Application Loading and Computing Allocation for Collaborative Edge Computing

YANZAN SUN\(^1\), (Member, IEEE), XINKUN XIE\(^1\), FAN WU\(^2\), SHUNQING ZHANG\(^1\), (Senior Member, IEEE), SHUGONG XU\(^1\), (Fellow, IEEE), AND YATING WU\(^1\), (Member, IEEE)

\(^1\)Key Laboratory of Specialty Fiber Optics and Optical Access Networks, Shanghai Institute for Advanced Communication and Data Science, Shanghai University, Shanghai 200444, China
\(^2\)School of Economics, Shanghai University, Shanghai 200444, China

Corresponding author: Fan Wu (fanwu6@shu.edu.cn)

This work was supported in part by the Pudong New Area Science and Technology Development Fund; in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101130012; in part by the Foshan Science and Technology Innovation Team Project under Grant F500-AJ919-4402-0060; in part by the National Natural Science Foundation of China (NSFC) under Grant 62071284, Grant 61801262, Grant 61901251, and Grant 61904101; and in part by the Innovation Program of Shanghai Municipal Science and Technology Commission under Grant 20JC1416400.

ABSTRACT The emergence of Mobile Edge Computing (MEC) provides a near-field tasking platform to meet the latency requirements of the growing number of compute-intensive mobile applications. However, physical memory constraints limit the number of application services that can be loaded on the edge server (ES) simultaneously, and the non-uniform distribution of traffic in the mobile network makes it difficult to fully utilize the resources in the edge network. In addition, for MEC platform operators, how to achieve the expected profit of application service providers (ASP) to attract more ASPs is also important. To address these issues, in this paper we propose an ASP profit-aware solution for jointly optimizing application loading, task allocation, and compute resource allocation across multi-ES, minimizing system latency while maintaining ASP profitability. We first formulate the problem as a long-term stochastic optimization problem with ASP profit constraints, transform it into a single time slot optimization problem using the Lyapunov optimization framework, and then, using the power of genetic algorithms (GA), we propose an online heuristic algorithm to obtain a near-optimal strategy for each time slot. Simulation results show that our algorithm is effective in reducing system latency in the long term, while demonstrating performance that ensures more ASPs with desired profits.

INDEX TERMS Edge computing, application loading, computing allocation, service provider profit-aware, load balance.

I. INTRODUCTION Mobile internet and artificial intelligence (AI) technology have prospered in recent years, and a growing number of emerging applications have been launched to facilitate people’s lives, such as real-time audio/video services, face recognition, and social interaction based on augmented reality [1]. Since AI technology has the characteristics of requiring sufficient computing resources, the further development of its application on the user equipment (UE) has enormous challenges [2]. With the advancement of 5G network deployment, as an effective solution to reduce UE’s request delay and improve its battery life, Mobile Edge Computing (MEC) has also been widely deployed. Based on the 5G network evolution architecture, MEC aims to integrate telecommunications with internet services. It deploys a cloud computing platform at the edge of the Radio Access Network (RAN) to provide computing, storage, network and communication resources for applications, allowing the services and functions originally located in the cloud data center to sink to the edge of the network [3]. Consequently, MEC shortens the distance between the UE and the service, thereby reducing the response delay and significantly improving the users’ quality of experience (QoE), while also promoting the...
application of AI technology on the UE. Therefore, MEC is widely regarded as an important technology to realize the vision of the next generation Internet.

The MEC platform generally consists of a base station (BS) and an edge server (ES), where ES provides computing resources. However, compared with ultra-large-scale remote cloud servers, the physical equipment performance of edge servers is far from sufficient. First, when deploying applications on the MEC platform, each type of application must be provided with the fixed running memory it needs. However, the operating memory of ES is limited, and when its allocation exceeds physical memory, it will experience severe performance degradation, which may cause memory overflow errors [4]. Therefore, ES cannot load all applications at the same time as the cloud server. Moreover, ES’s computing resources are limited, and the task traffic distribution in the mobile edge network is heterogeneous and dynamic in space and time. Therefore, it is difficult for a single ES to provide computing services in the constantly changing edge environment. These challenges have prompted us to study the collaboration between multiple MEC platforms and cloud servers to improve overall resource utilisation by coordinating task allocation between ES and cloud servers. And due to the high coupling of task requirements and resources, effective application loading and task allocation require careful coordination between all servers and applications.

In addition, when the task request of the UE is offloaded to the ES for execution, the corresponding application service providers (ASP) will obtain certain benefits, such as membership fees or increased number of clicks, so ASPs necessarily want to maximise task traffic to ESs. However, when ES’s workload is overly heavy, the corresponding request response time will increase [5], which violates the original intention of the MEC platform. Therefore, the MEC platform operator needs to weigh the benefits of the ASP and the system delay. Furthermore, the MEC platform operator will charge a certain resource fee [6] when running applications, which will also make ASPs reduce the number of applications loaded on multiple ESs, which will further increase the response time. Consequently, optimization primarily for user QoE related latency etc. while ignoring the expected profitability of the ASP is a common problem in current research, the solution to which is explored in this paper.

In this paper, we investigated the collaboration of multiple MEC platforms and remote clouds. As is shown in Figure 1, all ESs can be connected via a local area network (LAN) [7]. For ESs that are heavily loaded or have no memory to load applications, their computing tasks can be offloaded to cloud server or ESs with relatively idle resources, thereby improving the resource utilization of edge networks through the cooperation between MEC platforms. Our objective is to minimize the long-term average network delay under the long-term constraint of ensuring the expected profit of ASP. In addition, we designed an adaptive weight based on the historical profit loss to deal with scenarios when the request traffic is low. The main contributions are as follows:

1) In an ES collaboration network with multiple types of applications, we investigate the problem of joint application loading and task allocation among multiple servers to minimize task calculation delays under the constraints of ASP’s long-term expected profit. We express the problem as a mixed integer nonlinear programming problem and prove that it is NP-hard by analyzing simplified examples.

2) To solve this problem, based on the Lyapunov optimization theory, we have transformed it into a single-slot problem with ASP profit awareness. It can dynamically adjust the weight of ASP profit and task delay in accordance with the historical income of ASP to realize the profit perception of ASP, and can maximize the number of ASPs that satisfy the profit in the case of low number of tasks.

3) We decouple the single-slot problem into two phases, namely application loading and task allocation respectively. In the first stage, we use the power of genetic algorithms to obtain sub-optimal loading decisions. In the second stage, we propose an Online Application loading, Task allocation and Computing resource allocation (OATC) algorithm to obtain the asymptotically optimal solution. Through the above study, we will provide MEC operators with a solution that jointly balances the benefits of ASPs and the QoE of users.

The rest of the paper is organised as follows. In Section II, we first review the relevant work. Then the system model will be introduced in Section III. In Section IV, a description of the ASP profit-awareness problem will be given. Next, a single time slot transformation of the original problem will be performed in Section V and an online solution will be proposed in Section VI. Finally, we will give the results of the numerical analysis and summarise the work of this paper respectively in Section VII and Section VIII.

II. RELATED WORK

Mobile edge computing has been widely applied in various scenarios to reduce latency [8], [9] or energy consumption [1], [10]. For the application’s task module dependency problem, [2] designed an iterative heuristic algorithm to reduce application execution delay. Reference [11] optimized computing offloading and resource allocation for the vehicle network coordinated by MEC and cloud servers to reduce vehicle system delay. However, the above work only assumed the single-server scenario and cannot adapt to the rapidly growing demand for intensive tasks.

Therefore, multi-MEC server collaborative task offloading has been proposed to meet the computing requirements in the edge network [12]–[14]. In [13], the authors studied the energy-aware task offloading problem of multiple edge servers in the ultra-dense Internet of Things (IoT), and proposed a task offloading scheme based on iterative search to jointly optimize task offloading, computing resources and transmit power allocation. Reference [14] also optimized
the same variables and proposed a low complexity heuristic algorithm to maximize the weighted sum of task completion time and energy consumption using convex and quasi-convex optimization techniques. In addition, some work has investigated MEC heterogeneous networks. Reference [15] proposed a heterogeneous multilayer MEC to minimise system latency by coordinating task allocation and resource transfer between UE, multilayer MEC servers and central cloud, while [16] considered the collaborative computational offloading of D2D communications and MECs to increase the number of devices supported by the cellular network through a Hungarian algorithm.

Unfortunately, most of the above studies do not consider specific task types, and some simply assume that all applications can be cached and loaded simultaneously on each MEC server, which is actually difficult to achieve due to the physical limitations of the server equipment. The problem of caching and loading of services and applications has therefore inspired extensive research [17]–[19]. Reference [19] jointly performed service cache and load balancing optimisation and proposed an efficient online algorithm based on Gibbs sampling to reduce computational latency. Reference [20] jointly considered cache placement, computational offloading, and resource allocation, and designed a complexity reduction algorithm to reduce the energy consumption of MEC systems. However, most of the literature does not consider the overhead of service loading and caching. Reference [21] investigated the optimisation of service penalties and net profits for service providers while migrating interdependent virtual machines in real time, but neglected user latency.

Therefore, current research generally failed to consider the optimization of application loading and resource allocation by combining the benefits of ASP into a multi-MEC platform, so we investigated the problem of joint optimization of ASP benefits and user latency based on Lyapunov optimization and GA theory.

III. SYSTEM MODEL

In this section, we consider a distributed MEC network architecture. As is shown in Figure 1, it includes $N$ BSs, each of which is equipped with an ES for providing computing services for UEs in the coverage area. UE can transmit tasks to BS through wireless transmission and then forward to the corresponding ES for processing. Note that we assume each user device can only be connected to one BS at the same time and each edge server is only connected to one BS. Since some small base stations may be intensively deployed to increase the transmission rate and coverage in practice, there may be multiple BSs connected to the same ES, but those scenarios will not affect our system model and problem solving. Moreover, in order to simplify the calculation, the coverage overlap and interference between BSs are not considered.

We divide the time domain into multiple discrete time slots, defined as $T = \{1, 2, \ldots, T\}$, and the system status will not change in each time period. Let $\mathcal{N} = \{1, 2, \ldots, N\}$ denote the set of ES. Each ES has limited computing resources and limited running memory, and multiple different applications can be loaded in each time slot. We assume that ES provides a Container for each application it loads, like a Container in Docker, so it is heterogeneous and the computing resources of each Container can be dynamically adjusted in ES through [22]. Here we assume that each ES can only load one Container for the same application.

A. APPLICATION LOADING STRATEGY

Each UE may have several application tasks that need to be calculated, such as facial recognition, natural language processing, interactive gaming, and so on. Let $\mathcal{K} = \{1, 2, \ldots, K\}$ denote the set of different types of application tasks. Compared with the cloud server, the running memory of ES is limited, so it is impossible to load all applications in the same time slot when there are more applications deployed. Nevertheless, ES can dynamically load and delete applications in different time slots to achieve the same purpose. Note that we assume the ES physical storage is large enough, and all applications have been cached in ES. Let $a_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,K}\}$ denote the set of loading decisions of the application in all ESs in each time slot, where $a_{i,k}$ is a binary variable to represent the loading decision of application $k$ in ES $i$ in time slot $t$. $a_{i,k} = 0$ means that ES $i$ does not load or remove application $k$ in time slot $t$; $a_{i,k} = 1$ otherwise. Furthermore, we consider that all applications can be loaded simultaneously in the cloud server at any time slot. We assume that the running memory required for application $k$ is $c_k$, so the application loading decisions are subject to the following running memory capacity constraints

$$\sum_{k \in \mathcal{K}} a_{i,k} c_k \leq C_i \quad \forall n \in \mathcal{N}, \forall k \in \mathcal{K} \quad (1)$$

where $C_i$ is the maximum running memory of ES $i$.

B. TASK ALLOCATION STRATEGY

The number of compute task requests varies from region to region, so an ES requesting more tasks can transfer the
compute task to another ES requesting less to alleviate load pressure. In addition, for ES that are not loaded with a certain application, their corresponding tasks can be offloaded to nearby ES that are loaded with the application for processing, so that the computational workload in ESs can be balanced to fully utilize the computational resources in the network. Furthermore, we assume that each ES can connect to the cloud server, and that redundant computing tasks or tasks that do not load the appropriate application can be offloaded directly to the cloud server for processing.

In each time slot \( t \), for each application \( k \), we assume that task arrival per ES \( n \) is a Poisson process with expected rate \( \lambda^t_{n,k} \), which is the general assumption among current research [23]. We define \( \lambda^t \) as the arrival rate set of all application types of tasks in all ES in time slot \( t \), which may vary in time domain. According to [24], the non-uniform application task traffic in time domain can be modeled as Lognormal distribution. Thus, the task arrival rate set \( \lambda^t \) follows a Lognormal distribution in the time domain \( T \).

For simple modeling, we first consider scenarios where all ESs are tasks assigned, which are called load balancing below. We denote \( \nu^t_{n,k} \) and \( \nu^t_{n,k} \) as the arrival rates of tasks that the application \( k \) needs to perform locally in the ES \( n \) and those that need to be offloaded to the cloud server, after load balancing at time slot \( t \). Since the total UE task arrival remains the same, the task arrival rate before and after load balancing for each application \( k \) can be expressed as follows

\[
\sum_{n \in N} \nu^t_{n,k} + \nu^t_{n,k} = \sum_{n \in N} \lambda^t_{n,k} \tag{2}
\]

In the load balancing process, for each application, we assume that each ES cannot offload tasks and load tasks offloaded from other ES simultaneously in a time slot. This allows the system to avoid the cyclic unloading of tasks. We define \( \nu^t_{n,k} \) as the execution task arrival rate of the application \( k \) in time slot \( t \) which is the part exceeding the initial task arrival rate of ES \( n \), so it can be expressed as

\[
\nu^t_{n,k} = \nu^t_{n,k} - \lambda^t_{n,k} \tag{3}
\]

In addition, task requests for unloaded applications cannot be processed locally in ES, which is expressed mathematically as

\[
(1 - \alpha^t_{n,k}) \nu^t_{n,k} = 0 \tag{4}
\]

C. DELAY MODEL

In each time slot, the total delay spent is mainly divided into three parts: the computing delay in the ES, the LAN transmission delay between the ESs during load balancing, and the backhaul delay when the task is offloaded to the cloud server. Note that our main goal is task allocation between ESs, so the transmission delay of tasks transmitted wirelessly to the BS is not taken into account, and it has no effect on the scheme described in this paper.

1) COMPUTING DELAY

Since the computing resources of each ES are limited, the computing resources allocated to the Container of each loaded application are also limited. We model the task computing process of each loaded application container of each ES as M/M/1 queuing model according to [25]. Then the average computing delay of application \( k \) in ES \( n \) in time slot \( t \) can be expressed as

\[
D^t_{n,k} = \frac{1}{\beta^t_{n,k} - \nu^t_{n,k}} \tag{5}
\]

where \( \beta^t_{n,k} = f^t_{n,k}/\mu_k \) is the average service rate of application \( k \) in ES \( n \) in time slot \( t \), \( f^t_{n,k} \) is the computing resource allocated by ES \( n \) to application \( k \) in time slot \( t \), and \( \mu_k \) represents the average workload (cycles) required to accomplish the processing of application \( k \), which can be obtained using the methods in [13]. In order to keep the queue stable, the average service rate of ES should be less than its average service rate, i.e. \( \beta^t_{n,k} - \nu^t_{n,k} > 0 \).

2) LAN TRANSMISSION DELAY

Due to the limitation of sending rate and bandwidth, when ES offloads computing tasks to other servers through LAN, it will

| Symbol | Description |
|--------|-------------|
| \( N \) | set of ESs and connected BSs |
| \( K \) | set of different types of application tasks |
| \( c_k \) | running memory required for application \( k \) |
| \( \lambda^t_{n,k} \) | task arrival rate of application \( k \) in ES \( n \) in time slot \( t \) |
| \( s_k \) | input data size of the application \( k \) task |
| \( \mu_k \) | amount of workload(cycles) required to process application \( k \) |
| \( \alpha^t_{n,k} \) | loading decision of application \( k \) in ES \( n \) in time slot \( t \) |
| \( C_n \) | maximum running memory capacity of ES \( n \) |
| \( F_n \) | maximum computing resources of ES \( n \) |
| \( \nu^t_{n,k} \) | arrival rate of the task request of application \( k \) that needs to be calculated locally in ES \( n \) |
| \( \nu^t_{n,k} \) | task arrival rate of the application \( k \) in ES \( n \) that needs to be offloaded to other servers |
| \( \nu^t_{n,k} \) | task arrival rate of the application \( k \) in ES \( n \) that needs to be offloaded to the cloud server |
| \( f^t_{n,k} \) | the computing resource allocated by ES \( n \) to application \( k \) in time slot \( t \) |
| \( \xi_k \) | average loading delay of the corresponding Container for each application \( k \) |
| \( \xi_k \) | the profits of ASP when each unit task of application \( k \) is offloaded to ES for computing |
| \( \gamma \) | the fee charged by the ES operator when loading the application in each time slot |
| \( \nu^0_{k} \) | the fixed expected profit of ASP \( k \) |
cause the transmission delay of offloading tasks between LANs. We equate the task offloading process between ESs as a sending queue, which is modeled as an $M/M/1$ queuing model according to [23]. Then the average transmission delay of task offloading in the network at time slot $t$ is as follows

$$D^{t,\text{trans}} = \frac{\kappa}{1 - \kappa^{t,\text{trans}}} \tag{6}$$

where $\nu^{t,\text{trans}} = \sum_{n \in N} \sum_{k \in K} v_{n,k}^{t,\text{trans}} s_k$ represents the total task data size of all applications transmitted between LANs and $s_k$ is the average task data byte size of application $k$, and $\kappa$ represents the average transmission delay of per unit data size transmitted through LAN without congestion.

3) BACKHAUL DELAY

Each application task can be offloaded to the cloud server for processing through the backhaul link. Note that we assume the computing resources of the cloud server are unlimited, so only the transmission delay of the cloud server is considered. The process of offloading all ES tasks of each application to the cloud server can modeled as $M/M/1$ queuing model according to [23]. And we assume that its average transmission time per unit data size is $\psi$ times the transmission time through the LAN. Then the average transmission delay of all tasks offloaded to the cloud server in time slot $t$ is

$$D^{t,\text{cc}} = \frac{\psi \kappa}{1 - \psi \kappa^{t,\text{cc}}} \tag{7}$$

where $\nu^{t,\text{cc}} = \sum_{k \in K} v_{k}^{t,\text{cc}} s_k$ is the total task data size of all ES applications to be transferred to the cloud server.

4) APPLICATION LOADING DELAY

ES can dynamically switch loaded applications across time periods, requiring the corresponding Container to be loaded into the running memory space when loading, which will cause additional load delays. Since the time consumed by ES to remove the Container and clean up the memory is relatively short, the delay of application container deletion is not counted. Then the loading delay of application $k$ in all ES in time slot $t$ can be expressed as

$$D_{k}^{t,\text{load}} = \sum_{n \in N} (1 - d_{n,k}^{t-1}) v_{n,k}^{t,\text{load}} s_k \tag{8}$$

where $s_{k}$ represents the average loading delay of the corresponding Container for each application $k$, which can generally be obtained from the application statistics of the server system.

Based on the above discussion, we divide the application’s task delay into four parts, and then analyze the application in the following sections. Then the total task delay of the application $k$ in the system at time slot $t$ is calculated as

$$D_{k}^{t} = \sum_{n \in N} v_{n,k}^{t,\text{ec}} D_{n,k}^{t,\text{cc}} + \sum_{n \in N} v_{n,k}^{t,\text{trans}} s_k D^{t,\text{trans}} + v_{k}^{t,\text{cc}} s_k D^{t,\text{cc}} + D_{k}^{t,\text{load}} \tag{9}$$

D. ASP PROFIT MODEL

When the tasks of the UE are offloaded to the edge server for calculation, the corresponding ASP will charge certain profit, such as membership fees or increased clicks. In addition, when the application is deployed and run on the ES, the ES operator will also charge for physical storage and runtime resources, which will bring a certain amount of overhead to the ASP. Since the storage cost is fixed, here we only consider the cost of application loading and runtime. Then the net profit obtained by ASP $k$ in time slot $t$ can be expressed as

$$E_{k}^{t} = \sum_{n \in N} (\epsilon_{k} v_{n,k}^{t,\text{ec}} - d_{n,k}^{t} \gamma) \tag{10}$$

where $\epsilon_{k}$ represents the profits of ASP when each unit task of application $k$ is offloaded to ES for computing, and $\gamma$ represents the fee charged by the ES operator for loading the application at each time period. Considering the long-term profit of ASP, we define the loading state of application $k$ in all ESs at time slot $t$ as

$$l_{k}^{t} = I(\sum_{n \in N} d_{n,k}^{t} > 0) \tag{11}$$

where $I(x)$ is an indicator function, which is equal to 1 if $x$ is true and 0 otherwise. Therefore, the total load time of application $k$ in the time domain $T$ is $T_{k}^{\text{load}} = \sum_{t \in T} l_{k}^{t}$.

IV. PROBLEM FORMULATION

In the above section, we consider both ASPs and users. Through the MEC platform, ASPs definitely want to maximize its long-term benefits, while users hope to minimize the task delay, so a joint consideration is necessary. In this section, we will express the joint problem of ASP profit and user latency as a long-term stochastic optimization problem. The goal is to minimize the long-term total task delay in the MEC system by jointly optimizing application loading decisions, task allocation and ES computing resource allocation, while realizing the long-term fixed profit of ASPs. According to the system model defined in the previous section, the long-term joint optimization problem can be formulated as follows:

$$P1 \min_{a^{t},v^{t},f^{t}} \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} \sum_{k \in K} D_{k}^{t}$$

s.t. $C1 : \lim_{T \rightarrow \infty} \frac{1}{T_{k}^{\text{load}}} \sum_{t=0}^{T-1} E_{k}^{t} \geq \epsilon_{k}^{0} \ \forall k \in K$

$C2 : \sum_{k \in K} d_{n,k}^{t} \epsilon_{k} \leq C_{n} \ \forall n \in N$

$C3 : \sum_{n \in N} v_{n,k}^{t,\text{ec}} + v_{k}^{t,\text{cc}} = \sum_{n \in N} \lambda_{n,k}^{t} \ \forall k \in K$

$C4 : (1 - d_{n,k}^{t}) v_{n,k}^{t,\text{ec}} = 0 \ \forall n \in N, \forall k \in K$

$C5 : \sum_{k \in K} f_{n,k}^{t} \leq F_{n} \ \forall n \in N$

$C6 : \sum_{n \in N} d_{n,k}^{t} \leq N \ \forall k \in K$
In problem P1, C1 is the long-term benefit constraint of ASP, which ensures that the long-term average profit of each ASP is not lower than its fixed expected profit $c^0_k$ during the time period when the application is loaded. C2 is a runtime memory constraint for ESs such that the total runtime memory consumed by each ES should not exceed its runtime memory limit for each time slot when the application is loaded. C3 is the task arrival amount constraint, which ensures that the total task arrival amount will not change after load balancing between ESs. C4 ensures that ESs that do not load applications cannot handle the corresponding tasks locally. C5 and C7 limit the maximum computing resources of ES to $F_n$, C6 and C8 stipulate that only one application instance can be loaded per ES, and the total number of loads cannot exceed the total number of ESs.

Since the problem P1 is a multi-temporal optimization problem, the optimal solution requires all offline information, i.e., the distribution of request task arrival rates over all time periods, but this is difficult to predict in advance. In addition, P1 is a mixed-integer nonlinear problem, and it is generally hardly possible to obtain an optimal solution in finite time. To solve this problem, we propose an online algorithm that only requires information about the current time slot to optimize application loading and task allocation.

V. SINGLE SLOT CONVERSION BASED ON LYAPUNOV

In this section, we apply the Lyapunov optimization framework to perform a single time slot transformation of P1 so that it can approach the effect of global optimization when only the current time slot information is needed, and propose an online algorithm to solve the problem.

A. CONSTRUCTION OF VIRTUAL QUEUE

The main challenge in directly solving the P1 problem is the coupling of application loading policies and task allocation and computational resource allocation policies at multiple time slots due to ASP’s long-term profit constraint. To deal with this challenge, we employ Lyapunov optimization techniques to build a (virtual) profit-loss queue that directs application loading, task allocation, and computational resource allocation to follow the long-term profit constraint. Specifically, assuming that $q_k(0) = 0$, we construct a profit-loss queue that evolves dynamically as follows

$$q_k(t + 1) = \left[q_k(t) + l^0_k (e^0_k - E^1_k)\right]^+$$  \hspace{1cm} (13)

where $q_k(t)$ is the deviation between the accumulated profit of the ASP $k$ and the profit constraint in the time slot $t$.

Algorithm 1 Online Profit Adaptive Algorithm

Input:  
- Weight parameter $V$, Initial profit deficit queue $q(0) = 0$;

Output: 
- Application loading decision, $\{a^0, \ldots, a^{T-1}\}$;
- Task arrival rate of each ES, $\{v^{0,cc}, \ldots, v^{T-1,cc}\}$;
- Task arrival rate of cloud servers, $\{v^{0,cc}, \ldots, v^{T-1,cc}\}$;

1: for $t = 0$ to $T - 1$ do
2: Obtain task arrival rate $\lambda^t$;
3: Solve P2 at time slot $t$ to obtain the optimal $a^t, \nu^{t,cc}$;
4: Update the profit deficit queue for each application $k$:
5: \hspace{1cm} $q_k(t + 1) = \max \left[ q_k(t) + l^0_k (e^0_k - E^1_k) \right] +$
6: end for

first maximize the number of ASPs that meet the expected profit, i.e., applications with a relatively large number of requests will have a higher loading priority, since the loading of applications with fewer tasks will increase the congestion of the profit deficit queue. We determine the priority of each application by varying the weight of the profit deficit as follows:

$$q_k^t(t) = \begin{cases}
q_k(t), \\
\frac{e^0_k - \gamma}{e^0_k} \sum_{n \in N} \lambda^t_{n,k} \\
\frac{\sum_{n \in N} \lambda^t_{n,k}}{(e^0_k - \gamma)/e^0_k} \frac{1 - q_k^t(t)}{\max(q^t(t))} q_k(t),
\end{cases}$$  \hspace{1cm} (14)

where $\Gamma$ is the penalty weight for applications with fewer tasks, and use the minimum number of tasks to achieve the expected profit as the criterion for judging the number of tasks. In addition, the priority of application loading is inversely proportional to the congestion level of its corresponding deficit queue when it has fewer tasks, which can prevent further increase in profit deficit. And this can prevent ASPs from raising high expectations of profit.

B. PROBLEM TRANSFORMATION

The Lyapunov function is defined as $L(q_k^t(t)) = \frac{1}{2}(q_k^t(t))^2$, which is expressed as the degree of congestion in the profit-deficit queue. A small value of $L(q_k^t(t))$ indicates that the queue backlog is smaller, which means that the virtual queue is more stable. Thus, in order to maintain the stability of the profit deficit queue, it is necessary to continuously push the Lyapunov function to a lower value to enforce the profit constraint, which can be solved by introducing Lyapunov drift [19]. Then, according to the Lyapunov framework, P1 can be converted into a single-slot problem, i.e., we can get the best solution to the P1 problem, by solving the P2 application loading, task allocation, and computational resource
allocation problems in each time slot as follows:

\[
P_2 \min_{a^t, v^t, f^t} U = \sum_{k \in K} \left( VD_k^t - q_k^t(t)E_k^t \right)
\]
\[
s.t. (C2), (C3), (C4), (C5), (C6), (C7), (C8)
\]  (15)

where \( V \) is the weight parameter, which is used to adjust the trade-off in the network between minimizing system delay and maximizing the profit of ASP. Different from the \( P1 \) that requires information in all time slots, the objective function and constraint conditions of \( P2 \) are only related to the task arrival amount in the current time slot, thence online decision-making can be made.

The introduction of \( q_k(t) \) enables the MEC system to learn from historical state information, so that the ASP’s past profit deficits are considered in the decision-making process of each period. When the ASP’s previous income is lower than the expected profit, the backlog of \( q_k(t) \) will become larger, and the MEC system will focus more on maximizing the benefits of ASP \( k \), so as to maintain the profits of ASP \( k \) within the constraints of expected profits. Specifically as is shown in Algorithm 1, we solve \( P2 \) in each time slot \( t \), and update the profit deficit queue to guide the decision of subsequent time slots to achieve long-term optimization.

VI. ONLINE APPLICATION LOADING, TASK ALLOCATION AND RESOURCE ALLOCATION

In this section, we analyze \( P2 \) and divide it into three sub-problems, and propose an online algorithm, called the OATC algorithm, to solve the single time slot utility minimization problem through collaborative application loading, task allocation, and computational resource allocation.

A. OPTIMIZING APPLICATION LOADING VIA GA

We adopt genetic algorithms (GA) to obtain the loading strategy of the application, not only because traditional optimization algorithms require the value of the objective function, but also because the search process is often constrained by the continuity of the objective function. For GA, only the fitness function transformed from the objective function is needed to determine the further search range, without other auxiliary information such as the derivative value of the objective function. It directly uses the objective function value or fitness value to narrow down the search range in the search space with higher fitness, thereby improving the search efficiency.

Based on the theory of evolution, GA simulates natural selection and survival of the fittest. Starting from the initial population (strategy set), GA selects individual chromosomes (strategy) as parents, and uses genetic operations (such as crossover, mutation, and fitness evaluation) to produce the next generation. After successive iterations, the overall approach is close to the optimal solution. We will explain each term and describe the specific steps of GA-based application loading strategy optimization as follows:

1) Chromosomes and Population: In GA, a feasible solution to a problem is called a “chromosome”. A feasible solution is generally composed of multiple elements, and each element is called a “gene” on the chromosome. A population is a group composed of several chromosomes that simulate a biological population, and it is generally a small subset of the entire search space. Let the loading decision variable of each application be a gene, then a chromosome can be expressed as \( a_j = \{a_{1,1}, \ldots, a_{1,k}, \ldots, a_{1,K}, \ldots, a_{N,K}\} \), where \( j \) is the chromosome sequence number, and \( a_{n,k} \) is the loading decision variable of application \( k \) in ES \( n \). Then a population can be defined as \( P = \{c_1, \ldots, c_j, \ldots, c_P\} \), where \( P \) is the population size, i.e. the number of chromosomes.

2) Fitness Function: The fitness function will score all the chromosomes generated in each iteration to judge the fitness of these chromosomes. Then the chromosomes with lower fitness are eliminated, and only the chromosomes with higher fitness are retained. After several iterations, the quality of chromosomes in the population will get better. The objective of \( P2 \) is to minimize the tasks total delay, under the premise of satisfying running memory constraints. Therefore, we define the fitness function according to the objective function of \( P2 \), and add a penalty term to punish the chromosomes that violate the constraints of \( C2 \), similar to the work in [26] and [27]. Then the fitness function can be expressed as

\[
f = \sum_{k \in K} \left( VD_k^t - q_k^t(t)E_k^t \right) + \chi \left[ \sum_{k \in K} a_{n,k}e_{k}^{*} - C_{n} \right]^{+}
\]  (16)

where \( \chi \) is a large positive number. Note that the penalty function in (16) can make the population converge to a feasible region by avoiding the infeasible region, and that only the constraint \( C2 \) is added as a penalty term because the application of the loading policy is only related to the amount of running memory. The solution to the fitness function will be discussed in the next subsection.

3) Initialization and selection: The initial population is given by the greedy strategy, which greedily selects the application with the highest task arrival rate in each time slot for loading, and then obtains a feasible loading strategy and performs random mutation generation. For the selection of the population, we adopt the championship method. This selection strategy takes a certain number of individuals from the population at a time, selects the best one to enter the offspring population, and then repeats the operation until the new population reaches the number of the original population.

4) Crossover and Mutation: Crossover operation is to select the crossover point with probability \( p \) on a pair of parent chromosomes, and then exchange genes to form a new chromosome. The mutation operation is to randomly select and modify the value of the gene on the chromosome with probability \( q \) to break the current search limit, which is conducive to jumping out of the local optimal solution.

Finally, the workflow of genetic algorithm is given, as is shown in Algorithm 2. First, the initial chromosomes are generated based on Greedy strategy, and then the initial
population is generated by random mutation in line 2. In line 3, use the fitness function in (16) to calculate and select the largest chromosome fitness value in the initial population $P$. Then select parents for the next generation through the championship method in line 6, and cross and mutate them with a certain probability to generate new chromosomes in line 7 and 8. Calculate the fitness of the new population in line 9 and proceed to the next iteration. The algorithm continues until the maximum number of iterations $G$ or the minimum tolerance $\delta$ is reached.

**B. OPTIMIZING TASK ALLOCATION VIA HEURISTICS**

In this section, we will discuss the optimal task allocation between ESs to solve the fitness function in (16). For each iteration of GA, an application loading strategy will be given, and the fitness function can be further expressed as (17), shown at the bottom of the page. After the derivation, the fitness function of GA is divided into two parts: one part depends on the task allocation strategy between edge servers, i.e., the weighted sum of the calculation delay and the transmission delay between the LAN and the backhaul delay and the profit of the ASP. The other part is independent of the task allocation strategy, including the total load delay of the application, the total load cost of the application, and the penalty items of the server running memory constraints. Therefore, the second part can be considered as a fixed constant when the load policy of the application is given. Hence, the solution of the fitness function can be transformed into P3, which is expressed as

\[
P_3: \min S = \sum_{k \in K} \left( \mathbb{E} \left[ \sum_{n \in N} v_{n,k}^{t,ec} \right] + \frac{1}{\mu_k} \sum_{k \in K} \sum_{n \in N} v_{n,k}^{t,trans} s_k \right) + \left( \mathbb{E} \left[ \sum_{n \in N} v_{n,k}^{t,ec} \right] - q_k(t) \right) \sum_{n \in N} \epsilon_k v_{n,k}^{t,ec}
\]

\[
f = \sum_{k \in K} \left( \mathbb{E} \left[ D_k \right] - q_k(t) \mathbb{E} \left[ E_k^t \right] \right) + \chi \left( \sum_{k \in K} d_{n,k} c_k - C_n \right)^{+}
\]

\[
\sum_{k \in K} \left( \mathbb{E} \left[ \sum_{n \in N} v_{n,k}^{t,ec} D_{n,k}^{trans} \right] + \mathbb{E} \left[ \sum_{n \in N} v_{n,k}^{t,trans} s_k D_{n,k}^{trans} + v_{n,k}^{t,ec} s_k D_{n,k}^{load} + D_k^{load} \right] - q_k(t) \sum_{n \in N} \epsilon_k v_{n,k}^{t,ec} \right)
\]

\[
\sum_{k \in K} \left( \mathbb{E} \left[ \sum_{n \in N} v_{n,k}^{t,ec} \frac{1}{\mu_k} - v_{n,k}^{t,ec} \right] + \frac{1}{\mu_k} \sum_{k \in K} \sum_{n \in N} v_{n,k}^{t,trans} s_k \right) + \left( \mathbb{E} \left[ \sum_{n \in N} v_{n,k}^{t,ec} \right] - q_k(t) \sum_{n \in N} \epsilon_k v_{n,k}^{t,ec} \right)
\]

s.t. C3: \( \sum_{n \in N} v_{n,k}^{t,ec} + v_{k}^{t,cc} = \sum_{n \in N} \chi_{n,k}^{t} \forall k \in K \)

C4: \((1 - \alpha_{n,k}^{t}) v_{n,k}^{t,cc} = 0 \forall n \in N, \forall k \in K \) (18)

The optimization goal of problem P3 is to minimize the weighted sum of the profit of the ASP and the average response time of the task in the time slot $t$, which includes the impact of the profit deficit $q(t)$ of the previous time slot. In equation (18), because the number of tasks is discrete, P3 can be reduced to a multidimensional knapsack problem with weights, so it is NP-hard and achieving the optimal solution is a challenging task. In this regard, inspired by game theory methods [28], we propose a heuristic gradient iterative allocation algorithm to solve it.

During the initialization process, for each ES $n$, if the application $k$ is loaded within the time slot $t$, its computational tasks are offloaded to the local for execution, otherwise the corresponding tasks are offloaded to the cloud server for execution, thereby obtaining the initial task allocation policy $bmv$ and the total system utility (objective function value of P3) $S_{bmv}$. Then, we iteratively offload a certain number of tasks to other servers for all servers, choosing the location with the lowest system utility. Here, the number of tasks to be offloaded in each iteration is defined as a gradient, denoted by $\Delta d$, and the gradient is reduced for iteration until the minimum gradient is achieved to meet the best task allocation policy. The detailed process is shown in Algorithm 3, which is divided into three main parts as follows:

**Offload decision:** This process will select an application offloading strategy that can reduce the utility of the system. Let the set $A$ represent all loaded applications for all servers, i.e., $A = \{a_{n,k}^t \mid a_{n,k}^t = 1, n \in N^+, k \in K\}$, where $N^+ = N \cup \{N + 1\}$ and $N + 1$ represents the cloud server. First, randomly sample without replacement an application $k$ loaded by ES $n_0$ from $A$. After that, there are two situations: if its task arrival rate is less than $\Delta d$, the application will be reselected, otherwise its task amount of $\Delta d$ will be unloaded to server $n_i \in N^+ - \{n_0\}$ and respectively calculate their

\[
\sum_{k \in K} \sum_{n \in N} (1 - d_{n,k}^{t-1}) a_{n,k}^t \xi_k + \sum_{k \in K} q_k(t) \sum_{n \in N} d_{n,k} c_k + \chi \left( \sum_{k \in K} d_{n,k} c_k - C_n \right)^{+}
\]

\[
\text{depends on task allocation strategy independent of task allocation strategy}
\]
Algorithm 2 GA-Based Application Loading Scheduling

Input:
Maximum number of iterations $G_{\text{max}}$, ES set $\mathcal{N}$, Application set $\mathcal{K}$, Task arrival rate $\lambda$, ASP expected profit $\epsilon$;

Output:
Optimal (suboptimal) application loading decision $a$;

1. Encode $\mathcal{N}$ and $\mathcal{K}$ as chromosome;
2. Generate initial population $\mathcal{P}$ based on greedy loading strategy;
3. Calculate the fitness function of the initial population and obtain the minimum value $f_{\text{cur}}$
4. while $\text{iter} \leq G_{\text{max}}$ do
   5. $f_{\text{last}} = f_{\text{cur}}$;
   6. Select parent chromosomes through stochastic tournament method;
   7. Implement chromosome crossover with probability $p$;
   8. Implement mutation on each chromosome with probability $q$;
   9. Use formula (16) to calculate the fitness set $F_{\text{fitness}}$ of chromosomes in $\mathcal{P}$;
   10. $f_{\text{cur}} = \min F_{\text{fitness}}$;
   11. if $|f_{\text{last}} - f_{\text{cur}}| \leq \delta$ then
      12. break
   13. end if
   14. iter $\leftarrow$ iter + 1;
15. end while
16. Decode the chromosome with fitness $f_{\text{cur}}$ to obtain $a$;
17. return $a$;

system utility. Since the number of tasks for only one server has changed every time the task is offloading, the update of the system utility $S$ can be expressed as follows

$$S = S_{\text{cur}} + \Delta D_{n} - \Delta D_{t} - q_{k}^{\text{cur}}(t) * (I(n_{o} = N + 1) - I(n_{i} = N + 1))\Delta d$$

(19)

where $\Delta D_{n}$ and $\Delta D_{t}$ are the delay changes of the task output server $n_{o}$ and task receiving server $n_{i}$, respectively, after offloading, and $I(x)$ is the indicator function.

Then choose the strategy $S_{\text{min}}$ that minimizes the utility, and perform strategy update if it is less than the current system utility. Otherwise, the offload will be abandoned and the application will be resampled until $\mathcal{A} = \emptyset$.

strategy update: This process updates all offloading strategy and current system utility. When the offload decision obtains the available strategy, the initial task allocation strategy and system utility are updated, which is expressed as

$$v_{n_{o},k} = v_{n_{o},k} - \Delta d$$
$$v_{n_{i},k} = v_{n_{i},k} + \Delta d$$
$$v_{k}^{\text{trans}} = \max[v_{k}^{\text{cur}} - \lambda_{k}, 0]$$
$$S_{\text{cur}} = S_{\text{min}}$$

(20)

where $v_{n_{o},k}, v_{n_{i},k}$ represent ES or cloud server, and $v_{k}^{\text{trans}}$ includes $v_{n_{o},k}^{\text{trans}}$ and $v_{n_{i},k}^{\text{trans}}$.

Algorithm 3 Task Allocation Optimization

Input:
ES set $\mathcal{N}$, Loaded application set $\mathcal{A}$, Task arrival rate $\lambda$, Minimum gradient $\delta_{d}$;

Output:
Optimal (suboptimal) Task allocation strategy $\nu$;

1. Initialize task allocation strategy $\nu$ based on task arrival rate $\lambda$;
2. Calculate the utility $S_{\text{cur}}, S_{\text{last}}$ of the initial strategy;
3. Calculate the initial gradient $\Delta d$;
4. while $\Delta d > \delta_{d}$ do
   5. do
      6. Copy set $\mathcal{A}$ to set $\mathcal{A}_{\text{copy}}$;
      7. $S_{\text{cur}} = S_{\text{last}}$;
      8. while $\mathcal{A}_{\text{copy}} \neq \emptyset$ do
         9. Random sampling without replacement from $\mathcal{A}_{\text{copy}}$ to obtain the application $k$ in ES $n$;
         10. $v_{n,k} = v_{n,k} - \Delta d$;
         11. Respectively offload the task of $\Delta d$ to the cloud server and all of ES and calculate the utility $S$;
         12. $S_{\text{min}} = \min[S]$;
         13. if $S_{\text{last}} > S_{\text{min}}$ then
            14. Update strategy $\nu$;
            15. $S_{\text{last}} = S_{\text{min}}$;
            16. break
         17. end if
         18. $v_{n,k} = v_{n,k} + \Delta d$;
      19. end while
      20. while $S_{\text{cur}} > S_{\text{last}}$
         21. $\Delta d = \Delta d/2$;
      22. end while
   23. return $\nu$;

gradient descent: This process reduces the gradient after each parameter update to improve the accuracy of the allocation strategy. We get the initial gradient as follows

$$\Delta d = \max\{\max[k \lambda_{k} - \min[k \lambda_{k}], \forall k] / N$$

(21)

Then reduce the gradient after each strategy update and continue to iterate, until the current gradient is less than the minimum threshold $\delta_{d}$. Here, we set the gradient of each update to half of the previous gradient, i.e., $\Delta d = \Delta d / 2$.

Theorem 1: Algorithm 3 terminates after a finite number of iterations.

Proof: Since the task offload is divided into gradients, the set of strategies that can be updated is finite-dimensional, and each update in the parameter update process is performed in the direction of less system utility, that is, monotonically decreasing. Therefore, $S_{\text{cur}}$ approaches the minimum after a limited number of iterations.

C. OPTIMAL RESOURCE ALLOCATION

This paper focuses on the optimization of task allocation and application loading policies, but the proposed algorithm is compatible with any computational resource allocation.
scheme. Since allocating computational resources to each loaded application in a fixed proportion for each server may not be the optimal allocation method, we propose an optimal method to further reduce task latency based on convex optimization theory. After the edge server performs load balancing, the number of tasks to be processed by each server is constant, so for each ES, n task delay minimization problem can be expressed as P4

$$P4: \min_{f} \frac{1}{\sum_{k \in K} d_{n,k} f_{n,k}} - v_{n,k}$$

s.t. $C5: \sum_{k \in K} f_{n,k} = F_n \forall n \in \mathcal{N}$

$$C7: f_{n,k} \in [0, F_n] \forall n \in \mathcal{N}, \forall k \in \mathcal{K}$$ (22)

**Theorem 2:** When strategy $d_{n,k}$ is determined, P4 is transformed into a convex optimization problem.

**Proof:** For brevity, let $y(f) = \sum_{k \in K} d_{n,k} f_{n,k}/\mu_k - v_{n,k}$, and $H = \{h_{k,k'}, \forall k, k' \in \mathcal{K}\}$ is the Hessian matrix of $y(f)$. When $d_{n,k}$ is known, we have

$$h_{k,k'} = \begin{cases} \frac{2d_{n,k}/\mu_k^2}{(f_{n,k}/\mu_k - v_{n,k})^3} & k = k' \\ 0 & \text{otherwise} \end{cases}$$

It can be noticed that $H$ is a positive definite matrix, and $y(f)$ is convex over $f$ [29]. Furthermore, C5 and C7 are linear, so P4 is a convex optimization problem.

Since P4 is a convex problem, we can get the optimal solution of P4 by solving the KKT condition of P4 [29]. Therefore, the computing resources of each ES that each application should be allocated with can be obtained, so that the task execution time of each ES is minimized.

Finally, we discuss the complexity of the proposed algorithm. For the calculation of the fitness function for each iteration, that is, the optimal task offload, the number of gradient descent is given by $O(\log_2(\Delta M/N))$, where $\Delta M$ is the maximum value of the difference between the maximum and the minimum task arrival rate for each application. In the strategy update and offload decision, only select one application at a time for offload, since there are at most $KN$ applications, the worst-case complexity is $O(N(NK)^2)$. For optimized application loading, the initial population has P individuals, and Q iterations are performed. The number of the genetic operations of selection, crossover, and mutation in each iteration is given by $O(PQ)$. Therefore, the total complexity is $O(PQ) \times O(\log_2(\Delta M/N)) \times O(N(N^2K^2)/2)$, because $\log_2(\Delta M/N)$ is close to a constant, then the overall time complexity of the genetic-based solution can be approximated by $O(PQN^3K^2)$. It can be seen that it is only related to the number of edge servers and the types of applications and has nothing to do with the amount of tasks requested.

**VII. SIMULATION RESULTS**

In this section, we will evaluate the performance of the proposed algorithm in terms of the improvement of the ASP’s situation in meeting the expected profits, the reduction of user latency and the stability of the ASP’s profits loss backlog mentioned above to verify its enhancement of user latency reduction while guaranteeing the ASP’s expected profits.

**A. SIMULATION SETTING**

Unless otherwise specified, the parameters are set as follows. We consider a mobile edge network with $N = 4$ base stations, where each base station has an ES and each ES and cloud server can load $K = 10$ applications. For each ASP, its flow utility conversion ratio, and expected profit follows a uniform distribution of $\epsilon_k \sim U(0.008, 0.012)$ $$/task$$, and $\epsilon_k^0 \sim U(0.8, 1.0)$ $$. The loading fee charged by the ES operator per time slot is $\gamma = 0.3$ $. All ESs can be connected to each other via a LAN, and its average transmission time is $\tau = 0.001s$, while the ratio of unit return delay to LAN delay is $\psi = 5$. Since the application task arrival rate on each ES obeys lognormal distribution in time domain, we perform normalization operation $\lambda_{n,k} = \lambda_{n,k} \times \Lambda_{\text{expect}}/\Lambda_{\text{orig}}$ as the actual task arrival rate when observing the performance of the algorithm under different task arrivals, where $\Lambda_{\text{orig}}$ is the actual number of tasks reached and $\Lambda_{\text{expect}}$ is the expected total number of tasks. The number of time slots we simulate is $T = 200$, with an interval of 2 minutes per slot. The other simulation parameters are listed in Table 2.

| Parameters | Value |
|-----------|-------|
| Maximum running memory capacity of ES, $C_n$ | 32 GB |
| Computing resource of ES, $F$ | 200 GHz |
| Amount of workload(cycles) required for application, $\mu_k$ | $[0.6, 1.0]$ GHz |
| Running memory required for application, $c_k$ | $[4, 8]$ GB |
| Average data size of task, $\eta_k$ | $[1.0, 1.5]$ MB |
| Penalty factor for GA, $\chi$ | $10^0$ |
| Average loading delay of the application $k$, $\xi_k$ | 0.1 s |
| Penalty weight for deficit queue, $\Gamma$ | 0.1 |

For performance comparison, we consider the following five strategies:

- **Non-cooperative and Greedy algorithm (Non-cooperative):** Each ES greedily chooses a strategy to load applications based on the size of the task arrival rate, and the application’s computing workload is either processed locally or uploaded to the cloud server to execute.

- **Greedy and cooperative algorithm (Greedy):** Each edge node greedily chooses an application loading strategy, and the application’s computing workload can be performed locally or offloaded to other ESs and cloud servers.

- **Time-delay minimization algorithm (Delay-optimal):** Federate application loading, load balancing, and resource allocation at each time slot to minimize system latency. Note that the algorithm does not take into account
account ASP’s expected profits in each time slot or in the long run.

- **Non-resource allocation algorithm (Non-RA):** Federate application loading and load balancing to minimize the objective function at each time slot, but allocate resources proportionally to each application in ES.

- **ICE algorithm (ICE) [30]:** Minimize service response time and overall outsourcing traffic by jointly optimizing edge service caching and workload scheduling strategies.

### B. PERFORMANCE EVALUATION

Figure 2(a) shows the change over time in the number of ASPs that achieve the expected profit under different strategies. In Figure 2(a), we can observe that the OATC and Non-RA strategies can achieve the expected profits of more ASPs, and that OATC strategy can achieve the expected profits of almost all ASPs after time slot 82. Delay-optimal strategy can maintain 7 ASPs for a long time to obtain its expected profits. The Greedy and ICE strategies can only satisfy about half of the ASPs’ profits in the long run. The reason is that the OATC algorithm proposed in this paper optimizes the profit of ASP in each time slot, and introduces historical ASP profit loss to adjust the weight of profit optimization, so when the expected profit of more ASPs in the first 20 time slots cannot be reached, the weight of profit optimization will be increased to enlarge the profit of the ASP, so the expected profit of the ASP can be better guaranteed in the long run. However, the Delay-optimal and ICE strategies do not take into account the profit of ASP. And the Greedy strategy makes decisions based on the number of tasks without taking into account the overhead during loading, so it will cause a certain profit loss for ASP. This result verifies that the OATC algorithm can outperform other algorithms in ensuring as many ASPs as possible to achieve the desired benefits in the long run.

Figure 2(b) shows the variation of the average system delay over time under different strategies. It can be observed that over 200 time slots the system average delay of OATC and Non-RA strategies gradually increases with time and then stabilizes, while that of other strategies is relatively stable in time domain. Since OATC and Non-RA strategies have to maintain the stability of the profit loss queue \( q(t) \) in the early stage, and more consideration is given to ensuring the expected profit of ASP, which causes the increase of delay. Furthermore, the Non-RA strategy has a higher delay after stabilization because it has not optimized resource allocation. And the Greedy strategy ignores the cooperation between ESs and leads to poor performance. The outcomes appear that the OATC algorithm is superior to the ICE algorithm, indicating that the OATC algorithm can have better performance in minimizing the average system delay, and the slightly downward trend of the delay curve in Figure 2(b) may be the reason for the difference in the number of tasks in different time slots.

Figure 2(c) shows the change over time of the ASP profit loss backlog under different strategies. It can be seen that the ASP profit loss backlog of Greedy, Delay-optimal and ICE strategies increases linearly with time. However, the OATC strategy can keep the ASP’s loss of profit backlog at a low level, and the maximum in 200 time slots is only 1258, which proves the long-term stability of the OATC algorithm in this article. Although the Non-RA strategy also has a lower queue backlog, it will cause a larger delay.

Figure 3(a) shows the relationship between the number of ASPs that achieve the expected profit and the number of application tasks. It can be seen that the number of ASPs that achieve the expected profit under all strategies increases with the number of tasks. OATC and Non-RA strategies can almost guarantee that all ASPs get their expected profits after the average number of tasks is greater than 85, while Delay-optimal and Greedy strategies can achieve the same effect when there are more tasks. In addition, neither ICE nor Non-cooperative strategies can reach the expected profits of all ASPs. The reason is that OATC and Non-RA strategies take into account the interests of ASP when performing load balancing. When the number of tasks reached is small and cannot achieve the expected profit of the ASP, the profit will be maximized first, and the priority of the application will be defined according to the loss of profit and the number of tasks to maximize the number of ASPs that achieve the expected profit. However, the Greedy strategy does not cooperate with ES to optimize the application loading strategy, which may increase the loading cost of ASP, and other solutions ignore the optimization of ASP benefits. Therefore, the figure shows that the OATC algorithm has better performance in optimizing the benefits of ASP, and it can satisfy more ASPs to achieve the expected benefits even when the number of tasks is low.

Figure 3(b) shows the relationship between the average system delay and the number of application tasks. In Figure 3(b), when the average number of tasks is greater than 85, the delay curve of the OATC strategy is closer to the curve of the Delay-optimal strategy than other strategies. At this time, OATC latency is 8.36% and 7.72% lower than ICE and Non-RA strategies, while OATC performance is poor when the number of tasks is small. When the number of tasks is small, the OATC algorithm will focus on increasing the benefits of ASP, thereby reducing the number of applications loaded, leading to increased latency, but when there are more tasks, the weight of benefit optimization will approach 0, and the OATC strategy will approach Delay strategy. Since the Non-cooperative and Greedy strategies do not consider the joint ES for application loading, more tasks are offloaded to the cloud for execution, which increases system latency. The results show that the OATC algorithm has higher system delay when the number of tasks is small, but has better performance when there are more tasks.

Figure 4 shows the relationship between the average system delay, the number of ASPs that reach the expected profit and the running memory capacity of the ES when the average task arrival number is 160. Since the number of applications that can be loaded in each time slot increases as the running memory increases, Figure 4 also indicates the relationship...
between them and the number of applications that can be loaded.

In Figure 4(a), as the running memory capacity increases, the number of ASPs that achieve the expected profit under Greedy strategy first increases and then decreases, while other strategies gradually increase and become stable. It can be seen that the number of ASPs that achieve the expected benefits under OATC and Non-RA strategies is significantly higher than that under other algorithms. The reason is that the number of tasks executed locally at ES increases with running memory, which increases not only the profit of ASP, but also the application load overhead. OATC and Non-RA strategies can optimize application loading strategies between ESs, so they weigh the benefits and costs of ASP. Because loading too many applications will increase the delay, ICE and Delay-optimal strategies will also maintain a stable
number of application loads, while the Greedy strategy is greedy for application loading, so it will cause more overhead when loading too many applications. Therefore, Figure 4 verifies that the OATC algorithm can obtain lower system delay and guarantee more ASPs to obtain the expected benefits when the memory capacity is different.

The simulation results of Figure 4(b) show that with the increase of running memory capacity, the delay curve of Greedy strategy first decreases and then increases, the delay curve of Non-cooperative strategy gradually decreases, and the curves of other strategies gradually decrease and stabilize after 32, and compared with ICE and Non-RA strategies, the delay of OATC is reduced by 6.03% and 10.2%. Due to the increase in the number of applications that can be loaded, the number of tasks unloaded to the cloud server is reduced, thereby reducing the system latency, and the ES service rate will reach its peak when the running memory continues to increase. Since OATC, ICE, Delay-optimal and Non-RA strategies have optimized application loading between ESs, the number of loaded applications will not continue to increase, thus ensuring a low and stable delay.

Figure 5 depicts the effect of the number of servers at $N = 4$ and the amount of running memory at $C = 28 GB$ on the convergence of the algorithm. From Figure 5(a) it can be seen that larger amount of running memory leads to faster convergence of the algorithm, but the effect of decreasing the number of iterations required to reach convergence decreases with increasing memory capacity, with the algorithm having almost the same convergence speed at $28 GB$ and $32 GB$ of running memory. In Figure 5(b), the number of iterations required to reach convergence is almost the same as the...
number of servers increases, and it can be concluded that the number of servers has less effect on the convergence of the algorithm.

Figure 6 plots the relationship between the profit loss backlog and the average system delay as \( V \) increases. It can be seen that as \( V \) increases, the average latency decreases, due to the increased weight of latency optimisation as \( V \) increases. As a result, the system focuses on load balancing decisions to further reduce latency. At the same time, the weight of ASP profit optimisation decreases, which leads to a rise in the profit loss backlog. Nevertheless, we can see that the queue length stabilises as \( V \) rises, at which point the consideration of historical profit loss will approach zero and the optimisation objective will be transformed into a system delay minimisation like Delay-optimal. Furthermore, as can be seen in Figure 6, a trade-off between latency and profit loss backlog can be made by adjusting \( V \). In reality, edge server operators can set the value of \( V \) by considering the latency requirements of users and the profitability of the ASP.

VIII. CONCLUSION

In this paper, we aim to solve the problem of minimizing system latency by collaborating multi-ES application loading, task allocation, and computational resource allocation in edge networks while ensuring ASPs’ long-term expected profits. To solve this long-term stochastic optimization problem, we use the Lyapunov optimization framework for single time slot transformation and improve the virtual queue in the framework to support scenarios with low task traffic, and then propose an online optimization algorithm based on the theory of GA to obtain the optimal policy at each time slot. The results illustrate that in a dynamic edge environment, the algorithm is able to achieve lower system latency while better guaranteeing the desired gain of the ASP than other algorithms, and demonstrates its higher stability through the analysis of virtual queues, in addition to the 6% reduction in average latency compared to the ICE algorithm. In this work, we provide a solution to enhance user QoE while ensuring ASP benefits from the MEC operator’s perspective, and in future work, we will consider radio channel allocation optimisation to improve the joint edge network optimisation framework.

REFERENCES

[1] S. Yang, F. Li, M. Shen, X. Chen, and X. Fu, “Cloudlet placement and task allocation in mobile edge computing,” IEEE Internet Things J., vol. 6, no. 3, pp. 3583–3586, Jun. 2019.
[2] Z. Ning, P. Dong, X. Kong, and F. Xia, “A cooperative partial computation offloading scheme for mobile edge computing enabled Internet of Things,” IEEE Internet Things J., vol. 6, no. 3, pp. 4804–4814, Jun. 2019.
[3] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, “A survey on mobile edge computing: The communication perspective,” IEEE Commun. Surveys Tuts., vol. 19, no. 4, pp. 2322–2358, 4th Quatr., 2017.
[4] W. Cao and L. Liu, “Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud,” in Proc. IEEE Int. Conf. Big Data (Big Data), Dec. 2018, pp. 191–200.
[5] L. Chen, C. Shen, P. Zhou, and J. Xu, “Collaborative service placement for edge computing in dense small cell networks,” IEEE Trans. Mobile Comput., vol. 20, no. 2, pp. 377–390, Feb. 2021.
[6] T. X. Tran, K. Chan, and D. Pompili, “COSTA: Cost-aware service caching and task offloading assignment in mobile-edge computing,” in Proc. 16th Annu. IEEE Int. Conf. Sens., Commun., Netw. (SECON), Jun. 2019, pp. 1–9.
[7] L. Chen, S. Zhou, and J. Xu, “Computation peer offloading for energy-constrained mobile edge computing in small-cell networks,” IEEE/ACM Trans. Netw., vol. 26, no. 4, pp. 1619–1632, Aug. 2018.
[8] Y. Liu, S. Wang, Q. Zhao, S. Du, A. Zhou, X. Ma, and F. Yang, “Dependability-aware task scheduling in vehicular edge computing,” IEEE Internet Things J., vol. 7, no. 6, pp. 4961–4971, Jun. 2020.
[9] M. Chen and Y. Hao, “Task offloading for mobile edge computing in software defined ultra-dense network,” IEEE J. Sel. Areas Commun., vol. 36, no. 3, pp. 587–597, Mar. 2018.
[10] H. Guo and J. Liu, “Collaborative computation offloading for multiaccess edge computing over fiber–wireless networks,” IEEE Trans. Veh. Technol., vol. 67, no. 5, pp. 4514–4526, May 2018.
[11] J. Zhao, Q. Li, Y. Gong, and K. Zhang, “Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks,” IEEE Trans. Veh. Technol., vol. 68, no. 8, pp. 7944–7956, Aug. 2019.
[12] R. Guo, L. Li, Y. Cui, T. Wei, and S. Hu, “Exploring placement of heterogeneous edge servers for response time minimization in mobile edge-cloud computing,” IEEE Trans. Ind. Informat., vol. 17, no. 1, pp. 494–503, Jan. 2021.
[13] H. Guo, J. Zhang, J. Liu, and H. Zhang, “Energy-aware computation offloading and transmit power allocation in ultradense IoT networks,” IEEE Internet Things J., vol. 6, no. 3, pp. 4317–4329, Jun. 2019.
[14] T. X. Tran and D. Pompili, “Joint task offloading and resource allocation for multi-server mobile-edge computing networks,” IEEE Trans. Veh. Technol., vol. 68, no. 1, pp. 856–868, Jan. 2019.
[15] P. Wang, Z. Zheng, B. Di, and L. Song, “HetMEC: Latency-optimal task assignment and resource allocation for heterogeneous multi-layer mobile edge computing,” IEEE Trans. Wireless Commun., vol. 18, no. 10, pp. 4942–4956, Oct. 2019.
[16] Y. He, J. Ren, G. Yu, and Y. Cai, “D2D communications meet mobile edge computing for enhanced computation capacity in cellular networks,” IEEE Trans. Wireless Commun., vol. 18, no. 3, pp. 1750–1763, Mar. 2019.
[17] V. Farhadi, F. Mehmerti, T. He, T. L. Porta, H. Khanmroush, S. Wang, and K. S. Chan, “Service placement and request scheduling for data-intensive applications in edge clouds,” in Proc. IEEE Conf. Comput. Commun. (INFOCOM), Apr. 2019, pp. 1279–1287.
[18] D. Harutyunyan, N. Shahriari, R. Boutabaa, and R. Riggio, “Latency and mobility-aware service function chain placement in 5G networks,” IEEE Trans. Mobile Comput., early access, Oct. 1, 2020, doi: 10.1109/TMC.2020.3028216.
[19] J. Xu, L. Chen, and P. Zhou, “Joint service caching and task offloading for mobile edge computing in dense networks,” in Proc. IEEE Conf. Comput. Commun. (INFOCOM), Apr. 2018, pp. 207–215.
[20] S. Bi, L. Huang, and Y.-J. A. Zhang, “Joint optimization of service caching placement and computation offloading in mobile edge computing systems,” IEEE Trans. Wireless Commun., vol. 19, no. 7, pp. 4947–4963, Jul. 2020.
[21] S.-E. Benbrahim, A. Quintero, and M. Bellaiche, “Live placement of interdependent virtual machines to optimize cloud service profits and penalties on SLAs,” IEEE Trans. Cloud Comput., vol. 7, no. 1, pp. 237–249, Jan. 2019.
[22] Y. Wang, Y. Yang, T. Wo, W. Jiang, and C. Hu, “Improving utilization through dynamic VM resource allocation in hybrid cloud environment,” in Proc. 20th IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS), Dec. 2014, pp. 241–248.
[23] S. Wang, X. Zhang, Z. Yan, and W. Wenhao, “Cooperative edge computing with sleep control under nonuniform traffic in mobile edge networks,” IEEE Internet Things J., vol. 6, no. 3, pp. 4295–4306, Jun. 2019.
[24] S. Wang, X. Zhang, J. Zhang, J. Feng, W. Wang, and K. Xin, “An approach for spatial-temporal traffic modeling in mobile cellular networks,” in Proc. 27th Int. Teletraffic Congr., Sep. 2015, pp. 203–209.
[25] R. Jackson and R. Cooper, “Introduction to queuing theory,” J. Oper. Res. Soc., vol. 33, p. 675, Jul. 1982.
[26] F. Guo, H. Zhang, H. Ji, X. Li, and V. C. M. Leung, “An efficient computation offloading management scheme in the densely deployed small cell networks with mobile edge computing,” IEEE/ACM Trans. Netw., vol. 26, no. 6, pp. 2651–2664, Dec. 2018.
YANZAN SUN (Member, IEEE) received the Ph.D. degree in control theory and control engineering from Tongji University, China, in 2011. From 2007 to 2009, he was a Research Intern with the Shanghai Research Center for Wireless Communication (Wico) and worked on spectrum sensing and dynamic spectrum allocation in cognitive radio networks. In 2009, he was awarded the National Government Study Abroad Scholarship to pursue his research as a Visiting Student with the Electrical Engineering Department, Columbia University. From 2011 to 2012, he was a Research Engineer with Huawei Technologies Company Ltd., where he researched on enhanced inter-cell interference coordination (eICIC) in 3GPP long term evolution (LTE) systems. He is currently an Associate Professor with the School of Communications and Information Engineering, Shanghai University. His research interests include wireless resource/interference management, network intelligence, and green communication.

XINKUN XIE received the B.E. degree in electronic information science and technology from Qingdao University, in 2019. He is currently pursuing the M.S. degree with the School of Communications and Information Engineering, Shanghai University. His current research interests include mobile edge computing and resource allocation.

FAN WU received the Ph.D. degree in economics from The University of Alabama, USA, in 2020. After that, she has been working full-time as a Postdoctoral Researcher with the School of Economics, Shanghai University, China. She has published peer-reviewed articles in SSCI/SCI journals and international conferences, and participated in several national and provincial-level research programs. Her research interests cover interdisciplinary areas, such as behavioral and experimental economics, resource and environmental economics, energy economics, and industrial organization. She is a member of the Chinese Economists Society.

SHUNQING ZHANG (Senior Member, IEEE) received the B.S. degree from the Department of Microelectronics, Fudan University, Shanghai, China, in 2005, and the Ph.D. degree from the Department of Electrical and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong, in 2009. He was with the Communication Technologies Laboratory, Huawei Technologies, as a Research Engineer, and as a Senior Research Engineer, from 2009 to 2014. He is currently a Senior Research Scientist and the Associate Director of the Intel Collaborative Research Institute on Mobile Networking and Computing, Intel Labs. His current research interests include 5G communication networks, interference management, nonorthogonal waveform design, energy efficient communications, relay, cognitive radios, joint radio frequency, and baseband design. He has published over 50 peer-reviewed journals and conference papers and over 40 granted patents with over 40 patents in pending status in the above areas.

SHUGONG XU (Fellow, IEEE) received the Ph.D. degree from the Huazhong University of Science and Technology, in 1996. He was a Senior Research Scientist with the Sharp Laboratories of America. In 2008, he joined Huawei Technologies, where he was the Research Director and a Principal Scientist with the Communication Technologies Laboratory. Among his responsibilities with Huawei, he founded and directed Huawei Green Radio Research Program. He is currently the Director of ICRI-MNC. He has published over 80 peer-reviewed research papers in top international conferences and journals. One of his most referenced articles has over 1200 Google Scholar citations, in which the findings were among the major triggers for the research and standardization of the IEEE 802.11s. He has over 20 U.S. patents granted. Some of these technologies have been adopted in international standards, including the IEEE 802.11, 3GPP LTE, and DLNA. His recent research interests include mobile networking and computing, next generation wireless communication platform, network intelligence, and SDN/NFV.

YATING WU (Member, IEEE) received the B.E. degree from the Beijing University of Posts and Telecommunications, in 2004, and the Ph.D. degree from Peking University, in 2009. From 2009 to 2010, she worked as a Senior Research Assistant with the Department of Electronic Engineering, City University of Hong Kong. She joined Shanghai University, in 2011, where she is currently a Lecturer with the Department of Communication Engineering. Her current research interests include signal processing for wireless communications, cooperative MIMO communications, and limited feedback techniques. She has been a member of the China Broadband Wireless IP Standard Group (ChinaB-WIPS), since 2011.