Hindawi
Mobile Information Systems
Volume 2019, Article ID 8049804, 17 pages
https://doi.org/10.1155/2019/8049804

*Research Article*

# Cooperative Runtime Offloading Decision Algorithm for Mobile Cloud Computing

**Xiaomin Jin**,[1,2] **Zhongmin Wang**,[1,2] **and Wenqiang Hua**[1,2]

[1]*School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an, Shaanxi 710121, China*
[2]*Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an University of Posts and Telecommunications, Xi'an, Shaanxi 710121, China*

Correspondence should be addressed to Xiaomin Jin; xmjin@xupt.edu.cn

Received 24 January 2019; Revised 11 July 2019; Accepted 12 August 2019; Published 17 September 2019

Academic Editor: Carlos A. Gutierrez



Mobile cloud computing (MCC) provides a platform for resource-constrained mobile devices to offload their tasks. MCC has the characteristics of cloud computing and its own features such as mobility and wireless data transmission, which bring new challenges to offloading decision for MCC. However, most existing works on offloading decision assume that mobile cloud environments are stable and only focus on optimizing the consumption of offloaded applications but ignore the consumption caused by offloading decision algorithms themselves. This paper focuses on runtime offloading decision in dynamic mobile cloud environments with the consideration of reducing the offloading decision algorithm's consumption. A cooperative runtime offloading decision algorithm, which takes advantage of the cooperation of online machine learning and genetic algorithm to make offloading decisions, is proposed to address this problem. Simulations show that the proposed algorithm helps offloaded applications save more energy and time while consuming fewer computing resources.

## 1. Introduction

With the rapid development of wireless communication and computer technologies, using mobile devices (MDs) has become increasingly common in daily life. Cisco predicted that the number of MDs worldwide will grow from 8.6 billion in 2017 to 12.3 billion in 2022 [1]. With the popularity of MDs, mobile Internet has entered a stage of rapid development. For instance, according to Internet Trend Report 2018 [2], the number of mobile Internet users in China has exceeded 753 million with a year-on-year growth rate of 8%, accounting for more than half of the total population. At the same time, stimulated by mobile Internet, a large number of mobile applications that provide users with varieties of services are developed. People are spending more and more time on MDs based on the idea that they want to do everything with the help of mobile applications. In recent years, with the advancement of semiconductor manufacturing techniques, MD performance has been improved with high-speed CPUs and large-capacity memories. However, on the one hand, the fast hardware consumes more energy and causes more heat dissipation. MDs are usually powered by batteries, whose capacity is limited because their size is limited to support MDs' portability. Unlike the semiconductor technology, the battery technology has not made breakthroughs in the short term, and the annual growth rate of battery capacity is only 5% [3]. The development of battery technology lags far behind the semiconductor technology developed by Moore's Law. In addition, according to Andy-Bill's law [4], MD operating systems (e.g., Android and iOS) and applications will become more complex, which causes more energy consumption and shortens the working time of MDs after one charge further. On the other hand, because of a series of factors such as architecture and volume, although MD processing capacity has been improved, it is still weak compared with that of the ordinary computer, making MDs take much time and energy to execute some applications, and even cannot execute heavy applications. As a result, these constraints lead to a poor user experience and prevent the further development of MDs.

Computation offloading is an effective way to solve the problem of limited resources of MDs. MDs' tasks can be migrated to external machines by computation offloading. Cloud computing, as a business computing model, can provide powerful external computing resources to MDs. Cloud computing is a pay-per-use model that supplies available, convenient, on-demand network access to a shared pool of configurable computing and storage resources [5]. Users can consume these cloud resources through networks just as they consume water through water pipes, electricity through wires, and natural gas through gas pipes. Enterprise and individual users no longer need to buy their own expensive high-performance computing equipments, as they can access high-performance computing and storage services through the cloud. Mobile cloud computing (MCC), which is a combination of cloud computing, mobile computing, and wireless networks [6–8], enhances the MD performance through cloud computing. For MDs, MCC provides a rich pool of computing resources that can be accessed through wireless networks. MCC helps MDs break through their resource limitations, frees them from heavy local workloads, and makes them more responsible for connecting users and the information domain. Benefiting from MDs' portability, users can connect with the resource-rich cloud anytime and anywhere in MCC and enjoy the convenience of informatization better. MCC has attracted wide attention from industry and academia because of its tremendous potential. There has been a lot of research on MCC. For instance, Cuervo et al. implemented a computation offloading architecture named "MAUI" [9], which offloads some heavy functions to cloud infrastructures to save energy of smartphones. Kosta et al. proposed a new code offloading framework named "ThinkAir" [10], which offloads applications to the cloud and uses multiple virtual machine images to parallelize functions' execution. Kemp et al. implemented a computation offloading framework named "Cuckoo" [11] for Android smartphones, which offloads tasks to a remote server to reduce smartphones' power consumption and to increase their speed. Moreover, there have been many mobile cloud applications for e-commerce [12], mobile healthcare [13], and mobile education [14]. It is believed that more and more types of mobile cloud applications will appear with the further development of MCC.

One important problem in the research field of cloud computing is how to make offloading decisions, which determine where tasks should be executed, locally or remotely. Offloading decision-making in MCC is different from that in grid computing and multiprocessor. In these fields, the optimization goal of offloading decision-making is to balance the load and minimize the edge cut and the migration volume [15]. In traditional cloud computing, the user equipment (e.g., desktop computer) is connected to the cloud via wired networks and is electrified by plugs and sockets, which suppress the need for energy-efficient data transmission techniques. In MCC, MDs are connected to the cloud via wireless links, which have limited bandwidth and consume a high amount of energy available in the mobile battery. Wireless networks have a serious impact on the energy-saving and time-saving effects of computation offloading in MCC [16]. This feature makes MCC to take time and energy consumption of computation offloading into account when making offloading decisions. Additionally, mobile users move among different environments, and wireless network conditions change constantly, making the offloading decision-making in MCC a dynamic problem. The problem of how to make offloading decisions is usually converted to the application partitioning problem. Application partitioning problems are NP complete, which causes offloading decision algorithms to consume more computing resources. However, most existing research on offloading decision is focused on optimizing the consumption (energy or time) of offloaded applications, ignoring the overhead of decision-making algorithms.

This paper focuses on runtime offloading decision in dynamic mobile cloud environments and proposes a cooperative runtime offloading decision algorithm that combines online machine learning (ML) and genetic algorithm (GA) to achieve two purposes: (i) making dynamic offloading decisions and (ii) reducing the consumption of the offloading decision algorithm. In the cooperative algorithm, offloading strategies developed by GA are used as the training data for online ML and offloading strategies predicted by online ML are used to accelerate GA's convergence. Different from existing works that mainly focus on reducing the consumption of offloaded applications but ignore the offloading decision algorithms' consumption, the cooperative algorithm reduces the consumption of offloaded applications while reducing its own computing resource consumption with the help of the cooperation of online ML and GA. In the process of computation offloading, a lot of offloading strategy data are generated. If these historical offloading strategy data are not utilized, a large amount of repeated calculations will consume more computing resources. The cooperative algorithm is proposed based on the idea that mobile applications have their own offloading habits and rules, which can be learned from the historical offloading strategy data with the help of ML techniques. The offline ML, which requires a large amount of training data to be trained in batch, is not practical for a single mobile user because it is difficult for him/her to collect enough training data. Conversely, the online ML, which learns using one instance at one time and trains its predictor step by step, is appropriate for a single mobile user. The main contributions of this paper are twofold:

(1) A cooperative runtime decision model, which aims to minimize the weighted total cost of offloaded applications while reducing the computing resource consumption of the offloading decision algorithm, is established to formulate the multisite offloading decision problem in dynamic mobile cloud environments.

(2) A novel cooperative runtime offloading decision algorithm based on the cooperation of online ML and GA is proposed, in which offloading strategies developed by GA are used as the training data for online ML and offloading strategies predicted by online ML are used to accelerate GA's convergence.

The remainder of this paper is organized as follows: In Section 2, we review the related work. Section 3 introduces the computation offloading system. The runtime offloading decision algorithm based on the cooperation of online ML and GA is illustrated in Section 4. In Section 5, we evaluate the proposed algorithm. Section 6 concludes this paper.

## 2. Related Works

There are three fundamental issues [17] in MCC: (i) where to offload, (ii) what to offload, and (iii) when to offload. The first and second issues indicate what kind of MCC architecture is used and what is used as the offloading unit (e.g., process, class, and function), respectively. The third issue indicates when the offloading unit is offloaded to the cloud. In this paper, we mainly focus on the third issue, so we review the related work that focuses on offloading decision.

As mentioned above, offloading decision is usually converted to application partitioning. For instance, Li et al. constructed a cost graph of a given application through profiling the information of computing time and data sharing of its procedure calls and then divided the application into the server part and client part by the branch-and-bound algorithm such that the energy consumed by the application is minimized [18]. Li et al. also proposed a task partitioning and allocation scheme that divides the data-processing tasks between the server and the MD to optimize tasks' energy cost [19]. Some researchers focused on multisite offloading, in which the computation can be offloaded to multiple cloud servers. Goudarzi et al. established a weighted model for multisite offloading in MCC in terms of execution time, energy consumption, and weighted cost and then proposed an offloading strategy algorithm based on GA [20]. They used a reserve population besides the original GA population to diversify chromosomes and modified genetic operators to adapt to the multisite offloading problem. They initialized genetic operators for the multisite offloading problem to reduce the probability of generating ineffective chromosomes and optimized the crossover operator by means of a local fitness, inbreeding, and crossbreeding to generate better chromosomes. Enzai and Tang formulated the multisite computation offloading problem in MCC as a multiobjective combinatorial optimization problem such that the execution time of the mobile computation, the energy consumption of the MD, and the cost of using the cloud services are minimized and then transformed it into a weighted single-objective optimization problem [21]. To solve this optimization problem, they proposed a heuristic algorithm based on greedy hill climbing. Niu et al. formulated the multiway application partitioning problem as an integer linear programming (ILP) problem and proposed an energy-efficient offloading decision algorithm to solve it [22]. Kumari et al. considered a trade-off between completion time and energy-savings for offloading in the multisite environment and proposed a two-step algorithm that consists of the cost-and-time-constrained task partitioning and offloading algorithm, the multisite task scheduling algorithm based on teaching, learning-based optimization, and the energy-saving on multisite using dynamic voltage and frequency scaling technique [23]. Sinha and Kulkarni proposed a fine-grained and multisite application partitioning approach to minimize the computation cost [24]. They converted the application partitioning problem to a label-assignment problem and used two heuristic algorithms to make offloading decisions. Khoda et al. made offloading decisions with the goal of meeting application delay requirements while maximizing the energy conservation and used a nonlinear optimization method based on the Lagrange multiplier to develop the offloading strategy [25].

At present, most studies (e.g., [18–25]) focus on static offloading decision algorithms, which assume that mobile cloud environments do not change. These algorithms develop offloading strategies through program analysis during the application development phase, and the offloading strategies are fixed after the completion of the application development. There are few studies on dynamic offloading decision algorithms, which develop offloading strategies at runtime, and whose offloading strategies change constantly to adapt to dynamic mobile cloud environments. Kovachev et al. implemented a middleware named "Mobile Augmentation Cloud Services" (MACS), which offloads Android application to the cloud adaptively, to minimize the transfer cost, the memory cost, and the CPU cost [26]. MACS partitions the application at runtime, making it able to adapt to dynamic mobile cloud environments. At runtime, MACS forms an ILP problem, whose solution is used as the offloading strategy. MACS solves a new ILP problem to adapt to the new environment after the environmental parameters change. Yang et al. designed an online solution named "Foreseer," which repartitions application at runtime in dynamic mobile cloud environments, to shorten the application completion time [27]. Foreseer predicts the network status periodically by users' historical mobility information and then updates the application partitions. Jin et al. proposed a runtime offloading decision algorithm based on the memory-based immigrants adaptive GA to optimize the weighted cost and explored the usage of concurrent multiple transfer in computation offloading for MCC to combat the challenges caused by wireless communications [28]. Eom et al. proposed an adaptive scheduling algorithm based on the offline ML for the mobile offloading system [29]. In [30], Eom et al. proposed an adaptive scheduler based on online ML. This scheduler needs a module that forwards and executes computation tasks in both local and remote units to provide feedback in the online training phase.

This paper focuses on runtime offloading decision in dynamic mobile cloud environments and considers reducing the offloading decision algorithm's consumption, which goes beyond existing works. To optimize the consumption of offloaded applications while reducing the consumption of the offloading decision algorithm, advantages of online ML and GA are taken through their cooperation. Different from works that simply use ML, the proposed algorithm does not need to generate additional training data. The training data are obtained from historical offloading strategies that have been used. Moreover, relationships among application components and the situation where there are multiple cloud servers are also considered.

## 3. Computation Offloading System

In this section, the computation offloading system is introduced. First, we introduce the computation offloading architecture, which determines how to offload the computation and indicates "where to offload." Then, we illustrate the application model, which determines the offloading unit and indicates "what to offload."

*3.1. Computation Offloading Architecture.* The computation offloading architecture is the foundation of MCC and determines the way of computation offloading. The architecture, illustrated in Figure 1, is helpful in understanding the process of computation offloading. The MD is connected to the cloud via Internet, which can be accessed through wireless access points. Servers in the cloud are connected via wired networks. The offloading decision algorithm runs in the strategy server that can be a cloudlet [31] or a server near the access point. Many existing offloading decision algorithms run in MDs, which ignore the consumption of the decision algorithms themselves. Let us take a hypothetical case: the consumption of the offloading decision algorithm is larger than the application consumption saved by the computation offloading. In this case, the application consumption decreases, but the total MD consumption increases. The requested and replied data of a strategy are quite little and frequent. The roundtrip latency of the cloud is larger than the time required for the strategy transmission. Compared with the cloud, the strategy server is very close to the MD, is more suitable to execute the offloading decision algorithm, and allows the offloading decision to be made timely. Nevertheless, compared with the cloud, whose resources can be considered unlimited, the resources of the strategy server are limited. Therefore, this paper establishes a cooperative runtime decision model and proposes a cooperative runtime offloading decision algorithm to reduce the consumption of offloaded applications while reducing the computing resource consumption of the offloading decision algorithm with the help of the cooperation of online ML and GA.

Because the proposed offloading decision algorithm runs in the strategy server, it is possible that MDs disconnect from the strategy server. In this architecture, application components are deployed on both MDs and the cloud by MD and cloud patterns. There have been some studies on how to reconstruct applications into MD and cloud patterns. For example, Zhang et al. implemented a refactoring tool, named "DPartner," to automatically transform the bytecode of an Android application into MD and cloud patterns [32]. After processing by DPartner, an Android application is reconstructed into two files. The first is the refactored Android application file (.apk file), which is the MD pattern file and is deployed in the MD. The second is the cloud pattern file, which is an executable jar file and contains the offloadable Java bytecode files cloned from the refactored application. The cloud pattern file is deployed in multiple cloud servers. If the MD disconnects from the strategy server, one solution is using a timeout retransmission mechanism to reconnect to the strategy server and re-requesting for the offloading strategy after a timeout. If the timeout retransmission mechanism fails, the local execution strategy will be used before the connection is restored.

The typical process of the computation offloading is described as follows: When an application component is about to execute, an offloading request is sent to the strategy server to obtain the offloading strategy. After receiving an offloading request, the strategy server returns the offloading strategy of that component. The component is executed according to the offloading strategy returned. If the offloading strategy indicates that this component is executed locally, it will be executed in the MD. On the contrary, if the offloading strategy indicates that this component is executed remotely, it will be executed in the cloud server that is assigned by the offloading strategy. Some components need input data before execution. There will be a data transmission if the data are not in the same place as the component's execution. For example, if a component is executed in the MD but its input data are in a cloud server, the data are transmitted to the MD through the wireless network. If a component is executed in a cloud server but its input data are in another cloud server, the data are transmitted through the wired network.

*3.2. Application Model.* The application is composed of many components (e.g., processes, classes, and functions), which are the offloading units in the computation offloading system. An application is represented by a graph $G = (V, E)$ [33] illustrated in Figure 2. The target is to partition the application into several parts at runtime according to the optimization objective. A vertex $v$ ($v \in V$) represents a component, which is modeled as a 3-tuple $c_v = \{\rho_v, l_v, o_v\}$. $\rho_v$ represents the amount of the component $c_v$'s instructions or CPU cycles, which can be obtained through application analysis. This application model is a fine-grained model, and the case where there are unoffloadable components is also considered. $l_v$ is a binary variable that represents whether the component $c_v$ is offloadable. $l_v = 0$ means the component $c_v$ is offloadable. $l_v = 1$ means the component $c_v$ is unoffloadable and must be executed locally. Some components of a mobile application are unoffloadable because they have to operate MD hardware (e.g., sensors and screen). The slash-filled circles in Figure 2 represent the unoffloadable components. $o_v$ represents the execution sequence of the component $c_v$. For instance, the component $c_2$ is executed after the completion of the component $c_0$ and component $c_1$ and needs the output data of these two components. An edge $e = (u, v)$ $(u, v \in V)$ represents the interactive relationship between the component $c_u$ and the component $c_v$, and its weight $d_{u,v}$ denotes the amount of interactive data. If two components have no interactive relationship, the edge weight is set to zero. In some cases, the last component $c_N$ has the output data that need to be passed to the first component $c_0$ before the application's completion. In other cases, the application completes after the completion of the component $c_N$.

## 4. Runtime Offloading Decision Algorithm

In this section, we illustrate the runtime offloading decision algorithm based on the cooperation of online ML and GA. It

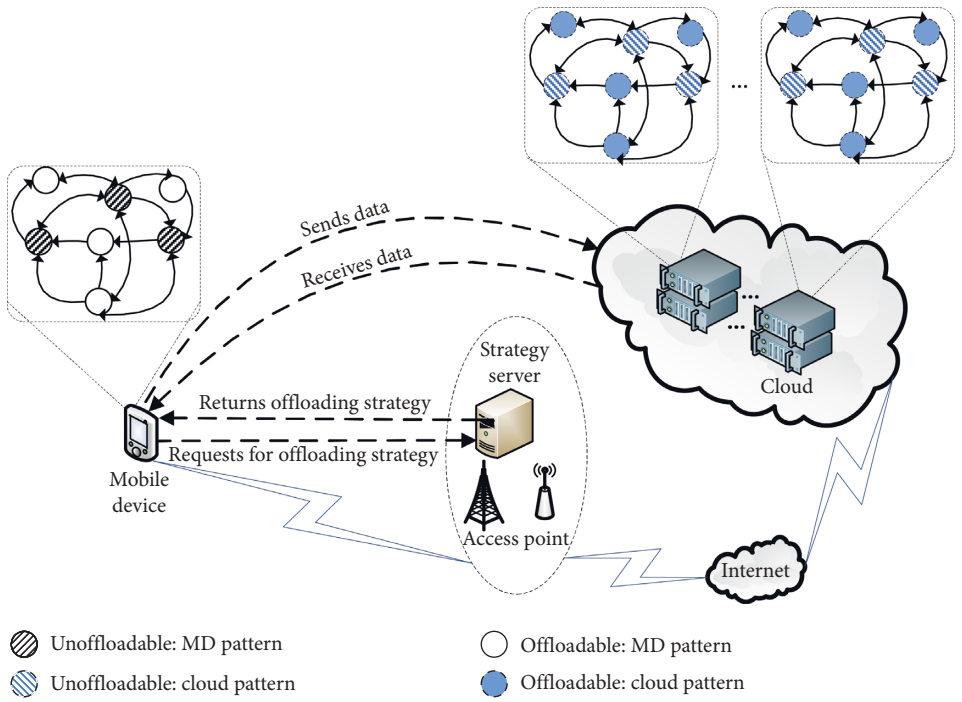


Figure 1: Computation offloading architecture.

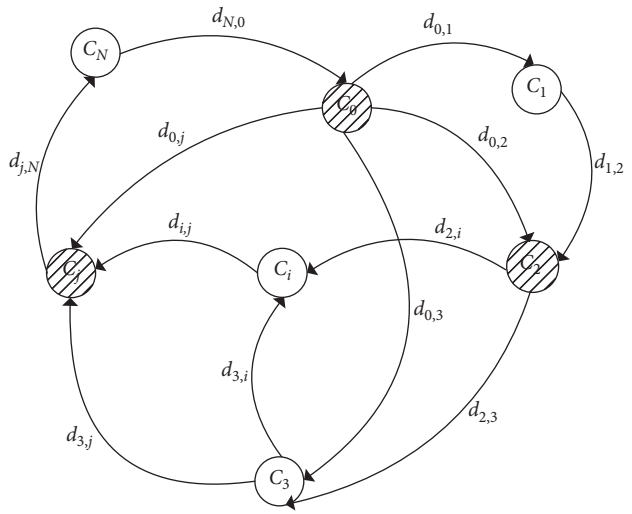


Figure 2: Application model.

determines when the offloading unit should be offloaded to the cloud and answers the question "when to offload." We first describe the optimization objective and then describe the cooperative runtime offloading decision algorithm in detail.

*4.1. Optimization Objective.* The offloading strategy of the application $G$ is modeled as a vector $X = (x_0, x_1, \ldots, x_N)$, in which $x_v$ $(0 \le x_v \le k,\ v \in V)$ represents the offloading strategy of the component $c_v$, and $k$ represents the maximum number of available cloud servers. $x_v = 0$ indicates that the component $c_v$ is executed in the MD, and $x_v = i$ $(1 \le i \le k)$ indicates that the component $c_v$ is executed in the cloud server $\mathrm{cs}_i$. In this paper, the optimization objective is to save both energy and time. These two optimization objectives are converted to one through two weights ($w_e$ and $w_t$, $w_e + w_t = 1$) based on the linear weighted sum method. $w_e$ represents the weight of energy cost, and $w_t$ represents the weight of time cost. The objective function, which represents the weighted total cost of the application $G$, is defined as equation (1), and the goal is to minimize it. $E_l(G)$ and $T_l(G)$ represent the energy and time costs when the application $G$ is executed locally, respectively. $E(G, X)$ and $T(G, X)$ represent the energy and time costs when the application $G$ is executed according to the offloading strategy $X$, respectively.

$$\begin{aligned} F(G,X) &= w_e \frac{E(G,X)}{E_l(G)} + w_t \frac{T(G,X)}{T_l(G)} \\ &= \sum_{v \in V} \left( w_e \frac{E_v^{x_v}}{E_l(G)} + w_t \frac{T_v^{x_v}}{T_l(G)} \right) \\ &\quad + \sum_{u \in V} \sum_{v \in V} \left( w_e \frac{E_{u,v}^{x_u,x_v}}{E_l(G)} + w_t \frac{T_{u,v}^{x_u,x_v}}{T_l(G)} \right). \end{aligned} \tag{1}$$

$T_v^{x_v}$ and $E_v^{x_v}$, which are illustrated in equations (2) and (3), respectively, represent the time and energy costs when the component $c_v$ is executed according to $x_v$. $f_m$ and $p_m$ represent the MD's working frequency and working power, respectively. When the component is executed in a cloud server, the MD enters the idle state, and its idle power is represented by $p_i$. $f_i^{cs}$ represents the cloud server $\mathrm{cs}_i$'s working frequency, and $dt_i$ represents the delay before a component's execution in the cloud server $\mathrm{cs}_i$. The delay contains queuing delay and some other preparation time. $t^{cs}$ represents the wired networks' throughput among these cloud servers. Cloud servers in the computation offloading system are represented by $\mathrm{CSs} = \{(f_i^{cs}, dt_i) \mid i = 1, 2, \ldots, k\}$.

$$T_v^{x_v} = \begin{cases} \dfrac{\rho_v}{f_{\mathrm{m}}}, & x_v = 0, \\ \dfrac{\rho_v}{f_{x_v}^{\mathrm{cs}}} + dt_{x_v}, & 1 \le x_v \le k, \end{cases} \tag{2}$$

$$E_v^{x_v} = T_v^{x_v} \times \begin{cases} p_{\mathrm{m}}, & x_v = 0, \\ p_i, & 1 \le x_v \le k. \end{cases} \tag{3}$$

$T_{u,v}^{x_u,x_v}$ and $E_{u,v}^{x_u,x_v}$, which are illustrated in equations (4) and (5), represent the time and energy costs caused by data transmission when the component $c_u$ is executed according to $x_u$ and the component $c_v$ is executed according to $x_v$, respectively. This also confirms that the consumption of data transmission in wireless networks has a serious impact on the effect of computation offloading. $t_i^{\mathrm{up}}$ ($t_i^{\mathrm{down}}$) represents the uplink (downlink) throughput between the cloud server $\mathrm{cs}_i$ and the MD. $p_i^{\mathrm{up}}$ ($p_i^{\mathrm{down}}$) represents the corresponding uplink (downlink) wireless network power. The uplink (downlink) wireless network power of the MD is modeled as $p = \alpha t + \beta$, which is a linear model obtained from practical experiments [34]. The strategy server, which is located near the access point, has only a "one hop" distance from the user. At the same time, the wireless bandwidth is large enough, and the requested and replied data (only the component sequence number and the component strategy are included) of a strategy are quite little. Therefore, the data transmission consumption caused by obtaining a strategy is ignored.

$$T_{u,v}^{x_u,x_v} = \begin{cases} \dfrac{d_{u,v}}{t_{x_v}^{\mathrm{up}}}, & x_u = 0 \wedge 1 \le x_v \le k \wedge u < v, \\ \dfrac{d_{u,v}}{t_{x_u}^{\mathrm{down}}}, & 1 \le x_u \le k \wedge x_v = 0 \wedge u < v, \\ \dfrac{d_{u,v}}{t_{x_v}^{\mathrm{down}}}, & x_u = 0 \wedge 1 \le x_v \le k \wedge u = 0 \wedge v = N, \\ \dfrac{d_{u,v}}{t_{x_u}^{\mathrm{up}}}, & 1 \le x_u \le k \wedge x_v = 0 \wedge u = 0 \wedge v = N, \\ \dfrac{d_{u,v}}{t^{\mathrm{cs}}}, & 1 \le x_u, x_v \le k \wedge x_u \ne x_v, \\ 0, & \text{otherwise}, \end{cases} \tag{4}$$

$$E_{u,v}^{x_u,x_v} = T_{u,v}^{x_u,x_v} \times \begin{cases} p_{x_v}^{\mathrm{up}}, & x_u = 0 \wedge 1 \le x_v \le k \wedge u < v, \\ p_{x_u}^{\mathrm{down}}, & 1 \le x_u \le k \wedge x_v = 0 \wedge u < v, \\ p_{x_v}^{\mathrm{down}}, & x_u = 0 \wedge 1 \le x_v \le k \wedge u = 0 \wedge v = N, \\ p_{x_u}^{\mathrm{up}}, & 1 \le x_u \le k \wedge x_v = 0 \wedge u = 0 \wedge v = N, \\ p_i, & 1 \le x_u, x_v \le k \wedge x_u \ne x_v, \\ 0, & \text{otherwise}. \end{cases} \tag{5}$$

The objective function in dynamic mobile cloud environments is introduced, as shown in Figure 3. $F_h^1$ represents the cost caused by the components that have been executed at time $h$. Each executed component is given an offloading strategy and has unique environmental parameters when it is executed. $F_h^1$ is the sum of all executed components' cost, and it is fixed. $F_h^2$ represents the cost caused by the component that is being executed. $F_h^2$ is a changing value because the environment may change during its execution. $F(G_h, X_h)$ represents the weighted total cost of components that are not executed at time $h$.

*4.2. Cooperative Runtime Offloading Decision Algorithm.* The pseudocode of the cooperative runtime offloading decision algorithm is illustrated in Figure 4. The offloading decision algorithm runs in the strategy server and returns the offloading strategy obtained from the current decision-making module (DMM) after receiving the offloading strategy request (Lines 06–15). In this algorithm, the online ML-based DMM and GA-based DMM cooperate with each other to make offloading decisions. The offloading strategy developed by the GA-based DMM is used to update the online ML-based DMM (Line 12). The online ML-based DMM enhances GA's convergence and ability to adapt to environmental changes by replacing GA's worst chromosome (Line 29). At the same time, inspired by the memory-based immigrants method [35], mutation and replacement operations are also used to enhance GA's ability further (Lines 31-32). $X_{\mathrm{OML}}$ represents the offloading strategy predicted by the online ML-based DMM. $X_{\mathrm{OML}}$ is used as the base to create immigrants to replace the worst $r_i \times M$ chromosomes in GA's population. $F_{\mathrm{OML}}$ represents the weighted total cost predicted by the online ML-based DMM. $F_{\mathrm{OML}} - F_{\mathrm{e}} \le F_{\mathrm{th}}$ means the weighted total cost $F_{\mathrm{OML}}$ predicted by the online ML-based DMM is good enough so that the online ML-based DMM is used as the current DMM (Lines 19-20). Otherwise, the GA-based DMM is used (Lines 21-22). $F_{\mathrm{th}}$ represents the threshold, and $F_{\mathrm{e}}$ represents the empirical weighted total cost that is calculated according to equation (6), in which $I$ represents the number of the application's completions and $F_i$ represents the weighted total cost of the $i$-th application's execution. If the current DMM is the GA-based DMM, the runtime offloading decision algorithm runs to find the optimal offloading strategy, which consumes more computing resources (Lines 25–34). On the contrary, if the current DMM is the online ML-based DMM, the offloading strategy is developed by its prediction when receiving the offloading strategy request. In this situation, the runtime offloading decision algorithm does not need to run and remains idle, which saves more computing resources. Since the online ML-based DMM and GA-based DMM are two key parts of this algorithm, these two DMMs are illustrated in detail in the following description.

$$F_{\mathrm{e}} = \begin{cases} 0, & I = 0, \\ \dfrac{\sum_{i=1}^{I} F_i}{I}, & I > 0. \end{cases} \tag{6}$$

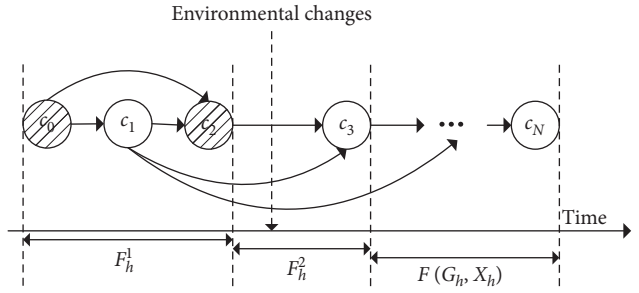


Figure 3: An example of the objective function in dynamic mobile cloud environments.

*4.2.1. GA-Based Decision-Making Module.* In this part, we illustrate the GA-based DMM, which makes offloading decisions using GA. The reason why we choose GA is explained as follows: The offloading decision problem is a dynamic combinatorial optimization problem, and GA is a classic optimization algorithm, which is easy to implement and has the relatively complete theory. In this paper, the offloading strategy can be directly used for the chromosome encoding of GA after simple mapping. Besides, GA has many advantages, such as finding the global optimal solution, being insensitive to the objective function (linear or nonlinear), and being robust. First, we introduce the encoding and fitness function. Then, we introduce the genetic operations, including selection, crossover, and mutation.

(1) *Encoding and Fitness Function*. In GA, a chromosome is a possible solution of the optimization problem, and it is usually encoded as a set of binaries. Since multiple cloud servers are considered in the computation offloading system, the genes are encoded as integers between 0 and $k$. As previously described, some application components cannot be offloaded such that their offloading strategies are always zero (local execution). There is no need to encode these components' offloading strategies into the chromosome. A vector that stores the mapping relationship between the complete offloading strategies and the chromosome's genes is utilized for the evaluation of the chromosome. For example, through the chromosome $Y = (y_0, y_1, \ldots y_i, \ldots, y_{n_o-1})$ $(0 \le y_i \le k,\ 0 \le i \le n_o - 1)$, in which $n_o$ represents the number of offloadable components, it can be known that $y_i$ is the offloading strategy of the component whose sequence number is $m_1(i)$. The mapping function $m_1(i)$ converts the gene's sequence number $i$ to the offloadable component's sequence number. The offloading strategy of the component $c_v$ can be obtained from formula (7). Fitness function, shown in equation (8), is used to evaluate the chromosome.

$$x_v = \begin{cases} y_i, & l_v = 0 \wedge o_v = m_1(i), \\ 0, & l_v = 1, \end{cases} \tag{7}$$

$$f_h = \frac{1}{F_h^1 + F_h^2 + F(G_h, X_h)}. \tag{8}$$

(2) *Genetic Operations*. GA has three basic operations: selection, crossover, and mutation. Selection operation selects chromosomes from the population for other operations. The roulette wheel selection method is a commonly used selection method, which generates a random number $r \in [0, 1)$ in each selection and selects the first chromosome $i$ that satisfies $r < \sum_{j=1}^{i} f_j / f_{\text{sum}}$ ($f_j$ represents the fitness of the chromosome $j$, and $f_{\text{sum}}$ represents the sum of all chromosomes' fitness). Crossover operation recombines part genes of two chromosomes to generate new chromosomes. In crossover operation, the crossover probability is a key parameter that affects the GA performance seriously. A large crossover probability generates more new chromosomes but reduces the convergence rate of GA. On the contrary, a small crossover probability generates fewer new chromosomes and leads to a local optimization result. However, the crossover probability is a constant in the standard GA. To solve this problem, the adaptive GA [36] is proposed to generate a changing and adaptive crossover probability. The adaptive crossover probability $\text{pr}_c$ is calculated by equation (9). $\text{pr}_{c1}$ and $\text{pr}_{c2}$ are two constants. $f_1$ is the larger one of two parent chromosomes' fitness. $f_{\text{avg}}$ and $f_{\max}$ are the average and maximum fitness of the population, respectively. Mutation operation changes genes of the selected chromosome between 0 and $k$ to generate a new chromosome. Similarly, a too large or too small mutation probability also affects the GA performance seriously. A large mutation probability maintains fewer genes of the parent chromosome and converts GA to be a random search algorithm. On the contrary, a small mutation probability prevents the generation of new chromosomes. The adaptive mutation probability $\text{pr}_m$ is calculated by equation (10). $\text{pr}_{m1}$ and $\text{pr}_{m2}$ are two constants. $f$ is the fitness of the parent chromosome. Compared with the constant probability, the following equations calculate the probabilities in the form of a right-angled trapezoid, which reduces the probability when the parent chromosomes' fitness is large and hence helps to retain good chromosomes:

$$\text{pr}_c = \begin{cases} \text{pr}_{c1} - \dfrac{(\text{pr}_{c1} - \text{pr}_{c2})(f_1 - f_{\text{avg}})}{f_{\max} - f_{\text{avg}}}, & f_1 \ge f_{\text{avg}}, \\ \text{pr}_{c1}, & f_1 < f_{\text{avg}}, \end{cases} \tag{9}$$

$$\text{pr}_m = \begin{cases} \text{pr}_{m1} - \dfrac{(\text{pr}_{m1} - \text{pr}_{m2})(f - f_{\text{avg}})}{f_{\max} - f_{\text{avg}}}, & f \ge f_{\text{avg}}, \\ \text{pr}_{m1}, & f < f_{\text{avg}}. \end{cases} \tag{10}$$

*4.2.2. Online ML-Based Decision-Making Module.* Attributes used in the online ML-based DMM are a $(2k + 1)$-tuple $\{F_h, t_1^{\text{up}}, t_1^{\text{down}}, \ldots, t_k^{\text{up}}, t_k^{\text{down}}\}$. $F_h$ $(F_h = F_h^1)$ represents the weighted total cost at time $h$, which is the cost of the components that have been executed at time $h$.$F_h$ is a comprehensive reflection of the offloading situations and environments before time $h$. $t_i^{\text{up}}$ and $t_i^{\text{down}}$ $(1 \le i \le k)$ represent the uplink and downlink throughput between the MD

```
(01) calculate E_l(G) and T_l(G);
(02) initialize GA's population;
(03) //initialize DMM
(04) DMM ⟵ GA
(05) while (running) do
(06)     if (receive the offloading strategy request) then
(07)         if (DMM == online ML) then
(08)             strategy = getStrategy(online ML);
(09)         else if (DMM == GA) then
(010)            strategy = getStrategy(GA);
(011)            //F_h = F_h^1, t_s is the throughput set, X is the offloading strategy
(012)            updateOnlineMLbasedDMM(F_h, t_s, X);
(013)        end if
(014)    sendStrategy(strategy);
(015)    end if
(016)    if (environment changes) then
(017)        recalculate F_h^2 and update environmental parameters;
(018)        F_OML = getPredictedFbyOnlineMLbasedDMM(F_h^1, F_h^2, t_s, X);
(019)        if (F_OML − F_e ≤ F_th) then
(020)            DMM ⟵ online ML;
(021)        else then
(022)            DMM ⟵ GA;
(023)        end if
(024)    end if
(025)    if (DMM == GA) then
(026)        GA operations: selection, crossover, and mutation, and evaluation;
(027)        //F_w is the worst objective function value in GA's population
(028)        if (F_OML ≤ F_w) then
(029)            replace the worst chromosome with X_OML;
(030)            //memory-based immigration
(031)            mutate(X_OML, r_i × M, p_m^i);
(032)            replace the worst r_i × M chromosomes in GA's population;
(033)        end if
(034)    end if
(035) end while
```

Figure 4: Cooperative runtime offloading decision algorithm.

and the cloud server $cs_i$, respectively. Some application components must be executed in the MD, and their offloading strategies are fixed (always zero) so that there is no need to make offloading decisions for them. Similar to the encoding of the chromosome in the GA-based DMM, the online ML-based DMM is composed of $n_o$ classifiers. From the perspective of the online ML-based DMM, the offloading decision-making problem is a classification problem, in which there is a need to classify the application components into $(k+1)$ categories. Here, "$k$" represents that a component can be offloaded to $k$ cloud servers, and "1" represents that a component can be executed locally (in the MD). For example, $k = 1$ ($(k+1) = 2$) means that the application components are classified into two parts. One part of components is executed in the cloud server, and another part of components is executed locally.

In the online training phase (Line 12 in Figure 4), the training data of classifier $cf_i$ ($m_2(o_h) \le i \le n_o - 1$) are a vector $(F_{h,i}, t_1^{up}, t_1^{down}, \ldots, t_k^{up}, t_k^{down}, y_i)$, in which $(F_{h,i}, t_1^{up}, t_1^{down}, \ldots, t_k^{up}, t_k^{down})$ is the input and $y_i$ $(0 \le y_i \le k)$ is the output that $cf_i$ uses for learning. $o_h$ represents the sequence number of the component that is to be executed at time $h$, and mapping function $m_2(o_v)$ converts the offloadable component's sequence number $o_v$ to the classifier's sequence number. $F_{h,i}$ is calculated by equation (11), in which $\Delta F_{o_v}$ represents the component $c_v$'s weighted total cost calculated according to its offloading strategy. $y_i$, generated by the GA-based DMM, represents the offloading strategy of the component whose sequence number is $m_1(i)$.

$$F_{h,i} = \begin{cases} F_h, i = m_2(o_h), \\ F_{h,i-1} + \sum_{m_1(i-1) \le o_v < m_1(i)} \Delta F_{o_v}, \quad m_2(o_h) < i \le n_o - 1. \end{cases} \tag{11}$$

In our algorithm, the online ML-based DMM needs to predict the offloading strategy $X_{OML}$ and calculate the weighted total cost $F_{OML}$ (Line 18 in Figure 4). The offloading strategy of the offloadable component $c_v$ is predicted by the classifier $cf_{m_2(o_v)}$ with the input attribute vector $(F_{h,m_2(o_v)}, t_1^{up}, t_1^{down}, \ldots, t_k^{up}, t_k^{down})$. $F_{OML}$ is calculated by

$$F_{OML} = F_h + \sum_{o_h \le o_v \le o_N} \Delta F_{o_v}. \tag{12}$$

## 5. Simulation and Analysis

In this section, a series of simulations in dynamic mobile cloud environments are conducted to evaluate the proposed offloading decision algorithm. First, the setup of these

simulations is described. Then, performance of our algorithm with different online ML techniques is compared. Finally, we evaluate our algorithm.

*5.1. Setup.* The default simulation parameter configuration is shown in Table 1. Applications are generated according to their parameters listed in Table 1. $\mathrm{pr}(l_v = 1)$ is the probability that the component $c_v$ is unoffloadable. The trajectory, generated based on the mobility model shown in Figure 5, is used to simulate the user's moving path. The user movement among different squares leads to changes in network parameters, which are used to simulate the environmental changes in dynamic mobile cloud environments. A 3-tuple $\{s, \Delta h, \gamma\}$ is used to model the users' mobility, in which $s$ represents the square where the user is located, $\Delta h$ represents the time the user takes to cross a square, and $\gamma$ represents the probability that the user still stays in the same square after $\Delta h$. $(1 - \gamma)/\theta$ is the probability that the user moves to an adjacent square, and $\theta$ is the number of a square's adjacent squares. $t^{\mathrm{up}}$ ($t^{\mathrm{down}}$) represents the uplink (downlink) throughput between the MD and cloud servers. The MD is connected to $k$ cloud servers, and $k$ uplink (downlink) throughput parameters are generated according to the distribution of $t^{\mathrm{up}}(t^{\mathrm{down}})$. To simplify the network environment, network parameters in a square are stable. Network parameters differ in different squares, and thus, network parameters change when the user moves from one square to another. If some squares are considered to constitute an area, then this change can be considered a change from the wireless network itself. This change can be regarded as a result of users' movement or changes in the wireless channel itself. Users in a square have eight movement directions, which are shown in the square $S_{12}$, and the number of directions decreases in marginal squares, which are shown in the square $S_0$. The user's trajectory is modeled as an array $((s_1, \Delta h_1), (s_2, \Delta h_2), \ldots, (s_n, \Delta h_n))$, and a simple trajectory example (from $S_2$ to $S_{23}$) is given in Figure 5. A 30-minute random trajectory is used in these simulations. In our algorithm, $M$ and $e_{\mathrm{ar}}$ represent the population size and accuracy requirement of the GA-based DMM, respectively.

As shown in Figure 6, a Java-based numerical simulation platform, which is built according to the computation offloading system introduced in Section 3, is used for simulation experiments. The application execution simulation module (AESM) simulates the application's execution. It sends a request to obtain the offloading strategy of the component that is about to be executed. The offloading decision algorithm module (ODAM) simulates the strategy server, and the offloading decision algorithms to be evaluated run in it. After receiving the offloading strategy request, the ODAM returns the offloading strategy of the component to the AESM. After receiving the offloading strategy, the AESM executes the component according to the offloading strategy. The random trajectory generation module (RTGM) generates the random user moving trajectory according to the mobility model introduced in Figure 5. The environment monitoring module (EMM) monitors the environmental changes and notifies the ODAM and AESM according to the

Table 1: Default simulation parameter configuration.

| Item | Configuration |
|---|---|
| Application | $\rho_v \sim N(2000, 1500)$Minstructions, $\mathrm{pr}(l_v = 1) = 10\%$, $d_{u,v} \sim N(300, 150)$ kbytes |
| Mobility | Number of squares $= 5 \times 5$, $\Delta h \sim U(0.25, 5)$ s, $\gamma = 0.1$, $t^{\mathrm{up}} \sim N(1.05, 0.95)$ Mbps, $t^{\mathrm{down}} \sim N(4.55, 4.45)$ Mbps |
| MD | $f_{\mathrm{m}} = 500$ MHZ, $p_{\mathrm{m}} = 800$ mW, $p_i = 50$ mW, $\alpha^{\mathrm{up}} = 283.17$ mW/Mbps, $\alpha^{\mathrm{down}} = 137.01$ mW/Mbps, $\beta = 132.86$ mW |
| Cloud servers | CSs = {(1000 MHZ, 0.1 s), (1875 MHZ, 0.125 s), (3500 MHZ, 0.175 s), (4000 MHZ, 0.2 s), (5000 MHZ, 0.225 s)}, $t^{\mathrm{cs}} = 100$ Mbps |
| Our algorithm | $F_{\mathrm{th}} = -0.005$, $r_i = 0.08$, $\mathrm{pr}^i_{\mathrm{m}} = 0.01$, $\mathrm{pr}_{\mathrm{c1}} = 0.9$, $\mathrm{pr}_{\mathrm{c2}} = 0.6$, $\mathrm{pr}_{\mathrm{m1}} = 0.1$, $\mathrm{pr}_{\mathrm{m2}} = 0.05$, $M = 50$, $e_{\mathrm{ar}} = 1 \times 10^{-3}$, $w_{\mathrm{e}} = 0.5$, $w_{\mathrm{t}} = 0.5$ |

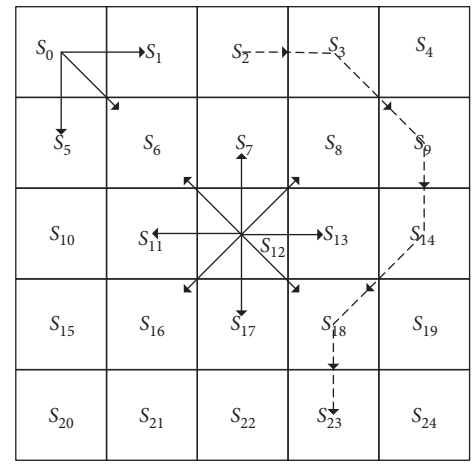


Figure 5: Mobility model.

trajectory generated by the RTGM. The EMM reads the array $\mathrm{tr} = \{(s_1, \Delta h_1), (s_2, \Delta h_2), \ldots, (s_n, \Delta h_n)\}$ that represents the user's moving path and sends network parameters of the square $s_i$ after waiting for $\Delta h_{i-1}$ ($\Delta h_0 = 0$). Network parameters differ in different squares, and thus, the environment constantly changes during the user's movement. The changes can simulate the variation of network parameters caused by users' movement as well as the variations of the network itself. The dynamic mobile cloud environments are simulated by these two modules. After receiving the environmental change notification, the offloading decision algorithm remakes offloading decisions with new environmental parameters. In the meanwhile, the AESM adjusts the component execution time and weighted total cost if there are data remaining to be transferred.

*5.2. Performance Comparison of Different Online ML Techniques.* The online ML-based classifier has a direct impact on the complexity and accuracy of the proposed algorithm. Therefore, the performance of the proposed algorithm with the commonly used online ML techniques implemented by Weka [37] is compared to choose the appropriate online ML technique. The weighted total cost and clock ticks with different online ML techniques are shown in Table 2. It can be seen that the weighted total cost

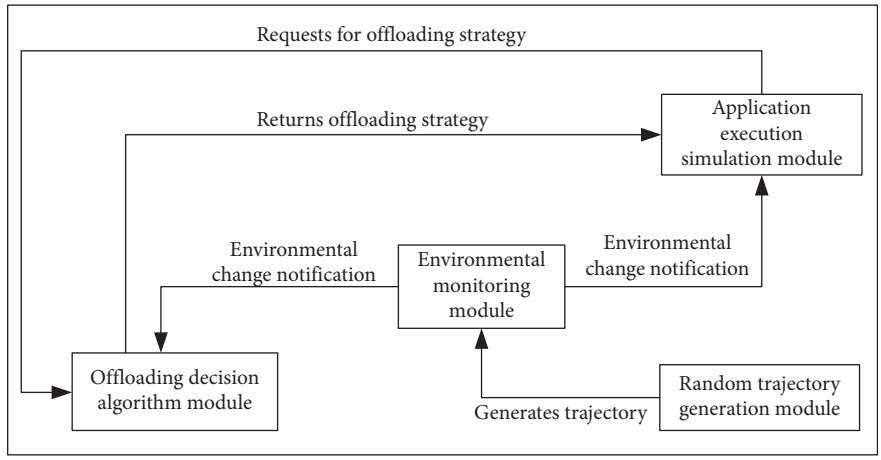


Figure 6: Numerical simulation platform framework.

Table 2: Weighted total cost and clock ticks with different online ML techniques.

| Performance | Techniques | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | KStar | NNge | NaiveBayes Updateable | RacedIncremental LogitBoost | LWL | IB1 | DMNBtext | NaiveBayesMultinomial Updateable |
| Cost (%) | 68.77 | 67.54 | 68.15 | 66.96 | 69.88 | 68.64 | 71.81 | 68.47 |
| Ticks ($10^3$) | 39.50 | 35.96 | 40.48 | 42.68 | 42.88 | 37.94 | 39.41 | 43.17 |

of RacedIncrementalLogitBoost is the smallest (66.96%) and the weighted total cost of DMNBtext is the largest (71.81%). In this paper, we focus on offloading decision algorithms' computing resource consumption, which is measured in units of CPU clock ticks. It can be seen that the clock ticks consumed by NNge is the least ($35.96 \times 10^3$ ticks) and the clock ticks consumed by NaiveBayesMultinomialUpdateable is the most ($43.17 \times 10^3$ ticks). The clock ticks consumed by LWL and NaiveBayesMultinomialUpdateable are larger than that consumed by RacedIncrementalLogitBoost, and their weighted total costs are also larger than that of RacedIncrementalLogitBoost, which shows that these two techniques consume more computing resources and do not achieve good results. The clock ticks consumed by KStar, NNge, NaiveBayesUpdateable, IB1, and DMNBtext are fewer than that consumed by RacedIncrementalLogitBoost, but their weighted total costs are larger than that of RacedIncrementalLogitBoost, which shows that these techniques consume fewer computing resources but do not achieve good results. For an offloading decision algorithm, it first needs to guarantee that the weighted total cost is optimal and then considers to reduce its computing resource consumption as much as possible. Therefore, RacedIncrementalLogitBoost, which has the smallest weighted total cost and consumes fewer clock ticks, is selected for the cooperative runtime offloading decision algorithm.

*5.3. Evaluation of the Cooperative Runtime Offloading Decision Algorithm.* To evaluate the proposed algorithm, it is compared with other runtime offloading decision algorithms. To express them succinctly in the following description, their abbreviations are used. AO is a traditional and intuitive runtime offloading decision algorithm. The reason for choosing AO is that we want to evaluate the performance of the offloading decision algorithm that does not consider the network conditions in MCC. The offloading decision problem is usually converted to the application partitioning problem, which is a combinatorial optimization problem. ILP and evolutionary algorithms are often used to solve the application partitioning problem. Therefore, ILP and GA are chosen as the comparison algorithms. Another reason for choosing GA is that we need to confirm that the proposed algorithm, which relies on the cooperation of online ML and GA, does work.

(1) AO: the thin client algorithm that *always offloads* offloadable components to cloud servers and selects cloud servers randomly in scenarios where there are more than one cloud servers

(2) ILP1: the algorithm that resolves the *ILP* problem when detecting the environmental changes and uses the local execution strategy as default

(3) ILP2: the algorithm that resolves the *ILP* problem periodically and uses the local execution strategy as default

(4) GA: the algorithm that reruns *GA* when detecting the environmental changes

(5) OMLGA: the proposed algorithm based on the cooperation of *online ML and GA*

Figure 7 shows the weighted total cost of five algorithms under different applications. In this simulation, these applications have different numbers of components. The

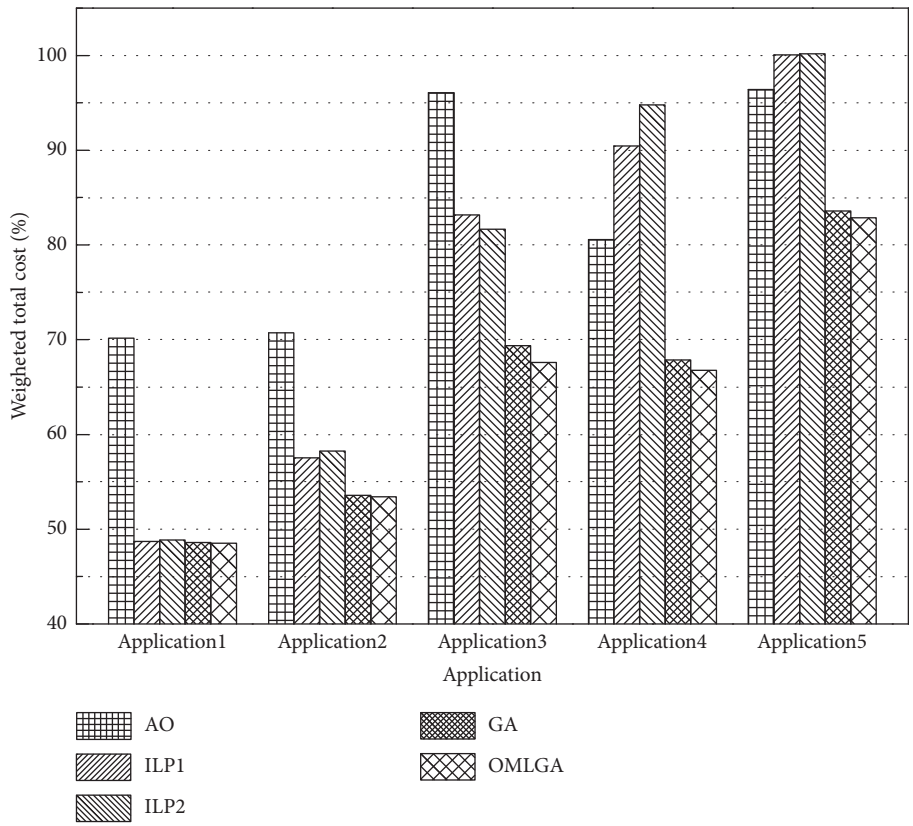


Figure 7: Weighted total cost under different applications.

numbers of components of these applications are 20, 40, 60, 80, and 100, respectively. It can be seen that weighted total costs of OMLGA are smaller than those of other algorithms. Weighted total costs of AO are larger than those of GA and OMLGA, which means AO saves less consumption than GA and OMLGA. When the environment changes, some components become unsuitable to be offloaded. AO offloads all offloadable components to the cloud servers without considering environmental changes and relationships among the components, which results in much additional consumption caused by data transmission. Weighted total costs of ILPs (ILP1: 48.71% and ILP2: 48.85%) are close to but larger than those of GA (48.60%) and OMLGA (48.52%) in Application1. However, as the number of components increases, ILPs' performance becomes bad, even worse than that of AO in Application4 and Application5. Running time ILPs take to find the optimal solution is exponential with the number of variables, which causes ILPs to take much time to partition an application that has a large number of components. Long running time makes ILPs unable to adapt to environmental changes timely and hence provides wrong offloading strategies. Weighted total costs of GA (48.60% and 53.56%) are close to those of OMLGA (48.52% and 53.42%) in Application1 and Application2. The reason is that Application1 and Application2 have relatively few components so that GA can find the optimal solution timely to adapt to environmental changes. Weighted total costs of GA are larger than those of OMLGA in Application3, Application4, and Application5, which results from that these applications' numbers of components are large such that GA cannot find the optimal solutions timely, but OMLGA can.

Figure 8 illustrates the clock ticks under different applications. In Application1, the clock ticks of OMLGA are smaller than those of ILP2 and larger than those of ILP1, which illustrates that ILP1 consumes fewer computing resources in small-scale applications. In other applications, clock ticks of OMLGA are smaller than those of ILPs. It is noticeable that clock ticks of OMLGA are smaller than those of GA. This is because that the online ML-based DMM in OMLGA helps to save a part of computing resources when making offloading decisions. GA's clock ticks ($45.74 \times 10^3$ ticks) are more than those of ILPs (ILP1: $10.85 \times 10^3$ ticks and ILP2: $26.99 \times 10^3$ ticks) in Application1, which illustrates that GA consumes more computing resources than ILPs in small-scale applications. However, in Application2, Application3, Application4, and Application5, clock ticks of GA are noticeably smaller than those of ILPs, which illustrates that GA consumes fewer computing resources than ILPs in large-scale applications. ILPs' clock ticks increase with the increasing number of application components and then remain stable. In Application1 and Application2, clock ticks of ILP1 are smaller than those of ILP2. The reason is that ILP2 needs to do more repeated calculations in small-scale applications. ILPs' clock ticks are close and stable in Application3, Application4, and Application5, which results from that ILPs make full use of computing resources to make offloading decisions in these large-scale applications. It is

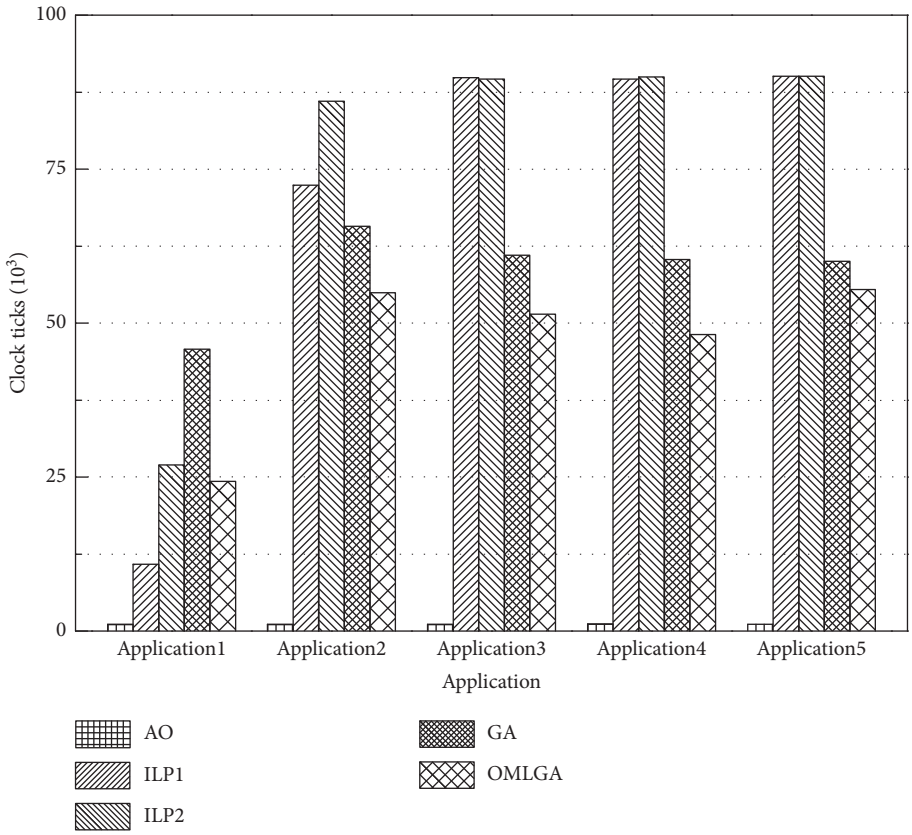


Figure 8: Clock ticks under different applications.

noticeable that the clock ticks AO takes are quite a few. The reason is that AO almost does not consume computing resources to make offloading decisions.

In this paper, we focus on the multisite computation offloading. Therefore, in the simulation below, we evaluate our algorithm under different numbers of cloud servers. Figure 9 illustrates the weighted total cost under different numbers of cloud servers. It is noticeable that OMLGA's weighted total costs are the smallest, which means OMLGA achieves the highest reduction in consumption among tested algorithms. Weighted total costs of OMLGA decrease (from 81.27% to 59.27%) with the increasing number of cloud servers. The reason is described as follows: In the multisite computation offloading, components can be offloaded to many cloud servers. On the one hand, when conditions of some wireless channels become bad, OMLGA continues to offload components to other cloud servers, which are connected by high-quality wireless networks. On the other hand, cloud servers are connected by wired networks, whose consumption is quite little, reducing much consumption caused by data transmission. Compared with the single-site computation offloading, in which there is only one single cloud server, the multisite computation offloading provides users with more choices. For example, as one extreme, if one of two cloud servers in the multisite computation offloading is always connected by the high-quality wireless network, and another one is always connected by the low-quality wireless network, components will be offloaded to the cloud server that is connected by the high-quality wireless network, and the multisite computation offloading can degenerate into the single-site computation offloading. Also, as the other extreme, if the only one cloud server in the single-site computation offloading is always connected by the low-quality wireless network, components will be executed locally, and the computation offloading does not work. Weighted total costs of ILPs are larger than those of OMLGA and GA and become quite large when the number of cloud servers is 5. This is because that the solution space becomes larger when the number of cloud servers is so large that ILPs cannot find the optimal solution in time. AO's weighted total costs are large, which illustrates that the commonly used thin client mode is not suitable for MCC. This simulation shows that advantages of the multisite computation offloading cannot be exploited or even lead to worse results if the offloading decision algorithm is not effective.

Figure 10 illustrates the clock ticks under different numbers of cloud servers. It is noticeable that clock ticks of OMLGA are smaller than those of ILPs and GA. Compared with ILPs and GA, OMLGA consumes fewer computing resources with the help of the online ML-based DMM, which consumes few computing resources to make offloading decisions. It can be seen that clock ticks of AO are still quite few. Clock ticks of ILPs increase with the increasing number of cloud servers, which results from that the solution space becomes large with the increasing number of cloud servers such that ILPs consume much computing resources to make offloading decisions. Similar to that shown in Figure 8, since ILP2 needs more repeated calculations in small-scale

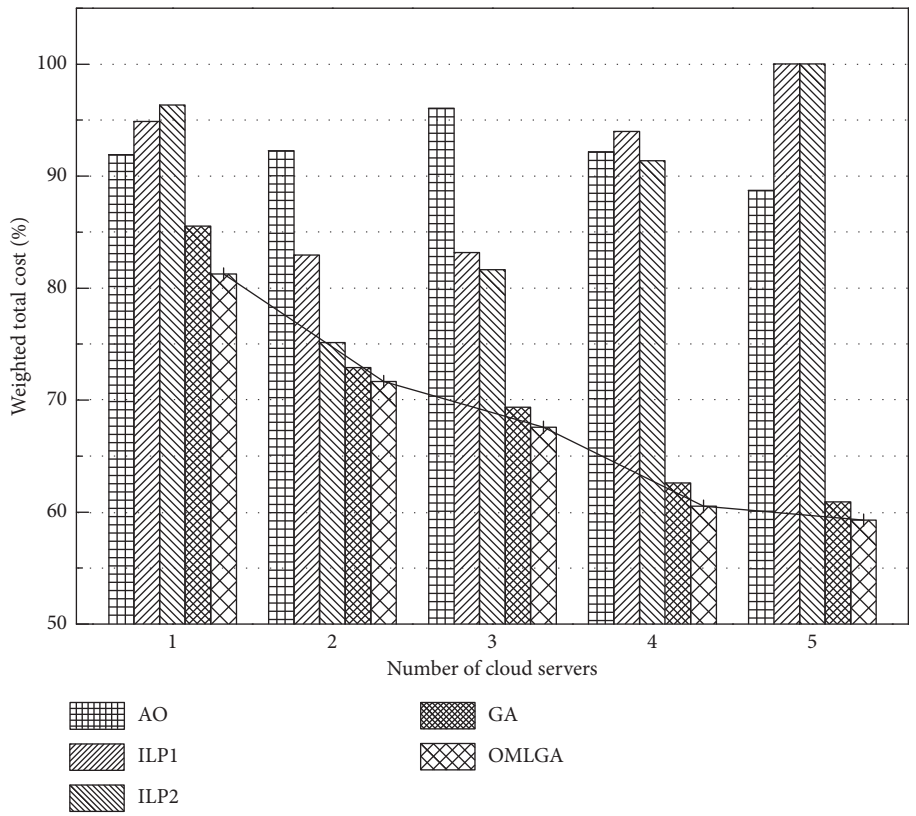


Figure 9: Weighted total cost under different numbers of cloud servers.

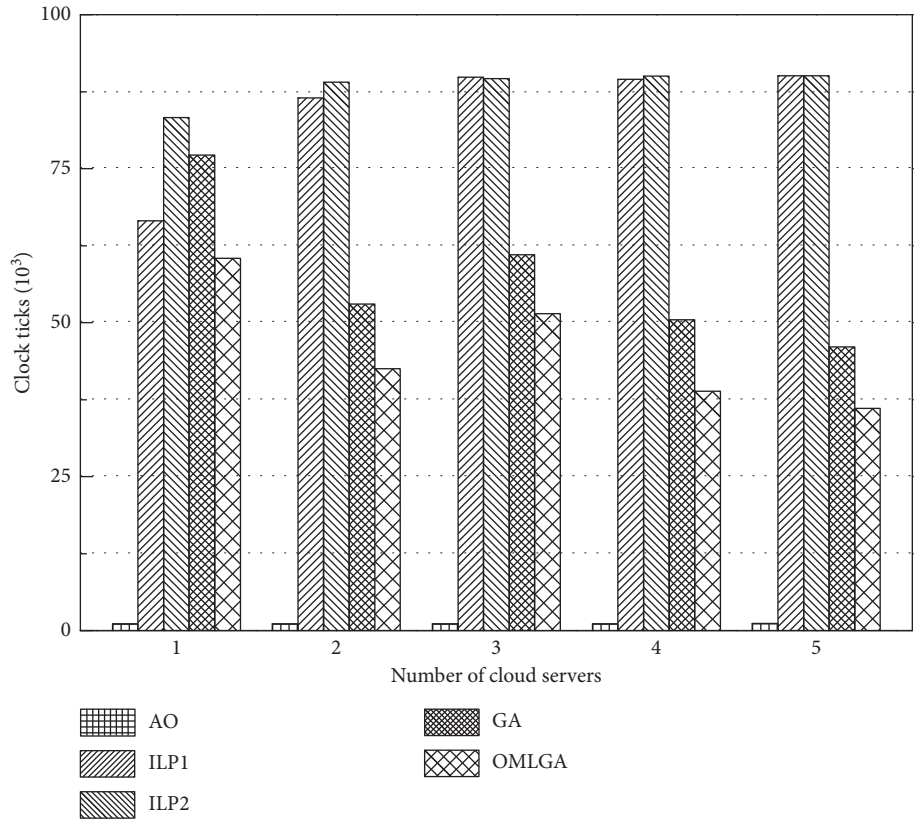


Figure 10: Clock ticks under different numbers of cloud servers.

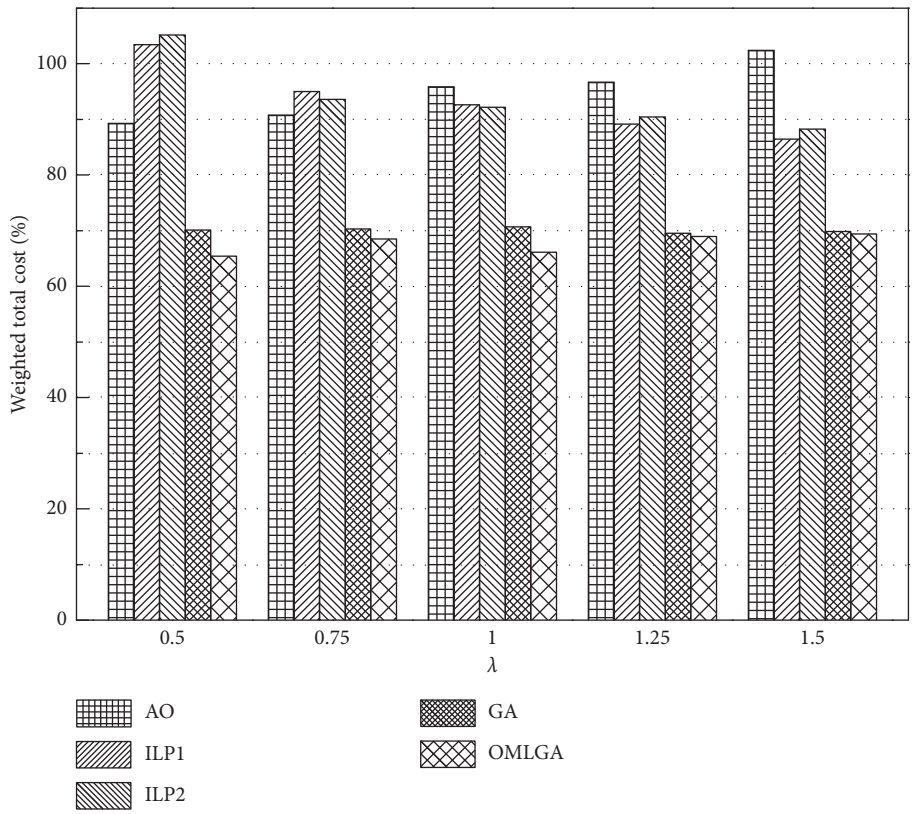


Figure 11: Weighted total cost under different user speeds.

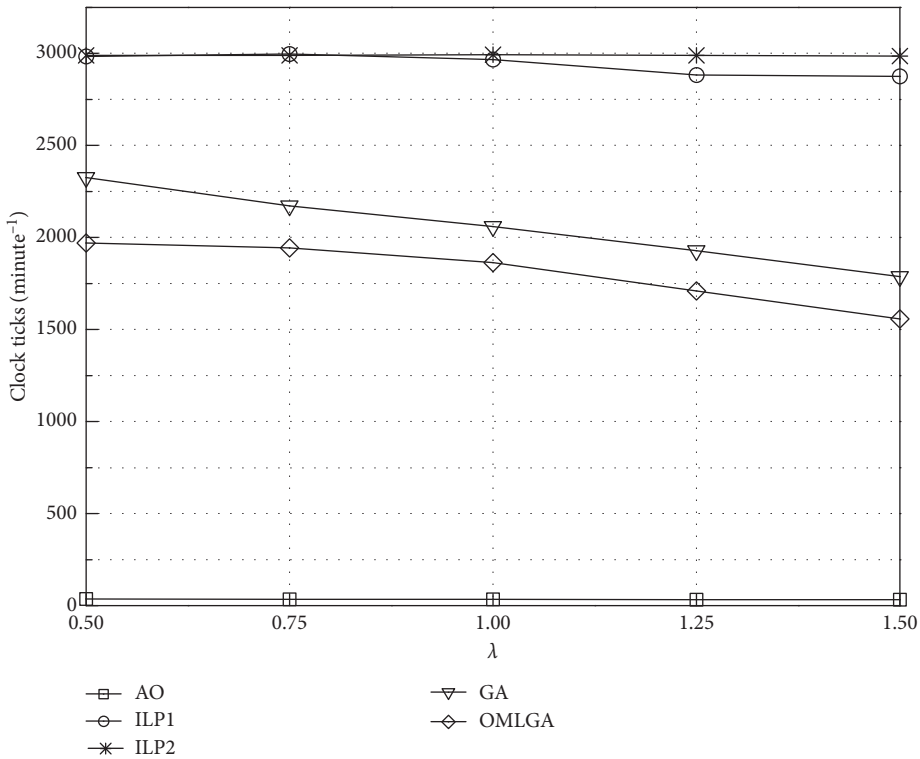


Figure 12: Clock ticks under different user speeds.

Table 3: Weighted total cost (%) in static environments.

| No. | Algorithm | |
|---|---|---|
| | Optimal (ILP) | OMLGA |
| 1 | 59.53 | 60.01 |
| 2 | 60.76 | 60.96 |
| 3 | 68.54 | 68.91 |
| 4 | 54.59 | 55.39 |
| 5 | 87.77 | 88.52 |
| 6 | 64.37 | 64.89 |
| 7 | 73.83 | 75.02 |
| 8 | 69.85 | 70.57 |
| 9 | 48.69 | 48.97 |
| 10 | 71.49 | 72.16 |

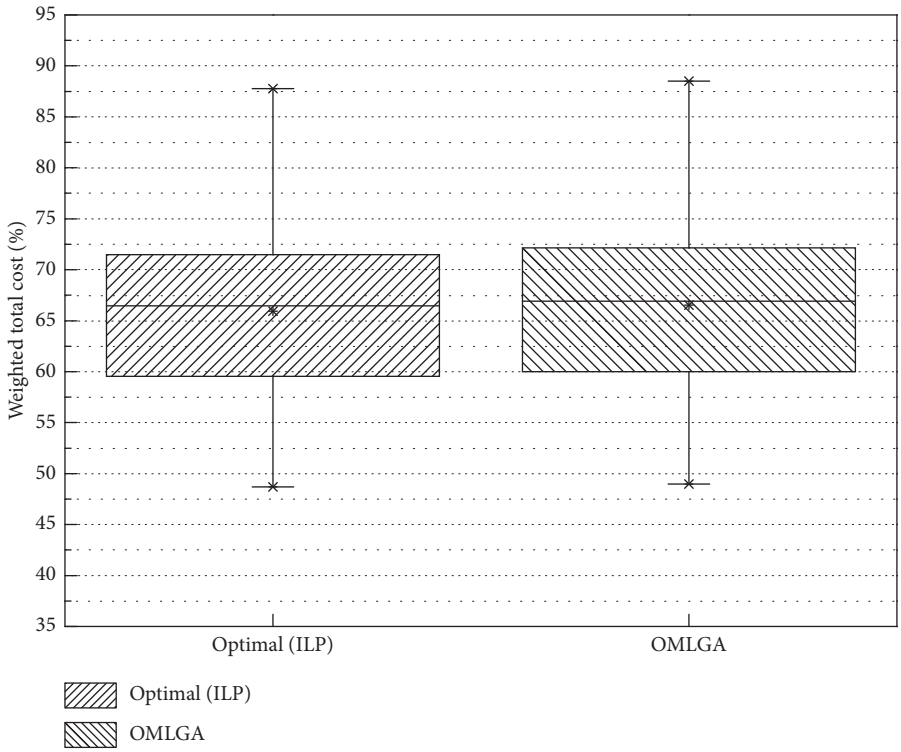


Figure 13: Weighted total cost in static environments.

offloading decision problems, clock ticks of ILP2 are larger than those of ILP1 when the numbers of cloud servers are 1 and 2. When the number of cloud servers is large, ILPs make full use of computing resources to make offloading decisions, which results in that ILPs' clock ticks are large and stable when the numbers of cloud servers are 3, 4, and 5.

Figure 11 illustrates the weighted total cost under different speeds of the user. In this simulation, the identical trajectory (moving path) with different user speeds is used. $\lambda$ represents the coefficient of the time ($\widehat{\Delta h}$) that the user takes to cross a square, that is, $\widehat{\Delta h} = \lambda \Delta h$($\lambda$ defaults to 1). A larger $\lambda$ means that the user takes more time to cross a square, the user's speed is slower, and the environment changes more slowly. It can be seen that OMLGA performs better than other algorithms under different speeds. When the user's speed becomes slow, GA can find excellent solutions timely to adapt to environmental changes, which is reflected in Figure 11 that GA's weighted total costs (69.52% and 69.89%) are close to but larger than those of OMLGA (68.98% and 69.43%) when $\lambda = 1.25$ and $\lambda = 1.5$. A slow speed leads to fewer environmental changes. It can be seen that weighted total costs of ILPs decrease (ILP1: from 103.46% to 86.45% and ILP2: from 105.17% to 88.28%) gradually with the decreasing $\lambda$. The reason is that ILPs have more time to make offloading decisions with fewer environmental changes, making the ILPs' offloading strategy adapt to the environment to a certain extent. Additionally, it can be seen that some weighted total costs of ILPs and AO exceed 100%, which means that the consumption caused by these algorithms' offloading strategies is greater than the consumption caused by the local execution strategy.

Figure 12 illustrates the clock ticks under different speeds of the user. As mentioned above, in this simulation, the user travels at different speeds with an identical trajectory, and

hence, the total time the user takes to travel is different. To evaluate the performance of these algorithms, clock ticks per minute is compared. ILP1 and GA rerun when detecting environmental changes, and a large $\lambda$ leads to fewer environmental changes. It can be seen that clock ticks of ILP1 (from 2983 ticks/minute to 2876 ticks/minute) and GA (from 2326 ticks/minute to 1788 ticks/minute) decrease with the increasing $\lambda$. Compared with ILPs, the GA-based DMM in OMLGA also helps to reduce the clock ticks. In addition, OMLGA makes offloading decisions based on the cooperation of online ML and GA. When the environment is stable, OMLGA makes more offloading decisions with the help of the online ML-based DMM, which consumes few computing resources. These two DMMs make clock ticks of OMLGA smallest and decrease (from 1971 ticks/minute to 1558 ticks/minute) with the increasing $\lambda$. Clock ticks of AO are still quite few, and clock ticks of AO and ILP2 are stable. AO makes offloading decisions when it receives the strategy request and remains idle at other time. ILP2 makes offloading decisions periodically. Therefore, the user's speed has no impact on these two algorithms.

ILP is a classic algorithm used to develop the accurate optimal solution in static environments, and these weighted total costs of ILP in static environments are used as the criteria to measure the accuracy of OMLGA. Ten simulations, in which environmental parameters are kept static, are conducted to evaluate the accuracy of OMLGA. To keep the environmental parameters static and simulate the static environments, the parameters in every square are set to be the same. In other words, when the user moves from one square to another, the parameters are constant. In this way, the static environments are simulated. The weighted total costs are shown in Table 3, and their box-plot is shown in Figure 13. As shown in Figure 13, weighted total costs of OMLGA are close to the optimal ones. Through the statistics of the data in Table 3, the maximum and minimum relative errors of OMLGA are 1.612% (No. 7) and 0.329% (No. 2), respectively. The average relative error of OMLGA is 0.896%.

## 6. Conclusion

The resource constraint prevents the further development of MDs and mobile applications. MCC, as an effective way to improve the MD performance, provides a good opportunity to promote it. This paper studies the runtime offloading decision problem for MCC and proposes a runtime offloading decision algorithm based on the cooperation of online ML and GA. On the one hand, with the help of GA, this algorithm obtains data to train the predictor of online ML. On the other hand, with the help of online ML, this algorithm saves more computing resources and enhances GA's ability to develop the offloading strategy. The results of this paper show that the commonly used thin client mode, which offloads all offloadable components to the cloud, is not suitable for MCC. The ILP-based offloading decision algorithms work well in small-scale problems but lead to bad results and consume more computing resources in large-scale problems. The cooperative runtime offloading decision algorithm, which minimizes the weighted total cost of offloaded applications while reducing the computing resource consumption and performs better in large-scale problems, can solve the runtime offloading decision problem efficiently. The established model and proposed algorithm can be extended to more general runtime decision problems in dynamic environments. This study provides a feasible approach to design a cooperative runtime decision algorithm, which combines online ML and GA to optimize the objective function while reducing the computing resource consumption. In this paper, we assume that there is only one application executed in MD. In the future, we will study the offloading decision problem with multiple applications based on the cooperative runtime offloading decision algorithm. In that problem, we need to consider the interaction among applications and add ML's attributes to combat the changing parameters of the MD.

## Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work was supported by the Special Scientific Research Program of Education Department of Shaanxi Province (Program No. 19JK0806), the Young Teachers Research Foundation of Xi'an University of Posts and Telecommunications, and the Special Funds for Construction of Key Disciplines in Universities in Shaanxi.

## References

[1] CISCO, *Cisco Visual Networking Index: Forecast and Trends*, 2017-2022 White Paper, Cisco, San Jose, USA, 2018, https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.

[2] Internet Trend Report 2018, http://www.kpcb.com/internet-trends.

[3] Energy Technology Perspectives 2017, https://www.iea.org/etp/tracking2017/energystorage/.

[4] Andy and Bill's Law, https://en.wikipedia.org/wiki/Andy_and_Bill%27s_law.

[5] P. Mell and T. Grance, "The NIST definition of cloud computing," *Communications of the ACM*, vol. 53, no. 6, pp. 50-51, 2011.

[6] K. Akherfi, M. Gerndt, and H. Harroud, "Mobile cloud computing for computation offloading: issues and challenges," *Applied Computing and Informatics*, vol. 14, no. 1, pp. 1–16, 2018.

[7] B. Zhou and R. Buyya, "Augmentation techniques for mobile cloud computing: a taxonomy, survey, and future directions," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–38, 2018.

[8] T. H. Noor, S. Zeadally, A. Alfazi, and Q. Z. Sheng, "Mobile cloud computing: challenges and future research directions," *Journal of Network and Computer Applications*, vol. 115, no. 1, pp. 70–85, 2018.

[9] E. Cuervo, A. Balasubramanian, D. Cho et al., "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pp. 49–62, San Francisco, CA, USA, June 2010.

[10] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of the 2012 IEEE INFOCOM*, pp. 945–953, Orlando, FL, USA, March 2012.

[11] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Proceedings of the 2010 International Conference on Mobile Computing, Applications, and Services*, pp. 59–79, Santa Clara, CA, USA, October 2010.

[12] K. Goel and M. Goel, "Cloud computing based e-commerce model," in *Proceedings of the 2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pp. 27–30, Bangalore, India, May 2016.

[13] Y. Liu, Y. Zhang, J. Ling, and Z. Liu, "Secure and fine-grained access control on e-healthcare records in mobile cloud computing," *Future Generation Computer Systems*, vol. 78, no. 3, pp. 1020–1026, 2018.

[14] L. Ramírez-Donoso, M. Pérez-Sanagustin, and A. Neyem, "MyMOOCSpace: mobile cloud-based system tool to improve collaboration and preparation of group assessments in traditional engineering courses in higher education," *Computer Applications in Engineering Education*, vol. 26, no. 5, pp. 1507–1518, 2018.

[15] C. Vuchener and A. Esnard, "Graph repartitioning with both dynamic load and dynamic processor allocation," in *Proceedings of the 2013 International Conference on Parallel Computing*, pp. 243–252, München, Germany, September 2013.

[16] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: can offloading computation save energy?," *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[17] G. Lewis and P. Lago, "Architectural tactics for cyber-foraging: results of a systematic literature review," *Journal of Systems and Software*, vol. 107, pp. 158–186, 2015.

[18] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: a partition scheme," in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 238–246, Atlanta, GA, USA, November 2001.

[19] Z. Li, C. Wang, and R. Xu, "Task allocation for distributed multimedia processing on wirelessly networked handheld devices," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pp. 6–11, Fort Lauderdale, FL, USA, April 2001.

[20] M. Goudarzi, M. Zamani, and A. T. Haghighat, "A genetic-based decision algorithm for multisite computation offloading in mobile cloud computing," *International Journal of Communication Systems*, vol. 30, no. 10, pp. 1–13, 2017.

[21] N. I. M. Enzai and M. Tang, "A heuristic algorithm for multi-site computation offloading in mobile cloud computing," *Procedia Computer Science*, vol. 80, pp. 1232–1241, 2016.

[22] R. Niu, W. Song, and Y. Liu, "An energy-efficient multisite offloading algorithm for mobile devices," *International Journal of Distributed Sensor Networks*, vol. 9, no. 3, Article ID 518518, 2013.

[23] R. Kumari, S. Kaushal, and N. Chilamkurti, "Energy conscious multi-site computation offloading for mobile cloud computing," *Soft Computing*, vol. 22, no. 20, pp. 6751–6764, 2018.

[24] K. Sinha and M. Kulkarni, "Techniques for fine-grained, multi-site computation offloading," in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 184–194, Newport Beach, CA, USA, May 2011.

[25] M. E. Khoda, M. A. Razzaque, A. Almogren, M. M. Hassan, A. Alamri, and A. Alelaiwi, "Efficient computation offloading decision in mobile cloud computing over 5G network," *Mobile Networks and Applications*, vol. 21, no. 5, pp. 777–792, 2016.

[26] D. Kovachev, T. Yu, and R. Klamma, "Adaptive computation offloading from mobile devices into the cloud," in *Proceedings of the 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 784–791, Leganés, Spain, July 2012.

[27] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri, "Runtime application repartitioning in dynamic mobile cloud environments," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 336–348, 2014.

[28] X. Jin, Y. Liu, W. Fan, F. Wu, and B. Tang, "Multisite computation offloading in dynamic mobile cloud environments," *Science China Information Sciences*, vol. 60, no. 8, article 089301, 2017.

[29] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer, "Machine learning-based runtime scheduler for mobile offloading framework," in *Proceedings of the 6th International Conference on Utility and Cloud Computing*, pp. 9–12, Dresden, Germany, December 2013.

[30] H. Eom, R. Figueiredo, H. Cai, Y. Zhang, and G. Huang, "MALMOS: machine learning-based mobile offloading scheduler with online training," in *Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pp. 51–60, San Francisco, CA, USA, April 2015.

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

[32] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android Java code for on-demand computation offloading," in *Proceedings of the 2012 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 233–248, Tucson, AZ, USA, October 2012.

[33] D. Huang, P. Wang, and D. Niyato, "A dynamic offloading algorithm for mobile computing," *IEEE Transactions on Wireless Communications*, vol. 11, no. 6, pp. 1991–1995, 2012.

[34] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pp. 225–238, Low Wood Bay, UK, June 2012.

[35] S. Yang, "Memory-based immigrants for genetic algorithms in dynamic environments," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, pp. 1115–1122, Washington, DC, USA, June 2005.

[36] M. Srinivas and L. M. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994.

[37] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.