Elastic Scaling of Resources for Energy-efficient Container Cloud Using Reinforcement Learning

Yanyu Shen, Chonglin Gu*, Xin Chen, Xiaoyu Gao, Zaixing Sun, Hejiao Huang

***Abstract*—In this paper, we aim to save the total energy consumption of servers through elastic scaling of CPU resources in container cloud. To be practical, we propose an online scheduling method, which consists of three parts: container placement, vertical scaling and migration. 1) For container placement, we design an algorithm based on dynamic threshold, resource balancing and delayed running. When there are PMs (Physical Machines) turned on, the CPU threshold increases so that the containers can be placed onto fewest possible PMs. To make full use of multi-dimensional resources of PM, we put forward a resource balancing strategy. Since the number of CPU cores can be scaled dynamically in containers' run time, the start time of containers can be delayed without violating deadlines. 2) For vertical scaling, a collaborative multi-agent reinforcement learning (MARL) algorithm is proposed to adjust the container's CPU, so that the containers on the same PM can finish simultaneously if possible. Then, the PM can be turned off to save energy. 3) To further reduce total energy consumption, we consider migrating the containers from underloaded PMs and overloaded PMs. Experiment results show the superior performance of our method to that of the state-of-the-art.**

***Index Terms*—Container cloud, Resources, Energy-efficient, Elastic scaling, Reinforcement learning.**

I. Introduction

In recent years, the high energy consumption has become a major issue faced by cloud data centers. The share of global data center electricity consumption will increase from 1.15% in 2016 to 1.86% by 2030, reaching 800TWh (within a margin of error of ±200TWh) [1]. It has been pointed out that servers consume 61% of the total energy in a data center, but their resources are not fully utilized [2]. Due to the growing number of users and the fluctuating demand for computing resources, servers are often over-provisioned. That is, servers are in idle most of the time [3], even in active state, only 20%∼30% of the resources are used [4], wasting a large amount of energy. Therefore, optimizing resource scheduling is the key to saving energy for cloud data center.

In modern cloud system, container has been extensively used due to its fast deployment feature. Its creation and destruction can be finished almost instantaneously. A container is just like a lightweight virtual machine (VM), so energy-efficient scheduling and migration is also applicable for container cloud. Different from VM, container usually shares the same operating system with its host, so its virtualization overhead is very low while with high flexibility when scaling at the run time [5]. Therefore, container can be a better choice especially for energy-efficient scaling.

For energy saving using elastic scaling of containers, most recent studies are mainly focused on meta-heuristics to reduce resource waste [6], [7]. However, these algorithms can not immediately make the right decisions to the dynamically changing environment since their modelings are usually too complex to be solved quickly [8]. Fortunately, reinforcement learning can well suit to such scenarios. It utilizes intelligent agents to build knowledge, take actions, and adapt to the changes of its environment through a load balance between exploration and exploitation, which can always generate adaptive and robust solution for autoscaling problems [9].

In this paper, we aim to save the total energy consumption of servers through vertical scaling of CPU resources in container cloud. In each scheduling time slot, there are many user requests arriving and each is composed of one or more tasks. Here each task requires a container with specified resources to complete within deadline. Container placement is an NP-hard problem considering the multi-dimensional resources. In fact, more energy can be saved if the containers can be placed properly while using other energy-saving methods. Another common energy-saving methods is to migrate all the containers from underloaded PMs and then turning off the idle PMs. Finally, we propose an online scheduling method, which consists of three parts:

1. *Container Placement*. We propose a container placement strategy based on *dynamic threshold*, *resource balancing* and *delayed running*. The higher the threshold, the more containers can be placed onto the server, but meanwhile the scaling extent of the running containers is limited since it depends on the remaining unallocated resource. When there are PMs turned on, the CPU threshold increases. Otherwise, the threshold will decrease with time. We also consider balancing different resources of PMs so as to make full use of the multi-dimensional resources. There are times when the tasks cannot be processed immediately after arriving due to limited resources. To lower down the number of task rejections, we delay the running of these tasks while speeding up their running through vertical scaling of the containers so as to meet their deadlines.
2. *Vertical Scaling*. In general, an active PM can be turned off only when all the containers running onside have finished their tasks. Therefore, vertical scaling is adopted to adjust the running speed of the containers, so that the containers on the same PM can finish simultaneously if possible, and then the PM can be turned off to save energy. Note that, we regard the problem as a decen-

tralized partially observable Markov decision progress and introduce a cooperative multi-agent reinforcement learning algorithm QMIX [10] to control the vertical scaling of CPU resources for containers that have been placed on the same PM.

3. *Migration*. For underloaded PM, we try to migrate all the containers onside until it is idle, and then turn off the PM to save energy. For overloaded PM, we try to reduce the high utilization of CPU through migrating the containers since the power of a PM is greatly affected by its CPU utilization. In this way, the scaling range of CPU will also be extended, making it more flexible for vertical scaling.

Specifically, the main **contributions** of our work can be summarized as follows:

- Considering the vertical scaling feature of container, we propose a multi-agent reinforcement learning (MARL) algorithm to dynamically adjust the CPU resources of the containers in the granularity of time slot, so that the total energy consumption of the servers can be minimized while satisfying the deadline requirements of the tasks as much as possible.
- We propose a new container placement algorithm that can accommodate as many containers as possible with fewest PMs using dynamic threshold. The ingenuity of the design of our vertical scaling lies in: 1) it can accelerate the running speed of the containers with delayed tasks to meet the deadlines so as to reduce the number of task rejections; 2) it can adjust the speed of the containers on the same PM to end the containers simultaneously, so that the PM can be turned off to save energy. We try to consolidate containers and flexibly adjust the scaling range of CPU resources through migration for underloaded PMs and overloaded PMs, respectively.
- Our method is evaluated with simulated data based on the real-world trace on various metrics. Experiment results show that our method has superior performance to the state-of-the-art methods.

The remainder of this paper is organized as follows. Section 2 summarizes related work. Section 3 gives the formulation of the problem in detail. Section 4 presents the design of our algorithm. In Section 5, the experimental setup is described and the results are analyzed. Section 6 concludes the whole paper.

II. Relate Work

There has been extensive research on energy saving for cloud data centers, which can be classfied into two categories: resource management and lifecycle management.

A. Resource Management

For resource management, the existing research can be classified into static resource management and dynamic resource management.

1) Static Resource Management: According to user requests, cloud data center usually place containers or VMs onto proper PMs to make full use of the resources. Container or VM placement can be regarded as a multi-dimensional bin packing problem, which is an NP-hard problem [11].

CPU and memory (Mem) are the main resources considered in most placement problems in cloud computing. Lin et al. [12] additionally considered bandwidth resource, and proposed a nonlinear programming model to solve the divisible task-scheduling problem. Azizi et al. [13] proposed a balancing strategy for CPU and Mem, so that both resources on the PM can be fully utilized and more requests can be processed. This strategy can be extended for three or more resources. EL-Taani et al. [14] proposed a statistical-based method to measure the similarity between PM and VM resources (CPU, Mem, bandwidth), based on which to reduce resource wasting in VM placement. Elsakhawy et al. [15] predicted the PM resource gap caused by VM release and then filled it with a new VM, so that the resource can keep in use. Both Karmakar et al. [16] and Hu et al. [17] used ACO-based algorithms to solve VM and container placement problems respectively, with the goal of saving energy and reducing communication costs. Zhou et al. [18] and Patel et al. [19] used undirected graphs to represent the communication between containers and the network topology of PMs. The former uses graph segmentation algorithms, while the latter uses graph multi-coloring methods, ultimately reducing communication costs and energy consumption. Benefiting from the development of AI, some recent researches [20]–[23] used various reinforcement learning to solve the placement problem.

However, static resource management mainly focuses on the initial placement of VMs or containers, which usually will be conducted at the beginning of each scheduling period. Different from the existing work, our algorithm achieves the delayed placement of containers. Containers that cannot be placed in current time slot will be delayed to the next time slot (at most once) so as to avoid turning on new PM. In spite of this, the deadlines of the containers can still be satisfied through vertical scaling of the CPU resources.

2) Dynamic Resource Management: To cope with the fluctuating demand, data centers often reserve resources which may lead to resource waste. Dynamic resource management adjusts the resources of the running containers or VMs to reduce waste.

Song et al. [24] proposed a two-layer on-demand resource provisioning method based on thresholds and feedback. The feedback mechanism will dynamically adjust the threshold. Guo et al. [25] proposed a shadow routing-based approach, which achieved horizontal scaling by dynamically adjusting the number of VMs. In literature [26], [27], the authors proposed a reactive-based vertical scaling algorithm, and experiments proved that it can reduce the resource waste of the PMs in container cloud. Zhang et al. [28] believed that the reactive auto-scaling strategy does not respond quickly, so they proposed a proactive method using ARIMA to predict the number of user requests and SARSA to make the scaling decisions. Rossi et al. [29] combined vertical and horizontal scaling and proposed a dynamic Q-learning algorithm, enhanc-

ing the resource utilization compared with that using only one scaling method. For microservice application, Xu et al. [30] proposed a multi-faceted scaling approach that combine the horizontal scaling, vertical scaling, and brownout to decreases response time and connection time of services. Song et al. [31] proposed ChainsFormer to dynamically adjust resource allocation for microservice. By combining deep learning and reinforcement learning, they optimizes resource usage while maintaining high-quality of service.

Elastic scaling is more common for containers. However, the purpose of elastic scaling in the existing works is more concerned with adjusting the resources on demand, rather than saving energy directly.

B. Life Cycle Management

Life cycle management in cloud refers to changing the state of PM (*on* state, *off* state and *sleep* state) and that of VM/container (creating, destroying and migrating). As a typical life cycle management technique, VM/container consolidation always migrates a certain instances onto fewer PMs and then shuts down or sleep the idle ones.

Li et al. [32] used a coefficient of variation metric called CV to reflect the load balancing degree of server resources in container consolidation. Mongia et al. [33] proposed an adaptive threshold technique for migration, which dynamically adjusted the threshold according to the past CPU utilization of the PM. Considering the migration cost in terms of energy consumption. Khan et al. [34] argued that migration should be triggered only if migration cost can be recovered. Shen et al. [35] considered that each resource on the PM may be overloaded, so they proposed a multi-criteria decision-making method to select the container to be migrated and destination PM. Haghshenas et al. [36] first used MARL to determine which state a PM should be in through migrating the VMs according to the state of the PM, and the PM will finally be transited to this state. However, these works ignore the switching delay and power of servers.

Different from the existing work, our method involves container placement, vertical scaling and migration, which are interrelated and work cooperatively to achieve the energy-saving goal.

III. PROBLEM FORMULATION

In this section, mathematical models and problem formulation are described in detail. Table.I lists the notations and definitions to be used in this section.

A. Data Center Model

In our model, it is assumed that there are N homogeneous PMs in total. The hardware resources of each PM can be denoted as follows:

$$PM_i = (C_i, M_i, B_i), \tag{1}$$

where i denotes the index of the PM, C_i, M_i and B_i denote the total CPU, Mem and bandwidth resources owned by this PM, respectively.

In real data center, the PMs may be divided into clusters and the PMs can only communicate with those in the same cluster. PMs can communicate with each other only if they are connected in the network. We use ζ^{ij} to denote whether there is connection between PM_i and PM_j as follows:

$$\zeta^{ij} = \begin{cases} 1 & \text{if } PM_i \text{ can communicate with } PM_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Note that, a PM may be connected with more than one PM, and each connection will take up a portion of the total bandwidth of each PM.

B. User Request Model

We consider a series of batch requests submitted by users. These requests come in the form of jobs, each is composed of tasks, and each task needs a container to process. It can be denoted as follows:

$$Job_m =< D_1, D_2, .., D_k, ., D_K >, \tag{3}$$

where m is the job index, K is the total number of tasks in the job, and D_k represents the container for the kth task. Here we only consider the parallel running tasks belonging to the same job and there are no topology dependencies among them.

The resource requirement of a task will finally be transformed to the requirement of a container running this task. Therefore, we can denote the resource requirement of a container as follows:

$$D_k = (C_k(r), M_k(r), B_k(r)), \tag{4}$$

where $C_k(r)$, $M_k(r)$ and $B_k(r)$ denote the CPU, Mem and bandwidth resources required by container D_k, respectively.

We assume jobs are independent of each other with no communication, while the tasks in the same job may communicate with each other through container. We use ξ to denote the communication relationship among the containers as follows:

$$\xi_m^{kl} = \begin{cases} 1 & \text{if } D_k \text{ and } D_l \text{ in } Job_m \text{ have communication} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Fig.1 is an example describing the communication relationships among the containers belonging to the same job using undirected graph. In this figure, there are 5 vertices, representing 5 containers in the job. The vertices are connected when the two containers have communications with each other, and the weight on the edge represents the bandwidth requirement of the communication. The bandwidth of a container is the

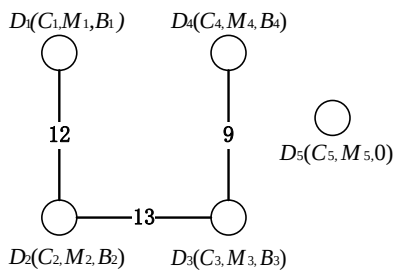


Fig. 1. An example of a job represented using undirected graph.

TABLE I
OVERVIEW OF THE USED NOTATION.*

Notation	Definition	Notation	Definition
D_k	kth container in job	L_k	Number of instructions of D_k
Job_m	mth user request	$F_k(r)$	Requested finish time of D_k
PM_i	ith PM	C_i	Total CPU resources owned by PM_i
N	Number of PM in data center	$C_k(r)$	Requested CPU resources by D_k
$N_i^t(c)$	Number of containers on PM_i at time slot t	$C_k^t(a)$	Actual CPU resource allocated to C_k at time slot t
N_{on}^t	Number of PM in on state at time slot t	M_i	Total Mem resources owned by PM_i
N_{off}^t	Number of PM in off state at time slot t	$M_k(r)$	Requested Mem resources by D_k
$N_{on\to off}^t$	Number of PM in $on \to off$ transition state at time slot t	$M_k^t(a)$	Actual Mem resource allocated to C_k at time slot t
$N_{off\to on}^t$	Number of PM in $off \to on$ transition state at time slot t	B_i	Total bandwidth resources owned by PM_i
E	Total energy consumption of data center	$B_k(r)$	Requested of bandwidth resources by D_k
P_i^t	Total power of PM_i at time slot t	$B_k^t(a)$	Actual bandwidth resource allocated to C_k at time slot t
$P_i^t(cpu)$	Power of CPU in PM_i	ζ^{ij}	Communication link between PM_i and PM_j
$P_i^t(mem)$	Power of Mem in PM_i	ξ_m^{kl}	Communication between D_k and D_l
$P_i^t(net)$	Power of bandwidth in PM_i	DL_m	Deadline for Job_m
$P_i^t(d)$	Dynamic power of PM_i	T	Total number of time slots
$P_i^t(s)$	Static power of PM_i	ι	Time of each time slot
$P_{on\to off}$	Transition power for turn off	$\iota_{on\to off}$	Transition delay for turn off
$P_{off\to on}$	Transition power for turn on	$\iota_{off\to on}$	Transition delay for turn on

* i and j are index for PM; k and l are index for container; m is index for job; t is index for time slot; r and a denote the resources requested and allocated for container, respectively.

sum of the weights of all the edges connected to it. Note that, the bandwidth utilization of a PMs depends on container placement. For two containers placed on the same PM, the communications between them will not consume any bandwidth of this PM. The two containers with communications between each other can be placed on different PMs if the two PMs are connected with enough bandwidth.

Considering the time characteristics of the container such as start time and finish time, a container D_k can be represented by the tuple as follows:

$$D_k = (C_k(r), M_k(r), B_k(r), S_k(r), F_k(r), L_k). \quad (6)$$

In this tuple, $S_k(r)$ denotes the default start time of container k, which equals to the arrival time of the job. It can be adjusted through delayed running when the resource requirement of this container can not be met in current time slot. L_k denotes the number of instructions to be executed by this container, which can be calculated in advance [20], [37]. $F_k(r)$ denotes the expected finish time of the container, which can be changed through vertical scaling of its CPU resource. The execution time of a container can be estimated given the number of instructions and the average CPU speed (instructions per second) [20]. Each user request is a job composed of tasks with different resource requirements, but having the same deadline DL_m. The speed of the containers for these tasks can be scaled separately as long as the finish times of these containers do not exceed DL_m.

C. Power Consumption Model

In general, a PM will transit between three states: *on* state, *off* state and *sleep* state. The power consumption of the PM in *off* state is lower than that in *sleep* state, but taking more time to wake up. Most studies choose the *sleep* state (rather than *off* state) as main energy-saving state to trade off between energy saving and QoS (Quality of Service). However, it proved that *off* state can save more energy in our work, meanwhile the delay of turning up can be compensated by the vertical scaling (more detail in Section 5.3.3). Therefore, we consider the following four states for PM: *on* state, *off* state, *on→off* transition state and *off→on* transition state. Note that, the server power is almost 0 when turned off, and the part of the energy consumed by the server after turned on within current time slot can not be ignored. Thus, the total energy consumption of data center can be represented as follows:

$$\begin{aligned} E = \sum_{t=1}^{T} \Bigg(& \sum_{i=1}^{N_{on}^t} P_i^t * \iota + \sum_{i=1}^{N_{on\to off}^t} P_{on\to off} * \iota_{on\to off} \\ & + \sum_{i=1}^{N_{off\to on}^t} P_{off\to on} * \iota_{off\to on} \\ & + \sum_{i=1}^{N_{off\to on}^t} P_i^t * (\iota - \iota_{off\to on}) \Bigg), \end{aligned} \quad (7)$$

where T and N are the total number of time slots and PMs in the data center, respectively. P_i^t is the power of PM_i in the tth time slot, and ι is the size of a time slot. N_{on}^t, N_{off}^t, $N_{on\to off}^t$ and $N_{off\to on}^t$ are the number of physical machines in the above four states in time slot t, respectively. $P_{on\to off}$ and $\iota_{on\to off}$ denote the transition power and delay when turning off a PM, while $P_{off\to on}$ and $\iota_{off\to on}$ denote the transition power and delay when turning on a PM. We assume that the transition delay of a PM is smaller than a time slot, and it is conducted at the beginning of each time slot so that the transition will not cross two adjacent time slots.

For a running PM, its power consumption consists of dynamic power and static power, denoted as $P_i^t(d)$ and $P_i^t(s)$, respectively. Thus, we have:

$$P_i^t = P_i^t(d) + P_i^t(s). \quad (8)$$

The dynamic power depends on the utilization of hardware components such as CPU, Mem and network, which account for about 66% of the total PM power [38]. As in literature [39], it can be modeled as follows:

$$P_i^t(d) = P_i^t(cpu) + P_i^t(mem) + P_i^t(net), \quad (9)$$

where $P_i^t(cpu)$, $P_i^t(mem)$ and $P_i^t(net)$ are the power of CPU,

Mem and bandwidth, respectively. P^t_{cpu} can be represented as a quadratic model, while P^t_{mem} and P^t_{net} can be represented as linear models [40], so we have:

$$\begin{cases} P^t_i(cpu) = \alpha_1 \mu^t_i(cpu) + \alpha_2 (\mu^t_i(cpu))^2 \\ P^t_i(mem) = \beta \mu^t_i(mem) \\ P^t_i(net) = \gamma \mu^t_i(net) \end{cases} \tag{10}$$

where $\mu^t_i(cpu)$, $\mu^t_i(mem)$ and $\mu^t_i(net)$ denote the utilization of CPU, Mem and bandwidth of PM_i, respectively. α_1, α_2, β and γ are the weights of the models.

D. Problem Formulation

In this paper, we try to minimize the total energy consumption of servers (denoted by E, which is calculated by equation(7)) , the number of container deadline violation (denoted by CDV, which is obtained by statistics) and rejection (denoted by CR, which is obtained by statistics) through elastic scaling of the resources of the containers that execute the tasks of user requests. Based on the models described above, our problem can be formulated as follows:

$$F(f_1, f_2, f_3) = \begin{cases} min\ f_1 = E \\ min\ f_2 = CDV \\ min\ f_2 = CR \end{cases} \tag{11}$$

Subject to:

$$N^t_{on} + N^t_{off} + N^t_{on \to off} + N^t_{off \to on} = N \tag{12}$$

$$C_i \geq \sum_{k=1}^{N^t_i(c)} C^t_k(a), \forall i \in \{1, ..., N\} \tag{13}$$

$$M_i \geq \sum_{k=1}^{N^t_i(c)} M^t_k(a), \forall i \in \{1, ..., N\} \tag{14}$$

$$B_i \geq \sum_{k=1}^{N^t_i(c)} B^t_k(a), \forall i \in \{i, ..., N\} \tag{15}$$

$$\zeta^{ij} = 1 \text{ if } D_k \in PM_i \ \&\& \ D_l \in PM_j \ \&\& \ \xi^{kl}_m = 1 \tag{16}$$

The constraint (12) means that the sum of the PMs in different states should be equal to the total number of PMs. The constraint (13)∼(15) means that the resources of all the containers in the same PM should not exceed the capacity of this PM. We use $C^t_k(a)$, $M^t_k(a)$ and $B^t_k(a)$ to denote the actual CPU, Mem and bandwidth resources allocated to the container in time slot t. The constraint (16) ensures that two communicating containers are either on the same PM ($i = j$) or two PMs ($i \neq j$) with connection.

Our scheduling is made in the granularity of time slot, and we determine:

- which containers should be placed onto which PMs;
- how many CPUs should be actually allocated to each containers in each time slot;
- when to trigger the migration and which containers should be migrated onto which PMs.

IV. ALGORITHM DESIGN

The architecture of our scheduling system consists of three parts: container placement, vertical scaling and migration, as shown in Fig. 2. Users submit their requests in the form of jobs, which will first be translated into the containers to run the tasks in each job, and then the connected ones will be bound into placement unit through *Job Processing Module*. The *Placement Module* will determine which PMs the placement unit should be deployed onto. When the containers are running on the PMs, their CPU resources can be scaled dynamically by multi-agents in cooperation. To achieve energy-saving goal, the *PM State Management Module* will collect the states of all the PMs and the containers, so as to determine whether to turn on or off a PM. The information will also be used by *Container Migration Module* to determine which containers should be migrated onto which PM to save energy.

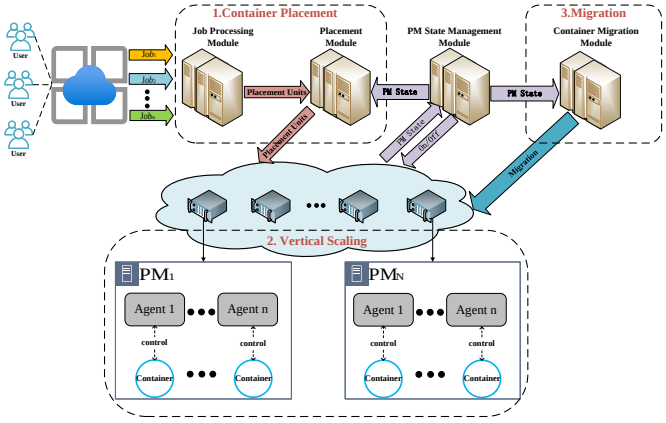


Fig. 2. The architecture of our scheduling system.

The algorithm execution process is shown in Fig.3. The container placement, scaling and migration algorithms are the three steps in our energy saving scheduling, and each makes different decisions independently. The decisions are made at the beginning of each time slot for placement and scaling, while at the end of each time slot for migration. The details of the algorithms for these three parts will be given in the following.

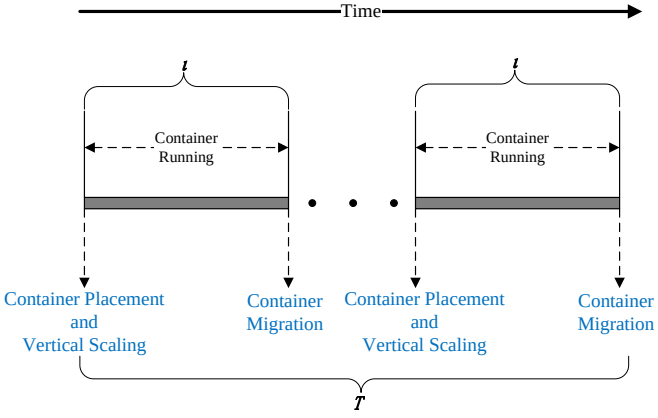


Fig. 3. Algorithm execution process

A. Container Placement

To finish a job, we need to allocate resources for each task in the job while considering the communications among different tasks, which can be processed through *Job Processing Module* and *Container Placement Module*, respectively.

1) Job Processing Module: When user requests arrive, the *Job Processing Module* will first translate each task in one job into the resource requirements of each container. Considering the communications between the containers, we preferentially place them on the same PM if possible. Therefore, the containers with communication with each other will first be bound together as a placement unit, which will then be deployed onto PMs by *Placement Module*. If a placement unit is too large to be placed directly onto any running PM, the containers in this units will be split and placed separately.

Algorithm 1: Job Processing

Input : $jobs$: the set of user requests arriving in current time slot.

Output: $placeUnits$: the set of placement units.

```
1  placeUnits ← ∅;
2  foreach job ∈ jobs do
3      jobGraph ← CreateGraph(job);
4      conGraphs ← GetConnectedGraph(jobGraph);
5      foreach connectGraph in conGraphs do
6          placeUnit ← ∅ ;
7          foreach container in connectGraph do
8              Add container into placeUnit;
9          Add placeUnit into placeUnits;
10 return placeUnits;
```

Algorithm 1 describes the process of the *Job Processing Module*. In this algorithm, the initialization is made in ($Line1$). Each job can be represented by an undirected graph, so we can obtain all the connected sub-graphs from user requests ($Line3 \sim Line4$). The containers in one connected sub-graph will then be bound together into a placement unit ($Line6 \sim Line8$). All the placement units will be added into a set called *placeUnits* as the final output of this algorithm ($Line9 \sim Line10$).

2) Placement Module: As given in equations (9) and (10), the power consumption of a server is in quadratic relation with its CPU utilization. When more containers are placed on a PM, the total energy consumption will increase dramatically. To reduce the sharply increase of energy consumption, we can lower down the CPU utilization through placing fewer containers on the PM. However, it also leads to an increase of static energy because we need more PMs to run these containers.

Therefore, we propose a *dynamic threshold* method to trade off the above two situations. The threshold divides the utilization of the CPU resource of a PM into two parts:

- *Static CPU Resource*: It refers to the available CPU resources that can be allocated to the containers at placement time but not exceeding the utilization threshold.
- *Dynamic CPU Resource*: It refers to the available CPU resources not yet allocated to the containers at placement time, but can be scaled at run time.

As in literature [18], our threshold of CPU utilization will be adjusted dynamically in range [80%, 100%]. The threshold will increase when there are PMs turned on, otherwise it will decrease with time but should not below 80%.

In order to make full use of multi-dimensional resources when placing containers on the PMs, we propose a *resource balancing* method to balance the utilization of different resources of a PM using balancing degree BD_i^t, which can be calculated as follows:

$$BD_i^t = \frac{\mu_i^t(mem)}{\mu_i^t(cpu)}, \tag{17}$$

where $\mu_i^t(mem)$ and $\mu_i^t(cpu)$ are the Mem and CPU utilization of PM_i in time slot t, respectively. The closer the BD is to 1, the more balancing this PM is. Therefore, we always select the PM with best BD after placing the container on it.

Since the number of CPU cores can be scaled dynamically to speed up the running of the container, its start time can be delayed without violating deadline. That is, the *delayed running* can be compensated through vertical scaling.

Algorithm 2 shows the details of container placement, which consists of three key methods such as *dynamic threshold*, *resource balancing* and *delayed running*.

1. The threshold θ is dynamically adjusted so as to control the maximum available resources that can be allocated to the containers at placement time. It decreases (the decrement is denoted as $\Delta\theta_1$) slowly with time but should not below the lower bound 80% ($Line1$). It will increase (the increment is denoted as $\Delta\theta_2$) when there is any PM turned on so as to accommodate more containers, but should not exceed the upper bound 100% ($Line8 \sim Line9, Line16 \sim Line17, Line28 \sim Line29$).
2. In each time slot, we try to allocate the placement units onto the PMs in previous time slot (denoted as $delayedPlaceUnits$) and that in current time slot (denoted as $placeUnits$), as shown in ($Line3 \sim Line21$) and ($Line22 \sim Line31$), respectively. We select a PM from all the servers for a placement unit or a container. If the selected PM is active, make the placement; or turn on a new PM and make placement. Note that, in the second case, the placement unit or container will actually be started in the next time slot considering the transition delay of turning on a new PM.
   When there are not enough remaining resources in all the PMs, there are ***two cases:***
   - For $placeUnit \in delayedPlaceUnits$, the placement of all the containers in $placeUnit$ will be cancelled. That is, the user request corresponding to $placeUnit$ will be rejected, and the number of rejected containers will be counted ($Line19 \sim Line20$).
   - For $placeUnit \in placeUnits$, the $placeUnit$ is added to $delayedPlaceUnits$ to be allocated in the next time slot ($Line31$), the deadlines of which can be compensated through vertical scaling of CPU.

Algorithm 2: Container Placement

Input : θ: the dynamic threshold;
$placeUnits$: the set of placement units;
$delayedPlaceUnits$: the set of placement units that was delayed in previous time slot.

Output: θ: the dynamic threshold;
$rejectNum$: the number of containers that could not be placed;
$delayedPlaceUnits$: the set of placement units that can not be placed in current time slot.

if $\theta - \Delta\theta_1 \geq 80\%$ **then** $\theta = \theta - \Delta\theta_1$;
$rejectNum \leftarrow 0$;
foreach *placeUnit* $\in$ *delayedPlaceUnits* **do**
  $PM \leftarrow PMSelection(placeUnit, \theta)$;
  if *PM* $\neq$ *null* && *PM* $\in$ *activePMs* **then**
    Place all containers in $placeUnit$ onto PM;
  else if *PM* $\neq$ *null* && *PM* $\notin$ *activePMs* **then**
    Turn on a PM and place all containers in $placeUnit$ onto PM;
    if $\theta + \Delta\theta_2 \leq 100\%$ **then** $\theta \leftarrow \theta + \Delta\theta_2$;
  else
    foreach *container* $\in$ *placeUnit* **do**
      $PM \leftarrow PMSelection(container, \theta)$;
      if *PM* $\neq$ *null* && *PM* $\in$ *activePMs* **then**
        Place $container$ in PM;
      else if *PM* $\neq$ *null* && *PM* $\notin$ *activePMs* **then**
        Turn on a PM and place$container$ onto PM;
        if $\theta + \Delta\theta_2 \leq 100\%$ **then** $\theta \leftarrow \theta + \Delta\theta_2$;
      else
        Cancel the placement of all containers in $placeUnit$;
        rejectNum $\leftarrow$ *rejectNum* + |*placeUnit*|;
        break;
$delayedPlaceUnits \leftarrow \varnothing$;
foreach *placeUnit* $\in$ *placeUnits* **do**
  $PM \leftarrow PMSelection(placeUnit, \theta)$;
  if *PM* $\neq$ *null* && *PM* $\in$ *activePMs* **then**
    Place all containers in $placeUnit$ onto PM;
  else if *PM* $\neq$ *null* && *PM* $\notin$ *activePMs* **then**
    Turn on the PM and place all containers in $placeUnit$ onto PM;
    if $\theta + \Delta\theta_2 \leq 100\%$ **then** $\theta \leftarrow \theta + \Delta\theta_2$;
  else
    Add $placeUnit$ to $delayedPlaceUnits$;
return θ, $rejectNum$, $delayedPlaceUnits$;

Note that, *PMSelection* will always select a PM for all the containers in a placement unit or a single container based on the balancing degree as mentioned above.

3. Finally, we output the dynamic threshold, the total energy consumption of all the servers in current time slot, the updated number of the rejected containers and the set of delayed placement units to be allocated in the next time slot.

B. Vertical Scaling

When all containers have been placed on the PMs, we can scale the CPU resources for each container for energy saving. If we assign an agent to control the scaling of CPU resources for each container, there can be one or more agents running on a PM to allocate the remaining resources in a sharing way. Thus, the whole scheduling process can be regarded as a decentralized partially observable Markov decision progress (Dec-POMDP) [41]. These agents need to observe the environment and perform actions, which will in turn change the environment. The changes of the environment and observation can be established into transition models based on Markov decision process and partial observation models, respectively.

The Dec-POMDP can be defined formally as a tuple $\mathcal{U} = \{\mathbb{D}, \mathbb{S}, \mathbb{O}, \mathbb{A}, T, R, n, \gamma\}$, where:

- $\mathbb{D}$ is the set of n agents.
- $\mathbb{S} = \{s_1, ..., s_t, ...\}$ is a (finite) set of environment states. s_t denotes the environment state in time slot t.
- $\mathbb{O} = \times_{i \in \mathbb{D}} \mathbb{O}_i$ is the set of joint observations for all agents. Here, $\mathbb{O}_i$ is the set of observations available to agent i.
- $\mathbb{A} = \times_{i \in \mathbb{D}} \mathbb{A}_i$ is the set of joint actions for all agents, and $\mathbb{A}_i$ is the set of actions available to agent i.
- $T : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{S}$ is the state transition function. $T(s_{t+1}|s_t, a^t)$ denotes transition to a new state s_{t+1} after taking action a^t in state s_t.
- $R : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R}$ is the reward function. It maps the state and joint action to a real number, meaning that a reward value can be obtained after taking the joint action under a certain environment state.
- $\gamma \in [0, 1)$ is a discount factor used to calculate the cumulative reward $R_t = \sum_{j=1}^{t} \gamma^{j-1} r_j$.

In each time slot t, each agent will observe the environment s_t and perform a action a^t based on a specific policy π, leading to one joint observations $o_{tot}^t =< o_1^t, ..., o_n^t >$ and one joint action $a_{tot}^t =< a_1^t, ..., a_n^t >$. How this action influences the environment is described by the transition function T. The next state of the environment is only determined by the current state and the action to be taken. is used to specify the goal of the agents. The rewards represent the benefits that can be achieved by transiting the state from s_t to s_{t+1}, which can is used to specify the goal of the agents. The objective of RL is to maximize the cumulative reward R_t by optimizing policy. Some RL-based approach is based the action value function called Q function to estimate the expected cumulative reward of state with action under policy π, which can be updated as:

$$Q(s_t, a^t) \leftarrow Q(s_t, a^t) + \alpha[R(s_t, a^t) + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a^t)] \tag{18}$$

where $\alpha \in (0, 1]$ is the learning rate.

Specifically, in our scenario, the environment is defined as a PM with containers running onside, each controlled by an agent. The agents on different PMs are independent of each

other in the data center. Let o_i^t denote the observations of agent i in time slot t, it can be denoted as follows:

$$o_i^t = [CPU, CPU_{avail}, L_i^t, C_i(r), L_1^t, C_1(r), ..., L_{i-1}^t, C_{i-1}(r), L_{i+1}^t, C_{i+1}(r), .., L_n^t, C_n(r)] \quad (19)$$

where CPU and CPU_{avail} are respectively the total CPU resources and available CPU resources of the PM running the container. L_i is the number of remaining instructions to be executed and C_i is the initial requested CPU resources for agent i, and n is the total number of agents on this PM.

In each PM, the available action set $\mathbb{A}$ for all agents may change in each scheduling, which depends on two factors: 1) the number of the allocated CPUs that exceed the initial demand of each container, which affects the decreasing action of the CPU cores for each agent. 2) the total number of the unallocated CPUs in the PM, which affects the actions to increase CPU cores for all agents. We use a $\epsilon-greedy$ method $\pi(o, a)$ to control the action selection process of an agent, as shown in the following:

$$\pi(o_i^t) = \begin{cases} \underset{a}{argmax}\, Q(o_i^t, a) & \text{if } \delta > \epsilon \\ random\ action\ from\ \mathbb{A}_i & \text{otherwise} \end{cases} \quad (20)$$

where δ is a uniform random number that can be used to balance the exploration and exploitation. $\epsilon \in [0, 1)$ is the probability to choose a random action for agent.

For a running container, its CPU resources can be increased or decreased dynamically, making it hard to judge whether the scaling operations save energy or not. The energy saved can be counted only when all the containers onside have finished their task and the PM is immediately shut down after that. In our scheduling system, the reward is designed considering the amount of energy saved and deadline violations as follows:

$$r = \begin{cases} V\Delta E & \text{if all containers on PM are finished \& } \Delta E > 0 \\ \Delta E & \text{if all containers on PM are finished \& } \Delta E \leq 0 \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

where $\Delta E = E_{origin} - E_{new}$. E_{origin} is the energy consumption of the PM to run all the containers from beginning to ending with no energy-saving operations, and E_{new} is the energy consumption with elastic scaling. V is the proportion of containers that do not violate their deadlines. When all the containers on the PM are finished and $\Delta E > 0$, we get a positive reward. The reward is larger if the proportion of deadline violations is low. Else when $\Delta E < 0$, we get a negative reward, which will guide the adjusting of the parameters for energy saving. When there is any container running on the PM, we can not count the energy saved, so the reward is 0 in such a time slot.

The vertical scaling of all the containers in a PM is controlled based on QMIX [10]. As shown in Fig. 4, the working principle of QMIX is to use the *Agent Networks* and *Mixing Network* to approximate the action-value and joint action-value function. In each time slot t, each agent has its own *Agent Network* that determines the action to be taken based on the observation o_i^t and the action a_i^{t-1}. All the agents are running cooperatively on a PM to achieve the energy-saving goal, and we use a *Mixing Network* to evaluate the actions of these agents.

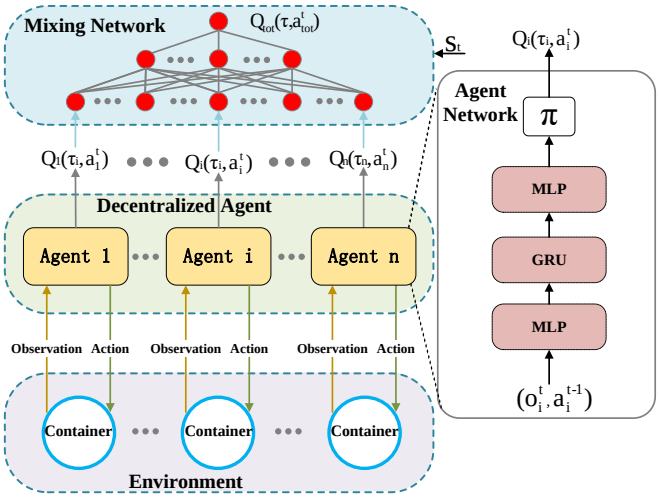


Fig. 4. The structure of QMIX.

Let $Q_i(\tau_i, a_i^t)$ denote the action-value function of agent i, where τ_i is action-observation history for agent i. The joint action-value function is denoted as $Q_{tot}(\tau, a_{tot}^t)$, where τ is the joint action-observation history. In QMIX, each network (including *Mixing network* and *Agent Network*) has two sets of parameters denoted as Θ and Θ', respectively. The network using Θ parameter is called *eval network*, which is used to calculate the Q value of the actually performed action at the current state. The network using Θ' parameter is called *target network*, which is used to calculate the Q value of the best action at the next state. It can also be used to calculate the real Q value (denoted as y_{tot}) in conjunction with reward to train the *eval network*. Therefore, *Agent Network* can further be classified into *eval Agent Network* and *target Agent Network*, and *Mixing Network* can further be classified into *eval Mixing Network* and *target Mixing Network*.

Algorithm 3 gives the details of our QMIX. The replay buffer and environment are initialized ($Line1, Line3$). Note that, the replay buffer is used to save the history record $(o_{tot}^t, o_{tot}^{t+1}, s_t, s_{t+1}, a_{tot}^t, r)$ during exploration ($Line15$). In fact, *target network* and *eval network* share the same network structure, but the parameter Θ' of *target network* will be synchronized with the parameter Θ of *eval network* every 200 episodes ($Line4 \sim Line5$). At the beginning of each time slot, each agent will get an observation and take an action to determine how many CPU cores the container should get in the current time slot ($Line8 \sim Line11$). Based on the joint observations and actions, the environment will be changed and the reward will be calculated ($Line12 \sim Line14$). When the number of records in the buffer exceeds $BatchSize$, randomly select a batch of records from the buffer to train the *eval Network* through gradient descend method ($Line17 \sim Line24$) based on the outputs from *eval Agent Networks*, *target Agent Networks*, *eval Mixing Networks* and *target Mixing Networks* ($Line19 \sim Line22$).

Algorithm 3: QMIX Algorithm

Initialize replay buffer *buffer*;
for *episode* $\leftarrow$ *1* **to** *M* **do**
  Initialize *environment* s;
  if *episode % 200 == 0* **then**
    Update target network based on eval network;
  $t \leftarrow 1$;
  while *the PM is active in time slot t* **do**
    $s_t \leftarrow$ Get environment state;
    foreach $agent_i \in PM$ **do**
      $o_i^t \leftarrow$ Observe current state;
      $a_i^t \leftarrow$ Select action by $\pi(o_i^t, a_i^{t-1})$;
    $o_{tot}^t \leftarrow [o_1^t, ..., o_n^t]$;
    $a_{tot}^t \leftarrow [a_1^t, ..., a_n^t]$;
    Execute a_{tot}^t to get new state s_{t+1} and calculate reward r;
    Store $(o_{tot}^t, o_{tot}^{t+1}, s_t, s_{t+1}, a_{tot}^t, r)$ to *buffer*;
    $t \leftarrow t + 1$;
  if *buffer.size* $>$ *batchSize* **then**
    Randomly extract a batch from *buffer*;
    Calculate each $Q_i(\tau_i, a_i; \Theta_i)$ by inputting (o_i^t, a_i^{t-1}) into *eval Agent Network*;
    Calculate each $\underset{a_i'}{max}\, Q_i(\tau_i', a_i'; \Theta_i')$ by inputting (o_i^t, a_i^{t-1}) into *target Agent Network*;
    Calculate $Q_{tot}(\tau, a_{tot}, s; \Theta)$ by inputting $< Q_1(\tau_1, a_1; \Theta_1), ..., Q_n(\tau_n, a_n; \Theta_n), s >$ into *eval Mixing Network*;
    Calculate $Q_{tot}(\tau', a_{tot}', s'; \Theta')$ by inputting $< \underset{a_i'}{max}\, Q_1(\tau_1', a_1'; \Theta_1'), ..., \underset{a_n'}{max}\, Q_n(\tau_n', a_n'; \Theta_n'), s' >$ into *target Mixing Network*;
    Set: $y^{tot} = r + \gamma \underset{a_{tot}'}{max}\, Q_{tot}(\tau', a_{tot}', s'; \Theta')$;
    Perform a gradient descend step on:
    $L(\Theta) = \sum_{i=1}^{b}[(y_i^{tot} - Q_{tot}(\tau, a_{tot}, s; \Theta))^2]$;

Algorithm 4: Container Migration

Input : θ_{over}: the dynamic threshold for overloaded;
θ_{under}: the dynamic threshold for underloaded;
Output: E: the energy consumption of all the servers;

1 **foreach** $PM_i \in$ *activePMs* **do**
2 $containerSet \leftarrow$ all containers in PM_i;
3 **while** $\mu_i(cpu) \geq \theta_{over}$&&$containerSet \neq \varnothing$ **do**
4 $container \leftarrow$*ContainerSelection*$(containerSet)$;
5 $PM_{des} \leftarrow DesPMSelection(container)$;
6 **if** $PM_{des} \neq$ *null* **then**
7 Migrate the $container$ onto PM_{des};
8 $containerSet \leftarrow containerSet \setminus \{container\}$;
9 **if** $\mu_i(cpu) \leq \theta_{under}$ **then**
10 **foreach** *container* $\in PM_i$ **do**
11 $PM_{des} \leftarrow DesPMSelection(container)$;
12 **if** $PM_{des} \neq$ *null* **then**
13 Migrate the $container$ onto PM_{des};
14 **else**
15 Cancel all the migrations of the containers on PM_i;
16 break;
17 **if** PM_i *is idle* **then**
18 Turn off PM_i;
19 $E \leftarrow$ CalEnergy();
20 **return** E

C. Migration

To further reduce the energy consumption of PMs, we try to consolidate the containers onto fewer PMs through migration, as shown in Algorithm 4. Migration is always triggered at the end of each time slot when there is any PM overloaded (CPU utilization exceeds the threshold θ_{over}) or underloaded (CPU utilization below the threshold θ_{under}). *For overloaded PM*, we always select the container from the PM with best balance degree BD after migrating the container ($Line4$). All candidate destination PMs should meet two conditions: 1) it is not overloaded or underloaded; 2) it will not be overloaded after migration. Then, we select the destination PM PM_{des} with best BD after migrating the container onto it ($Line5, Line11$). The container migration from the overloaded PM continues until its CPU utilization is no longer higher than θ_{over} ($Line3 \sim Line8$).

For underloaded PM, we try to migrate all the containers onside ($Line9 \sim Line16$) until it is idle, and then turn off the PM to save energy ($Line17 \sim Line18$). If there is any container that can't be migrated, we cancel all the migration decisions for this underloaded PM. Note that, the migration cost can't be ignored, which usually causes an increase of 10% running time for the remaining instructions to be executed by this container using the initially requested CPU resources [34].

V. PERFORMANCE EVALUATION

A. Experiment Setup

In our simulation, the data is generated based on the analysis of real data sets (Alibaba 2017&2018 [42]) and the literature of data statistics [35], [43], as shown in Table.II. In our scheduling, the size of each time slot is 30 seconds. We assume all the PMs are homogeneous, each with 68 logical CPU cores, 68 GB memory and bandwidth capacity is 100M. The power and delay of the homogeneous PMs in different states are listed in Table.III [44], [45]. For container placement, the threshold of CPU utilization is dynamically adjusted with decrement $\Delta\theta_1$=1% and increment $\Delta\theta_2$=5%, respectively. For container migration, the overloaded and underloaded thresholds are 80% and 30%, respectively.

TABLE II
GENERATION METHOD OF SIMULATION DATA

Data	Generation Method
Communication situation	Random graph G(n, p=0.3)
Number of jobs in each time slot	Exponential distribution (λ=0.2)
Number of tasks in job	Burr distribution (c=5.95, d=0.17)
CPU and Mem requested by container	Statistical probability distribution
Brandwidth requested by container	Zip distribution (a=2)
Container runtime	Negative binomial distribution (p=0.027, loc=167)

TABLE III
POWER AND DELAY FOR PM IN DIFFERENT STATES

State	Power(W)	Delay(s)
off	0	-
sleep	102	-
on $\rightarrow$ sleep	100	9
sleep $\rightarrow$ on	138	6
on $\rightarrow$ off	21	10
off $\rightarrow$ on	55	30

TABLE IV
SIMULATED DATA OF DIFFERENT SCALES

	Number of Jobs	Number of PMs
Scale 1	100	25
Scale 2	500	100
Scale 3	5000	500

To test the performance of different placement algorithms, we designed three scales of traces in experiment, as shown in Table.IV. In each scale, there are two traces generated based on Alibaba 2017&2018 analysis. Thus, there are totally 6 request traces named *trace1* $\sim$ *trace6*.

B. Compared Approaches

When there are new requests arriving in each time slot, new containers will be placed on the PMs to run each task of the requests. To evaluate different energy-saving methods, we first select a placement algorithm for each method. Here are the most commonly used placement methods:

- FCFS (First Come First Served) Algorithm: As a classical algorithm, it always places the containers on the PMs with sufficient resources one by one according to their arriving order.
- Balance Algorithm [13]: To fully utilize multi-dimensional resources of the PM, this algorithm always selects the PM for the container that still satisfies the balancing conditions after placement.
- Two-sided Matching Algorithm [46]: This algorithm calculates VM-to-PM and PM-to-VM satisfaction degrees according to different placement preference rules. The VMs will finally be placed on the PMs based on the two satisfaction degrees, which can easily be extended for container placement.

When all containers have been placed on the PMs, the energy-saving methods will be taken. Here are the most commonly used methods including classical and state-of-the-art algorithms:

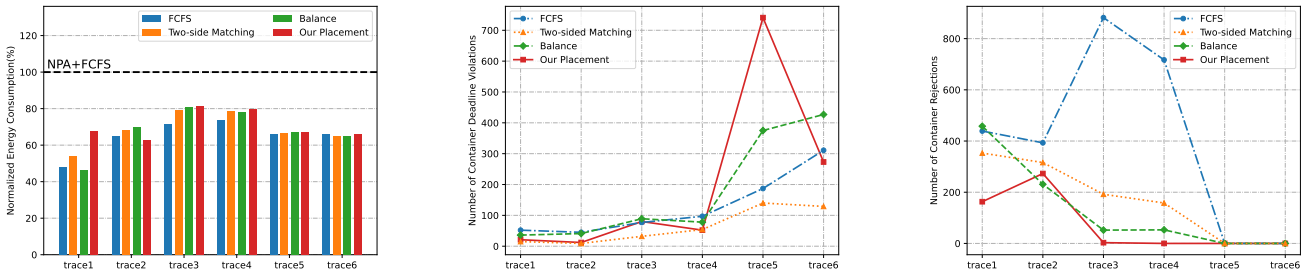
- RR-Scaling: Round Robin (RR) is a classical strategy that allocates all the remaining CPUs of the PM to a container through polling in each time slot.
- Threshold-Scaling [27]: This method always allocates the remaining CPUs to the containers averagely as long as CPU utilization of the PM does not exceed threshold.
- Q-Threshold [47]: Based Q-Learning, Q-Threshold sets the threshold of CPU utilization dynamically for each PM. If the CPU utilization on the PM does not reach the threshold, it will allocate the remaining CPUs to the containers averagely.
- CVFFC [32]: It saves energy through container migration for the overloaded and underloaded PMs using a coefficient of variation (CV) index to select the migrated container and the destination PM.
- RIAL [35]: It dynamically assigns different weights to the resources based on their utilization. Once any resource in the PM is overloaded, it will combine the Multi-Criteria Decision Making and weights to determine which VMs should be migrated to which PMs.
- MAGNETIC [36]: The approach uses MARL to determine whether a PM should be turned off. If the PM is determined to be off, all containers in the PM will execute migration.

In addition, we use the Non-power aware solution (NPA) as the baseline for normalization, which places all containers through FCFS but with no energy-saving methods taken.

C. Experiment Analysis and Result

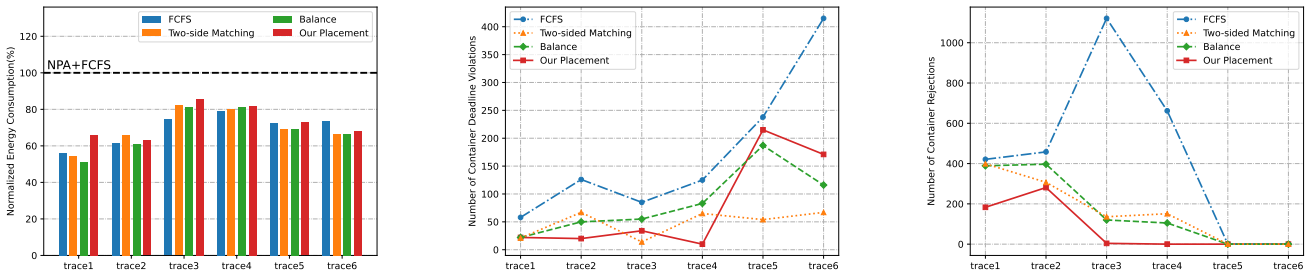
1) Placement Algorithm Selection: Fig.5, Fig.6, Fig.7 and Fig.8 show the comparisons of different placement algorithms for the energy-saving methods based on the three metrics including: normalized energy consumption against NPA, the number of deadline violations and the number of container rejections. We find that the energy consumption of each method with different placement algorithms seems almost the same, but quite different for deadline violations and rejections. For most energy-saving methods using FCFS, the number of container rejections is much larger than using other placement algorithms, because FCFS is too naive to consider the utilization of multi-dimensional resources. It can be found that our placement algorithm supports delayed placement of

This article has been accepted for publication in IEEE Transactions on Green Communications and Networking. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TGCN.2025.3552594



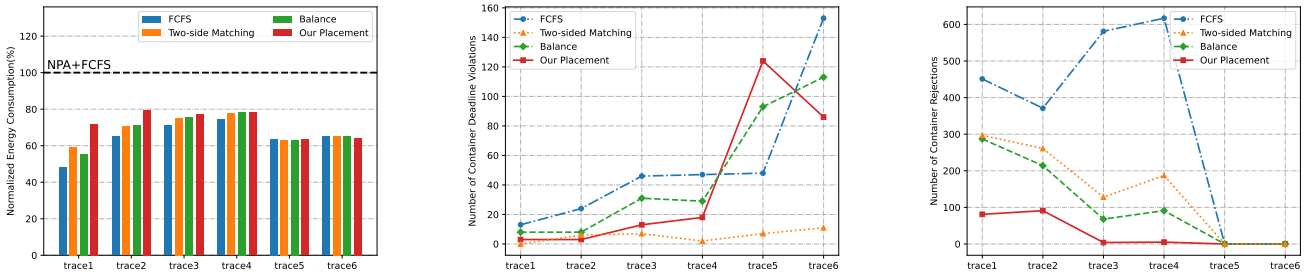
(a) Energy Consumption (b) Container Deadline Violation (c) Container Rejection

Fig. 5. Comparisons of different placement algorithms for CVFFC



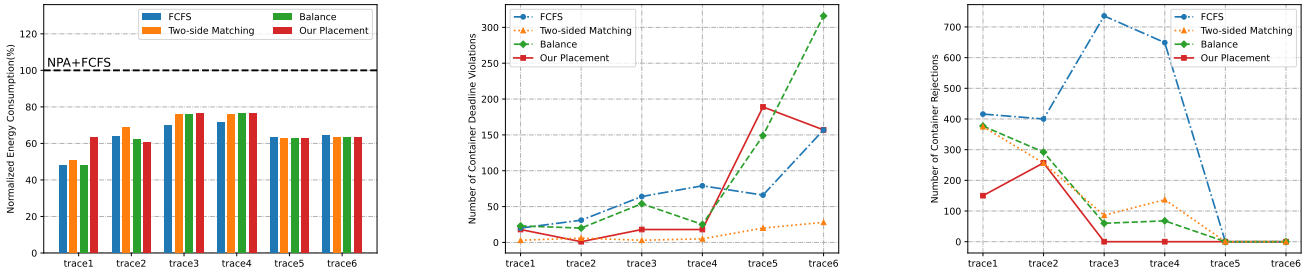
(a) Energy Consumption (b) Container Deadline Violation (c) Container Rejection

Fig. 6. Comparisons of different placement algorithms for RIAL



(a) Energy Consumption (b) Container Deadline Violation (c) Container Rejection

Fig. 7. Comparisons of different placement algorithms for RR-Scaling



(a) Energy Consumption (b) Container Deadline Violation (c) Container Rejection

Fig. 8. Comparisons of different placement algorithms for Threshold-Scaling

containers. The containers that cannot be placed in the current time slot can be delayed to the next time slot (at most once) to avoid turning on PM as much as possible. Thus, our placement method usually has lower container rejection rate in most cases. These results also show that different energy-saving methods in cooperation with these placement algorithms performs different in various metrics. Taking all the three metrics into account, we select the best placement algorithm for each energy-saving method as follows: *CVFFC+Two-Side Matching*, *RIAL+Balance*, *RR-Scaling+Two-Side Matching* and *Threshold-Scaling+Our Placement*.

2) Training Performance: The running of QMIX can be divided into two phases: training phase and evaluation phase. In the training phase, an action is selected using formula (20); in the evaluation phase, the action is selected using the trained network and the evaluation is performed every 5000 episodes. In formula (20), there is an important parameter ϵ representing the probability of randomly performing an action that can be collected as training data. It will converge to sub-optimal solutions if ϵ is too small, while the model can't converge if ϵ is too large. In view of this, we set this parameter by exploration decay. That is, initialize a large ϵ at the beginning of training, and then reduce it gradually to a small value, and keep on training using the small ϵ to until converging. Fig.9 shows the trend of reward with episode under different ϵ. Our design with $epsilon = 0.99 \rightarrow 0.1$ enables the model to converge to a bigger reward. In fact,

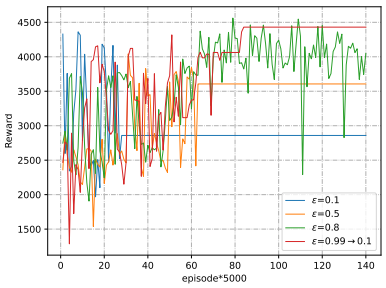


Fig. 9. The reward under different ϵ

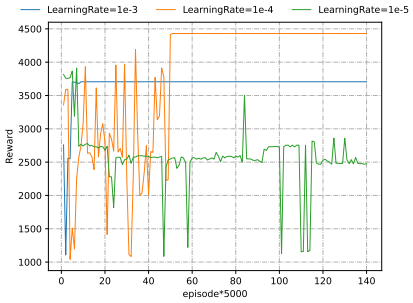


Fig. 10. The reward under different learning rates

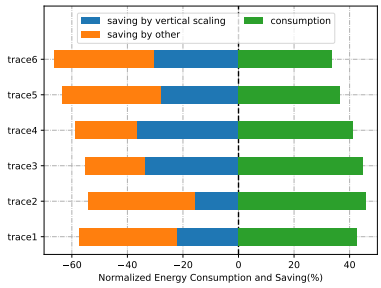
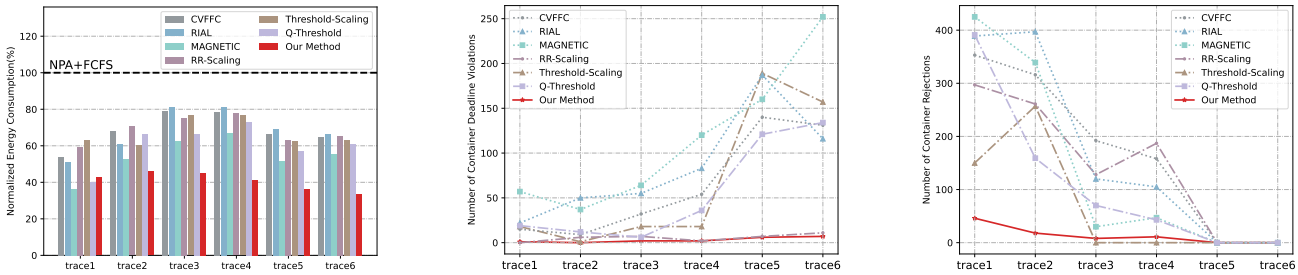


Fig. 11. The ablation experiment

TABLE V
COMPARISONS BETWEEN USING *Off* AND *Sleep*

	Off			Sleep		
	Energy Consumption(kWh)	**Deadline Violation**	**Container Rejection**	**Energy Consumption(kWh)**	**Deadline Violation**	**Container Rejection**
trace1	5681	1	46	6946	0	44
trace2	5274	0	18	6671	0	18
trace3	29903	5	8	35519	1	8
trace4	20298	1	11	24593	1	11
trace5	299532	11	0	354404	4	0
trace6	203669	3	0	258837	0	0



(a) Energy Consumption (b) Container Deadline Violation (c) Container Rejection

Fig. 12. Performance Comparison with the existing work

the learning rate of the network also affects the result. Fig.10 compares the convergence under different learning rates. The model can't converge when $LearningRate = 1e{-}5$, and it will converge when $LearningRate = 1e{-}3$ and $LearningRate = 1e{-}4$. Considering convergence and training reward, we select $LearningRate = 1e{-}4$ for our experiment.

3) Turn Off or Sleep PM: In general, turning on a PM from *off* state takes more time than waking up it from *sleep* state, which may cause deadline violation of containers. However, our method can speed up the running of containers through vertical scaling, which can compensate for the delay when turning on a PM. Table.V shows the difference between *off* and *sleep*. It can be clearly seen that using the *off* state saves more energy without incurring many more deadline violations. Therefore, we select *off* state as our energy-saving state.

4) Performance Comparison: Fig.12 (a) shows the comparisons of our method with other works in normalized power consumption against NPA. It can be found that our method is more energy-efficient in most traces. In trace1, however, our method consumes more energy than MAGNETIC and Q-Threshold. It is because the energy saved by these two algorithms is achieved at the expense of rejecting much more containers. In elastic scaling, the energy can be saved only when the idle CPUs are reasonably allocated, rather than using them all. One way for energy saving is achieved through speeding up the execution of all the containers on a PM through vertical scaling and then turn off the PM as soon as possible. To validate the effectiveness of our scaling, we conducted an ablation experiment, as shown in Fig.11. In this Figure, the *normalized saved energy* is a negative value (in blue+orange color) and the *normalized total energy* is a positive value (in green color). The blue part is the proportion of energy saved by vertical scaling while the orange part is that saved by container placement and migrations. It can be found that vertical scaling contributes the largest to energy saving in most traces (even exceeding 40% of the total energy saved).

Fig.12 (b) shows the comparisons of the number of deadline violations of the containers. Although there are additional cost for placement (e.g. delayed running) and migration (e.g. prolonged running time), vertical scaling can speed up the running of the containers to compensate for this while saving more energy. Besides saving more energy, it can be found that our method has the fewest deadline violations and achieves the lowest rejection rate of the containers in most traces, as shown in Fig.12 (b) and Fig.12 (c). In fact, the rejection rate is highly dependent on the placement algorithm. In our placement algorithm, we consider balancing multi-dimensional

TABLE VI
QUANTITATIVE ANALYSIS OF COMPARISON RESULTS

	Normalized Average Energy Consumption(%)	Normalized Average Deadline Violation(%)	Normalized Average Rejection(%)
CVFFC	68.3	0.92	13.78
RIAL	68.4	1.96	15.32
MAGNETIC	54.3	2.69	14.48
RR-scaling	68.5	0.14	11.59
Threshod-Scaling	67.1	0.67	7.26
Q-Threshold	60.7	0.86	10.98
Our Method	40.8	0.04	1.3

resources of PMs and delaying the running the containers if they can't be placed on any active PM when arriving. Besides, the tasks on the containers can be finished more quickly through vertical scaling, so that the resources of the containers can be released quickly to place more containers.

To further clarify and quantify the comparison results shown in Fig.12, we draw the Table VI which shows a quantitative analysis of the results. For each method, we average the normalized energy consumption, normalized deadline violation and normalized rejection rate against NPA under the six traces. Our method can save 27.5%, 27.6%, 13.5%, 27.7%, 26.3%, 19.9% more energy consumption than other compared method respectively, while without sacrificing service quality in terms of deadline violation and rejections.

VI. CONCLUSION

In this paper, an online scheduling scheme including container placement, vertical scaling and migration have been designed to save the total energy consumption of the servers in container cloud. To accommodate as many containers as possible using fewest PMs, we design a container placement algorithm based on dynamic threshold, resource balancing and delayed running for container placement. Benefited from the delayed running, our algorithm usually has a lower container rejection rate. Since reinforcement learning can well suit the autoscaling problems, we propose a collaborative MARL algorithm to control the vertical scaling of container, which can substantially reduce the total energy consumption. To further reduce energy consumption, we consolidate the containers onto fewer PMs through migration. Experiment results show that our method is more energy-efficient than state-of-the-art algorithms while with lower rates in deadline violations and rejections for the containers. In future work, we would like to explore how to use both vertical and horizontal scaling to achieve better performance. Furthermore, enhancing the performance of QMIX is another way for improvement.

REFERENCES

[1] M. Koot and F. Wijnhoven, "Usage impact on data center electricity needs: A system dynamic forecasting model," *Applied Energy*, vol. 291, p. 116798, 2021.

[2] S. Pelley, D. Meisner, T. F. Wenisch, and J. W. VanGilder, "Understanding and abstracting total data center power," in *Workshop on Energy-Efficient Design*, vol. 11, 2009, pp. 1–6.

[3] C. Gu, Z. Li, H. Huang, and X. Jia, "Energy efficient scheduling of servers with multi-sleep modes for cloud data center," *IEEE Transactions on Cloud Computing*, vol. 8, no. 3, pp. 833–846, 2018.

[4] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 205–216, 2009.

[5] R. Zhang, A.-m. Zhong, B. Dong, F. Tian, R. Li *et al.*, "Container-VM-PM architecture: A novel architecture for docker container placement," in *International Conference on Cloud Computing*. Springer, 2018, pp. 128–140.

[6] K. Li, Y.-m. Ji, S.-d. Liu, H.-c. Yao, H. Li, S. You, and S.-s. Shao, "Acea: A queueing model-based elastic scaling algorithm for container cluster," *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–11, 2021.

[7] C. Jiang and P. Wu, "A fine-grained horizontal scaling method for container-based cloud," *Scientific Programming*, vol. 2021, pp. 1–10, 2021.

[8] X. Chen, L. Cheng, C. Liu, Q. Liu, J. Liu, Y. Mao, and J. Murphy, "A woa-based optimization approach for task scheduling in cloud computing systems," *IEEE Systems journal*, vol. 14, no. 3, pp. 3117–3128, 2020.

[9] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Fscaler: Automatic resource scaling of containers in fog clusters using reinforcement learning," in *2020 international wireless communications and mobile computing (IWCMC)*. IEEE, 2020, pp. 1824–1829.

[10] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *International conference on machine learning*. PMLR, 2018, pp. 4295–4304.

[11] M. J. Magazine and M.-S. Chern, "A note on approximation schemes for multidimensional knapsack problems," *Mathematics of Operations Research*, vol. 9, no. 2, pp. 244–247, 1984.

[12] W. Lin, C. Liang, J. Z. Wang, and R. Buyya, "Bandwidth-aware divisible task scheduling for cloud computing," *Software: Practice and Experience*, vol. 44, no. 2, pp. 163–174, 2014.

[13] S. Azizi, M. Zandsalimi, and D. Li, "An energy-efficient algorithm for virtual machine placement optimization in cloud data centers," *Cluster Computing*, vol. 23, no. 4, pp. 3421–3434, 2020.

[14] I. El-Taani, M.-C. Boukala, and S. Bouzefrane, "Energy-aware vm placement based on intra-balanced resource allocation in data centers," in *2021 8th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2021, pp. 400–405.

[15] M. Elsakhawy and M. Bauer, "Usage trends aware vm placement in academic research computing clouds," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 688–697.

[16] K. Karmakar, R. K. Das, and S. Khatua, "An ACO-based multi-objective optimization for cooperating VM placement in cloud data center," *The Journal of Supercomputing*, vol. 78, no. 3, pp. 3093–3121, 2022.

[17] Y. Hu and Y. Lei, "A container cloud scheduling strategy based on QoS," in *The 2nd International Conference on Computing and Data Science*, 2021, pp. 1–5.

[18] L. Zhou, L. N. Bhuyan, and K. Ramakrishnan, "Goldilocks: Adaptive resource provisioning in containerized data centers," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 666–677.

[19] Y. S. Patel, A. Baheti, and R. Misra, "Interval graph multi-coloring-based resource reservation for energy-efficient containerized cloud data centers," *The Journal of Supercomputing*, vol. 77, no. 5, pp. 4484–4532, 2021.

[20] J. Yan, Y. Huang, A. Gupta, A. Gupta, C. Liu, J. Li, and L. Cheng, "Energy-aware systems for real-time job scheduling in cloud data centers: A deep reinforcement learning approach," *Computers and Electrical Engineering*, vol. 99, p. 107688, 2022.

[21] P. Song, C. Chi, K. Ji, Z. Liu, F. Zhang, S. Zhang, D. Qiu, and X. Wan, "A deep reinforcement learning-based task scheduling algorithm for energy efficiency in data centers," in *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021, pp. 1–9.

[22] S. Long, Z. Li, Y. Xing, S. Tian, D. Li, and R. Yu, "A reinforcement learning-based virtual machine placement strategy in cloud data centers,"

in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2020, pp. 223–230.

[23] D. Yi, X. Zhou, Y. Wen, and R. Tan, "Efficient compute-intensive job allocation in data centers via deep reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1474–1485, 2020.

[24] Y. Song, Y. Sun, and W. Shi, "A two-tiered on-demand resource allocation mechanism for vm-based data centers," *IEEE Transactions on Services Computing*, vol. 6, no. 1, pp. 116–129, 2011.

[25] Y. Guo, A. L. Stolyar, and A. Walid, "Online VM auto-scaling algorithms for application hosting in a cloud," *IEEE Transactions on Cloud Computing*, vol. 8, no. 3, pp. 889–898, 2018.

[26] C. H. Nicodemus, C. Boeres, and V. E. Rebello, "Managing vertical memory elasticity in containers," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2020, pp. 132–142.

[27] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Generation Computer Systems*, vol. 98, pp. 672–681, 2019.

[28] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-SARSA: A predictive container auto-scaling algorithm based on reinforcement learning," in *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 2020, pp. 489–497.

[29] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 329–338.

[30] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "Coscal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 3995–4009, 2022.

[31] C. Song, M. Xu, K. Ye, H. Wu, S. S. Gill, R. Buyya, and C. Xu, "Chainsformer: A chain latency-aware resource provisioning approach for microservices cluster," in *International Conference on Service-Oriented Computing*. Springer, 2023, pp. 197–211.

[32] Y. Li, Y. Xu, and X. Zhou, "CVFCC: CV-based framework for container consolidation in cloud data centers," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 795–796.

[33] V. Mongia and A. Sharma, "An adaptive performance aware threshold policy based on Q_n estimator in cloud data centers," *SN Computer Science*, vol. 2, no. 4, pp. 1–10, 2021.

[34] A. A. Khan, M. Zakarya, R. Buyya, R. Khan, M. Khan, and O. Rana, "An energy and performance aware consolidation technique for containerized datacenters," *IEEE Transactions on Cloud Computing*, vol. 9, no. 4, pp. 1305–1322, 2019.

[35] H. Shen and L. Chen, "A resource usage intensity aware load balancing method for virtual machine migration in cloud datacenters," *IEEE Transactions on Cloud Computing*, vol. 8, no. 01, pp. 17–31, 2020.

[36] K. Haghshenas, A. Pahlevan, M. Zapater, S. Mohammadi, and D. Atienza, "MAGNETIC: Multi-agent machine learning-based approach for energy efficient dynamic consolidation in data centers," *IEEE Transactions on Services Computing*, vol. 15, no. 01, pp. 30–44, 2022.

[37] Y. Ding, X. Qin, L. Liu, and T. Wang, "Energy efficient scheduling of virtual machines in cloud with deadline constraint," *Future Generation Computer Systems*, vol. 50, pp. 62–74, 2015.

[38] T. L. Vasques, P. Moura, and A. de Almeida, "A review on energy efficiency and demand response with focus on small and medium data centers," *Energy Efficiency*, vol. 12, no. 5, pp. 1399–1428, 2019.

[39] L. Ismail and H. Materwala, "Computing server power modeling in a data center: Survey, taxonomy, and performance evaluation," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–34, 2020.

[40] H. Zhao, J. Wang, F. Liu, Q. Wang, W. Zhang, and Q. Zheng, "Power-aware and performance-guaranteed virtual machine placement in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1385–1400, 2018.

[41] F. A. Oliehoek and C. Amato, *A concise introduction to decentralized POMDPs*. Springer, 2016.

[42] A. Inc, "Alibaba production cluster data," 2017-2018, https://github.com/alibaba/clusterdata.

[43] Z. Wu, Y. Deng, H. Feng, Y. Zhou, and G. Min, "Blender: A traffic-aware container placement for containerized data centers," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 986–989.

[44] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, V. Bahl, and R. Gupta, "Somniloquy: augmenting network interfaces to reduce pc energy usage," in *Networked Systems Design & Implementation (NSDI)*, 2009.

[45] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," *ACM SIGARCH computer architecture news*, vol. 35, no. 2, pp. 13–23, 2007.

[46] C. Zhang, Y. Wang, H. Wu, and H. Guo, "An energy-aware host resource management framework for two-tier virtualized cloud data centers," *IEEE Access*, vol. 9, pp. 3526–3544, 2020.

[47] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2018, pp. 85–92.

Yanyu shen received the B.S. degree in the Department of Computer Science and Technology in Nanchang University. And now he is a master student in School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. His research interests are in the fields of cloud computing and intelligent optimization.

Chonglin Gu received PhD degree in computer science and technology from Harbin Institute of Technology, Shenzhen in 2018. After that, he has been a postdoctoral fellow in the Chinese University of Hong Kong, Shenzhen, China. He is currently an assistant professor in the school of computer science and technology in Harbin Institute of Technology, Shenzhen. His research interests include cloud computing, especially algorithm design and system implementation.

Xin Chen received the B.S. degree in School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. And now he is a master student in School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. His research interests are in the fields of cloud computing and algorithm design.

Xiaoyu Gao received the B.S. degree in School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. And now he is a master student in School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. His research interests are in the fields of cloud computing and green data centers.

Zaixing Sun received the M.S. degree in Control Engineering from the Kunming University of Scence and Technology, Kunming, China, in 2019. Currently, He is a Ph.D. candidate in Harbin Institute of Technology, Shenzhen, China. His research interests include cloud computing, intelligent optimization and scheduling.

Hejiao Huang received her PhD degree in Coputer Science from City University of Hong Kong in 2004. She is currently a professor in Harbin Institute of Technology, Shenzhen, China, and previously was an invited professor at INRIA, France. Her research interests include network security, cloud computing security, trustworthy computing, big data security, formal methods for system design and wireless networks.