napier.ac.uk).*
- *Keping Yu is with the Graduate School of Science and Engineering, Hosei University, Tokyo 184-8584, Japan. (email: keping.yu@ieee.org)*
- *Ammar Hawbani is the corresponding author.*

directly depend on the results of each other. They also fail to consider resource competition among subtasks of multiple applications. As a result, the challenge lies in effectively handling subtask dependencies to maximize the utilization of computation resources.

Meanwhile, edge content caching is proposed with the aim of further enhancing the computation performance of MUs by pre-caching the content frequently needed by offloaded tasks on edge servers to effectively assist task offloading decisions [11]. However, current research on edge caching still faces many challenges. First, the caching resources of edge servers are limited, and it is useful to implement coordination among different servers in caching schemes, but this aspect is usually ignored by current researches [12]. In addition, most existing schemes adopt static caching schemes that prioritize cache high popularity content based on the popularity of historical offloading tasks [13]. That is, the cached content on edge servers remains unchanged and infrequently updated over long periods of time, which does not achieve a relatively fair effect on the popularity ranking results and at the same time exhibits deficiencies in complex MEC environments. Therefore, there is an urgent need to design a dynamic caching scheme that can make more frequent cache adjustments according to the actual needs of the offloading task to achieve better assisted offloading effects and thus reduce latency.

In fact, offloading and caching are complementary, and their effective combination can significantly improve the performance of MEC system [14]. Only the content that are frequently reused by the offloaded tasks can be successfully cached. Conversely, if the content required by the offloaded tasks has been cached, user latency and energy consumption can be effectively reduced [15]. Given the synergistic benefits of the combination of offloading and caching, some schemes have emerged in the field of MEC, however there are still some issues. First, existing static caching schemes are difficult to flexibly adapt to uncertain user service requests in MEC [16]. Second, most schemes are based on the assumption of unlimited computational and storage resources for edge servers, which is significantly unrealistic to the real situation [17]. Finally, there is a lack of a caching scheme specialized for DAG-dependent tasks to effectively assist the offloading of such tasks [18]. Therefore, it is necessary to design a solution that combines task offloading and dynamic caching to better meet the complex demands of DAG offloading tasks in MEC.

In light of the existing research landscape, which often addresses individual aspects in isolation, this paper endeavors to present a comprehensive approach. We focus on a persistent but frequently overlooked challenge—the tasks with dependencies. We introduce a dynamic caching dependency-aware task offloading (CachOf) scheme to minimize the computation latency of MUs in MEC. Specifically, we formulate a quantitative computation scheme for subtask priority. In addition, we propose a dynamic caching approach by comprehensively considering the popularity of task requests, the priority of subtasks, and the mutual coordination of cached resources on adjacent edge servers. Finally, we model the task offloading process as a Markov Decision Process (MDP) and design a resource allocation scheme based on Deep Reinforcement Learning (DRL) [19]. To the best of our knowledge, this is the first study to comprehensively addresses task dependency, dynamic caching, task offloading, and resource allocation in MEC. The main contributions of this paper are summarized as follows.

- To enhance resource utilization and minimize latency, we introduce a scheme for subtasks execution and offloading order. This scheme offers two key advantages: (i) the execution priority of subtasks on an individual application takes into account the dependency relationship and also improves the parallelism of subtasks processing as much as possible, and (ii) the offloading priority of subtasks across multiple applications considers edge server computing capability and the latest start time of subtasks.
- A dynamic caching scheme based on 0-1 knapsack for DAGs is proposed. This scheme is based on the results of subtasks offloading priority and dynamically caches content under the constraint of limited cache capacity, while also adjust the offloading decisions by coordinating cached content across various edge servers to achieve load balancing.
- The offloading process of dependent tasks is modeled as a MDP, and the Deep Deterministic Policy Gradient (DDPG) algorithm is utilized for offloading decision. The objective is the minimization of system delay by combining caching results and offloading priority results.

The remaining part of this paper is structured as follows. In Section 2, we describe the related work. In Section 3, the system model and problem formulation are introduced for clarity. Section 4 outlines the algorithm design and proposed solutions. Sections 5 and 6 provide the experimental analysis and draw conclusion, respectively.

2 Related Work

In this section, we first analyze the priority computation scheme, and then we delve into the task dependency-based offloading scheme in MEC. Subsequently, existing studies on cache-assisted offloading are surveyed, and finally the combined performance of these studies with our scheme is compared in Table 1.

2.1 Prioritization Scheme Based on Dependent Tasks

The execution order of tasks is significantly influenced by task dependencies. In recent years, various studies have explored methods for prioritizing dependent tasks. Liu *et al.* [20] accurately prioritize all tasks based on their expected completion time and efficiently assign tasks to appropriate servers. Zhao *et al.* [21] employ the earliest start time of a task as their prioritization criterion, aiming to optimize the earliest completion time of subtasks; thus, tasks with smaller earliest start times are granted higher priority. Liu *et al.* [22] design a dynamic downward sorting mechanism and can dynamically adjust the prioritization result according to the resource distribution of the surrounding vehicles. Sahni *et al.* [23] adopt a holistic approach, considering both the earliest start time of a task and the start time of a network flow, to design a priority calculation method for task scheduling based on the priority list order.

TABLE 1
COMPARISON OF SOLUTIONS FOR RELATED WORK

Ref.	Algorithm	Offload	Caching	Considered Factors			Optimization Objective
				Dependency	RSU Collaboration	Limited C2 Resource	
[20]	COFE	Partial	NA	✓	✗	✗	Minimize the average makespan.
[21]	FS	Partial	NA	✓	✗	✗	Minimize overall completion time.
[22]	RFID	Partial	NA	✓	✗	✗	Minimize overall completion time.
[23]	JDOFH	Partial	NA	✓	✗	✗	Minimize the average delay.
[24]	DAAS	Partial	NA	✓	✗	✗	Maximize the number of tasks.
[25]	VTSPO	Partial	NA	✓	✗	✗	Minimize delay and energy.
[26]	BAFOS	Partial	NA	✓	✗	✗	Minimize the average delay.
[27]	DQNTS	Partial	NA	✓	✗	✗	Minimize the execution latency.
[16]	LRRS	Binary	Static	✗	✗	✓	Maximize the frames number.
[17]	TOSC-CF	Binary	Dynamic	✗	✗	✗	Minimize delay and energy.
[18]	0-1 ILP	Binary	Static	✗	✗	✓	Minimize energy consumption.
[28]	COOR	Binary	Static	✗	✓	✓	Minimize the cost and delay.
[29]	CGP	Binary	Dynamic	✗	✓	✗	Maximize the total QoS.
Ours	DDPG	Partial	Dynamic	✓	✓	✓	Minimize the average delay.

2.2 Offloading Scheme Based on Dependent Tasks

The impact of task dependencies on the execution latency of applications seeking offloading has garnered substantial attention in recent years. This heightened interest reflects the increasing emphasis on developing specialized offloading schemes designed to address the challenges posed by dependent tasks in MEC. Liao *et al.* [24] introduce a scheme that integrates task dependency, assignment, and scheduling, proposing an algorithm named DAAS to address this issue. Li *et al.* [25] put forward the VTSPO algorithm, which is based on PPO, to handle vehicle requests for offloading services after subtask ordering. Bi *et al.* [26] model the offloading sequence of dependent tasks as MDP and use the Q-learning algorithm to solve it. Shang *et al.* [27] conceptualize the application as a DAG, based on which a DQN-based scheduling algorithm, DQNTS, is proposed to minimize the execution delay of dependent tasks.

2.3 Service Caching Placement-assisted Offloading

Jointly optimized service caching placement and offloading solutions have been extensively studied in recent years. Service placement refers to the deployment of an entire service (e.g., an application or a functional module) to a specific edge node, and cache placement refers to storage the data or computation results needed for the tasks that frequently request service on the edge servers. Farhadi *et al.* [16] propose a dual time-scale solution to handle both service placement and task scheduling requests simultaneously. Dai *et al.* [17] introduce a dynamic caching approach and dynamically adjusted the offloading scheme based on what is cached in the server. Bi *et al.* [18] investigate the joint optimization problem involving offloading policy, cache placement and resource allocation. Li *et al.* [28] propose a scheme called CooR to resolve the collaboration problem between offloading and caching by utilizing the mutual collaboration between edge servers. Hudson *et al.* [29] propose a Federation Learning (FL)-based algorithm to dynamically predict future incoming requests from users to assist placement decisions. We summarize these literatures in Table 1, where the column "Limited C2 Resource" refers specifically to studies that consider both edge server computing and caching resource constraints. This design aims to highlight the advantages of our scheme in considering dual resource constraints in real-world scenarios, thus enhancing its utility when compared with other schemes.

2.4 Summary

Based on our analysis of existing research, we summarize their shortcomings as follows. First, most current task prioritization schemes calculate subtask priorities based on task attributes and execute them sequentially, overlooking potential parallel execution and scheduling priority. Second, many offloading schemes for dependent tasks neglect interoperability between edge servers and dynamic allocation of limited computational resources. Finally, existing joint caching placement and offloading optimization schemes typically do not account for the influence of dependencies on limited caching and offloading resources, and they often have long cache update cycles, which poses challenges in adapting to dynamic environments. In this work, we propose a dynamic cache-assisted offloading scheme for dependent tasks, which comprehensively considers the dependencies between tasks, limited resource allocation, and dynamic caching with the aim of minimizing system latency.

3 SYSTEM MODEL AND PROBLEM FORMULATION

This section introduces a generalized MEC network model considering sub-task dependencies represented as DAGs. It then details the caching and computing models for dependent tasks, followed by the optimization objective. Key notations are summarized in Table 2.

3.1 System Model

3.1.1 Network Model

As illustrated in Fig. 1, in this paper, we consider an edge network consisting N Base Stations (BSs), denoted as $B = \{b_1, b_2, \ldots, b_N\}$. Each base station b_n contains M mobile devices in its coverage area and its index is denoted as $M = \{1, 2, \ldots, M\}$, and the number of mobile users within

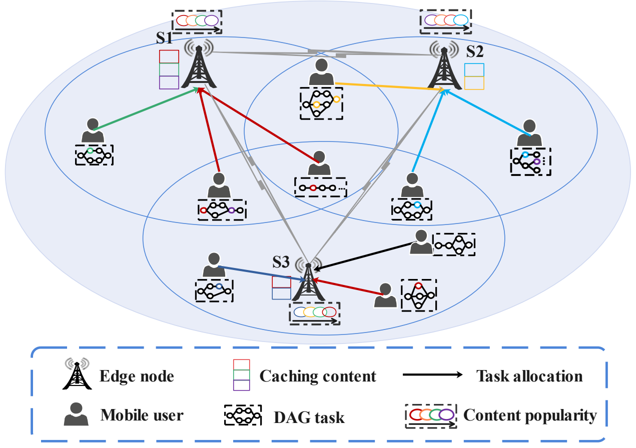

Fig. 1. An illustration of our MEC system model.

TABLE 2
KEY NOTATIONS LIST

Notation	Description
N	Number of RSUs
M	Number of MUs
C_n	Computing power of RSUs
S_n	Storage capacity of RSUs
V_m	The set of nodes of DAG
E_m	The set of edges of DAG
U_m	The set of weights of the edges of the DAG
$v_{m,i}$	A subtask of the application
$d_{m,i}$	Input data size of $v_{m,i}$
$c_{m,i}$	Required CPU cycles by $v_{m,i}$
$t_{m,i}$	Maximum acceptable latency of $v_{m,i}$
$e_{v_{m,i},v_{m,j}}$	A directed edge of DAG
$u_{v_{m,i}}^{v_{m,j}}$	Weight of a directed edge in DAG
$t_{v_{m,i}}^{trans}$	Data transmission delay before execution
$t_{v_{m,i}}^{wait}$	Total waiting delay
$EST_{v_{m,i}}$	The starting execution time of $v_{m,i}$
$EFT_{v_{m,i}}$	The end execution time of $v_{m,i}$
$t_{v_{m,i},loc}^{exe}$	The local execution delay of $v_{m,i}$
$t_{v_{m,i},off}^{exe}$	The offloading execution delay of $v_{m,i}$
$h(v_{m,i})$	The offloading priority value of $v_{m,i}$
$Q(m)$	Offloading priority sorting set of MUs
$g(v_{m,i})$	The scheduling priority value of $v_{m,i}$
$LST_{v_{m,i}}$	The latest start time of $v_{m,i}$
$LFT_{v_{m,i}}$	The latest finish time of $v_{m,i}$
p_n^{pre}	Popularity of historical offloaded content
p_n^t	Ranking of popularity of current time slot
p_n^{cache}	Cache allocation results
S	State space
A	Action space
$R(S,A)$	Reward function

each BS is different. Each base station b_n is equipped with an edge server, noting its computational and storage capacities as C_n and S_n, respectively. So that the base station can provide computational services to MUs such as cell phones, computers, and smart vehicles in its wireless coverage area. The coverage of different base stations is intersected and they are able to communicate with each other, denote the transmission rate between two edge servers as r_k^l.

3.1.2 Task Model

For simplicity, we assume that there is only one application on each MU and denote the deadline completion time for each application as T_m. These applications consist of

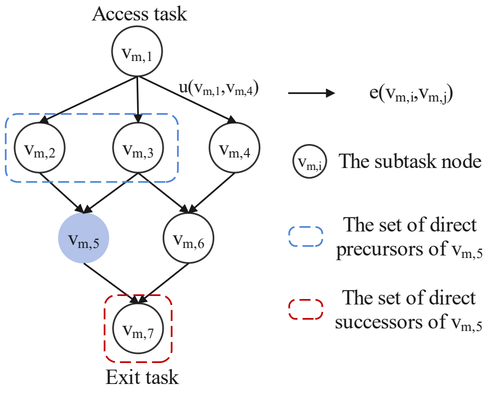

Fig. 2. Example of DAG task of the application.

subtasks with dependencies, so they can be modeled as DAGs, denoted $DAG = \{V_m, E_m, U_m\}$. As Fig. 2 shows the example of an application modeled as DAG [30]. Where $V_m = \{v_{m,1}, v_{m,2}, \ldots, v_{m,I}\}$ is the set of subtasks and each subtask is denoted as $v_{m,i} = \{d_{m,i}, c_{m,i}, t_{m,i}\}$, where $d_{m,i}$ is the size of the input data of the subtask $v_{m,i}$, $c_{m,i}$ is the number of CPU cycles required to complete the subtask $v_{m,i}$, and $t_{m,i}$ is the maximum latency tolerated by the subtask $v_{m,i}$. $E_m = \{(e_{v_{m,i},v_{m,j}}) \mid v_{m,i}, v_{m,j} \in V_m\}$ is the set of all directed edges, $(e_{v_{m,i},v_{m,j}})$ denotes that subtask $v_{m,i}$ is the direct predecessor of $v_{m,j}$, denoted as $v_{m,i} = \mathrm{pre}(v_{m,j})$, and subtask $v_{m,j}$ is the direct successor of $v_{m,i}$, denoted as $v_{m,j} = \mathrm{suc}(v_{m,i})$. Subtask $v_{m,j}$ can be executed only after subtask $v_{m,i}$ is completed. $U_m = \{u_{v_{m,i}}^{v_{m,j}} \mid v_{m,i}, v_{m,j} \in V_m\}$ is the set of weights of all directed edges set, i.e., the set of execution result sizes from the predecessor to the successor, e.g., $u_{v_{m,i}}^{v_{m,j}}$ denotes the size of the amount of data transferred to subtask $v_{m,j}$ after the computation of subtask $v_{m,i}$ is completed. If neighboring subtasks $v_{m,i}$ and $v_{m,j}$ are executed at the same location, no additional transmission time is required; if neighboring subtasks $v_{m,i}$ and $v_{m,j}$ are not executed at the same location, additional transmission time is required. Also without loss of generality, we assume that each DAG has an entry subtask v_m^{access} and an exit subtask v_m^{exit}.

3.2 Caching Model

In this paper, we consider that the cache capacity of edge server is limited, the total cache resource size is denoted as S_n, and the capacity occupied by each sub-task is denoted as s_n^i, thus subject to the following capacity constraint.

$$\sum_{i=1}^{f} s_n^i \leq S_n \tag{1}$$

where f represents the number of cached contents. Since the edge server's caching resources are limited, the edge server needs to decide what to cache.

The edge caching adopted in this paper is the content caching of computational tasks, which can reasonably utilize the resources and reduce the latency of task execution by pre-caching the data and computational results required by frequently requested tasks on the edge servers. Different from the previous approach of static caching of content with high popularity, this paper designs a dynamic caching approach. The specific process is shown in Fig. 3.

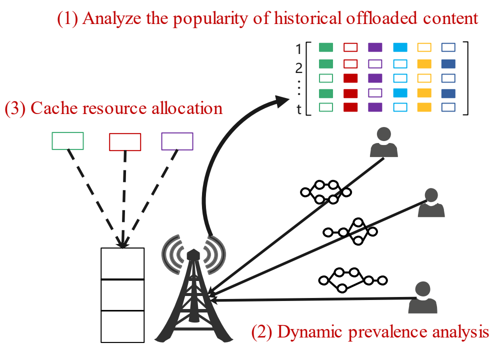
(1) Analyze the popularity of historical offloaded content
(3) Cache resource allocation
(2) Dynamic prevalence analysis

Fig. 3. The dynamic caching process.

First, b_n analyzes the tasks received over a period of time in history, calculates the number of times each type of task has been requested, and considers tasks with a high number of requests as tasks with high popularity.

Second, in each offloading cycle, the time slots are divided according to the offloading order of the subtasks, and the dynamic high popularity content is obtained in each time slot based on the result of combining the current requested offloading content with the historical high popularity content.

Then, b_n selects a more appropriate content cache based on the computed high popularity sequence and the limited storage space. Based on the caching results the offloading decision can be adjusted to decide which subtasks are offloaded and which subtasks are executed locally. When the computation results of the tasks requested to be offloaded are cached, the task execution latency can be saved.

3.3 Computing Model

In this subsection, we focus on the latency cost imposed on the system by task communication and computation after the arrival of an application service request. Given that applications can be aptly represented as DAGs, it is imperative to acknowledge the inter-dependencies among subtasks. These inter-dependencies inherently constrain the execution order. In light of this, we first model the task completion time as two parts: waiting time and execution time, on the basis of which we formulate the optimization objective. Of these, the computation of waiting time is described in the supplementary material.

Any sub-task can choose to be executed locally or offloaded to the edge server, and the completion time of the task will be different for different choices, so we discuss the local computation and offloading computation separately. First, regardless of whether a subtask is executed locally or offloaded, the earliest start time is given by (2).

$$EST_{v_{m,i}} = t_{wait,v_{m,i}} \tag{2}$$

Second, the delay for local execution of subtasks is expressed as in (3), where f_m is the local CPU frequency of the MU.

$$t^{exe}_{v_{m,i},loc} = \frac{c_{m,i}}{f_m} \tag{3}$$

Finally, the delay for offloading of subtasks is given by (4), where f_n is the CPU frequency of the edge server.

$$t^{exe}_{v_{m,i},off} = \frac{c_{m,i}}{f_n} \tag{4}$$

In summary, combining local computation, offloading computation and caching, the execution delay is denoted by (5).

$$t^{exe}_{v_{m,i}} = \begin{cases} 0, & \text{If it is cached;} \\ t^{exe}_{v_{m,i},loc}, & \text{If it is executed locally;} \\ t^{exe}_{v_{m,i},off}, & \text{If it is offloaded.} \end{cases} \tag{5}$$

Since the size of the output data is much smaller than the size of the input data after the computation is completed, the downlink delay can be ignored when computing the execution delay [31] [32]. Finally, the formula for the earliest completion time of a subtask is expressed by (6).

$$EFT_{v_{m,i}} = EST_{v_{m,i}} + t^{exe}_{v_{m,i}} \tag{6}$$

3.4 Problem Formulation

In this paper, we address the challenge of managing multiple applications with dependencies in MEC. Our approach involves designing a subtask offloading priority computation method and establishing a mechanism for resource allocation priority. These serve as the basis for formulating both caching and offloading schemes. Our objective is to minimize the system average delay while satisfying the maximum delay constraint for each task. To achieve this, we formulate the optimization problem as shown in (7). And then it is optimized based on the DRL algorithm in Section 4.3.

$$(P): \min\ \Omega = \frac{1}{M} \sum_{m=1}^{M} EFT_{v_{m,exit}} \tag{7}$$

$$s.t. \qquad EFT_{v_{m,exit}} \le T_m, \forall m \in M, \tag{7a}$$

$$EST_{v_{m,i}} \ge 0, \forall m \in M, \forall v_{m,i} \in V_m, \tag{7b}$$

$$EST_{v_{m,i}} \ge EFT_{v_{m,j}} + t^{trans}_{v_{m,j}}, \forall v_{m,i}, v_{m,j} \in V_m, \tag{7c}$$

where condition (7a) dictates that the completion delay of all applications should be less than the maximum delay constraint, otherwise it is considered to be a failure and is computed as the maximum delay tolerance in the computation of the average delay. Condition (7b) indicates that all subtasks start executing at a moment greater than or equal to 0. Condition (7c) means that each subtask initiates its execution once all of its predecessors have completed their execution and handed over the results.

4 PROPOSED SOLUTION

In this section, we focus on introducing the CachOf scheme, as depicted in Fig. 4, the primary design steps of CachOf are as follows. First, we establish a prioritization scheme for both the execution and offloading order of dependent tasks (Section 4.1). Second, after obtaining the offloading priority order, we design a dynamic caching method adapted to the dependent tasks (Section 4.2). Finally, we employ a DRL-based offloading method to achieve an optimal solution (Section 4.3).

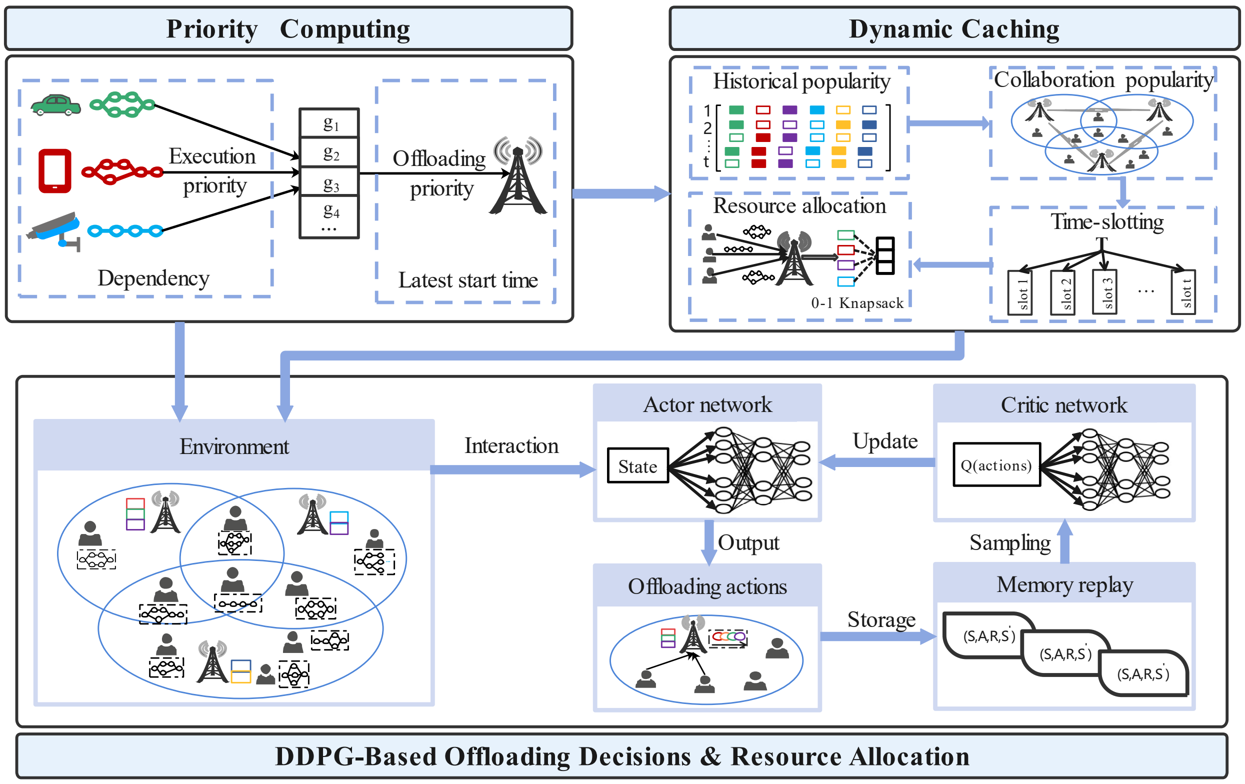

Fig. 4. A comprehensive strategy for DAG task offloading in MEC network.

4.1 Prioritization Method

There are two main parts in this subsection. First, computing the execution priority of subtasks on the same application based on the dependencies between subtasks. Second, computing the offloading priority of subtasks on different applications based on the latest start time.

4.1.1 *Execution Priority of Subtasks within Application*

In our research scenario, each application that requests services is comprised of subtasks with dependencies, which are representable as DAGs. Differing from conventional offloading priority computation methods, our approach considers not only the influence of dependencies on subtasks offloading order but also aims to optimize the utilization of local and server resources in the device, it allows for parallel execution of subtasks with mutually independent relationships in the DAG structure. As a result, we introduce a quantitative calculation method that systematically assigns offloading priorities to each subtask within the DAG. This method takes into account both the intricate dependency relationships and the possibilities of parallel computation. The specific quantitative calculation method is shown below.

If a subtask $v_{m,i}$ has no direct predecessor, set its priority value to $h(v_{m,i}) = 0$. Otherwise, its priority value formula is defined as (8).

$$h(v_{m,i}) = \max_{v_{m,j} \in pred(v_{m,i})} \{h(v_{m,j})\} + 1 \tag{8}$$

That is, subtasks with the same priority value represent that they can be executed in parallel, and subtasks with smaller priority values are given an earlier execution order. Finally each application generates a execution queue of subtasks based on the priority value.

$$Q(m) = \{h(v_{m,1}), h(v_{m,2}), ..., h(v_{m,n})\} \tag{9}$$

This means that subtasks in the queue are executed sequentially and those with the same priority value can be executed in parallel.

4.1.2 *Offloading Priority of Subtasks across Applications*

The previous researches either assume that the resources of edge servers are unlimited and all offloaded tasks can share these resources, or adopt a first-come-first-served scheduling strategy. However, when multiple subtasks request resources at the same time, some of them have to wait for the completion of other tasks to obtain resources due to limited resources. How to effectively allocate resources for task scheduling remains an urgent problem. Therefore, we propose a method for calculating the task offloading priority. The urgency of a subtask's execution is calculated by considering its latest start time. The earlier the latest start time, the greater the urgency, the higher the offloading priority, and thus the greater the urgency of caching and resource allocation. First, the latest start time of a task is computed as (10), where the larger the $LST_{v_{m,i}}$, the more urgent the task is.

$$LST_{v_{m,i}} = LFT_{v_{m,i}} - t^{\text{exe}}_{v_{m,i}} \tag{10}$$

Second, $LFT_{v_{m,i}}$ represents the latest finish time of the task, which is formulated as (11).

$$LFT_{v_{m,i}} = \min_{j \in \text{suc}(v_{m,i})} \left(LST_{v_{m,j}} - t^{\text{trans}}_{v_{m,j}} \right) \tag{11}$$

Finally, the priority value of $v_{m,i}$ is described as (12).

$$g(v_{m,i}) = LST_{v_{m,i}} \tag{12}$$

when making caching and offloading decisions, the offloading priority of the task needs to be taken into account, which is conducive to the rational allocation of resources. Specifically, in dynamic caching schemes, time slots should be allocated based on offloading priorities to ensure that subtasks with higher offloading priorities are assigned to the forefront time slots, thereby enhancing their execution efficiency. When making offloading decisions, different applications are independent of each other, and the edge server will receive requests from multiple applications. In this scenario, priority should be given to executing offloading decisions for subtasks with higher offloading priorities to maximize overall system performance.

4.2 DAG-based Dynamic Caching Scheme

In this subsection, we first build a popularity analysis scheme to optimize task offloading in DAGs by computing task caching. Second, a caching resource allocation scheme is designed to maximize cache resource utilization and load balancing by dynamically changing the cache content with limited edge server cache capacity. The detailed process is outlined in the Algorithm 1.

4.2.1 Dynamic Popularity Analysis Method

Existing caching schemes are relatively static as they mainly cache content with high historical popularity rankings and usually have longer caching update cycles than task offloading cycles. However, due to limited cache resources and dynamic offloading requests, this approach may result in unfair handling of slightly lower popularity content and is difficult to adapt to highly dynamic offloading environments. To address this challenge, we develop a dynamic popularity analysis scheme for dependent tasks. The scheme is able to subdivide the entire offloading cycle into multiple caching update cycles based on the offloading priority of the subtasks, and dynamically adjusts the content popularity in each caching cycle by considering the popularity of the historical offloaded content and the load of the edge servers.

First, the edge server calculates the popularity of historical offloaded content by tracking the number of times a specific offloaded content, denoted as p_i, has been requested for service over a defined period, represented as P_i^T in (13), where p_i^t is the number of requests for offloaded content p_i at moment t.

$$P_i^T = \sum_{t=0}^{T} p_i^t \tag{13}$$

Next, P_i^T in descending order, and its average value is set to the threshold $\overline{P_i^T}$. Any offloaded content of $P_i^T \geq \overline{P_i^T}$ is listed as a high-popularity object and saved as a set.

$$P_n^{pre} = \{p_1, p_2, ..., p_i\} \tag{14}$$

Subsequently, the system assigns tasks in the duplicate coverage area to edge servers with relatively lower loads based on the load status of neighboring edge servers. In this case, the load of the edge servers is quantified by the number of tasks currently requesting processing as well as the remaining computation resources, and their load information is broadcast periodically. This can achieve dynamic load balancing among servers, ensure reasonable distribution of tasks, and improve overall resource utilization efficiency.

Then, once the edge server acquires the offloading priority value from nearby MUs, tasks with high priority are executed first and low priority tasks are executed later, resulting in time intervals. Our caching strategy utilizes these intervals to divide the offload cycle into T small time slots. Since the offloading priority of a task represents the urgency of its execution, dividing the time according to this can be a better representation of the dynamics, and can reasonably utilize the cache resources.

Finally, at the beginning of each time slot t, each edge server compares its calculated historical popularity set P_n^{pre} with the subtasks within the region requesting offloading. Then, recalculate the number of requests for each content in the set P_n^{pre}, and record the content with high popularity as (15), where $j < i$.

$$P_n^t = \{p_1, p_2, ..., p_j\} \tag{15}$$

Update the set P_n^t at the start of each time slot by removing content that is no longer popular and adding new popular content. Compared to traditional caching methods, this method caches more content relatively fairly with limited caching resources and is more suitable for offloading tasks with dependencies.

It can be seen that the caching scheme we devised is similar to the prefetching operation in some ways, but the key difference between the two is the way they are implemented. Pre-fetching is usually based on prediction algorithms that load content that is likely to be used in advance before it is actually needed. Our proposed scheme, on the other hand, is based on actual task requests and historical high-popularity set, and dynamically adjusts the cached content by comparing the task requests in the current slot, thus ensuring faster access when subsequently needed. Thus, our scheme is more consistent with the definition of caching operations, focusing on optimizing resource utilization and improving access efficiency, rather than simply relying on prediction to load data.

4.2.2 Caching Strategy based on 0-1 Knapsack

Since the storage space of the edge server is limited and the storage space required for each content to be cached is different. How to cache as much more valuable content as possible under the limited cache resource constraints is the main issue discussed in this subsection. We adopt a strategy for cache resource allocation based on 0-1 knapsack [33]. The 0-1 knapsack problem refers to the optimization of selecting non-repeating contents, each with unique values and capacities, to achieve the maximum total value within a fixed and limited capacity.

First, the storage capacity of the edge server is S_n, and the content popularity ranking computed at each time slot t is $P_n^t = \{p_1, p_2, ..., p_j\}$. Denote the storage capacity required for each content to be cached as (16).

$$s_n^t = \{s_{p,1}, s_{p,2}, ..., s_{p,j}\} \tag{16}$$

Algorithm 1 Dynamic caching scheme

```
Input: $N$ RSUs with duplicate coverage, $M$ DAG tasks within each RSU, the cache capacity $S_n$, DAG task execution order $Q(m)$ and offloading priority value $g(v_{m,i})$
Output: Cache contents within each RSU
1: Initialize the MUs within $N$ RSUs to $N$ sets $\{f(1)\},\{f(2)\},...,\{f(N)\}$;
2: Initialize historical popularity set $P_n^{pre}$, dynamic popularity set $P_n^t$, cache set $P_n^{cache}$;
3: Initialize the content set $I$, the set of subtasks $J$ within each RSU;
4: Computing historical offloaded content popularity
5: for $i = 1$ to $I$ do
6:     if $P_i^T > \overline{P_i^T}$ then
7:         for $j = len(P_n^{pre})$ to 0 do
8:             if $p_n^{pre}[j] < P_i^T$ then
9:                 $p_n^{pre}[j+1]$=$p_n^{pre}[j]$;
10:                $j - -$;
11:            end if
12:            $p_n^{pre}[j+1]$=$P_i^T$;
13:        end for
14:    end if
15: end for
16: Edge servers collaborate with one another
17: for $n = 1$ to $N-1$ do
18:     $\eta$ = intersection$(f(n), f(n+1))$;
19:     if len$(f(n))$ < len$(f(n+1))$ then
20:         $f(n)$ = union$(f(n), \eta)$;
21:         $f(n+1)$ = difference$(f(n+1), \eta)$;
22:     else
23:         $f(n+1)$ = union$(f(n+1), \eta)$;
24:         $f(n)$ = difference$(f(n), \eta)$;
25:     end if
26:     for $j = 1$ to $J$ do
27:         $j$ = random$(I)$;
28:         $counts(j) = counts(j) + 1$;
29:     end for
30:     $P_n^t$ = sorted$(P_n^{pre}, key = counts.get, reverse = True)$;
31: end for
32: Cache resource allocation based on 0-1 knapsack
33: for $n = 1$ to $N$ do
34:     $dp = [[0] * (S_n + 1)$ for in range(len$(P_n^t) + 1)]$;
35:     for $i = 1$ to len$(P_n^t) + 1$ do
36:         for $j = 1$ to $S_n + 1$ do
37:             if $p_n^t[i-1][c] > j$ then
38:                 $dp$[i][j] = $dp$[i-1][j];
39:             else
40:                 $dp[i][j] = \max(dp[i-1][j], dp[i-1][j - P_n^t[i-1][c]] + P_n^t[i-1][v])$;
41:             end if
42:         end for
43:     end for
44:     for $i$ = len$(P_n^t), j = S_n$ to 0 do
45:         if $dp[i][j] \neq dp[i-1][j]$ then
46:             $P_n^{cache}.append(P_n^t[i-1])$;
47:             $j = j - P_n^t[i-1][c]$;
48:         end if
49:         $i = i - 1$;
50:     end for
51: end for
```

The sum of the storage capacity of the content selected for caching decision cannot exceed the total storage capacity of the edge server. The value of them is expressed as (17).

$$v_n^t = \{v_{p,1}, v_{p,2}, ..., v_{p,j}\} \tag{17}$$

Denote the popularity as its value, i.e., the number of times it has been requested in this time slot. Second, we solve this 0-1 knapsack problem using a dynamic programming approach to maximize the value of limited resources by caching more reasonable content. Instead of the traditional approach of just caching the content with the highest popularity ranking without considering the limited resources. Finally, the content of the cache within the edge server is represented as (18).

$$P_n^{cache} = \{p_1, p_2, ..., p_k\} \tag{18}$$

Algorithm 1 describes the detailed process. Of these, the detailed explanation and complexity analysis of Algorithm 1 is described in the supplementary material.

4.3 Partial Offloading Scheme based on DDPG

This paper models offloading decisions and dynamic resource allocation for tasks with dependencies as an MDP with continuous action spaces, using a model-free DRL framework for partial offloading. Unlike DQN, which is unsuitable for continuous action spaces, we apply a DDPG-based algorithm to address the DAG-based offloading problem [34]. DDPG, based on the Actor-Critic framework, effectively handles continuous actions and improves convergence through dual actor and critic networks.

4.3.1 Design of SAR

When making offloading decisions, the designed state space, action space, and reward function for the DDPG algorithm are as follows.

(1)*State space*. After the offloading and scheduling priority of the dependent tasks and the content caching of the edge servers are completed, the state of the whole system is thus determined and represented as (19), where $S_U = \{DAG_m, g(v_{m,i}), Q(m)\}$, DAG_m represents the DAG structure of each mobile user, $g(v_{m,i})$ represents the scheduling priority, $Q(m)$ represents the offloading priority of the subtasks in each user device. $S_C = \{P_n^{pre}, P_n^t, P_n^{cache}\}$, representing the historical popularity analysis results, current popularity analysis results, and the content eventually cached in the edge server, respectively. $S_V = \{C_n, S_n\}$,represent the computational capacity of the edge server and the storage capacity respectively.

$$S = \{S_U, S_C, S_V\} \tag{19}$$

(2)*Action space*. The MUs makes appropriate actions provided that the current state is known. That is, in the case of limited edge server and local computational resources, subtasks to be offloaded to the edge server and sub-tasks to be executed locally are selected to achieve rational allocation of computational resources.

$$A = \{x_{m,i}^n\} \tag{20}$$

It represents the i-th subtask of the m-th MU that chooses where to offload to for execution. Where $n \in \{0, 1, ..., n\}$

denotes offloading to be executed locally or on one of the RSUs within its RSU range, respectively.

(3)*Reward function*. In the current state S, the reward function obtained by performing a certain action A is set to R(S,A), which determines the convergence speed and the degree of convergence of the DRL algorithm, so the goal of the DRL algorithm is to maximize the reward, and in this algorithm, the reward function is set as (21).

$$R=-\sum_{m=1}^{M} EFT_{v_{m,exit}} \tag{21}$$

The reward function is associated with the earliest completion time of the last subtask, meaning that when the optimization goal is to minimize, the reward function is maximized.

Algorithm 2 CachOf offloading scheme

```
Input: Dynamic cache-optimized MEC networks
Output: Offloading decisions
 1: Initialize actor network, critic network, target actor network and target critic network with $\theta^{\mu}$, $\theta^{Q}$, $\theta^{\mu'}$, $\theta^{Q'}$;
 2: Initialize $B = 10000$, $T = 1000$, $\gamma$, and $\tau$;
 3: for episode = 1 to 1000 do
 4:     Reconfigure offloading and scheduling priority of DAGs and the output of Algorithm 1;
 5:     Note the initial state $s_0$;
 6:     for $t$ = 1 to $T$ do
 7:         Select action $a_n(t)$ from the environment;
 8:         Perform $a_n(t)$ and obtain reward $r_t$;
 9:         Acquire the next state $s_{t+1}$;
10:         if $B$ is not completely occupied then
11:             Store the entry $(s_t, a_t, r_t, s_{t+1})$ into $B$;
12:         else
13:             Randomly substitute records $(s_t, a_t, r_t, s_{t+1})$ in $B$ and sample mini-batch records from $B$;
14:             Calculate the target $Q$ value using Eq.(22);
15:             Update critic policy parameters using Eq.(23);
16:             Update actor policy parameters using Eq.(25);
17:             if $t$ mod $d$ then
18:                 Perform a soft update on the parameters of the target network using Eq.(24) and Eq.(26);
19:             end if
20:         end if
21:     end for
22: end for
```

4.3.2 *The AC Framework for the DDPG Algorithm*

The aim of the DDPG algorithm is to maximize the reward for actions made based on states by building an optimal policy model. Here, the actor network is the strategy model, which has inputs of states and outputs of actions. And the critic network is the evaluation network, whose inputs are states and actions, and whose output is an evaluation value, based on which the actor network is optimized.

(1)*Critic network*. A stochastic sample (s_t, a_t, r_t, s_{t+1}) is selected from experience pool, critic network evaluates (s_t, a_t) to obtain Q^{real}, and target critic network evaluates (s_t, a_t) to obtain Q^{target}.

$$Q^{\text{target}}=r_t+\gamma Q'\left(s_{t+1}, \mu'\left(s_{t+1} \mid \theta^{\mu'}\right) \mid \theta^{Q'}\right) \tag{22}$$

Here the neural networks with parameters Q and Q' are used to simulate the Q function and the target Q value, in order to reduce the gap between Q^{real} and Q^{target}, the loss function of the critic network is reduced and the mean square error loss function is used to update the current critic network, where N represent the size of the mini-batch.

$$Loss=\frac{1}{N}\sum_{i=1}^{N}\left(Q_i^{\text{target}}-Q\left(s_i, a_i \mid \theta^{Q}\right)\right)^2 \tag{23}$$

And the gradient ascent method is used to softly update the target critic network.

$$\theta^{Q'}=\tau\theta^{Q}+(1-\tau)\theta^{Q'} \tag{24}$$

(2)*Actor network*. The current idea of updating the actor network is to maximize the Q value obtained from the output of the actor network after entering into the critic network by optimizing the policy parameters in the actor network, where the gradient ascent method is used for updating.

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{T}\sum_{i}\nabla_a Q\left(s_i, a_i \mid \theta^{Q}\right)\nabla_{\theta^{\mu}}\mu\left(s_i \mid \theta^{\mu}\right) \tag{25}$$

Also, the gradient ascent method is used to softly update the target actor network.

$$\theta^{\mu'}=\tau\theta^{\mu}+(1-\tau)\theta^{\mu'} \tag{26}$$

The specific process is described in Algorithm 2. And, the detailed explanation and complexity analysis of Algorithm 2 is described in the supplementary material.

5 PERFORMANCE EVALUATION

In this section, we assess the performance of the proposed CachOf scheme through simulation experiments. We begin by outlining the simulation setup, including parameter settings, evaluation metrics, and constraints. Then, we introduce five comparison scenarios to analyze the performance differences between CachOf and other baseline schemes. Finally, we conduct a comprehensive performance evaluation based on these configurations.

5.1 Simulation Environment

In this subsection, based on the existing research, we perform simulation experiments by setting various parameters. The simulation platform is built using Python 3.7, and the corresponding source code and documentation can be accessed at link[1]. First, we define the target scenario to encompass ten heterogeneous RSUs, each equipped with an MEC server. Second, to simulate applications with various DAGs, we design a DAG generator to generate DAGs with different structures and numbers of subtasks. We set the latency constraint for each application to be T_m = 10 s, the data size of each subtask to be $d_{m,i}$ = (0.8, 1.2) MB, and the computational resource requirement of each subtask to be $c_{m,i}$ = (0.1, 1) G. In addition, to model the randomness of task requests in real scenarios, we assume that the arrival rate γ of user requests conforms to a Poisson distribution [35]. Next, we assume that the local computational resource

1. https://github.com/NetworkCommunication/CachOf

of the mobile user is f_m = 0.5 GHz, the computational and cache resources of the MEC server are C_n = (2, 2.8) GHz and S_n = (20, 30) MB, respectively. Regarding the parameter settings of the edge server, the parameters we used are consistent with the common settings in current MEC research [9, 11, 13, 17, 27]. This fact shows that our parameter settings are in line with the common standards in the field. Regarding the parameter settings of the application programs, our parameter settings are consistent with some of the typical vehicular application program parameters in current in-vehicle edge computing [9, 25, 26]. This further validates the reasonableness of our parameter settings and their high degree of fit with practical applications. The transmission power between the mobile user and the MEC server is P_m = 0.1 W, and the channel bandwidth is W_m = 10 MHz. Finally, in the deep reinforcement learning algorithm DDPG, we used a deep neural network containing 4 layers and chose tanh as the activation function to apply to a continuous action space. Meanwhile, in order to obtain superior system performance and faster convergence, we adjust the size of the empirical pool to 6400 and the batch size to 64. In addition, we set the learning rate of the actor network and the critic network to 0.0001 and 0.001, respectively. More detailed simulation parameters can be found in Table 3. In addition, we put in the supplementary material about the evaluation metrics and parameter constraints.

TABLE 3
MAIN PARAMETERS SETTING

Parameter	Notation	Value
Number of edge servers	N	10
Edge server computing power	C_n	$(2.0, 2.8)$ GHz
Edge server caching capability	S_n	$(20, 30)$ MB
Computing power of the local	f_m	0.5 GHz
Latency constraints	T_m	10 s
Input data of subtasks	$d_{m,i}$	$(0.8, 1.2)$ MB
CPU cycles required of subtasks	$c_{m,i}$	$(0.1, 1.0)$ G
Data upload power	P_m	0.1 W
Channel bandwidth	W_m	10 MHz
Noise power density	N_0	-174 dBm/Hz
Channel gain	h_n	$(10^{-8}, 10^{-7})$
Arrival rate of tasks	γ	$(0.4, 0.8)$
Discount factor	λ	0.99
Experience pool size	e	6400
Batch size	b	64
Learning rate	α, β	0.0001,0.001

5.2 Comparison Schemes

This paper presents a dynamic cache-assisted offloading solution for tasks with dependencies. We compare our CachOf scheme with five baseline scenarios, each addressing key aspects of the research. To ensure reliable results, all scenarios use consistent parameter settings. The evaluation highlights the significant performance advantages of CachOf.

(1)*StCach.* This scheme does not consider a dynamic caching scheme and still adopts the traditional static caching approach, i.e., it does not consider collaborative caching among edge caching servers and does not dynamically update the cached content. In contrast, our scheme demonstrates that dynamic caching can better adapt to dynamic environments by effectively utilizing limited cache resources to cache more valuable content, thereby assisting in offloading and reducing the latency of task execution.

(2)*RdmOf.* This scheme does not consider parallel processing of subtasks with dependencies and resource competition between multiple applications. The application performs offloading in order based on dependencies only, and randomly selects subtasks for scheduling when multiple subtasks experience resource preemption on the edge server. In contrast, our scheme demonstrates that the use of parallel offloading and scheduling based on the LST effectively improves the performance of computational offloading. The scheme is able to fully utilize both edge and local resources to ensure that the subtasks complete the computation as much as possible within the latency constraints.

(3)*CachDQN.* The scheme uses DQN algorithm for making offloading decisions. In contrast, we demonstrate that the application of DDPG algorithm is more applicable in the continuous action space. In the mobile edge computing offloading scheme, using the DDPG algorithm can make better offloading decisions, thus rationally utilizing computational resources and reducing computational latency.

(4)*CachGA.* This scheme employs a task offloading strategy based on genetic algorithms, and its design is built on an iterative optimization framework. In comparison, it can be demonstrated that the deep reinforcement learning-based solution outperforms the genetic algorithm-based solution in reducing the overall system latency when solving the MEC offloading problem.

(5)*G+DQN [28].* This scheme optimizes the cache replacement strategy by Gibbs sampling algorithm and optimizes the offloading decision by DQN algorithm. Under the premise of keeping the parameter settings consistent, it can be demonstrated that our scheme takes full advantage of edge and local resources and makes more intelligent offloading decisions in continuous action space through dynamic cache optimization, parallel offload scheduling, and the application of DDPG algorithm.

Meanwhile, the evaluation analyses of all the experiments described below are the average of the results of 10 independent runs with the same configuration. And in order to improve the validity of the data, the rewards, delay and offloading success rate generated per hundred episodes are averaged in this paper.

5.3 Performance Evaluation

In this section, we evaluate the performance of the proposed scheme in this paper with several other comparative schemes in terms of the total reward, average delay, and offloading success rate.

5.3.1 Evaluation of Total Reward

Fig. 5(a) shows the TR of four different schemes under 1000 episodes. To improve the reliability of the data, we use the average of the rewards per 100 episodes. As shown in the figure, the total rewards of all the schemes start to show an increasing trend over time as the number of episodes increases. This is because they are all trained using DRL methods, and the DRL algorithms continue to optimize their strategies through experience, so the total reward continues to rise. It is clear from the figure that our proposed CachOf

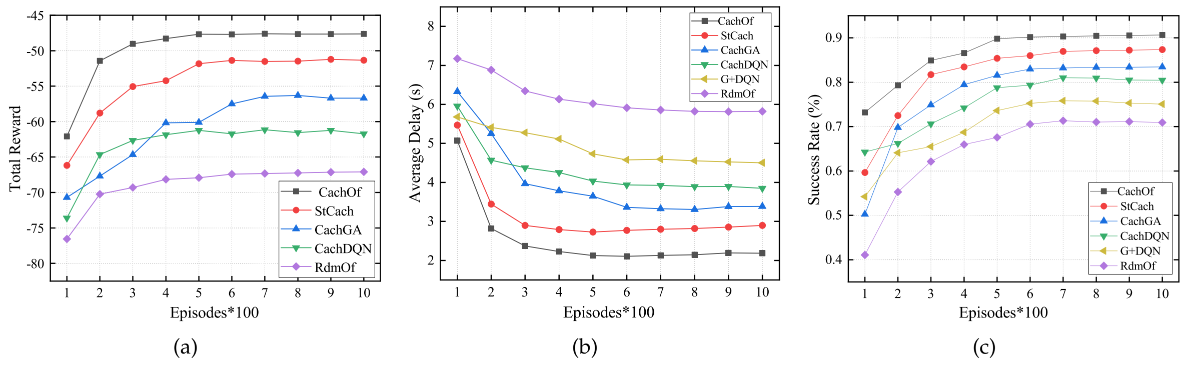

Fig. 5. Comparative analysis of distinct schemes regarding (a) TR, (b)AD, and (c) SR under various number of episodes.

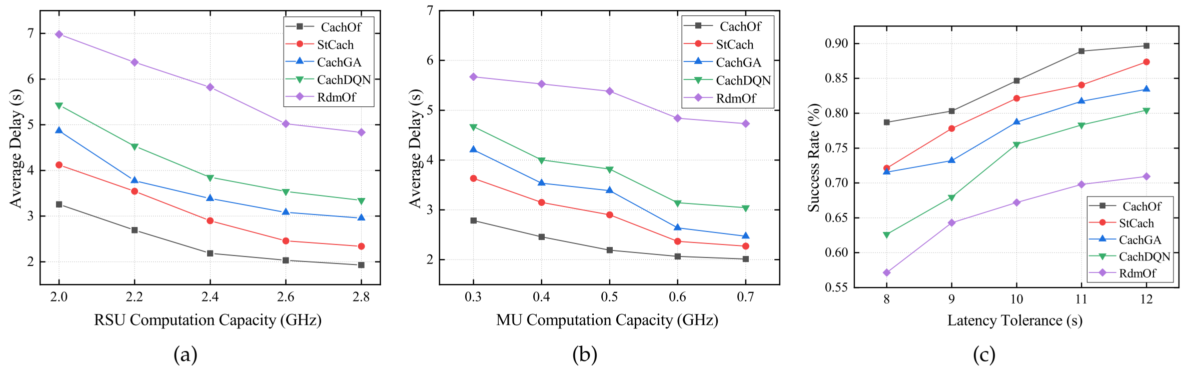

Fig. 6. (a) Total delay under different RSU computation capacity. (b) Total delay under different MU computation capacity. (c) Success rate under different latency tolerance.

scheme consistently outperforms the other schemes in terms of total reward and tends to converge at the 500th episode, whereas the other schemes start to converge at about the 600th episode. This is because the CachOf scheme not only optimizes task offloading priority, but also adopts a dynamic caching strategy, i.e., it focuses on the optimization of the whole MEC network scenario, which makes its state space more concise compared to the StCach scheme and the RdmOf scheme. Meanwhile, it can be observed from the figure that under the same constraints, the CachGA scheme outperforms the CachDQN scheme, but is still inferior to the CachOf scheme. This is because the GA algorithm can effectively optimize the problem of complex search space and overcome the deficiency of the DQN algorithm in terms of local minimum and parameter sensitivity. On the contrary, the DDPG algorithm employs deterministic strategy and AC structure to deal with action continuity more effectively and improve the learning efficiency. As a result, the CachOf scheme is superior to other schemes in terms of total reward and convergence speed.

5.3.2 *Evaluation of Success Rate*

In Fig. 5(c), the SR of the five different schemes under 1000 episodes is presented. From the figure, it is evident that the SR of all the schemes shows a gradual increasing trend as the number of episodes increases, and finally tends to be stabilized. Notably, the SR of the CachOf scheme is consistently higher than other schemes, which is attributed to three key factors. First, the dynamic caching strategy employed by the CachOf scheme computes and caches more popular content in each cycle, thus ensuring that more tasks are completed in the same time period. Second, the offloading priority policy proposed by the CachOf scheme enables more tasks to be completed within the limited delay constraint by optimizing the order of subtask scheduling. Finally, the DDPG algorithm employed by the CachOf scheme achieves more efficient offloading decisions in the continuous action space.

In Fig. 6(c), the SR variation of the five schemes is compared under different delay constraints. It can be observed from the figure that the SR of all the schemes tends to increase as the latency constraint increases. This is because a larger latency constraint indicates that the execution of the application is relatively less urgent and more tolerant to offloading decisions, and thus more tasks can be accomplished within the latency constraint. Meanwhile, it is particularly noteworthy that the SR of the CachOf scheme is consistently the highest, especially when the latency constraint is small, the SR performance of the CachOf scheme is superior compared to the other schemes. This indicates that the CachOf scheme has higher reliability in handling delay-sensitive tasks, thanks to the dynamic caching and prioritization scheme adopted by the CachOf scheme, which can effectively reduce the delay of computational offloading.

5.3.3 *Evaluation of Average Delay*

Fig. 5(b) illustrates the comparison of AD results for the five schemes. The vertical coordinate represents the average latency of each application and the horizontal coordinate is the episode. The graphical representation highlights a

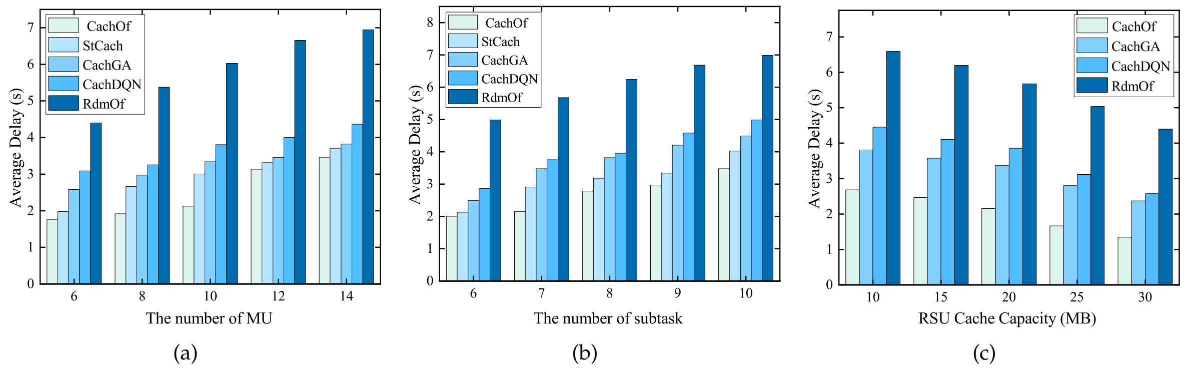

Fig. 7. Performance comparison of average delay under (a) different numbers of mobile users, (b) different numbers of subtasks, and (c) different edge server cache capacity.

consistent decline in the average latency across all schemes. Among them, the CachOf scheme always has the lowest average latency, followed by the StCach scheme, the CachGA scheme, and the CachDQN, while the G+DQN scheme and the RdmOf scheme have the highest latency. This is attributed to the efficient caching and offloading prioritization implemented in the CachOf scheme. This strategy maximizes the use of limited resources, enabling the completion of multiple subtasks. In addition, the DDPG algorithm outperforms other algorithms in offloading decisions, resulting in further latency reduction.

Fig. 6(a) compares the the AD of the five solutions under different edge server computation power. The findings indicate a diminishing trend in the average computation latency of all solutions with the augmentation of edge server computation power. This is due to the fact that more tasks can be offloaded to the edge server for execution when the computational power of the edge server is increased, which is much higher than that of the local users, leading to a decrease in latency. It is especially noteworthy that the trend of computation delay reduction is more significant for all schemes with priority computation, due to their ability to maximize the computational resources of edge servers by designing scheduling prioritization schemes.

Fig. 6(b) analyzes the AD of the five scenarios under different mobile user computing power. The results show that the computational delay of all the scenarios shows a decreasing trend as the mobile user computing power increases. This arises from the fact that local has faster processing speed when the edge server has insufficient computing power and needs to perform tasks locally. Meanwhile, it is worth noting that the CachOf scheme has a significant performance improvement compared to the RdmOf scheme. This is because the parallel offloading scheme used in the CachOf scheme is able to maximize the utilization of both edge servers and local computational resources at the same time, holding all other conditions constant.

Fig. 7(a) depicts the AD comparison among five schemes under varying mobile user request numbers within edge servers. It is apparent that as the number of mobile users increases, the average delay of each scheme gradually rises. Significantly, beyond 10 applications, the growth of AD becomes less pronounced. This is due to the surge in application numbers exceeding the computational capacity provided by edge servers, rendering some tasks incapable of meeting latency constraints. Moreover, the CachOf scheme demonstrates a significant computational latency advantage over the StCach scheme. This is attributed to the strategic utilization of cache resources, effectively facilitating offloading and resulting in reduced computational latency under severe computational resource constraints on edge servers.

Fig. 7(b) shows the results of the AD comparison of the five schemes with different numbers of subtasks within the application. In the figure, it is evident that with the escalation in the number of subtasks, there is a tendency for the average delay to increase across all schemes. However, the AD of the CachOf scheme always remains at a low level, much lower than that of the other schemes. This is because as the number of subtasks increases, more dependent subtasks are involved in the system, the overall system complexity increases, and the latency required to perform the tasks increases accordingly. At the same time, the CachOf can effectively utilize resources and make the best offloading decisions by using dynamic caching and offloading optimization schemes. Therefore, the performance of CachOf is better than other schemes in all aspects.

Fig. 7(c) illustrates the comparative results of AD for five schemes under various cache capacity conditions. Clearly, as the cache capacity increases, there is a consistent downward trend in the average delays for all schemes. Notably, this downward trend becomes more pronounced with larger cache capacities. The rationale behind this trend lies in the increased cache capacity facilitating advanced pre-caching of content supportive of task offloading on edge servers. Consequently, this diminishes the complexity associated with computation offloading, ultimately resulting in a reduction in overall delay.

6 Conclusion

In this paper, we propose a scheme named CachOf, an offloading scheme for DAG-structured tasks aimed at reducing latency under limited computing and caching resources. It includes a strategy for subtask execution and offloading order, a dynamic caching approach for edge server collaboration, and an intelligent offloading decision scheme. The main contribution lies in comprehensively considering task dependency, resource allocation, dynamic caching, and offloading decisions, making it highly applicable to real-world MEC scenarios. Simulation results demonstrate that CachOf outperforms other baseline schemes in terms of latency, reward, and offloading success rate. Future work will focus on MU collaboration and edge server load prediction to further enhance system performance. In addition, we plan to expand our research to include scenarios that account for both uplink and downlink delays, as well as communication bandwidth constraints and allocation.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China under Grant 62372310, in part by the Liaoning Province Applied Basic Research Program under Grant 2023JH2/101300194, in part by the UK Engineering and Physical Sciences Research Council (EPSRC) funded COG-MHEAR Programme (EPSRC Grant Reference: EP/T021063/1), and in part by the LiaoNing Revitalization Talents Program under Grant XLYC2203151.

References

[1] X. Chen, M. Li, H. Zhong, X. Chen, Y. Ma, and C.-H. Hsu, “Funoff: Offloading applications at function granularity for mobile edge computing,” *IEEE Transactions on Mobile Computing*, pp. 1–18, 2023.

[2] L. Zhao, T. Li, E. Zhang, Y. Lin, S. Wan, A. Hawbani, and M. Guizani, “Adaptive swarm intelligent offloading based on digital twin-assisted prediction in vec,” *IEEE Transactions on Mobile Computing*, pp. 1–18, 2023.

[3] S. Duan, F. Lyu, H. Wu, W. Chen, H. Lu, Z. Dong, and X. Shen, “Moto: Mobility-aware online task offloading with adaptive load balancing in small-cell mec,” *IEEE Transactions on Mobile Computing*, pp. 1–16, 2022.

[4] H. Ke, J. Wang, L. Deng, Y. Ge, and H. Wang, “Deep reinforcement learning-based adaptive computation offloading for mec in heterogeneous vehicular networks,” *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7916–7929, 2020.

[5] L. Zhao, E. Zhang, S. Wan, A. Hawbani, A. Y. Al-Dubai, G. Min, and A. Y. Zomaya, “Meson: A mobility-aware dependent task offloading scheme for urban vehicular edge computing,” *IEEE Transactions on Mobile Computing*, pp. 1–15, 2023.

[6] L. Zhao, Z. Zhao, E. Zhang, A. Hawbani, A. Al-Dubai, Z. Tan, and A. Hussain, “A digital twin-assisted intelligent partial offloading approach for vehicular edge computing,” *IEEE Journal on Selected Areas in Communications*, pp. 1–1, 2023.

[7] Y. Sahni, J. Cao, L. Yang, and Y. Ji, “Multihop offloading of multiple dag tasks in collaborative edge computing,” *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4893–4905, 2021.

[8] X. Lv, H. Du, and Q. Ye, “Tbtoa: A dag-based task offloading scheme for mobile edge computing,” in *ICC 2022 - IEEE International Conference on Communications*, 2022, pp. 4607–4612.

[9] Y. Liu, S. Wang, Q. Zhao, S. Du, A. Zhou, X. Ma, and F. Yang, “Dependency-aware task scheduling in vehicular edge computing,” *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4961–4971, 2020.

[10] 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)*, 2020, pp. 1–9.

[11] T.-Y. Kuo, M.-C. Lee, J.-H. Kim, and T.-S. Lee, “Quality-aware joint caching, computing and communication optimization for video delivery in vehicular networks,” *IEEE Transactions on Vehicular Technology*, vol. 72, no. 4, pp. 5240–5256, 2023.

[12] H. Tian, X. Xu, L. Qi, X. Zhang, W. Dou, S. Yu, and Q. Ni, “Copace: Edge computation offloading and caching for self-driving with deep reinforcement learning,” *IEEE Transactions on Vehicular Technology*, vol. 70, no. 12, pp. 13 281–13 293, 2021.

[13] Y. Dong, S. Guo, Q. Wang, S. Yu, and Y. Yang, “Content caching-enhanced computation offloading in mobile edge service networks,” *IEEE Transactions on Vehicular Technology*, vol. 71, no. 1, pp. 872–886, 2022.

[14] X. Ma, A. Zhou, S. Zhang, and S. Wang, “Cooperative service caching and workload scheduling in mobile edge computing,” in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 2076–2085.

[15] Y. Liu, Q. He, D. Zheng, X. Xia, F. Chen, and B. Zhang, “Data caching optimization in the edge computing environment,” *IEEE Transactions on Services Computing*, vol. 15, no. 4, pp. 2074–2085, 2022.

[16] V. Farhadi, F. Mehmeti, T. He, T. F. L. Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis, “Service placement and request scheduling for data-intensive applications in edge clouds,” *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 779–792, 2021.

[17] X. Dai, Z. Xiao, H. Jiang, M. Alazab, J. C. S. Lui, G. Min, S. Dustdar, and J. Liu, “Task offloading for cloud-assisted fog computing with dynamic service caching in enterprise management systems,” *IEEE Transactions on Industrial Informatics*, vol. 19, no. 1, pp. 662–672, 2023.

[18] S. Bi, L. Huang, and Y.-J. A. Zhang, “Joint optimization of service caching placement and computation offloading in mobile edge computing systems,” *IEEE Transactions on Wireless Communications*, vol. 19, no. 7, pp. 4947–4963, 2020.

[19] J. Liu, M. Ahmed, M. A. Mirza, W. U. Khan, D. Xu, J. Li, A. Aziz, and Z. Han, “Rl/drl meets vehicular task offloading using edge and vehicular cloudlet: A survey,” *IEEE Internet of Things Journal*, vol. 9, no. 11, pp. 8315–8338, 2022.

[20] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, “Efficient dependent task offloading for multiple applications in mec-cloud system,” *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2147–2162, 2023.

[21] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, “Offloading dependent tasks in mobile edge computing with service caching,” in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 1997–2006.

[22] Z. Liu, M. Liwang, S. Hosseinalipour, H. Dai, Z. Gao, and L. Huang, “Rfid: Towards low latency and reliable dag task scheduling over dynamic vehicular clouds,” *IEEE Transactions on Vehicular Technology*, pp. 1–15, 2023.

[23] Y. Sahni, J. Cao, L. Yang, and Y. Ji, “Multihop offloading

of multiple dag tasks in collaborative edge computing," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4893–4905, 2021.

[24] H. Liao, X. Li, D. Guo, W. Kang, and J. Li, "Dependency-aware application assigning and scheduling in edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4451–4463, 2022.

[25] C. Li, F. Liu, B. Wang, C. L. P. Chen, X. Tang, J. Jiang, and J. Liu, "Dependency-aware vehicular task scheduling policy for tracking service vec networks," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 3, pp. 2400–2414, 2023.

[26] X. Bi, X. Sun, Z. Lyu, B. Zhang, and X. Wei, "A back adjustment based dependent task offloading scheduling algorithm with fairness constraints in vec networks," *Computer Networks*, vol. 223, pp. 1389–1286, 2023.

[27] Y. Shang, J. Li, M. Qin, and Q. Yang, "Deep reinforcement learning-based task scheduling in heterogeneous mec networks," in *2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring)*, 2022, pp. 1–6.

[28] Z. Li, C. Yang, X. Huang, W. Zeng, and S. Xie, "Coor: Collaborative task offloading and service caching replacement for vehicular edge computing networks," *IEEE Transactions on Vehicular Technology*, vol. 72, no. 7, pp. 9676–9681, 2023.

[29] N. Hudson, H. Khamfroush, M. Baughman, D. E. Lucani, K. Chard, and I. Foster, "Qos-aware edge ai placement and scheduling with multiple implementations in faas-based edge computing," *Future Generation Computer Systems*, vol. 157, pp. 250–263, 2024.

[30] Y. Sahni, J. Cao, L. Yang, and Y. Ji, "Multihop offloading of multiple dag tasks in collaborative edge computing," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4893–4905, 2021.

[31] G. Ma, X. Wang, M. Hu, W. Ouyang, X. Chen, and Y. Li, "Drl-based computation offloading with queue stability for vehicular-cloud-assisted mobile edge computing systems," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 4, pp. 2797–2809, 2023.

[32] J. Zhang, X. Xu, S. Han, K. Zhang, P. Zhang, and S. Ren, "Intelligent ultra-reliable and low latency communications: Security and flexibility," *IEEE Transactions on Wireless Communications*, vol. 22, no. 11, pp. 8392–8406, 2023.

[33] L. Ale, S. A. King, N. Zhang, A. R. Sattar, and J. Skandaraniyam, "D3pg: Dirichlet ddpg for task partitioning and offloading with constrained hybrid action space in mobile-edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 19, pp. 19 260–19 272, 2022.

[34] H. Tabatabaee Malazi, S. R. Chaudhry, A. Kazmi, A. Palade, C. Cabrera, G. White, and S. Clarke, "Dynamic service placement in multi-access edge computing: A systematic literature review," *IEEE Access*, vol. 10, pp. 32 639–32 688, 2022.

[35] B. Han, V. Sciancalepore, Y. Xu, D. Feng, and H. D. Schotten, "Impatient queuing for intelligent task offloading in multiaccess edge computing," *IEEE Transactions on Wireless Communications*, vol. 22, no. 1, pp. 59–72, 2022.

Liang Zhao (S'09-M'17) is a Professor at Shenyang Aerospace University, China. He received his Ph.D. degree from the School of Computing at Edinburgh Napier University in 2011. He is also a JSPS Invitational Fellow (2023). He was listed as Top 2 % of scientists in the world by Standford University (2022 and 2023). He served as the Chair of several international conferences and workshops, including 2022 IEEE BigDataSE (Steering Co-Chair), 2021 IEEE TrustCom (Program Co-Chair), 2019 IEEE IUCC (Program Co-Chair). He is Associate Editor of Frontiers in Communications and Networking and Journal of Circuits Systems and Computers. He is/has been a guest editor of IEEE Transactions on Network Science and Engineering, Springer Journal of Computing, etc.

Zijia Zhao is a student at Shenyang Aerospace University, China. She is currently studying for her M.S. degree in Computer Science, Shenyang Aerospace University. Her research interests mainly include mobile edge computing, vehicle edge computing, computation offloading and caching.

Ammar Hawbani is a Full Professor at the School of Computer Science at Shenyang Aerospace University. He earned his B.S. in Computer Software and Theory from the University of Science and Technology of China (USTC) in 2009. His academic journey continued with an M.S. in 2012 and a Ph.D. in 2016, all from USTC. Following his Ph.D. completion, he served as a Postdoctoral Researcher in the School of Computer Science and Technology at USTC from 2016 to 2019. Later, he worked as an Associate Researcher in the School of Computer Science and Technology at USTC from 2019 to 2023. Currently, he holds the position of Full Professor at the School of Computer Science in Shenyang Aerospace University. His research interests span IoT, WSNs, WBANs, WMNs, VANETs, and SDN.

Zhi Liu (S'11-M'14-SM'19) received a Ph.D. degree in informatics in National Institute of Informatics. He is currently an Associate Professor at The University of Electro-Communications. His research interest includes video network transmission and mobile edge computing. He is now an editorial board member of IEEE Transactions on Multimedia. He is a senior member of IEEE.

Zhiyuan Tan is an Associate Professor with the School of Computing, Engineering and the Built Environment, Edinburgh Napier University, UK. He received his Ph.D. degree from the University of Technology Sydney, Australia, in 2014, and was a Postdoctoral Researcher with the University of Twente, NL between 2014 and 2016. He is an Associate Editor of IEEE Transactions on Reliability, IEEE Open Journal of the Computer Society, Journal of Ambient Intelligence and Humanized Computing and the Journal of Ambient Intelligence and Humanized Computing, as well as an Academic Editor of Security and Communication Networks. He is a Senior Member of the IEEE and a Member of the ACM.

Keping Yu (Senior Member, IEEE) received the M.E. and Ph.D. degrees from the Graduate School of Global Information and Telecommunication Studies, Waseda University, Tokyo, Japan, in 2012 and 2016, respectively. He was a Research Associate, a Junior Researcher, and a Researcher with the Global Information and Telecommunication Institute, Waseda University, from 2015 to 2019, from 2019 to 2020, and from 2020 to 2022, respectively. He is currently an Associate Professor, the Vice Director of Institute of Integrated Science and Technology, and the Director of the Network Intelligence and Security Laboratory (YU Lab), Hosei University, Japan.