Received: 29 October 2021 | Accepted: 23 June 2022

DOI: 10.1002/int.22954

RESEARCH ARTICLE

WILEY

PARA: Performability-aware resource allocation on the edges for cloud-native services

Yeting Guo[1] | **Fang Liu**[2] | **Nong Xiao**[1] | **Zhaogeng Li**[3] | **Zhiping Cai**[1] | **Guoming Tang**[4] | **Ning Liu**[3]

[1]College of Computer, National University of Defense Technology, Hunan, China

[2]School of Design, Hunan University, Hunan, China

[3]System Department, Baidu Inc., Beijing, China

[4]Science & Technology on Information Systems Engineering Lab, National University of Defense Technology, Hunan, China

Correspondence
Fang Liu, School of Design, Hunan University, 410082 Lushan South Road, Changsha, Hunan, China.
Email: fangl@hnu.edu.cn

Funding information
Natural Science Foundation of Guangdong Province, Grant/Award Numbers: NSFCU1811461, 2018B030312002; National Natural Science Foundation of China, Grant/Award Numbers: 62172155, 62072465, 61832020; Key Technologies Research and Development Program, Grant/Award Number: 2018YFB0204301

Abstract

This paper explores resource allocation strategy in the Baidu Over The Edge system to enable mobile edge computing (MEC) datacenters to effectively support cloud-native services downstream to the network edge. There are many challenges to this issue. First, MEC datacenters are resource-constrained to fully meet resource demands. Second, previous works regard the resource requirements of each service as an indivisible unit, resulting in idle MEC resources, even if the resources can meet the demands of some microservices decoupled by the service. Third, they are confined to optimize the allocation for a single slot, failing to adapt to the dynamic demands. To improve resource utilization, we propose performability-aware resource allocation (PARA), a PARA on the edges for cloud-native services. It takes microservices as the unit of resource allocation and allows services to perform with degraded services when only part of microservices' demands are met. It also considers dependency among microservices, dynamic resource requirements, and resource supply characteristics of MEC and cloud. Performability is a unified

performance-reliability measure for evaluating such degradable systems. To maximize the long-term overall performability, we model the resource optimization problem and then develop an online greedy heuristic algorithm. The algorithm predicts services' resource demands and then adapts the online allocation. The experimental results show that PARA reduces the reallocation overhead by 47.7%–53.6%, and improves the long-term overall performability by 23.14%–43.25% of existing state-of-the-art works.

KEYWORDS
degradable system, microservice, mobile edge computing, performability, resource allocation

1 | INTRODUCTION

Mobile edge computing (MEC)[1] is becoming an important part of ubiquitous computing infrastructure in the Internet-of-Things era. Cloud vendors have built and developed their own MEC datacenters and platforms, for example, Baidu Over The Edge (OTE) system. With these systems, various cloud-native intelligent services, for example, Baidu multimode search and video surveillance, are offloaded from the cloud to MEC datacenters. Since MEC datacenters are often colocated with mobile access networks and are closer to users than the cloud, these offloaded services improve performance (as viewed by the throughput, latency, and energy consumption) and reliability (ratio of successful service request completions before a deadline).[2]

Particularly, we focus on resource allocation issues on the edges for cloud-native services. Most studies have discussed how to offload tasks from terminal devices to MEC datacenters and allocate MEC resources to these tasks. But few discuss the resource allocation for offloaded cloud naive services. The issue is very important for cloud vendors and service providers. Since MEC datacenters are usually composed of a few servers, the resource-limited MEC datacenter needs to collaborate with the remote but resource-rich cloud datacenter to meet all services' resource demands.[3] The servers in MEC datacenters are called edge servers in our paper. At first glance, the allocation appears to be a MultiDimensional Knapsack Problem (MDKP). Here, the different dimensions correspond to different types of resources. However, two factors make the problem more complex.

1.1 | Service degradability

Services can perform with downgraded services when they are not provided sufficient resources.[4] Performablity[5,6] is defined as a unified performance-reliability measure for these degradable services to evaluate the MEC system's effectiveness.

Previous studies[7–9] allocate resources based on service granularity. They neglect service degradability, while we introduce it into MEC resource allocation issue to improve MEC resource utilization for cloud-native services. Baidu OTE platform adopts microservice architecture, in which each service is decoupled into a group of functional independent microservices. The global microservices market is reported to have maintained a compound annual growth rate of 22.4%.[10] With the acceleration and deepening of digitization and the increase of service complexity, more and more services are suitable for microservice architecture. For each cloud service, the MEC datacenter can provide some of the microservices but not necessarily all of them. Different selections of offloaded microservices exhibit different levels of performability.[11] Note that the selection is not arbitrary. Specifically, since the MEC datacenter usually plays the role of preprocessing service requests by geographical advantages, the microservices offloaded to the MEC datacenter should always be executed before the microservices in the cloud. Otherwise, it is meaningless for the service to occupy the MEC resources. Resource allocation based on microservice granularity leverages service degradability to improve MEC resource utilization; however, it also escalates the problem to a MultiChoice MultiDimensional Knapsack Problem (MCMDKP). Here, the different choices correspond to different selections of offloaded microservices.

1.2 | Dynamic resource demands

The resource demands of cloud services fluctuate with the number of service requests. To adopt dynamic demands, the allocation decision needs to be updated. In reallocation, some microservices may be migrated from one edge server to another, that is, their original instances will be deleted, and new instances will be created in target edge servers. The boot time of a new instance can be of the order of hundreds of milliseconds, an order of magnitude higher than the duration of many latency-critical service requests.[12] This degrades the service's performability in the next period. In this case, a MEC datacenter must take deployment resilience and reallocation overhead into consideration to guarantee satisfactory long-term overall performability.

In this paper, we study the resource allocation issue in Baidu OTE platform and propose performability-aware resource allocation (PARA), a PARA scheme for cloud naive services. It jointly considers the cloud service degradability, dynamic resource requirements, and resource supply characteristics of MEC and the cloud, to optimize the resource allocation and maximize the long-term overall performability. Our main contributions are summarized as follows:

1. We highlight the significance of resource allocation with the awareness of service degradability and the characteristics of MEC datacenters and the cloud. To maximize the utilization of MEC resources, we note that the MEC system should consider the execution sequence of microservices when allocating resources to services.
2. We further extend the resource allocation issue from a single snapshot problem to a practical dynamic model that considers reallocation overhead as a correction term to avoid degrading the long-term performability. We develop an online algorithm to solve the problem.
3. We evaluate PARA with the Google Cluster data set. Numerical results demonstrate that PARA outperforms the state-of-the-art works in terms of reallocation overhead and long-term overall performability in most cases.

The remainder of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 summarizes related works. Then, we present the resource allocation problem formulation in our PARA scheme and the design of our solution algorithm in Sections 4 and 5. The evaluation results are presented in Section 6. Finally, Section 7 concludes our work.

2 | BACKGROUND AND MOTIVATION

In this section, we introduce the background of MEC resource allocation and our motivations. The motivations mainly consist of service degradability (reflected in Sections 2.1 and 2.2) and dynamic resource demands (reflected in Sections 2.2 and 2.3).

2.1 | Multiple offloading alternatives

High-level functions of cloud-native services are always composed of several functional components, referred to as microservice in this paper. When designing Baidu OTE platform, we observed that microservice architecture is very suitable for rapid iteration of edge computing scenarios owing to its compactness, rapid development, and low coupling.[13,14] Each microservice achieves certain specific subfunctions and is encapsulated into independent virtual machines or containers because they were very likely to be developed by different research and development teams. These microservices are also correlated to achieve the overall function. Sequential relationships are very common among microservices.[15] Examples of such cloud services are given in Figure 1. More examples could be derived from these examples. We briefly describe one of them. In the Baidu multimode search service, terminals request cloud to translate the character in their images. They are asking for two microservices, optical character recognition, and character translation. In the first microservice, the image is preprocessed, and the characters in the image are segmented and recognized. In the second microservice, the recognized characters would be translated into the language that the mobile user wants.

In most cases, the MEC completes the microservices at the top of the execution sequence as much as possible to reduce the amount of data transmitted to the cloud. Indeed there exist other complex relationships among microservices, but we focus on the sequential relationship which adds the most basic constraints to task-offloading mechanisms in MEC[11,16,17] and hopes

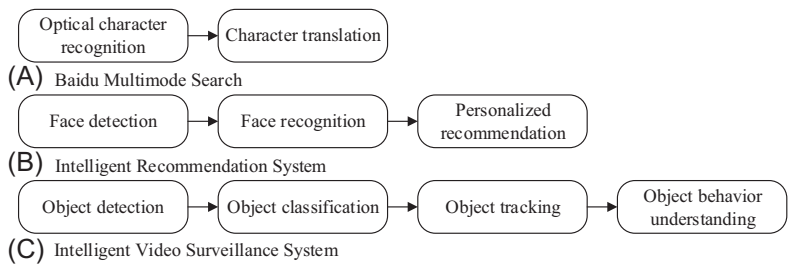

FIGURE 1 Examples of microservices in cloud-native services

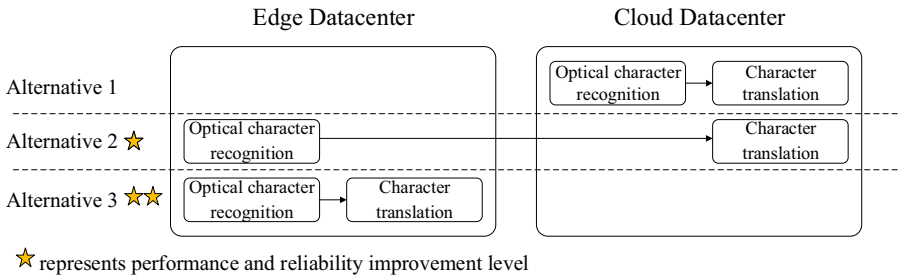

FIGURE 2 Examples of offloading alternatives [Color figure can be viewed at wileyonlinelibrary.com]

that our research can draw more attention to the effect of the microservice relationship on edge resource allocation.

As shown in Figure 2, the service is composed of two sequential microservices, and there exist three offloading alternatives. Different offloading alternatives correspond to different improvements in service performance and reliability. Online service providers always want to offload the whole services at the edge of the network to maximum decrease the packet-loss rates, delays, and delay jitter. In this example, Alternative 3 brings the maximum performability improvement for the service. Previous work considered cloud service offloading and resource allocation mechanisms in a coarse-grained manner. They simply assumed that the edge datacenter either satisfies the full demands of one service, that is, Alternative 3, or allocates no resources, that is, Alternative 1. These works neglect the possibility that idle MEC resources may be able to satisfy the resource demands of Alternative 2. The service could be degraded from Alternative 3 to Alternative 2 rather than directly to Alternative 1. If the service were to adopt Alternative 2 and obtain the corresponding idle resources from the edge datacenter, it would be a win–win for both the offloaded cloud service and edge datacenter.

Such service degradability inspires us to allocate MEC resources in microservice granularity. For cloud services, they have more alternatives to increase competitiveness and decrease the risk that they obtain no MEC resources. For the MEC datacenter, it has more alternatives to allocate resources, and the segmented idle resources can be utilized.

2.2 | Dynamic resource demands

As mentioned above, cloud service providers encapsulate their microservices into virtual machines or containers. For the convenience of description, we assume that each microservice is encapsulated in a Docker container. The encapsulation method and types of containers do not affect the resource allocation in our paper. The images of these microservices are cached in the MEC datacenter. When necessary, container instances are created to respond to service requests based on these cached images.[18]

The resource demand is not fixed but dynamically and elastically adjusted with the number of requests.[19] For one cloud service, with the increasing number of user requests, it requires more resources to increase the scale of container instances because each container instance has its maximum number of service requests that can be handled at a time, and the number of user requests may exceed the maximum number. Different numbers of instances bring different throughput and reliability for the service.[20]

New conflict generates between the increasing resource demands and the limited resource supply. Previous works simply attribute the increasing demands to the increase in the number of services. However, few studies discuss the change in elastic resource demand of a single service. Suppose that one service provider has deployed one container instance in the MEC datacenter and requires creating a new instance to serve the increasing requests. But the MEC datacenter has no extra resources to support the new demands. In the reallocation, previous works take back all the MEC resources from this service even though the MEC can support to deploy one service instance. It seems unreasonable because the service provider can assign the instance to handle some high-priority requests and the cloud to handle low-priority requests. This motivation is similar to the above one, which can be attributed to the awareness of service degradability when resource shortage. In the former, the logical order of microservice execution is incremental, and in this motivation, the number of container instances for microservice execution is incremental, as shown in Figure 3. Solutions to these two problems can be cross-referenced. Thus, to make our model more concise, we focus on the former motivation to discuss resource allocation.

2.3 | Dynamic container placement

When the resource demands change, the resources should be reallocated to maximize resource utilization. Figure 4 gives an example of different resource allocations at two time points. Circles of different colors represent different types of container instances. Due to the change of service requests, the online service provider would adjust the type or scale of container instances. Some increase and some decrease. Then, the edge datacenter makes a reallocation for MEC resources. Some container placements may be updated, which we call migration. For stateless services, it could simply delete one container at one edge server and create a new one at another edge server to continue handling requests. However, it still makes the corresponding service unavailable in a short period. For stateful services, they will have to consume more time to synchronize state information.[21] Frequent migration would harm the service performability.[22] We call such overhead reallocation overhead. It motivates us to decrease unnecessary migration when reallocating resources.

Compared with the existing work, we take into account service degradability in the microservcie architecture to allocate the MEC resources in more finely and reduce resource

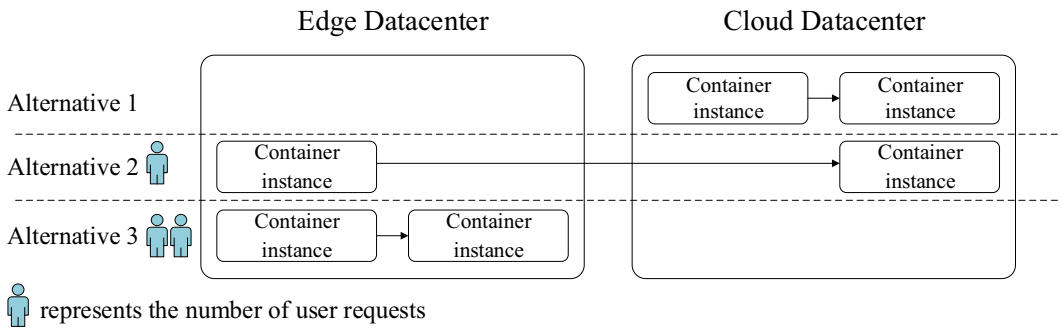

FIGURE 3 Dynamic resource demands [Color figure can be viewed at wileyonlinelibrary.com]

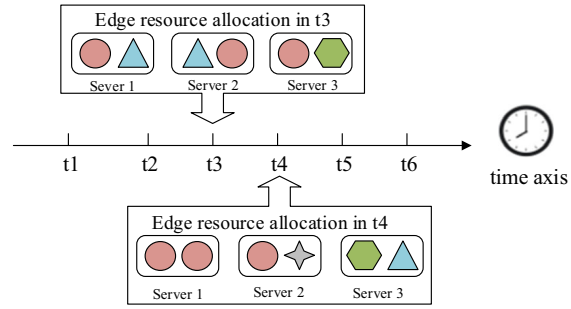

FIGURE 4 Dynamic resource allocation [Color figure can be viewed at wileyonlinelibrary.com]

waste. Besides, we consider decreasing the reallocation overheads to optimize performability from a long-term perspective rather than just a single time slot.

3 | RELATED WORK

In this section, we briefly review the related work of edge resource allocation from three aspects, that is, service offloading, service placement, and allocation evaluation.

3.1 | Service offloading

There are two categories of service offloading. One is offloading services from terminals to the MEC datacenter and the cloud, and the other is offloading cloud services to the MEC datacenter.

In the first category, the terminal requests to offload its service due to limited resources, and it splits and deploys the service among terminals, the MEC, and the cloud. Yi et al.[11] proposed to reduce one application's latency by offloading subtasks between terminals and edge servers based on required computational resources and the network bandwidth. Rahman et al.[23] considered network connectivity and on-demand mobility to achieve superior system performance and minimum consumption of resources. Chen et al.[24] further discussed multiuser computation service offloading decision problems. They formulated this problem with the game theory. And then they designed a distributed computation offloading algorithm to achieve Nash equilibrium among multiple users. These studies are designed for terminals to decrease local computation burden and improve local task execution efficiency.

In the second category, the online service provider offloads its service to the MEC to improve its performance and reliability. Multiple service providers compete for limited MEC resources. From the perspective of the MEC datacenters, Cardoso et al.[25] proposed a virtual resource management scheme for infrastructure and spectrum sharing. They found that the visualization could greatly minimize the cost and idle resource capacity of the MEC providers. Zhang et al.[8] proposed a decentralized resource allocation scheme for multiple resource

providers, which combines multiple decentralized edge data center resources to maximize the profits of all resource providers and provide users with high-quality services. For online cloud service providers, Xu et al.[26] designed Zenith, in which the infrastructure management in the MEC datacenter is independent with the service provisioning and service management performed by the cloud service providers. On the basis of the architecture, they further proposed an auction-based mechanism to establish resource contact and a latency-aware scheduling technique to optimize resource allocation and reduce overall service latency. Castellano et al.[27] propose DRAGON, a distributed resource allocation and management strategy. It seeks the best partition of shared resources for different services running in the edge data center, so as to maximize the service quality. These works regard service as granularity and neglect the service degradability in service offloading and resource allocation. Samanta et al.[28] proposed an online microservice-based auction mechanism to encourage microservice to release their occupied edge resources, so that edge data centers can reclaim these resources and reallocate them to other microservices that need resources, so as to realize flexible and dynamic allocation of edge resources. However, it does not fully realize the relationship between microservices.

3.2 | Service placement

The above works discussed how to offload services among different network locations but neglected the dynamic placement of services within one location, especially in the MEC datacenter. Each microservice is encapsulated in an independent container or virtual machine and placed at one physical server to handle the request. The placement in the cloud datacenter has been fully discussed in Anuradha and Sumathi[29] and Xiao et al.[30] Xiao et al.[30] proposed to perceive the nonuniformity of multidimensional resource utilization of servers, and designed a heuristic algorithm to combine different workload types, so as to reduce the number of online servers and support green computing. However, these proposals are not applicable to MEC datacenters. The reason is that a cloud datacenter is assumed to be resource-rich to satisfy the resource demands of all microservices independently, and the optimization goal is to achieve load balancing,[31] reduce migration overhead,[32] and so forth. But a MEC datacenter has fewer servers than a cloud datacenter and cooperates with the cloud to meet resource demands, causing the optimization goal to be different and complex. He et al.[3] discussed service placement in the MEC datacenter to maximize the number of served requests. In particular, they focused on discussing the impact of sharable resources (e.g., storage) and nonsharable resources (e.g., computation) on service placement. Wang et al.[33] considered service placement for social virtual reality applications based on users' mobility to support the economic operation of the MEC datacenter and achieve satisfactory quality-of-service for the users. In addition, due to dynamic resource demands, the resource allocation was updated to maximize the utility of the resources. To decrease the reallocation overhead, some workload prediction methods have been proposed. Kecskemeti et al.[34] observed the long-running scientific workflows and their behavior discrepancies, and then summarized the generic background workload for workload prediction. In the formalization of resource allocation problems, Duong-Ba et al.[35] introduced reallocation overhead as one of the objective functions to solve. Beloglazov and Buyya[36] set the number of service migrations as a constraint in the resource optimization problem. Yang et al.[20] considered the service migration overhead as a

correction term in allocating resources. However, they also simply regard the service as an atom-demanding resource without considering service degradability. The placement of microservices in the MEC datacenter is rarely discussed, which could provide some new insights to decrease the reallocation overhead.

3.3 | Allocation evaluation

As we described above, various metrics have been used to evaluate MEC resource allocation, such as service latency,[11] the profit of multiple participants.[8] In the current development stage of edge computing, for cloud service providers, they are more interested in the service both performance and reliability improvement that the MEC can bring. And since we leverage service degradability in the MEC in our paper, we use performability as our evaluation metric, a unified performance-reliability measure for degradable services. Existing researches have investigated the definition and measure of performability. Kim et al.[37] measured the performability of the Infrastructure as a Service cloud based on the failure of each level, that is, datacenter, hosts, virtual machine, and task. It facilitates the cloud service providers to predict how many cloud resources need to be prepared when a failure occurs. Silva et al.[38] studied performability in geographically distributed cloud datacenters with the consideration of the possibility of natural disasters. It adopts a hybrid heterogeneous modeling method, including reliability block diagrams, stochastic Petri nets, and cloud system high-level models, to evaluate this metric. Raei and Yazdani[22] discuss the performability analysis of the MEC architecture. In their analysis, they considered the impact of many factors such as the quality of wireless connection, user mobility, and resource capacity on the availability and performance of the edge computing architecture. These works provide important references for cloud service providers to evaluate the MEC.

Our proposal is designed for allocating MEC resources to cloud-native services. Different from existing works, we regard microservice as allocation granularity based on service degradability, and consider the microservices' dependency and placement in the allocation.

4 | SYSTEM MODEL AND PROBLEM FORMULATION

In this section, based on the above motivations, we first define our system model. Then we formulate the resource allocation problem in this model from static and dynamic aspects, respectively. At last, we discuss the uniqueness of the problem.

4.1 | System model

The model is shown in Figure 5 and contains three layers: the mobile devices, MEC microdatacenter, and cloud datacenter. The mobile devices (such as cameras) connect to the MEC microdatacenter and submit their requests. The MEC microdatacenter is *close to users* but *resource-limited*. It pulls some microservice images from the cloud datacenter to preprocess the received requests (such as pedestrian detection). But due to the resource limitations and cloud service degradability, the MEC microdatacenter can only create part of the microservice instances. The cloud datacenter is *remote* but *resource-rich*. It contains all the microservice

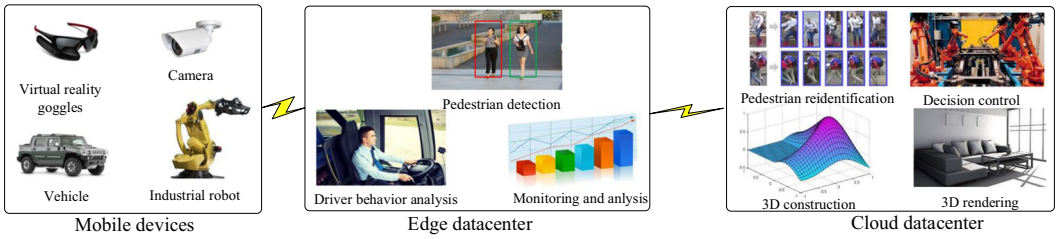

FIGURE 5 Cloud-native service placement in edge computing microservice architecture [Color figure can be viewed at wileyonlinelibrary.com]

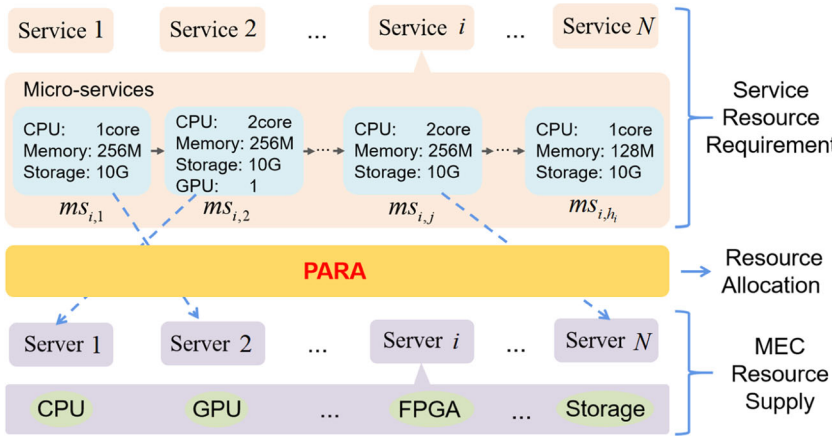

FIGURE 6 Illustration of the resource allocation in PARA. CPU, central processing unit; FPGA, field-programmable gate array; GPU, graphics processing unit; MEC, enable mobile edge computing; PARA, performability-aware resource allocation. [Color figure can be viewed at wileyonlinelibrary.com]

images. And in this architecture, if the request is not fully processed, the cloud datacenter will create the corresponding microservice instances and continue to process the request (such as pedestrian reidentification). In the figure, the behavior analysis service required by vehicles is offloaded on the edge, the industrial service required by robots and the surveillance service required by cameras are partially offloaded to the edge, and the virtual reality service is placed in the cloud.

The key challenge lies in which microservice instances are created in the MEC microdatacenter. We illustrate PARA in Figure 6, and Table 1 summarizes the main notation. In terms of MEC resource supply, we consider that the MEC microdatacenter is a cluster of C homogeneous servers. Each server supplies K types of resources. The second type of resource refers to memory resources. $\{s_l^1, s_l^2, ..., s_l^K\}$ $(l \in 1, 2, ..., C)$ denotes the resource supply at the lth server. Regarding service resource demand, there are N services attempting to obtain MEC resources. Considering the number of requests in the current time slot, deployment cost, or other factors, one service i $(i \in \{1, 2, ..., N\})$ attempts to offload h_i microservices to the MEC. Note that h_i changes dynamically over time. $\{d_{i,j}^1, d_{i,j}^2, ..., d_{i,j}^K\}$ $(j \in \{1, 2, ..., h_i\})$ denotes the resource demands of the microservice $ms_{i,j}$.

TABLE 1 Model notations

Notation	Definitions
C	Number of edge servers
K	Number of resource types
$\left\{s_l^1, s_l^2, ..., s_l^K\right\}$	Resource supply at the edge server l
N	Number of services
h_i	Number of microservices in service i
$\left\{d_{i,j}^1, d_{i,j}^2, ..., d_{i,j}^K\right\}$	Resource demands of the microservice $ms_{i,j}$
T	Number of time slots
$u_{i,j}$	Performability improvement to service i when deploying the microservice $ms_{i,j}$
pf	Overall performability
α	Parameter used to evaluate the reallocation overhead
$re\_overheads(t)$	Reallocation overhead in the tth time slot
$x_{i,j,l}$	Decision on whether the microservice $ms_{i,j}$ is deployed on the edge server l

Depending on the microservices in the MEC microdatacenter, the services will present different levels of performability. $u_{i,j}$ denotes the performability improvement to service i when deploying the microservices $ms_{i,j}$. Performability refers to the possibility that service performance reaches one measurable set of service accomplishment levels. The specific calculation of performability could be obtained in Meyer,[5] Oliveira et al.,[6] and Raei and Yazdani,[22] and these measurements are compatible with our work. Here we just simplify our problem formulation and regard it as a known value that has been measured. Given the service resource demands and MEC resource supplies, the MEC microdatacenter will allocate resources based on PARA.

4.2 | Optimization formulation

To model our optimization target and allocation constraints, we introduce the decision variable $x_{i,j,l}$ ($x_{i,j,l} \in \{0, 1\}$), which indicates whether server l deploys the microservices $ms_{i,j}$. If $x_{i,j,l}$ is equal to 1, the microservice is placed in server l in this time slot; otherwise, it is not. If $x_{i,j,l}^t$ is equal to 0 for any edge server, then the microservice remains in the cloud.

First, we introduce some main constraints in PARA.

1. *Each resource for each edge server is limited*: This is described by

$$\sum_{i=1}^{N} \sum_{j=1}^{h_i} x_{i,j,l} \times d_{i,j}^v \le s_l^v, \quad \forall l \in \{1, 2, ..., C\}, \quad \forall v \in \{1, 2, ..., K\}. \tag{1}$$

2. *Each microservice instance is deployed on one edge server at most*: Microservices are regarded as the minimum resource allocation granularity. If one microservice requires two CPU cores, two edge servers are not allowed to satisfy this requirement using a single core each. The constraint is described by (2). It is worth noting that, for the sake of concise description, we mainly discuss in conjunction with Motivation A, simply assuming that there is only one instance of each microservice. This is easily extended to Motivation B, that is, each microservice has multiple identical instances, and the reason has been explained before.

$$\sum_{l=1}^{C} x_{i,j,l} \leq 1, \quad \forall i \in \{1, 2, ..., N\}, \quad \forall j \in \{1, ..., h_i\}. \tag{2}$$

3. *The selection of offloaded microservices should consider the dependency of microservices*: Generally, to maximize the benefit of the close proximity of the MEC datacenter to mobile devices, predecessor microservices are offloaded to the MEC, and successor microservices are offloaded to the cloud. If the jth microservice is offloaded to the MEC, the $(j-1)$th microservice should also be offloaded to the MEC. Otherwise, it would not be possible to improve the performability. Thus, the constraint is described by

$$\sum_{l=1}^{C} x_{i,j,l} \leq \sum_{l=1}^{C} x_{i,j-1,l}, \quad \forall i \in \{1, 2, ..., N\}, \quad \forall j \in \{2, ..., h_i\}. \tag{3}$$

Then, we introduce our optimization target for resource allocation in two steps.

1. *Static Resource Allocation Problem (SRAP)*: Given the resource supply and demand in one time slot, the problem is to maximize the overall performability in the slot, formulated as

$$\max pf = \sum_{i=1}^{N} \sum_{j=1}^{h_i} \sum_{l=1}^{C} x_{i,j,l} \times u_{i,j} \quad \text{s.t.} \quad (1)\text{–}(3). \tag{4}$$

2. *Dynamic Resource Allocation Problem (DRAP)*: Given the resource supply and demand in each time slot during one time period, the problem is to decide the allocation in each time slot and maximize the long-term overall performability.

The allocation is not a simple sum of performability in each time slot. During the reallocation in the next slot, some instances may be dynamically migrated to a new server, and the migration time causes a latency surge. The reallocation overhead is estimated using a linear function of the average memory footprint.[39] In this problem, it can be described as (5). α denotes the slope in the linear function. $d_{i,j}^2$ denotes the memory demand and is used to estimate the average memory footprint. $x_{i,j,l}(t)$ and $h_i(t)$ denote the decision and the number of microservices for service i in the tth slot, respectively.

$$re\_overheads(t) = \alpha \cdot \sum_{l=1}^{C} \sum_{i=1}^{N} \sum_{j=1}^{h_i(t)} \left[x_{i,j,l}(t) \times (1 - x_{i,j,l}(t-1)) \times d_{i,j}^2 \right]. \tag{5}$$

Thus, the global optimization goal is expressed as

$$\max \sum_{t=1}^{T} (pf(t) - re\_overheads(t)) \quad \text{s.t.} \quad \forall t, (1)\text{–}(3). \tag{6}$$

Both *SRAP* and *DRAP* are based on our PARA scheme, which is shown in Figure 6. *SRAP* discusses the service performability maximization in one single cycle. It can be regarded as a basic version of PARA. On the basis of *SRAP*, *DRAP* discusses the service performability maximization from the long-term point of view. It can be regarded as a final version of PARA.

4.3 | Uniqueness of the problem

The MEC resource allocation problem is always abstracted as an MKDP. This approach regards the MEC microdatacenter as a knapsack and each service as a thing to be placed in the knapsack. Actually, one service can be composed of several microservices; thus, each service has more than one offloading alternative, and at most one of these alternatives will be selected. Different alternatives are corresponding to different service performability. Thus PARA proposes to escalate this problem to an MCMDKP. Each offloading alternative is regarded as a choice in an MCMDKP. Moreover, PARA takes into account that the microdatacenter is also composed of several edge servers. Each edge server can be regarded as a knapsack for deploying services. This makes existing solutions to the MCMDKP less applicable to PARA. The dynamic resource demands further complicate the problem. If the decision maker considers the allocation in each time slot *independently*, described in *SRAP*, it is very likely to create unnecessary container migrations among edge servers, decrease the service performability and degrade users' experience during migration. Thus, we discuss *DRAP* in which the allocation decision in the current time slot will be *correlated* with the decision in the next time slot to reduce unnecessary overheads. In conclusion, the problem discussed in our PARA scheme is different from the classical knapsack problem.

5 | ALGORITHM DESIGN

In this section, we propose our algorithm designs for solving *SRAP* and *DRAP* in PARA scheme.

5.1 | Solution to SRAP

A greedy heuristic algorithm is simple but highly efficient for solving optimization problems. Many studies have applied greedy algorithms to solve resource allocation and service placement problems.[3,9,30] We have also considered many other algorithms, such as genetic algorithm and enhanced learning algorithms. But due to the very huge search space, it has to take these algorithms a large amount of time to find the solution, which is not so practical to the high real-time edge resource allocation scenario. Thus, in our proposal, we consider the characteristics of *SRAP* and then design a greedy algorithm to efficiently find a quasioptimal solution to the *SRAP*, given in Algorithm 1. The input consists of the number of microservices h_i in each service, the resource demands $d_{i,j}^{v}$ and the performability improvement $u_{i,j}$ of each required microservice.

The output is the allocation decision $x_{i,j,l}$. The algorithm is periodically called with a time slot having a specified duration. The decision is valid for the current time slot.

Algorithm 1: Greedy Heuristic Algorithm

Input: Vectors of the number of micro-services h_i, matrix of the resource demands $d_{i,j}^v$, matrix of the performability improvement $u_{i,j}$

Output: Matrix of the allocation decision $x_{i,j,l}$

1 Initialize all $x_{i,j,l}$ to zero, $choice = \phi$
2 **for** $v = 1, 2 \cdots K$ **do**
3 $\quad w_v = \frac{\sum_{i=1}^{N} \sum_{j=1}^{h_i} d_{i,j}^v}{\sum_{l=1}^{C} s_l^v}$
4 **for** $i = 1, 2 \cdots N$ **do**
5 $\quad$ **for** $j = 1, 2 \cdots h_i$ **do**
6 $\quad\quad ap_{i,j} = \frac{\sum_{m=1}^{j} u_{i,m}}{\sum_{m=1}^{j} \sum_{v=1}^{K} w_v \times d_{i,m}^v}$, $choice = choice \cup [i, j, ap_{i,j}]$
7 Sort $choice$ in non-increasing order of $ap_{i,j}$
8 **while** $choice! = \phi$ **do**
9 $\quad i, j, ap_{i,j} = choice[0]$, $available = True$
10 $\quad$ **for** $m = 1, 2 \cdots j$ **do**
11 $\quad\quad available = True$
12 $\quad\quad$ **for** $l = 1, 2 \cdots C$ **do**
13 $\quad\quad\quad$ **if** $x_{i,j,l} == 1$ **then**
14 $\quad\quad\quad\quad$ break
15 $\quad\quad\quad available = True$
16 $\quad\quad\quad$ **for** $v = 1, 2 \cdots K$ **do**
17 $\quad\quad\quad\quad$ **if** $s_l^v \le d_{i,m}^v$ **then**
18 $\quad\quad\quad\quad\quad available = False$
19 $\quad\quad\quad$ **if** $available == True$ **then**
20 $\quad\quad\quad\quad$ **for** $v = 1, 2 \cdots K$ **do**
21 $\quad\quad\quad\quad\quad s_l^v = s_l^v - d_{i,m}^v$, $x_{i,m,l} = 1$
22 $\quad\quad\quad\quad$ break
23 $\quad\quad$ **if** $available == False$ **then**
24 $\quad\quad\quad$ break
25 $\quad$ **if** $available == False$ **then**
26 $\quad\quad$ **for** *item in choice* **do**
27 $\quad\quad\quad$ **if** $item[0] == i$ *and* $item[1] >= j$ **then**
28 $\quad\quad\quad\quad$ remove *item* from *choice*
29 $\quad$ **else**
30 $\quad\quad$ **for** *item in choice* **do**
31 $\quad\quad\quad$ **if** $item[0] == i$ **then**
32 $\quad\quad\quad\quad$ **if** $item[1] <= j$ **then**
33 $\quad\quad\quad\quad\quad$ remove *item* from *choice*
34 $\quad\quad\quad\quad$ **else**
35 $\quad\quad\quad\quad\quad item[2] = \frac{\sum_{m=j+1}^{item[1]} u_{i,m}}{\sum_{m=j+1}^{item[1]} \sum_{v=1}^{K} w_v \times d_{i,m}^v}$
36 $\quad\quad\quad\quad\quad$ Sort *choice* again
37 **return** all $x_{i,j,l}$

First, the algorithm determines the weight of each type of resource (lines 2 and 3) and the average performability of different offloaded microservices choices for each service (lines 4–6) based on the resource supply and demand.

Then, the algorithm sorts these choices in nonincreasing order of their average performability (line 7) and prioritizes the allocation of the choice that gives the highest average performability (line 9). In the allocation, the algorithm checks whether the edge servers can satisfy the resource demands of all microservices contained in the choice (lines 12–18). The algorithm places the microservice to the first fit server with the expected resource demands (lines 19–22). If no servers can satisfy the demands, the algorithm decides that the selected choice is invalid. Thus, it should return to the previous state and remove some other choices based on constraint (3) (lines 25–28).

If all the microservices in the choice are satisfied, the algorithm decides that the selected choice is valid and that some relevant choices should be updated. Specifically, it is unnecessary to consider choices that are subsets of the valid selected choice (lines 32 and 33) because the microservices in these choices have been satisfied. The remaining choices should be updated in terms of their average performability and be sorted again (lines 34–36) to continue to compete for the resource. The allocation decision ends when no choice is competitive (line 8). These microservices unable to obtain MEC resources would be satisfied by the remote cloud.

5.2 | Solution to DRAP

To solve *DRAP*, we first predict the resource demands at the next time slot online and then design the solution to the above greedy heuristic algorithm. For convenience, we call the solution from the online greedy heuristic algorithm.

5.2.1 | Online prediction

The number of requested microservices $h_{i,j}$ changes dynamically over time and invokes a new allocation in edge servers. To avoid frequent immigration and emigration harming the long-term overall performability, we should consider the sustainability of microservice demands. Thus, we record the historical data $\{h_i(1), h_i(2), ..., h_i(t)\}$ and predict whether $ms_{i,j}$ still requires resources on the next time slot. Here $h_i(1)$ denotes the number of microservices for service i at the first time slot. We introduce a new parameter $hp_i(t+1)$ to represent the number of microservices that require resources at the tth and $(t+1)$th time slots. The online prediction algorithm is shown in Algorithm 2.

Algorithm 2: Online Prediction Algorithm

Input: The historical data $\{h_i(1), h_i(2) \cdots h_i(t)\}$
Output: The prediction $hp_i(t+1)$

```
for m = 0, 1, 2 ··· h_i(t) do
    count = 0
    for b = 1, 2 ··· (t − 1) do
        if h_i(b) ≤ m then
            count− = 1
        else
            count+ = 1
    if count ≤ 0 then
        return m
return h_i(t)
```

If the service i attempts to offload $ms_{i,j}$ to the edge server at the tth time slot, $h_i(t)$ will be at least j; otherwise, $h_i(t)$ will be smaller than j. We calculate the frequency at which the service requests the deployment of the microservice for two consecutive time slots (lines 4 and 5) and the frequency at which the service requested the deployment of the microservice in the previous slot but did not request it in the latter slot (lines 6 and 7). By comparing the two frequencies, we can judge whether the microservice is more likely to continue requesting edge resources at the next time slot (lines 9 and 10).

5.2.2 | Online allocation

On the basis of the prediction, we improved Algorithm 1 to allocate resources from two aspects.

First, we prioritize microservices with resource requests that have a certain continuity. We introduce $u'_{i,j}$, the combination of the performance improvement $u_{i,j}$ and the reallocation overhead $\alpha \times d_{i,j}^2$, to replace $u_{i,j}$ in Algorithm 1. The calculation of $u'_{i,j}$ is shown in (7). If the microservice $ms_{i,j}$ has been offloaded in one edge server at the last time slot, we assume that $ms_{i,j}$ can still be deployed on the server without reallocation overhead, and $u'_{i,j}$ is equal to $u_{i,j}$. If $ms_{i,j}$ only requires edge resources for the current time slot, the reallocation overhead is valid for a single time slot, and $u'_{i,j}$ should be the difference between $u_{i,j}$ and $\alpha \times d_{i,j}^2$. Supposing that $ms_{i,j}$ requires edge resources at multiple continuous time slots, the reallocation overhead can be divided equally into these slots. We found that the inaccuracy of multislot prediction strongly affects the performance. Thus, we only predict the resource demands for the next time slot in our paper.

$$u'_{i,j} = \begin{cases} u_{i,j}, & j \le h_i(t-1), \\ u_{i,j} - \alpha \times d_{i,j}^2, & h_i(t-1) < j \le hp_i(t+1), \\ u_{i,j} - \dfrac{\alpha \times d_{i,j}^2}{2}, & h_i(t-1) < j \text{ and } hp_i(t+1) < j. \end{cases} \tag{7}$$

Second, unnecessary migration between edge servers decreases. We record the allocation decision at the last time slot. If one microservice has been offloaded to one edge server at the last time slot and if the server can still satisfy the microservice's resource demands at the current time slot, the server will continue to allocate resources to the microservice rather than simply applying the first-fit rule.

6 | PERFORMANCE EVALUATION

In this section, we perform numerous simulations to evaluate the performance of our solutions to *SRAP* and *DRAP*. The two solutions discuss the resource allocation problem in PARA scheme from the perspectives of a single time slot and long-term effect, respectively.

6.1 | Evaluation of SRAP

6.1.1 | Simulation setting

We evaluate the performance of our designed greedy heuristic algorithm for *SRAP* through large-scale test cases.

In the default simulation setting, the MEC datacenter has 10 edge servers whose configuration is the same as that of a Dell PowerEdge R740 Rack Server. Each edge server could provide three types of resources: 128 vCPUs, 512 GB of memory and 3000 GB of storage. The resource demands refer to the AWS ES2 m5ad family. We select five instance configurations from the family: m5ad.large, m5ad.xlarge, m5ad.2xlarge, m5ad.4xlarge, and m5ad.12xlarge. Their corresponding resource demands are shown in Table 2, and their prices are 0.103, 0.206, 0.412, 0.824, and 2.472 USD/h, respectively. For each microservice, we randomly select one instance type to set the resource demands. To reflect the variety of performability, the performability of the microservices follows a normal distribution, whose mean value and standard deviation are the prices of the instance. Each service has 5–10 microservices. We introduce a new parameter, p, the vCPU supply-to-demand ratio, to evaluate the overall resource supply-to-demand ratio. This simulation method follows.[9,26,35] We repeated each set of simulations 50 times independently and recorded the median results.

6.1.2 | Baseline

We introduce a set of algorithms to solve the resource allocation problem. The first baseline is used to prove the necessity of considering service degradability. The following three baselines are used to demonstrate the efficiency of our algorithm with other top-ranked algorithms.

Baseline1: Most resource allocation research does not consider service degradability. *G-ERAP*, proposed in Bahreini et al.,[9] is representative. *G-ERAP* determines the average bid per unit of resource for each user and then allocates resources to the users in the nonincreasing order of their average bids. If one service's resource demands cannot fully be satisfied by the MEC datacenter, the MEC datacenter will not allocate any resources to the service or other services whose average bid is lower. The bid by the users can be understood as the performability of services in our problem. *G-ERAP* does not specifically discuss the service placement; thus, we also place instances on the first-fit server.

TABLE 2 Configurations of resource demands

Instances	vCPU	Memory (GB)	Storage (GB)
m5ad.large	2	8	75
m5ad.xlarge	4	16	150
m5ad.2xlarge	8	32	300
m5ad.4xlarge	16	64	600
m5ad.12xlarge	48	192	1800

Baseline2: We consider service degradability based on *G-ERAP*. Specifically, when one service's demands are not fully satisfied, the MEC datacenter still allocates resources to some microservices of the service as much as possible to improve service performability.

Baseline3: We adopt a classical genetic algorithm to solve *SRAP*. This algorithm randomly generates a population of individuals that determine which microservice is deployed on which server and then repairs each individual to obey the constraints. In the repair process, if the resource demands of one server exceed the resource supplies, the server will remove the microservices with the lowest average bid. If a microservice is not deployed on any edge server, the subsequent microservices of the service will also not be allocated resources from edge servers. The algorithm selects individuals from the population by an elitist strategy[40] and generates new individuals by a uniform crossover strategy[41] and simple mutation strategy.[42] If the fitness of the new individual is higher than that of its parents, the new individual will replace its parents. The fitness is the overall performability in *SRAP*. The number of individuals and iterations is set as 80 and 500, respectively. The setting is large enough to guarantee satisfactory performance.

Baseline4: In essence, the *SRAP* is a mixed integer programming problem. We use the classical solver *Mosek*[43] as one of the baselines. The time taken to find a solution is always expected to be as short as possible. We conducted numerous preliminary experiments to find the shortest solution time that can ensure relatively good solution results under the *Mosek* method. Thus, we set an upper limit to the solution time of *Mosek* to be 120 s. A longer solution time may get better results, but it is usually intolerable in the latency-critical edge environment.

6.1.3 | Evaluation results

We define three metrics to evaluate the above algorithms: the normalized performability, the satisfaction, and the solution time. The normalized performability is the ratio of the obtained performability using these algorithms to the ideal performability when all the resource demands are met. Besides we calculate for each service the ratio of the actually obtained performability to its expectation. And the satisfaction metric is defined as the average of these ratios to indicate whether these services are satisfactory to the resource allocation. The solution time is the time taken to find an allocation scheme.

Figure 7 shows the evaluation results on *SRAP* with a different demand–supply ratio p. Among these algorithms, baseline1 always has the lowest normalized performability because it neglects the service degradability and allocates resources in a coarse manner. Our designed greedy heuristic algorithm retains greater than 97% of the optimal performability given in baseline4. When the resource supply-to-demand ratio decreases from 1.6 to 0.4, the performability of our algorithm can be up to 1.11–1.28 times that of baseline1, 1.02–1.11 times that of baseline2, and 1.10–1.22 times that of baseline3. And the satisfaction of our algorithm is 1.14–1.52 times that of baseline1, 1.03–1.25 times that of baseline2, and 1.10–1.18 times that of baseline3. In terms of solution time, baseline3 and baseline4 are quite time consuming, being several orders of magnitude higher than the other algorithms.

To further analyze the impact of the number of microservices h_i on the algorithm performance, we set p to 1 and range the value of h_i from 5 to 15. As shown in Figure 8, with

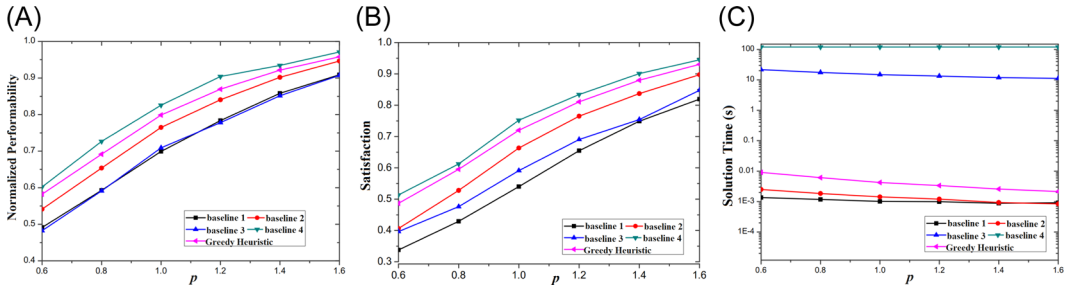

FIGURE 7 Evaluation results on SRAP with different p. (A) Normalized performability with different algorithms, (B) satisfaction with different algorithms, and (C) solution time with different algorithms. SRAP, Static Resource Allocation Problem. SRAP, Static Resource Allocation Problem. [Color figure can be viewed at wileyonlinelibrary.com]

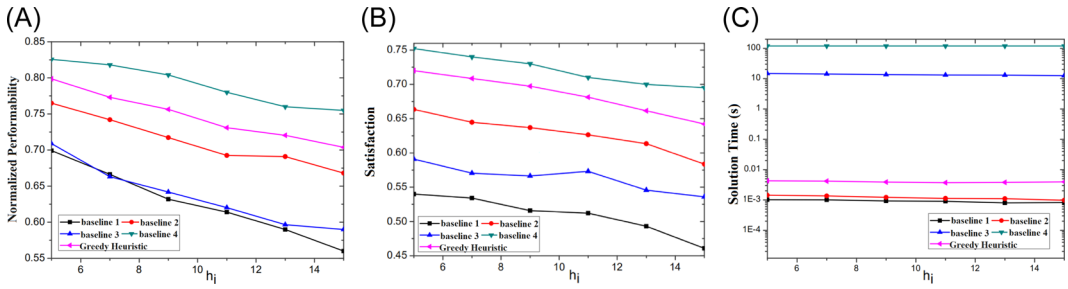

FIGURE 8 Evaluation results on SRAP with different h_i. (A) Normalized performability with different algorithms, (B) satisfaction with different algorithms, and (C) solution time with different algorithms. SRAP, Static Resource Allocation Problem. [Color figure can be viewed at wileyonlinelibrary.com]

the increase of h_i, the normalized performability and the satisfaction are all declined. Specifically, the normalized performability of baseline1–4 and our algorithm declined, respectively, 19.8%, 12.6%, 16.7%, 8.5%, and 11.87%, the satisfaction decreased by 14.6%, 12.0%, 9.3%, 7.6%, and 10.8%, respectively. Baseline1 decreased the most, followed by baseline2, baseline3, and our algorithm, and baseline4 decreased the least. The reason is that baseline1 does not perceive the service degradeability. When the resource demand of one service expands, it increases the possibility that the edge resources cannot fully meet the demand of the service. Baseline2–4 and our algorithm can make use of the degradability to slow down its decline speed, but they are still affected to a certain extent. The solution time of the problem remains stable on the whole and is not greatly affected by h_i.

6.2 | Evaluation of DRAP

6.2.1 | Simulation setting

We use the Google Cluster data set[44] to simulate dynamic scenarios and evaluate the performance of our online greedy heuristic algorithm.

For resource supplies, we refer to the machine attributes given in the data set and set the CPU, memory, and storage configuration of the edge server as 0.5, 0.5, and 0.0015. These values are not real values but rather are values normalized to 1. The default number of edge servers is 20. In terms of resource demands, we track 591 jobs in 269 consecutive time slots. These jobs require resources in at least 10 slots. The jobs and tasks are regarded as services and microservices. The corresponding resource demands and priorities simulate the resource demands and performability improvement of the microservices. Because the priority ranges from 0 to 11, we add 1 to this value to make it at least positive to participate in the resource competition, set the performability improvement according to a normal distribution whose mean value and standard deviation are the modified priority and ensure that the set value is positive. The default value of α is set to 200, which can reflect the importance and consideration of reallocation overheads in the problem solution.

6.2.2 | Baseline

To prove the importance of considering reallocation overhead in optimizing the ultimate overall performability, we adopt the baseline1, baseline2, baseline3, and greedy heuristic algorithms as baselines. Because *mosek* in baseline4 seeks the globally optimal solution in an offline situation whereby resource demands in each time slot are known first, it is not really applicable to dynamic situations. The dimensionality limit is readily exceeded when solving the problem. Thus, we do not list baseline4 as one of the baselines in this subsection.

6.2.3 | Evaluation results

We first evaluate the accuracy of online prediction and then compare our online greedy heuristic algorithm with other baselines; finally, we analyze the impact of the parameter settings on the overall performability.

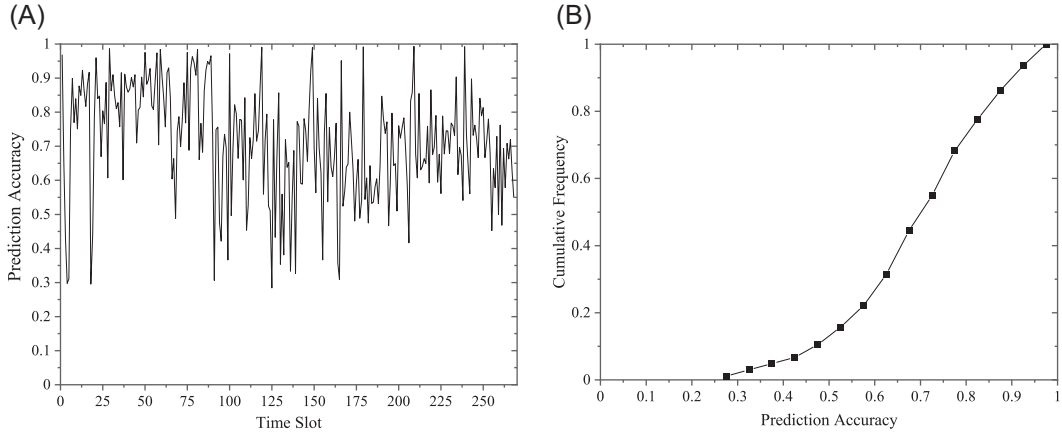

FIGURE 9 Evaluation results in the prediction function. (A) The prediction accuracy at different time slots and (B) the cumulative distribution function

Figure 9A presents the prediction accuracy at different time slots. The accuracy fluctuates with time but remains stable overall. The average accuracy is 76.5%. To further analyze the accuracy distribution, we present the cumulative distribution function of the prediction accuracy in Figure 9B. The accuracy mainly ranges from 63.0%–82.5%. The probability that the prediction accuracy is less than 60% is only 29%. The prediction results still need to be improved for resource allocation optimization in the highly dynamic network edge environment. There always exists a trade-off between prediction accuracy and prediction efficiency. We would further study the prediction mechanism in PARA in the future. There is no baseline that predicts workload in dynamic MEC resource allocation and service placement, thus we then select reallocation overheads and overall performability as metrics to compare with other baselines.

On the basis of the prediction, our online greedy heuristic algorithm can greatly decrease the reallocation overhead. Figure 10 shows the performance under different algorithms. Each histogram represents the corresponding algorithm's raw performability, which is the sum of the performability improvement of all offloaded microservices *without consideration of the reallocation overhead*. This consists of two parts: reallocation overhead and ultimate overall performability. The ultimate overall performability is the most important metric for edge computing systems. We found that the online greedy algorithm can almost achieve the raw performability of baseline3. Due to the significant decrease in overhead, the online greedy algorithm achieves the highest ultimate overall performability, being 19.45%, 14.06%, 15.13%, and 12.11% higher than the other four baselines.

We further evaluate these algorithms' performances under different numbers of edge servers C and parameter α to analyze their impact. As shown in Figure 11, the raw performability increases with an increasing number of edge servers. Baseline2, the greedy heuristic algorithm and the online greedy heuristic algorithm are superior to baseline1 and baseline3. The reallocation overhead also increases; however, that of the online algorithm increases more steadily. When the number of edge servers increases from 14 to 24, the increases in the reallocation overhead with these algorithms are 9.98%, 5.1%, 6.5%, 5.08%, and 2.3%. The gap between the online algorithm and the other algorithms widens. The ultimate overall performability of the online algorithm was up to 22.1%, 14.2%, 19.51%, and 12.9% higher than the other four baselines. Figure 12 illustrates the performance under

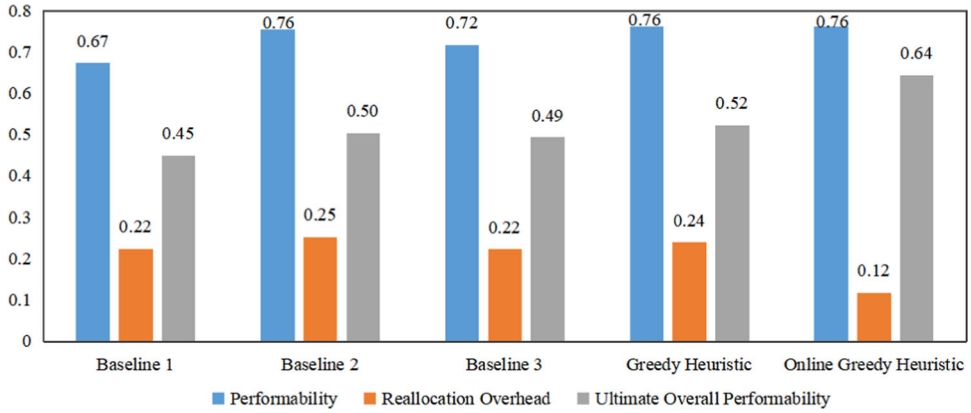

FIGURE 10 Performability of different algorithms [Color figure can be viewed at wileyonlinelibrary.com]

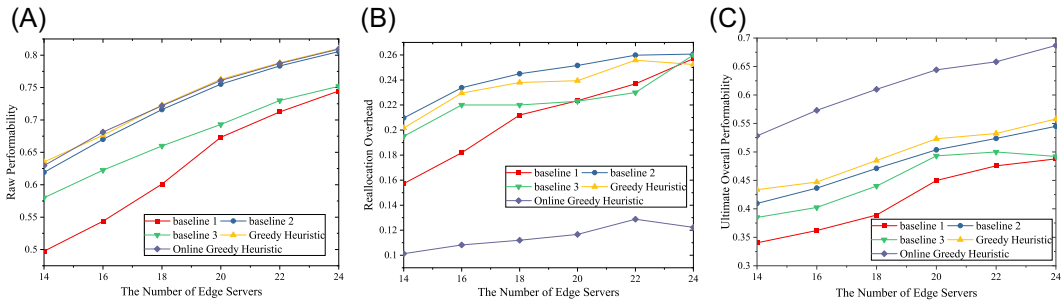

FIGURE 11 Performance under different numbers of edge servers. (A) Raw performability, (B) reallocation overhead, and (C) ultimate overall performability. [Color figure can be viewed at wileyonlinelibrary.com]

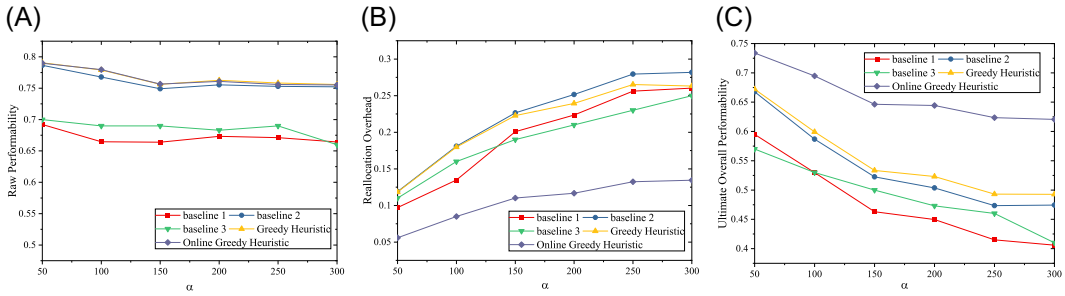

FIGURE 12 Performance under different parameters α. (A) Raw performability, (B) reallocation overhead, and (C) ultimate overall performability. [Color figure can be viewed at wileyonlinelibrary.com]

different α. We found that α has little to no effect on raw performability but does affect the reallocation overhead. Because these algorithms, except for the online algorithm, do not consider overhead, the reallocation overhead increases linearly with increasing α. The gap between the online algorithm and the other algorithms also widens from 2.9%–6.2% to 7.8%–16.2%. Thus, the online algorithm achieves the highest ultimate overall performability among these baselines.

7 | CONCLUSION

We designed a PARA strategy to satisfy the resource requirements for cloud naive service offloading on edge computing platforms. It takes into account the three factors of service degradability, dynamic resource requirements, and resource supply characteristics of MEC and cloud. We formulated the resource allocation model to maximize the long-term overall performability, and then designed an enhanced heuristic algorithm. We observe the historical resource demand and analyze its behavior for demand prediction. The prediction empowers the heuristic algorithm to efficiently finds a quasioptimal solution to the allocation model. The results of numerical experiments demonstrated that our solution was simple but efficient to improve the long-term overall performability of multiple competing services with a great decrease in reallocation overhead. It is very promising for the edge computing microservice

architecture to optimize resource allocation when there are sequential execution order constraints between microservices.

In the future, we will further deploy it on Baidu OTE platform for verification and promote our design with some important issues, for example, the more general case with multiple heterogeneous MEC datacenters, highly dynamic resource demands, nonlinear dependency in microservices, application content caching, and so on.

ACKNOWLEDGMENTS

This study is supported by National Natural Science Foundation of China-Guangdong Joint Fund (NSFCU1811461), National Natural Science Foundation of China (62172155, 61832020, and 62072465), National Key Research and Development Program of China (2018YFB0204301), and Natural Science Foundation of Guangdong Province (2018B030312002).

ORCID

Yeting Guo http://orcid.org/0000-0002-1877-7796
Fang Liu http://orcid.org/0000-0001-8753-3878
Nong Xiao http://orcid.org/0000-0002-2166-977X
Zhaogeng Li http://orcid.org/0000-0003-0331-5852
Zhiping Cai http://orcid.org/0000-0001-5726-833X
Guoming Tang http://orcid.org/0000-0001-9801-1055
Ning Liu http://orcid.org/0000-0003-4140-9489

REFERENCES

1. Liu F, Tang G, Li Y, Cai Z, Zhang X, Zhou T. A survey on edge computing systems and tools. *Proc IEEE*. 2019;107(8):1537-1562.
2. Kerö N, Puhm A, Kernen T, Mroczkowski A. Performance and reliability aspects of clock synchronization techniques for industrial automation. *Proc IEEE*. 2019;107(6):1011-1026.
3. He T, Khamfroush H, Wang S, Porta TL, Stein S. It's hard to share: joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In: Pietzuch P, ed. *38th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE; 2018:365-375.
4. Shen H, Chen L. A resource usage intensity aware load balancing method for virtual machine migration in cloud datacenters. *IEEE Trans Cloud Comput*. 2020;8(1):17-31.
5. Meyer JF. On evaluating the performability of degradable computing systems. *IEEE Trans Comput*. 1980;29(8):720-731.
6. Oliveira D, Brinkmann A, Rosa N, Maciel P. Performability evaluation and optimization of workflow applications in cloud environments. *J Grid Comput*. 2019;17(4):749-770.
7. Ren J, Yu G, Cai Y, He Y. Latency optimization for resource allocation in mobile-edge computation offloading. *IEEE Trans Wireless Commun*. 2018;17(8):5506-5519.
8. Zhang C, Du H, Ye Q, Liu C, Yuan H. DMRA: a decentralized resource allocation scheme for multi-SP mobile edge computing. In: Du D, ed. *39th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE; 2019:390-398.
9. Bahreini T, Badri H, Grosu D. An envy-free auction mechanism for resource allocation in edge computing systems. In: Shi W, Verma D, eds. *3rd IEEE/ACM Symposium on Edge Computing*. IEEE; 2018:313-322.
10. Markets and Markets. *Cloud Microservices Market by Component (Platform and Services), Deployment Mode (Public Cloud, Private Cloud, and Hybrid Cloud), Organization Size (Large Enterprises and SMEs), Vertical, and Region—Global Forecast to 2023*; 2018.
11. Yi S, Hao Z, Zhang Q, Zhang Q, Shi W, Li Q. LAVEA: latency-aware video analytics on edge computing platform. In: Zhang J, ed. *2nd ACM/IEEE Symposium on Edge Computing*. IEEE; 2017:15:1-15:13.
12. Simon S. A provider-friendly serverless framework for latency-critical applications. In: Giceva J, ed. *12th Eurosys Doctoral Workshop*. IEEE; 2018.

13. Qu Q, Xu R, Nikouei SY, Chen Y. An experimental study on microservices based edge computing platforms. In: Liu Y, Vuran MC, Wu H, eds. *39th IEEE Conference on Computer Communications, INFOCOM Workshops*. IEEE; 2020: 836-841.
14. Xu R, Jin W, Kim D. Enhanced service framework based on microservice management and client support provider for efficient user experiment in edge computing environment. *IEEE Access*. 2021;9:110683-110694.
15. Ma S, Fan C, Chuang Y, Lee W, Lee S, Hsueh N. Using service dependency graph to analyze and test microservices. In: Reisman S, Ahamed SI, eds. *42nd IEEE Annual Computer Software and Applications Conference*. IEEE; 2018:81-86.
16. Mahmoodi SE, Uma RN, Subbalakshmi KP. Optimal joint scheduling and cloud offloading for mobile applications. *IEEE Trans Cloud Comput.* 2019;7(2):301-313.
17. Tian S, Deng X, Chen P, Pei T, Oh S, Xue W. A dynamic task offloading algorithm based on greedy matching in vehicle network. *Ad Hoc Networks*. 2021;123:102639.
18. Zhang J, Hu X, Ning Z, et al. Joint resource allocation for latency-sensitive services over mobile edge computing networks with caching. *IEEE Internet Things J*. 2019;6(3):4283-4294.
19. Farhadi V, Mehmeti F, He T, et al. Service placement and request scheduling for data-intensive applications in edge clouds. In: Lou W, Wolf T, eds. *IEEE Conference on Computer Communications, INFOCOM*. IEEE; 2019:1279-1287.
20. Yang L, Cao J, Liang G, Han X. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Trans Comput*. 2016;65(5):1440-1452.
21. Deshpande U. Caravel: burst tolerant scheduling for containerized stateful applications. In: Du D, ed. *39th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE; 2019:1432-1442.
22. Raei H, Yazdani N. Performability analysis of cloudlet in mobile cloud computing. *Inf Sci*. 2017;388:99-117.
23. Rahman A, Jin J, Cricenti AL, Rahman A, Kulkarni A. Communication-aware cloud robotic task offloading with on-demand mobility for smart factory maintenance. *IEEE Trans Ind Inf*. 2019;15(5):2500-2511.
24. Chen X, Jiao L, Li W, Fu X. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Trans Networking.* 2015;24(5):2795-2808.
25. Cardoso KV, Abdel-Rahman MJ, MacKenzie AB, DaSilva LA. Virtualization and programmability in mobile wireless networks: architecture and resource management. In: Katz-Bassett E, Sherry J, eds. Proceedings of the Workshop on Mobile Edge Communications, MECOMM@SIGCOMM. IEEE; 2017:1-6.
26. Xu J, Palanisamy B, Ludwig H, Wang Q. Zenith: utility-aware resource allocation for edge computing. In: Altintas I, Chen S, eds. *International Conference on Edge Computing*; 2017:47-54.
27. Castellano G, Esposito F, Risso F. A distributed orchestration algorithm for edge computing resources with guarantees. In: Lou W, Wolf T, eds. *2019 IEEE Conference on Computer Communications (INFOCOM)*. IEEE; 2019:2548-2556.
28. Samanta A, Jiao L, Mühlhäuser M, Wang L. Incentivizing microservices for online resource sharing in edge clouds. In: Du D, ed. *39th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE; 2019:420-430.
29. Anuradha VP, Sumathi D. A survey on resource allocation strategies in cloud computing. In: *International Conference on Information Communication and Embedded Systems*; 2014:1-7.
30. Xiao Z, Song W, Chen Q. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Trans Parallel Distrib Syst*. 2013;24(6):1107-1117.
31. Zhao D, Zhou J, Li K. An energy-aware algorithm for virtual machine placement in cloud computing. *IEEE Access*. 2019;7:55659-55668.
32. Ruan X, Chen H, Tian Y, Yin S. Virtual machine allocation and migration based on performance-to-power ratio in energy-efficient clouds. *Future Gener Comput Syst.* 2019;100:380-394.
33. Wang L, Jiao L, He T, Li J, Mühlhäuser M. Service entity placement for social virtual reality applications in edge computing. In: Mao S, Melodia T, Sinha P, eds. *2018 IEEE Conference on Computer Communications (INFOCOM)*; 2018:468-476.
34. Kecskemeti G, Németh Z, Kertész A, Ranjan R. Cloud workload prediction based on workflow execution time discrepancies. *Clust Comput.* 2019;22(3):737-755.
35. Duong-Ba TH, Nguyen T, Bose B, Tran T. A dynamic virtual machine placement and migration scheme for data centers. *IEEE Trans Serv Comput.* 2021;14(2):329-341.

36. Beloglazov A, Buyya R. Energy efficient allocation of virtual machines in cloud data centers. In: Parashar M, ed. *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*; 2010:577-578.
37. Kim J, Lee S, Kim D, Park J. Performability analysis of IaaS cloud. In: Tang F, Xhafa F, eds. *5th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*; 2011:36-43.
38. Silva B, Maciel P, Brilhante J, Zimmermann A. GeoClouds Modcs: A perfomability evaluation tool for disaster tolerant IaaS clouds. In: Zhang R, ed. *IEEE International Systems Conference*; 2014:116-122.
39. Shen Z, Subbiah S, Gu X, Wilkes J. CloudScale: elastic resource scaling for multi-tenant cloud systems. In: Chase JS, El Abbadi A, eds. *ACM Symposium on Cloud Computing in conjunction with SOSP*; 2011:1-5.
40. Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput*. 2002;6(2):182-197.
41. Syswerda G. Uniform crossover in genetic algorithms. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*; 1989:2-9.
42. Al-Afandi J, Horváth A. Adaptive gene level mutation. *Algorithms*. 2021;14(1):16.
43. Andersen E, Andersen K. The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm. *High Perform Optim*. 2000;33:197-232.
44. Reiss C, Wilkes J, Hellerstein J. *Google cluster-usage traces: format+ schema* [White paper, 1-14]. Google Inc.

How to cite this article: Guo Y, Liu F, Xiao N, et al. PARA: performability-aware resource allocation on the edges for cloud-native services. *Int J Intell Syst*. 2022;37:8523-8547. doi:10.1002/int.22954