Tian WH, Xu MX, Zhou GY *et al.* Prepartition: Load balancing approach for virtual machine reservations in a cloud data center. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(4): 773-792 July 2023. DOI: 10.1007/s11390-022-1214-x

Prepartition: Load Balancing Approach for Virtual Machine Reservations in a Cloud Data Center

Wen-Hong Tian[1, 2] (田文洪), *Senior Member, CCF, Member, ACM, IEEE*
Min-Xian Xu[3, *] (徐敏贤), *Member, CCF, IEEE*, Guang-Yao Zhou[1] (周光耀)
Kui Wu[4] (吴　逵), *Senior Member, IEEE*, Cheng-Zhong Xu[5] (须成忠), *Fellow, IEEE*, and
Rajkumar Buyya[6, 1], *Fellow, IEEE*

[1] *School of Information and Software Engineering, University of Electronic Science and Technology of China Chengdu 610054, China*

[2] *Yangtze Delta Region Institute (Huzhou), University of Electronic Science and Technology of China, Huzhou 313001, China*

[3] *Institute of Advanced Computing and Digital Engineering, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China*

[4] *Department of Computer Science, University of Victoria, Victoria, BC, V8W 3P6, Canada*

[5] *State Key Laboratory of Internet of Things for Smart City, University of Macau, Macau 999078, China*

[6] *School of Computing and Information Systems, University of Melbourne, Melbourne 3010, Australia*

E-mail: tian_wenhong@uestc.edu.cn; mx.xu@siat.ac.cn; guangyao_zhou@std.uestc.edu.cn; wkui@uvic.ca; czxu@um.edu.mo rbuyya@unimelb.edu.au

Received December 10, 2020; accepted April 26, 2022.

Abstract Load balancing is vital for the efficient and long-term operation of cloud data centers. With virtualization, post (reactive) migration of virtual machines (VMs) after allocation is the traditional way for load balancing and consolidation. However, it is not easy for reactive migration to obtain predefined load balance objectives and it may interrupt services and bring instability. Therefore, we provide a new approach, called Prepartition, for load balancing. It partitions a VM request into a few sub-requests sequentially with start time, end time and capacity demands, and treats each sub-request as a regular VM request. In this way, it can proactively set a bound for each VM request on each physical machine and makes the scheduler get ready before VM migration to obtain the predefined load balancing goal, which supports the resource allocation in a fine-grained manner. Simulations with real-world trace and synthetic data show that our proposed approach with offline version (PrepartitionOff) scheduling has 10%–20% better performance than the existing load balancing baselines under several metrics, including average utilization, imbalance degree, makespan and Capacity_makespan. We also extend Prepartition to online load balancing. Evaluation results show that our proposed approach also outperforms state-of-the-art online algorithms.

Keywords cloud computing, physical machine (PM), virtual machine (VM), reservation, load balancing, Prepartition

1 Introduction

Cloud data centers have become the foundation for modern IT services, ranging from general-purpose web services to many critical applications, such as online banking and health systems. The service operator of a cloud data center always faces with a difficult trade-off between high performance and low operational cost[1, 2]. On the one hand, to maintain high-quality services, a data center is usually over-engineered to be capable of handling peak workload. Such up-bound configuration can bring high expense on

Regular Paper

This work was supported by Shenzhen Industrial Application Projects of undertaking the National Key Research and Development Program of China under Grant No. CJGJZD20210408091600002, the National Natural Science Foundation of China under Grant No. 62102408, and Shenzhen Science and Technology Program under Grant No. RCBS20210609104609044.

*Corresponding Author

maintenance and energy as well as low utilization to data centers[3]. On the other hand, to reduce cost, the data center needs to increase server utilization and shut down idle servers[4]. The key tuning knob in making the above trade-off is data center load balancing.

Due to the importance of data center load balancing, tremendous research and development have been devoted to this domain in the past decades[5]. Yet, load balancing for cloud data centers is still one of the prominent challenges that need more attention. The difficulty is compounded by several issues such as virtual machine (VM) migration, service availability, algorithm complexity, and resource utilization. The complexity in cloud data center load balancing has fostered a new industry dedicating to offer load balance services[6].

Ignoring the subtle differences in detailed implementation of load balancing, let us first have a high-level view of how cloud data centers perform resource scheduling and load balancing. The process is illustrated in Fig.1, which includes the following main steps.

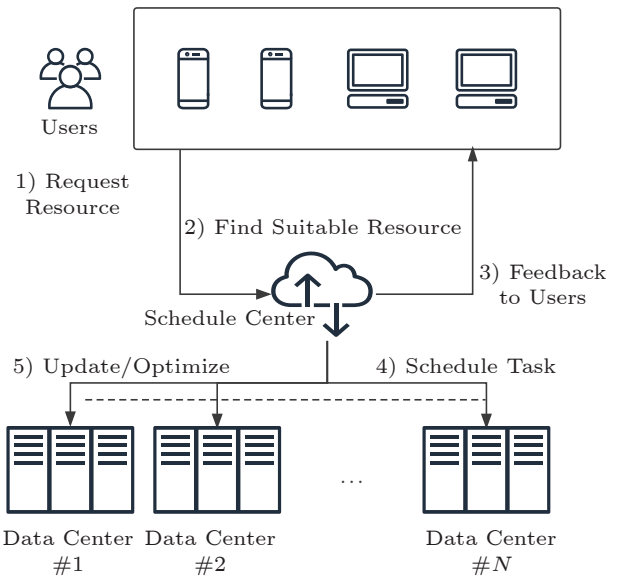

Fig.1. A high-level view on resource scheduling/load balancing in cloud data centers.

1) *Initializing Requests*. A user submits a VM request through a provider's web portal.

2) *Matching Suitable Resources*. Based on the user's features (such as geographic location, VM quantity and quality requirements), the scheduling center sends the VM request to an appropriate data center, in which the management program submits the request to a scheduling domain. In the scheduling domain, a scheduling algorithm is performed and resources are allocated to the request.

3) *Sending Feedback* (*e.g., Whether or Not the Request Has Been Satisfied*) *to Users*.

4) *Scheduling Tasks*. This step determines when a VM should run on which physical machine (PM).

5) *Optimization*. The scheduling center executes optimization in the back-end and makes decisions (e.g., VM migration) for load balancing.

In the above process, most existing work on load balancing is reactive, i.e., performing load balancing with VM migration when unbalancing or other exceptional things happen after VM deployment. Reactive migration of VMs is one of the practical methods for load balancing and traffic consolidation such as in VMWare. Nevertheless, it is well known that reactive VM migration is not easy to obtain predefined load balance objectives and may interrupt services and bring instability[5]. Our observation is that if load balancing is considered as one of the key criteria before VM allocation, we should not only reduce the frequency of (post) VM migration (thus less service interruption), but also reach a better balanced VM allocation among different physical machines (PMs).

Motivated by the above observation, we propose a new load balancing approach called Prepartition. By combining interval scheduling and lifecycles characteristics of both VMs and PMs, Prepartition handles load balancing from a different angle. Starkly different from previous approaches like in [7] and [8], it handles the VM load balancing in a more proactive way.

Fig.2 shows the illustrative example based on the above observation and motivation. At the requests submission stage, the users firstly submit their reserved VM requests, including the capacity and duration information. Based on the information, then the service provider can generate the original VM request at the requests generation stage (e.g., VM1, VM2 and VM3). Our approach focuses on the prepartition stage where the original VM requests can be partitioned into sub-requests and allocated to PMs before the VM migration stage, for instance, VM1 is partitioned into VM1-1 and VM1-2, and allocated to PM1. And finally, to further optimize VM locations, VM migration can be further applied.

As the prepartition process happens before the final requests generation stage, and it does not need to execute the jobs, the costs are rather low compared with the overall job execution. The prepartition costs

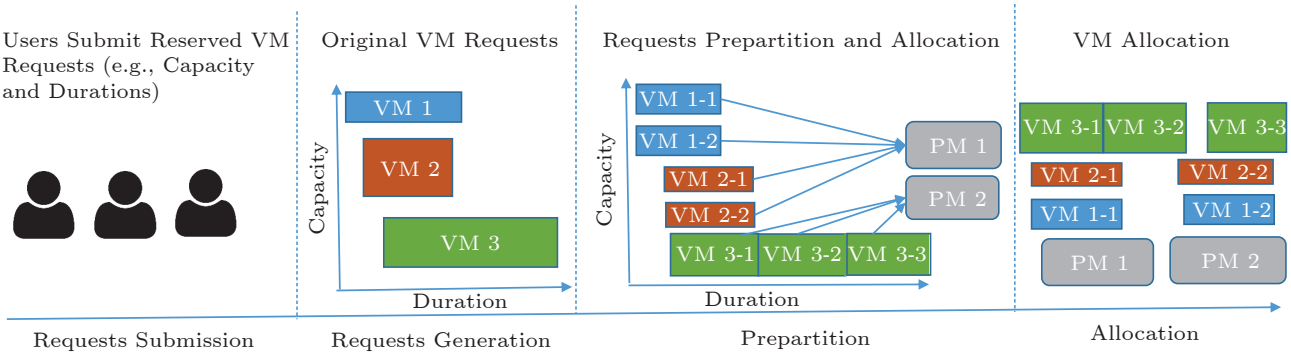

Fig.2. Illustrative example for Prepartition

will not be the bottleneck of the system if the algorithm complexity is low. The prepartition operations can be done on the master node with powerful capability in a short time (e.g., several seconds), which is much shorter compared with the execution time of jobs.

The novelty of Prepartition is that it proactively sets a process-time bound (as per Capacity_makespan defined in Section 3) by pre-partitioning each VM request and therefore helps the scheduler get ready before the VM migration to achieve the predefined load balancing goal. Pre-partitioning here means that a VM request may be partitioned into a few sub-requests sequentially with start time, end time and capacity demands, where the scheduler treats each sub-request as a regular VM request and may allocate the sub-requests to different PMs①. In this way, the scheduler can prepare in advance, without waiting for the VM migration signals as in traditional VM allocation/migration schemes. In addition, the resources can be allocated at the fine granularity and the migration costs can be reduced.

To the best of our knowledge, we are the first to introduce the concept of pre-partitioning VM requests to achieve better load balancing performance in cloud data centers. This paper has the following key contributions:

• proposing a modeling approach to schedule VM reservation with sharing capacity by combining interval scheduling and lifecycles characteristics of both VMs and PMs;

• designing novel prepartition-based algorithms for both offline and online scheduling which can prepare migration in advance and set process time bound for each VM on a PM; thus the resource allocation can be made in a more fine-grained manner;

• deriving computational complexity and quality analysis for both offline and online prepartition-based approaches;

• carrying out performance evaluation in terms of average utilization, imbalance degree, makespan, time costs as well as Capacity_makespan (a metric to represent loads, whose details will be given in Section 3) by simulating different algorithms with trace-driven and synthetic data.

The organization of the remaining paper is as follows. Section 2 presents related work on load balancing in cloud data centers, and Section 3 introduces problem formulation. Section 4 presents the prepartition-based approach in detail for both offline and online algorithms. Performance evaluations are demonstrated in Section 5. Finally, conclusions and future work are given in Section 6.

2 Related Work

As introduced in several popular surveys, resource scheduling and load balancing in cloud computing have been widely studied. Xu *et al.*[9] had a survey for the state-of-the-art VM placement algorithms. Ghomi *et al.*[10] have recently made a comprehensive survey on load balancing algorithms in cloud computing. A taxonomic survey related to load balancing in cloud is studied by Thakur and Goraya[11]. Noshy *et al.*[7] reviewed the latest optimization technology dedicated to developing live VM migration. They also emphasized a further investigation, which aims to optimize the VMs migration process. [12] dis-

①Note that in practice we need to copy data and running state information from a VM (corresponding to a sub-request) to another VM (corresponding to the next sub-request), i.e., the operations for VM migration. But since the scheduler knows the information of all sub-requests, it can prepare early so that the VM state/data transition can be finished smoothly. The implementation detail is beyond the focus of this paper.

cusses the issues and challenges associated with existing load balancing techniques. In general, approaches for VM load balancing can be categorized into two categories: online and offline. The online ones assume that only the current requests and PMs status are known, while the offline ones assume all the information is known in advance.

Online Approach for VM Loading Balancing. Song *et al.*[8] proposed a VM migration method to dynamically balance VM loads for high-level application federations. Thiruvenkadam and Kamalakkannan[13] showed a hybrid genetic VMs balancing algorithm, which aims to minimize the times of migration. Cho *et al.*[14] tried to maximize the balance of loads in cloud computing by combining ant colony with particle swarm optimization. Xu *et al.*[15] proposed iAware, which is a lightweight interference model for VM migration. iAware can capture the relationship between VM performance interference and the important factors. Zhou *et al.*[16] presented a carbon-aware online approach based on Lyapunov optimization to achieve geographical load balancing. Mathematical analysis and experiments based on realistic traces have validated the effectiveness of the proposed approach. Liu *et al.*[17] proposed a framework to characterize and optimize the trade-offs between power and performance in cloud platforms, which can improve operating profits while reducing energy consumption.

Offline Approach for VM Load Balancing. Tian *et al.*[18] presented an offline algorithm on VM allocation within the reservation mode, in which the VM information is known before placement. Derived from the ant colony optimization, Wen *et al.*[19] proposed a distributed VM load balancing strategy with the goals of utilizing resources in a balanced manager and minimizing the times of migration. By estimating resource usage, Chhabra and Singh[20] developed a VMs placement method for loading balancing according to maximum likelihood estimation for parallel and distributed applications. Bala and Chana[21] presented an approach to improving proactive load balancing by predicting multiple resource types in the cloud. Ebadfard and Babamir[22] developed a task scheduling approach derived from a particle swarm optimization algorithm and the tasks are independent and non-preemptive. Ray *et al.*[23] presented a genetic-based load balancing approach to distribute VM requests uniformly among the PMs. Deng *et al.*[24] introduced a server consolidation approach to achieve energy efficient server consolidation in a reliable and profitable manner.

Different from all the above approaches, 1) we investigate the reservation model where makespan and VM capacity are considered together for optimization instead of considering them separately; 2) our approach can be applied to both online and offline scenarios rather than a single scenario; 3) we also perform theoretical analysis for the proposed approach; 4) we evaluate comprehensive performance in terms of metrics. A qualitative comparison between our approach and others is listed in Table 1.

Table 1. Comparison of Closely Related Work

Approach	Algorithm Type	VM Type	Resource Type	Theoretical Analysis	Metric
Song *et al.*[8]	Online	Homogeneous	Single	No	Utilization
Thiruvenkadam and Kamalakkannan[13]	Online	Heterogeneous	Multiple	Yes	Utilization, service level agreement violations
Cho *et al.*[14]	Offline	Heterogeneous	Multiple	No	Utilization
Xu *et al.*[15]	Online	Heterogeneous	Multiple	No	Utilization, total cost
Zhou *et al.*[16]	Online	Heterogeneous	Single	Yes	Service level agreement violations
Liu *et al.*[17]	Online	Heterogeneous	Single	Yes	Total cost
Tian *et al.*[18]	Offline	Heterogeneous	Multiple	Yes	Imbalance degree, makespan
Wen *et al.*[19]	Offline	Heterogeneous	Multiple	No	Service level agreement violations
Chhabra and Singh[20]	Online	Heterogeneous	Single	No	Utilization, imbalance degree
Bala and Chana[21]	Offline	Heterogeneous	Multiple	No	Utilization, imbalance degree, service level agreement violations
Ebadfard and Babamir[22]	Offline	Homogeneous	Multiple	Yes	Utilization, makespan
Ray *et al.*[23]	Offline	Heterogeneous	Multiple	No	Imbalance degree
Deng *et al.*[24]	Offline	Homogeneous	Multiple	No	Total cost, service level agreement violations
Our approach (Prepartition)	Online, offline	Heterogeneous	Multiple	Yes	Utilization, imbalance degree, makespan, total cost, Capacity_makespan

3 Problem Description and Formulation

VMs reservation is considered as that users submit their VM requests by specifying required capacity and duration. The VM allocations are modeled as a fixed processing time problem based on a modified interval scheduling problem (MISP). Details on traditional interval scheduling problems with fixed processing time are introduced in [25]. In the following, a general formulation of the MISP is introduced and evaluated against some known algorithms. The key symbols used throughout this work are summarized in Table 2.

Table 2. Key Notations in This Paper

Notation	Definition
T	Whole observation time period
sl_0	Length of each time slot
n	Maximum number of requests
s_i	Start time of request i
f_i	Finishing time of request i
$A(i)$	Set of VMs requests scheduled to PM i
d_j	Capacity demand of VM j
CM_j^{r}	Capacity_makespan of VM request j
CM_i	Capacity_makespan of PM i
$PCPU_i$	CPU capacity of PM i
$PMem_i$	Memory capacity of PM i
$PSto_i$	Storage capacity of PM i
$VCPU_j$	CPU demand of VM j
$VMem_j$	Memory demand of VM j
$VSto_j$	Storage demand of VM j
T_j^{start}	Start time of VM request j
T_j^{end}	Finishing time of VM request j
T_r	Time span between time slot t_{r-1} and t_r
CM^{p}	Maximum Capacity_makespan of all PMs
IMD	Imbalance degree
k	Partition value
P_0	Lower bound of the optimal solution OPT
B_{d}	Dynamic balance value based on Capacity_makespan
L	Amount of VM requests that have already arrived
m	Number of PMs in use
I	Set of VM requests
CM_{b}	Predefined Capacity_makespan threshold for partition
f	Constant value to avoid too frequent partitions

3.1 Assumptions

The key assumptions are as follows.

1) The time is given in a discrete fashion; all data is given deterministically. The whole time period [0, T] is partitioned into equal-length (sl_0), and the total number of slots is then $t{=}T/sl_0$. The start time s_i and the end time f_i are the multiples of the minimum slot. Then the interval of demand can be expressed in a slot fashion with (start time, end time). For instance, if $sl_0 = 10$ min, an interval [5, 12] represents that the task starts at the 5th time slot and finishes at the 12th time slot. The duration of this demand is $(12-5)\times 10 = 70$ min.

2) For all VM requests generated by users, they have the start time and the end time to represent their life-cycles, and the capacity to show the required amount of resources.

3) The capacity of a single PM is normalized to be 1 and the required capacity of a VM can be 1/8, 1/4, 1/2 or other portions of the total capacity of a PM. This is consistent with applications in Amazon EC2[26] and [27].

3.2 Key Definitions

A few key definitions are given here.

Definition 1 (Traditional Interval Scheduling Problem (TISP) with Fixed Processing Time). *In a batch of demands* {1, 2, ..., n}, *the i-th demand refers to an interval of time starting at s_i and ending at f_i* ($\forall i, s_i < f_i$). *Besides each demand requires a capacity of* 100%, *i.e., utilizing the full capacity of a server during the interval.*

Definition 2 (Interval Scheduling with Capacity Sharing, ISWCS). *Different from TISP, ISWCS can share the capacities among demands if the sum of all demands scheduled on the single server at any time is still not fully utilized.*

Definition 3 (Compatible Sharing Intervals for ISWCS, CSI-ISWCS). *A batch of intervals with requested capacities below the whole capacity of a PM during the intervals can be compatibly scheduled on a PM. Compared against ISWCS, the requests in CSI-ISWCS can be modelled as the ones with lifecycles, which can be represented as sharing the subset of intervals.*

In the existing literature[18], makespan, i.e., the maximum total load (processing time) on any machine, is applied to measure load balancing.

In this paper, we aim to solve the problem based on the ISWCS manner and apply a new metric Capacity_makespan.

Definition 4 (Capacity_makespan of PM_i). *In the schedule of VM requests to PMs, $A(i)$ is denoted as the set of VM requests scheduled to PM_i. With this scheduling, PM_i will have load as the sum of the prod-*

uct of each requested capacity and its duration, called Capacity_makespan (CM)*, as shown in* (1):

$$CM_i = \sum_{j \in A(i)} d_j t_j, \tag{1}$$

where d_j *is the capacity demand (some portion of total capacity) of* VM_j *from a PM where the capacity can be CPU or memory or storage in this paper.* d_j *can also be simplified as a capacity based on assumption* 3*, and* t_j *represents the span of demand* j*, being the length of processing time of the demand* j*.*

Similarly, the Capacity_makespan of a given VM request is simply the product of the requested capacity and its duration.

3.3 Optimization Objective

Then, the objective of load balancing is to minimize the maximum load (Capacity_makespan) on all PMs as noted in (2). Considering that m PMs are in the data center, we can formulate the problem as:

$$\min \left(\max_{1 \leqslant i \leqslant m} (CM_i) \right), \tag{2}$$

$$\text{subject to } \forall \text{ slot } s, \sum_{j \in A(i)} d_j \leqslant 1, \tag{3}$$

where (3) shows the sharing capacity constraint that in any time interval, the shared resources should not use up all the provisioned resources (100%).

From (1) amd (2) , we see that lifecycle and capacity sharing are key metrics different from traditional ones like makespan which focuses on the process time. Traditionally Longest Process Time first (LPT)[28] is widely adopted in offline multi-processor load balancing. Reactive migration of VMs is another way to compensate after allocation.

3.4 Metrics for ISWCS Load Balancing

The key metrics based on [29] for ISWCS load balancing are given in the following.

1) PM resources: $PM_i(i,\ PCPU_i,\ PMem_i,\ PSto_i)$, where $PCPU_i$, $PMem_i$, and $PSto_i$ are the CPU, memory, and storage capacity that a PM can offer respectively.

2) VM resources: $VM_j(j,\ VCPU_j,\ VMem_j,\ VSto_j,\ T_j^{\text{start}},\ T_j^{\text{end}})$, where $VCPU_j$, $VMem_j$, and $VSto_j$ are the CPU, memory, and storage demand of VM_j respectively, and $T_j^{\text{start}}, T_j^{\text{end}}$ are the start time and the end time respectively.

3) Discrete time: a time span can be partitioned into equal length of slots. The slots with size s can be considered as $[(t_0, t_1), (t_1, t_2), \ldots, (t_{s-1}, t_s)]$, and each time slot T_r represents the time span (t_{r-1}, t_r).

4) Average CPU utilization of PM_i during slot 0 and T_s: it is defined in (4):

$$PCPU_i^{\text{U}} = \frac{\sum_{r=0}^{s} (PCPU_i^{T_r} \times T_r)}{\sum_{r=0}^{s} T_r}, \tag{4}$$

where $PCPU_i^{T_r}$ is the average CPU utilization monitored and computed in slot T_r which may be a few minutes long, and it can be obtained by monitoring CPU utilization in slot T_r. Average memory utilization ($PMem_i^{\text{U}}$) and storage utilization ($PSto_i^{\text{U}}$) of PMs can be calculated similarly. Similarly, the average CPU (memory and storage) utilization of a VM can be calculated.

5) Makespan: it represents the whole length of the scheduled VM reservations, i.e., the difference between the start time of the first request[②] and the end time of the last request.

6) The maximum Capacity_makespan (CM^{P}) of all PMs is calculated in (5) as below:

$$CM^{\text{P}} = \max_i (CM_i), \tag{5}$$

which we can apply CPU, memory and storage utilization to substitute CM_i too.

7) Imbalance degree (IMD): it is a metric that measures how far a set of values are spread out from each other in statistics. Imbalance degree is the normalized variance (regarding its average) of CPU, memory and storage utilization for all PMs. It measures load imbalance effect and is defined as shown in (6):

$$IMD = \left(\sum_{i=0}^{m} \left(\frac{(Avg_i - CPU_u)^2}{3} + \frac{(Avg_i - Mem_u)^2}{3} + \frac{(Avg_i - Sto_u)^2}{3} \right) \right) \Big/ m, \tag{6}$$

where Avg_i is defined in (7) as:

$$Avg_i = \frac{PCPU_i^{\text{U}} + PMem_i^{\text{U}} + PSto_i^{\text{U}}}{3}, \tag{7}$$

and CPU_u, Mem_u, Sto_u are the average utilization of CPU, memory and storage in a cloud data center re-

[②] In this paper, we interchange demands and requests, both of which are referred to as VM requests.

spectively and can be computed using utilization of all PMs in a cloud data center.

Theorem 1. *Minimizing the makespan in the offline scheduling problem is NP-hard*[29].

The proof was provided in our previous work[29] and we omit it here. Our approach in this paper differs from [29] in several perspectives: 1) we consider that the multiple VM requests are allowed to be executed on the same host simultaneously rather than a single VM request in [29]; 2) our objective is minimizing the Capacity_makespan rather than the longest processing time; 3) we extend [29] to be suitable for both the offline and online scenarios rather than for only the online one.

Combining the properties of both fixed process time intervals and capacity sharing, we present new offline and online algorithms in Section 4.

4 Prepartition Algorithms

In the following, we introduce one algorithm for the offline scenario and two algorithms for the online scenario. These algorithms can handle both the offline and online requests, and achieve good performance in load balancing.

4.1 Algorithm PrepartitionOff

First, we introduce the PrepartitionOff algorithm that aims to partition the VM requests under the situation that the information of all VM requests is known in advance. The processed order of VM requests and the prepartition operations can be managed by the algorithm.

Considering a set of VM reservations, there are m PMs in a data center and we denote OPT as the optimal solution with regard to minimizing Capacity_makespan. Firstly we define

$$P_0 = \frac{1}{m}\sum_{j=1}^{J} CM_j^{\mathrm{r}} \leqslant OPT, \tag{8}$$

where J denotes the total number of allocated VMs, and P_0 denotes the lower bound for OPT as shown in (8).

Algorithm 1 gives the pseudocodes of the PrepartitionOff algorithm which measures the ideal load balancing among m PMs. The algorithm firstly calculates the balancing value by (8), sets a partition value (k) and computes the length of each partition, i.e., $\lceil P_0/k \rceil$, representing the maximum VM's CM that can be allocated on a PM (line 1). For every demand, PrepartitionOff divides it into multiple $\lceil P_0/k \rceil$ subintervals when its CM is equal to or larger than P_0, and each subinterval is treated as a new request (lines 2–4). Then the algorithm sorts the newly generated requests in decreasing order based on the CM for further scheduling (line 5). After sorting of requests, the algorithm will pick up the VM with the earliest start time, and allocates the VM to the PM with the lowest average CM and enough available resources (lines 6–8), thus achieving the load balancing objective. The CM of PM will also be updated accordingly (line 9). Finally, the algorithm calculates the CM of each PM when all requests are assigned and finds the total partition number (line 10). In practice, the scheduler records all possible subintervals and their hosting PMs so that the migration of VMs can be prepared beforehand to alleviate overheads.

Algorithm 1. PrepartitionOff

Input: m: the total number of PMs; n: the total number of VM requests; CM_j^{r}: the CM of request j; CM_i: the CM of PM i;

Output: assign PM IDs to all requests and their partitions

1 Initialization: computing the bound value P_0 and partition value k (e.g., 1, 2,...);
2 **forall** i from 1 to m **do**
3 **if** $CM_i \geqslant P_0$ **then**
4 Divide it by $\lceil P_0/k \rceil$ subintervals equally and treat each subinterval as a new request
5 All intervals are sorted in decreasing order of CM, and ties are broken arbitrarily;
6 **forall** j from 1 to n **do**
7 Pick up the VM with the earliest start time in the VM queue for execution;
8 Allocate j to the PM with the smallest load and enough capacity;
9 Update load (CM) of the PM;
10 Calculate CM of every PM and find the total number of partitions

Theorem 2. *Applying the priority queue data structure, the PrepartitionOff algorithm has a computational complexity of $O(n\log m)$, where n is the number of VM requests after pre-partition and m is the total number of PMs used.*

Proof. The priority queue is adopted so that each PM has a priority value (average Capacity_makespan), and each time when the algorithm chooses an item from it, the algorithm selects the one with the highest priority. It costs $O(n)$ time to sort n elements, and $O(\log n)$ steps for insertion and the extraction of minima in a priority queue[25]. Then, by adopting a priority queue, the algorithm picks a PM with the lowest average Capacity_makespan in $O(\log m)$ time. In total, the time complexity of the PrepartitionOff algorithm is $O(n\log m)$ for n demands. □

Theorem 3. *The PrepartitionOff algorithm has the*

approximation ratio of $1+\epsilon$ *regarding the Capacity_makespan where* $\epsilon=1/k$ *and* k *is the partition value* (*a preset constant*).

Proof. It can be seen that every demand has bounded Capacity_makespan by Preparition applying the lower bound P_0. Every request has start time s_i, end time f_i and process time $p_i=f_i-s_i$. The start time of the last completion job (later than all the other jobs) is T_0. We also assume that all the other servers are allocated with VM requests and denote the maximum Capacity_makespan as CM_{m}, that is, $CM_{\mathrm{m}} \leqslant OPT$. Since, for all requests $i \in J$, we have $CM_i \leqslant \epsilon OPT$ (by the setting of the PrepartitionOff algorithm in (9)), this job finishes with load $(CM_m+\epsilon OPT)$. Therefore, the schedule with Capacity_makespan $CM_m+\epsilon OPT \leqslant (1+\epsilon)OPT$. □

4.2 Algorithm PrepartitionOn1

Apart from the offline scenario, the online scenario is also quite common in a realistic environment. For online VM allocations, scheduling decisions must be made without complete information about the entire job instances because jobs arrive one by one. We firstly extend the PrepartitionOff algorithm to the online scenario as the PrepartitionOn1 algorithm, which can only have the information of VM requests when the requests come into the system.

Given m PMs and L VMs (including the one that just came) in a data center, we can have the dynamic balance value as shown in (9):

$$B_{\mathrm{d}} = \min\left(\max_{1\leqslant j\leqslant L}(CM_j^{\mathrm{r}})/2, \sum_{j=1}^{L}(CM_j^{\mathrm{r}})/m\right), \quad (9)$$

where B_{d} is one half of the maximum Capacity_makespan of all current PMs or the ideal load balance value of all current PMs in the system, and L is the number of VMs requests that have already arrived. Notice that the reason to set B_{d} as one half of the maximum Capacity_makespan of all current PMs is to avoid the extra management costs caused by a large number of partitions.

Algorithm 2 shows the pseudocodes of the PrepartitionOn1 algorithm. Since in an online scenario, the requests come one by one, the system can only capture the information of arrived requests. The algorithm firstly predefines the prepartition value of k and the total partition number P as 0 (line 1). When a new request comes into the system, the algorithm picks up the VM with the earliest start time in the queue for scheduling and computes dynamic balance value (B_{d}) by (9) (lines 2 and 3). After B_{d} is computed, if the Capacity_maskespan of the VM request is too large (larger than $\lceil(B_{\mathrm{d}}/k)\rceil$), then the initial request is partitioned into several requests (segments) based on the partition value k. In these partitioned requests, if some requests are still with large Capacity_maskespan, they would be put back into the queue waiting to be executed, and follow the same partition and allocation process (lines 4 and 5). The VM requests with small Capacity_maskespan after partition would be executed when their start time begins, and will be assigned to the PM that has the minimum value of Capacity_makespan (lines 6–8). After all demands are allocated, the PrepartitionOn1 algorithm calculates the Capacity_makespan value of all the PMs and outputs all the partition values for n demands (line 9). Since the numbers of partitions and segments of each VM request are known at the moment of allocation, the system can prepare VM migration in advance so that the processing time and instability of migration can be reduced.

Algorithm 2. PrepartitionOn1

Input: m: the total number of PMs; n: the total number of VM requests; CM_j^{r}: the CM of request j; CM_i: the CM of PM i;

Output: assign PM IDs to all requests and their partitions

1 Set the partition value k, total partition number $P=0$;
2 **for** each arrived job j **do**
3 Pick up the VM with the start time equal to the system time in the VM queue to schedule; compute CM_j^{r} of VM_j and B_{d} using (9);
4 **if** $CM_j^{\mathrm{r}} > \lceil(B_{\mathrm{d}}/k)\rceil$ **then**
5 Partition VM_j into multiple $\lceil(B_{\mathrm{d}}/k)\rceil$ equal subintervals, treat each subinterval as a new demand and add them into the VM queue, $P=P+\lceil\frac{CM_j^{\mathrm{r}}}{B_{\mathrm{d}}/k}\rceil$, update load (CM) of the PM;
6 **else**
7 Allocate j to PM with the minimum load and enough capacity;
8 Update load (CM) of the PM;
9 Output the total number of partitions P

To analyze algorithm performance based on theoretical analysis, we conduct competitive ratio analysis that represents the performance ratio between an online algorithm and an optimal offline algorithm. An online algorithm is competitive if its competitive ratio is bounded.

Theorem 4. *The PrepartitionOn1 algorithm has a competitive ratio of* $(1+(1/k)-(1/mk))$ *with regarding to the Capacity_makespan.*

Proof. Without loss of generality, we label PMs in order of non-decreasing final loads (CM) in the PrepartitionOn1 algorithm. OPT and $PrepartitionOn1(I)$ are denoted as the optimal load balance value of corresponding offline scheduling and

the load balance value of PrepartitionOn1 for a given set of jobs I, respectively. Then the load of PM_m defines the Capacity_makespan. Each of the $m-1$ PMs processes a subset of the jobs and then experiences an (possibly none) idle period. All PMs together finish a total Capacity_makespan $\sum_{i=1}^{n} CM_i$ during their busy periods. Let us consider the allocation of the last job j to PM_m. By the scheduling rule of the PrepartitionOn1 algorithm, PM_m has the minimum load at the time of allocation. Hence, any idle period on the first $m-1$ PMs cannot be bigger than the Capacity_makespan of the last job allocated on PM_m and hence cannot exceed the maximum Capacity_makespan divided by k (partition value), i.e., $\max_{1\leqslant i\leqslant n} CM_i/k$. Based on (10), we have

$$m\times PrepartitionOn1(I)\leqslant\sum_{i=1}^{n} CM_i+(m-1)\frac{\max(CM_i)}{k}, \tag{10}$$

which is equivalent to (11) as below:

$$PrepartitionOn1(I)\leqslant\sum_{i=1}^{n}\frac{CM_i}{m}+(m-1)\frac{\max(CM_i)}{mk}, \tag{11}$$

which is also equivalent to (12):

$$PrepartitionOn1(I)\leqslant\left(OPT+\left(\frac{1}{k}-\frac{1}{mk}\right)OPT\right). \tag{12}$$

Note that $\sum_{i=1}^{n} CM_i/m$ is the lower bound on $OPT(I)$ because the optimum Capacity_makespan cannot be smaller than the average Capacity_ makespan on all PMs. Besides $OPT(I)\geqslant \max_{1\leqslant i\leqslant n} CM_i$ since the largest job must be processed on a PM. We therefore have $PrepartitionOn1(I) \leqslant (1+(1/k)-(1/mk))OPT$. □

Theorem 5. *By using the priority queue, the computational complexity of the PrepartitionOn1 algorithm is $O(n\log m)$, where n is the number of VM requests after the pre-partition operations and m is the total number of used PMs.*

Proof. It is similar to the proof for Theorem 2, and we omit it here. □

4.3 Algorithm PrepartitionOn2

Observing that the PrepartitionOn1 algorithm may bring too many partitions in some cases, we present the PrepartitionOn2 algorithm by introducing a parameter to control the number of partitions in a more flexible manner. The differences between the PrepartitionOn1 algorithm and the PrepartitionOn2 algorithm are the followings.

1) To avoid a large number of partitions, we bring a constant value f (for instance 0.125, 0.25) for measuring load balancing.

2) Setting a CM bound for each PM, for instance, each PM has a CM as 1×24 in each day within 24 h, but we consider a PM can at most run with 100% CPU utilization in 16 hours, i.e., we set a CM bound for each PM for each day as $CM_{\mathrm{B}}=16$. If overloading happens to a PM according to predefined thresholds $1+f$ and CM_{B}, then a new request should be partitioned into x (the number of active PMs) subintervals equally and the scheduler allocates each subinterval to every PM.

The pseudocodes of the PrepartitionOn2 algorithm are shown in Algorithm 3. The algorithm firstly initializes the predefined Capacity_makespan bound of PMs and the constant value f as introduced above (line 1). For the arrived VMs, the algorithm picks up the VM with the earliest start time for execution, and calculates the Capacity_makespan of both VMs and PMs (lines 2–4). The picked VM will be supposed to be allocated to the PM with the minimum Capacity_makespan value, and the Capacity_makespan of the PM and the PM with the minimum Capacity_makespan are calculated with that supposition (lines 5–7). If the increased Capacity_makespan of the PM is too large (line 8), the VM will be partitioned into the number of active PMs, and the partitioned VMs are allocated to PMs one by one (line 9). Otherwise, the VM can be allocated directly to the PM with the minimum loads (lines 10 and 11). Finally, the scheduling results and the number of partitions can be obtained (line 12).

Theorem 6. *PrepartitionOn2 has a computational complexity of $O(n\log m)$ by applying a priority queue, where n is the number of VM requests after the prepartition operations and m is the total number of used PMs.*

Proof. It is also similar to the proof for Theorem 2, and we omit it here. □

Theorem 7. *The competitive ratio of the PrepartitionOn2 algorithm is at most $1+f$ and each PM has maximum CM as CM_{B}.*

Proof. According to Algorithm 3, whenever a PM has CM larger than CM_B or the competitive ratio of the algorithm is larger than $1+f$, the allocating VMs

will be pre-partitioned into multiple sub-instances and allocated. Therefore, the competitive ratio of the PrepartitionOn2 algorithm is at most $1+f$. □

Algorithm 3. PrepartitionOn2

Input: m: total number of PMs; n: total number of VM requests; CM_j^{r}: the CM of request j; CM_i: the CM of PM i;

Output: assign PM IDs to all requests and their partitions

1 Initialization: set the CM bound CM_{B} for each PM, a constant value $f(\leqslant 0.125)$, total partition number $P=0$;
2 **for** each arrived job j **do**
3 Pick up the VM with the earliest start time in the VM queue to schedule;
4 Compute CM_j^{r} of VM j, and CM of each PM;
5 Choose the minimum value CM of PM named CM_{oldmin};
6 Suppose to allocate VM j to the PM which has CM_{oldmin} and compute its new value of CM named $CM_{\mathrm{oldmin}+}$;
7 Get the new minimum value of CM of PM named CM_{newmin};
8 **if** $(CM_{\mathrm{oldmin}+}/CM_{\mathrm{newmin}})>(1+f)$ or $CM_{\mathrm{oldmin}+}>CM_{\mathrm{B}}$ **then**
9 Partition VMj into multiple x (the number of PM turned on) subintervals equally, and allocate each subinterval to every PM, $P=P+x$;
10 **else**
11 Allocate j to PM with the minimum load and available capacity;
12 Output the total number of partitions P

5 Performance Evaluation

Notice that there are eight types of VMs in Table 3 and three types of PMs in Table 4, where each type of VM occupies 1/16 or 1/8 or 1/4 or 1/2 of the whole capacity of the corresponding PM considering all three dimension resources of CPU, memory, and storage. Therefore the three dimension resources become one dimension in this case. In the following, the simulation results of the Prepartition algorithms and a few existing algorithms are provided. To conduct simulation, a Java simulator called CloudSched (refer to Tian *et al.*[30]) is used.

5.1 Offline Algorithm Performance Evaluation

Table 3. Eight Types of VMs Derived from Amazon EC2

VM Type (No.)	Computing Capacity (Cores)	Memory (GB)	Storage (GB)
1-1(1)	1.0	1.875	211.25
1-2(2)	4.0	7.500	845.00
1-3(3)	8.0	15.000	1 690.00
2-1(4)	6.5	17.100	422.50
2-2(5)	13.0	34.200	845.00
2-3(6)	26.0	68.400	1 690.00
3-1(7)	5.0	1.875	422.50
3-2(8)	20.0	7.000	1 690.00

Table 4. Three Types of Suggested PMs Specification

PM Type	Computing Capacity (Cores)	Memory (GB)	Storage (GB)
1	16	30.0	3 380
2	52	136.8	3 380
3	40	14.0	3 380

All simulations are conducted on a computer configured with an Intel i5 processor at 2.5 GHz and 4 GB memory. All VM requests are generated by following normal distribution. To compare the offline algorithms, Round-Robin (R-R) algorithm, LPT algorithm and post migration (PMG) algorithm are implemented.

1) *R-R Algorithm.* It is a load balancing scheduling algorithm by allocating the VM demands in turn to each PM that can offer demanded resources.

2) *LPT Algorithm.* LPT is one of the best practices for offline scheduling algorithms without migration, which has an approximation ratio of 4/3. All the VM demands are sorted by processing time in decreasing order firstly. Then demands are allocated to the PM with the minimum load in the sorted order. The minimum load indicates the minimum Capacity_makespan among all the PMs.

3) *PMG Algorithm.* The PMG algorithm comes from the VMware distributed resource scheduling (DRS) algorithm[31], which adopts migration to achieve load balancing regarding makespan. In the beginning, it allocates the demands in the same way as LPT does. Here we replace makespan by Capacity_makespan. Then the algorithm calculates the average Capacity_makespan of all demands. In the PMG algorithm, the up-threshold and the low-threshold are configured to achieve the load balancing effects, which are configured based on the average Capacity_makespan and factor. In our experiments, we configure the factor as 0.1 (which can be configured dynamically to meet the demands), which represents that the up-threshold is 1.1 times of the average Capacity_makespan and the low-threshold is 0.9 times the average Capacity_makespan. The algorithm also maintains a migration list containing the VMs on the PMs with higher Capacity_makespan values than the low-threshold. The VM migration is triggered to make the Capacity_makespan smaller than the low-threshold. Thereafter, the VMs in the migration list will be re-allocated to a PM with Capacity_makespan smaller than the up-threshold. Migrating VMs to a new PM is triggered if the operation would not lead the Capacity_makespan of the

PM to be higher than the up-threshold. To be noted, some VMs can be left in the list; thus finally the algorithm allocates the left VMs to the PMs with the minimum Capacity_makespan in sequence to balance the loads.

VMs and PMs have the same configuration with Amazon EC2. The configurations are shown in Table 3 and Table 4, in which one unit (core) of computing capacity is equivalent to 1.0 GHz–1.2 GHz 2007 Xeon or 2007 Opteron processors[26].

Remarks. We adopt the typical recommended VM types suggested by Amazon EC2. Amazon EC2 has a variety of VM types, and it classifies them into general purpose, compute optimized (computational intensive VMs), memory optimized (memory-intensive VMs), and storage optimized (storage-intensive VMs). Although we adopt the Amazon EC2 classification, our approach can still be extended to other classifications.

5.1.1 *Replay with ESL Data Trace*

To reflect realistic data generation, we utilize the data derived from Hebrew University Experimental System Lab (ESL)[32] that has been widely used for realistic data. The data with monthly records collected by the Linux cluster has characteristics that can be fitted into our reservation model. In the log file, each line contains 18 elements where we only need parts of them, such as the requested ID, start time, duration and the number of processors (capacity demands) in our simulation. Because the time slot length mentioned previously is set to 5 min, the time units of the original data are converted from seconds to minutes.

Fig.3 shows the comparison of different algorithms in average utilization, imbalance degree, makespan and Capacity_makespan. According to the results, we can observe that the PrepartitionOff algorithm can achieve better performance than the other algorithms in four aspects. For average utilization, the PrepartitionOff algorithm is 10%–20% higher than PMG, LPT, and R-R. The reason for different algorithms to have different average CPU utilization lies in that we consider heterogeneous PMs and different algorithms may use the different numbers of total PMs. For makespan and Capacity_makespan, the PrepartitionOff algorithm is 10%–20% lower than PMG and LPI, and 30%–40% lower than R-R. And for imbalance degree, it is 30%–40% lower than LPT.

Observation 1. As shown in the above performance evaluations, PMG is one of the best heuristic approaches to balancing loads; however, it cannot assure a bounded or predefined target.

Observation 2. PMG does not obtain the same good performance as PrepartitionOff in terms of average utilization, makespan and Capacity_makespan, no matter how many times of migration have been taken.

The main reason is that PrepartitionOff takes actions in a much more refined and desired scale by prepartition based on reservation data while PMG is just a best-effort trial by migration. In addition, PrepartitionOff is much more precise and desired with the aid of prepartition while PMG is just a trial to balance load as much as possible. To compare imbalance degree (IMD) change as time goes, we also conduct the tests about consecutive imbalance degree using 1 000 VMs and 2 000 VMs among four different offline algorithms as shown in Fig.4. In Fig.4(a) and Fig.4(b), the X-axis is for time and the Y-axis is for imbalance degree. We can see that PrepartitionOff (with $k = 8$) has the minimum makespan and minimum imbalance degree most of the time during tests, except for the initial period. Notice that the value of k can be set differently. Here we just present the results for $k = 8$.

5.1.2 *Results Comparison by Synthetic Data*

We configure the time slot to be 5 minutes as mentioned before; therefore, an hour has 12 slots and a day has 288 slots. All requests are subject to normal distribution with mean μ as 864 (three days) and standard deviation δ as 288 (one day) respectively. After requests are generated in this way, we start the simulator to simulate the scheduling effects of different algorithms and the comparison results are collected. For data collection, first we set k of the PrepartitionOff algorithm as 4 (we configure the value as 4 because in previous research[18], this value has been validated to be an effective value to improve performance). Besides the different types of VMs are with equal probabilities. We also vary the number of VMs from 100 to 200, 400 and 800 to analyze the trend. Each dataset is an average of 10 runs.

Fig.5 displays the comparison of different algorithms in average utilization, imbalance degree, makespan and Capacity_makespan. From Figs.5(a)–5(d), we can know that for average utilization, the PrepartitionOff algorithm is 10%–20% higher than

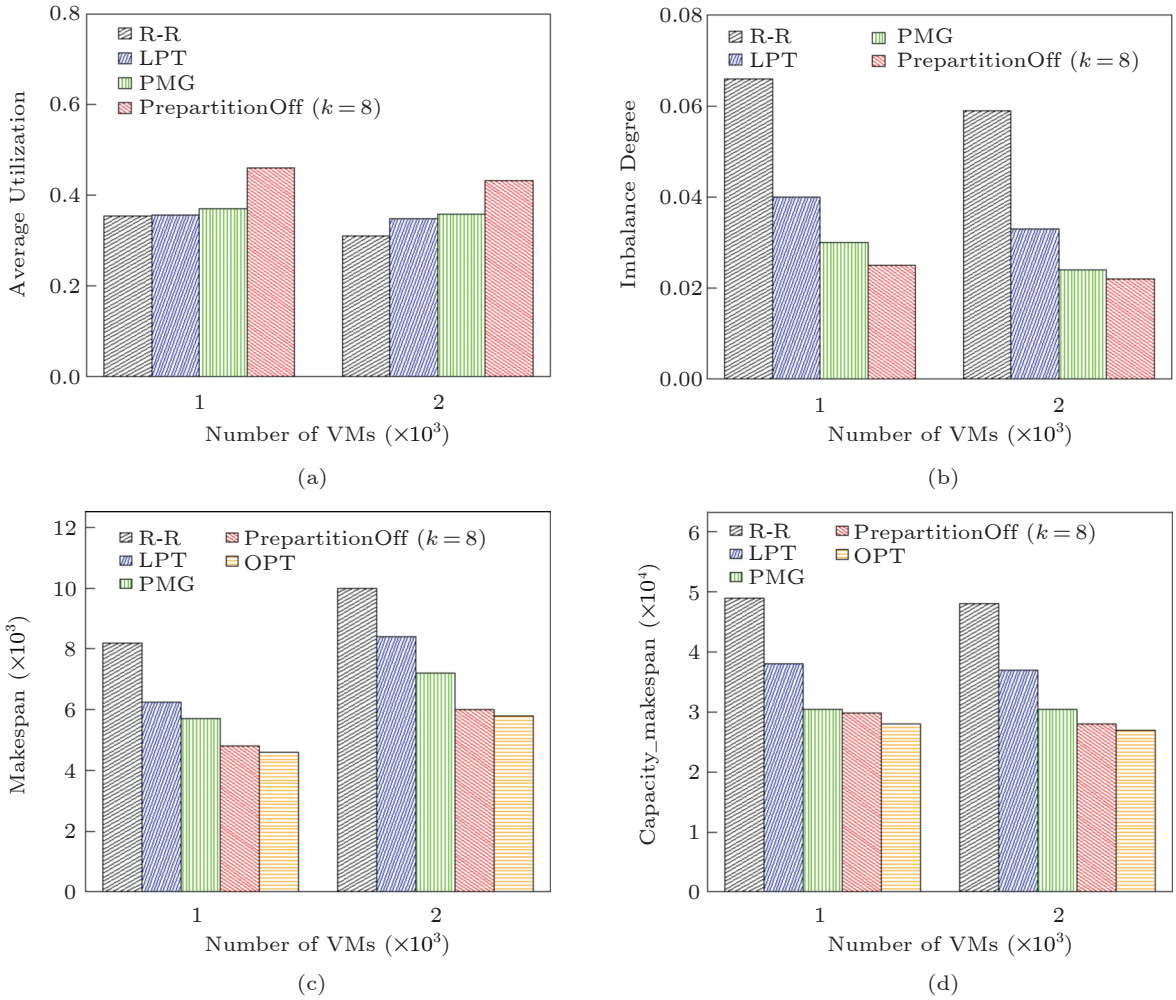

Fig.3. Comparison of offline algorithms with ESL trace. (a) Average utilization. (b) Imbalance degree with ESL trace. (c) Makespan with ESL trace. (d) Capacity_makespan.

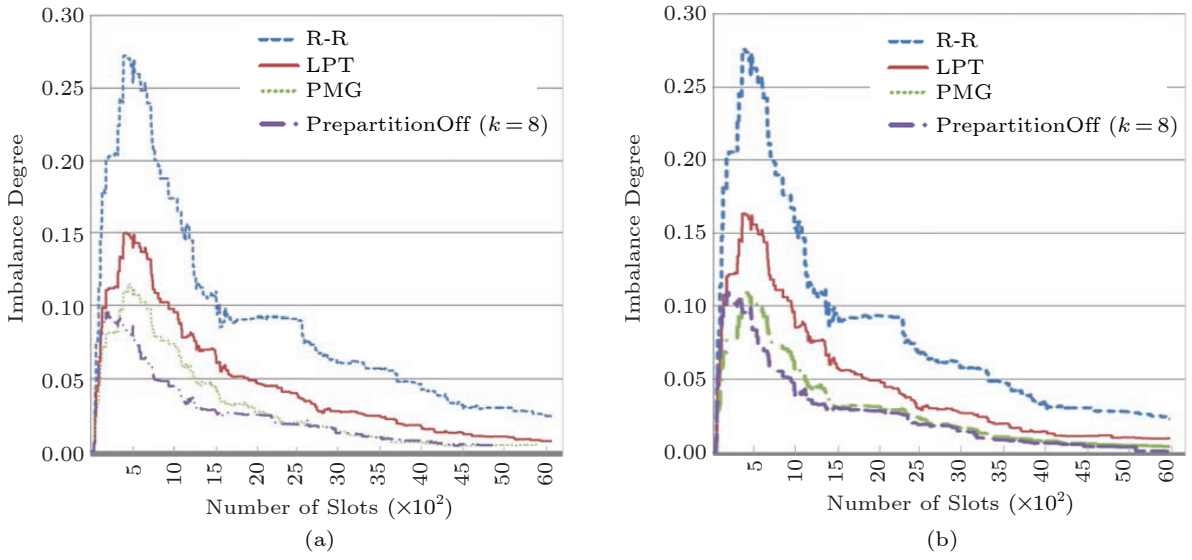

Fig.4. Consecutive imbalance degree under different numbers of VMs of four different offline algorithms. (a) 1 000 VMs. (b) 2 000 VMs.

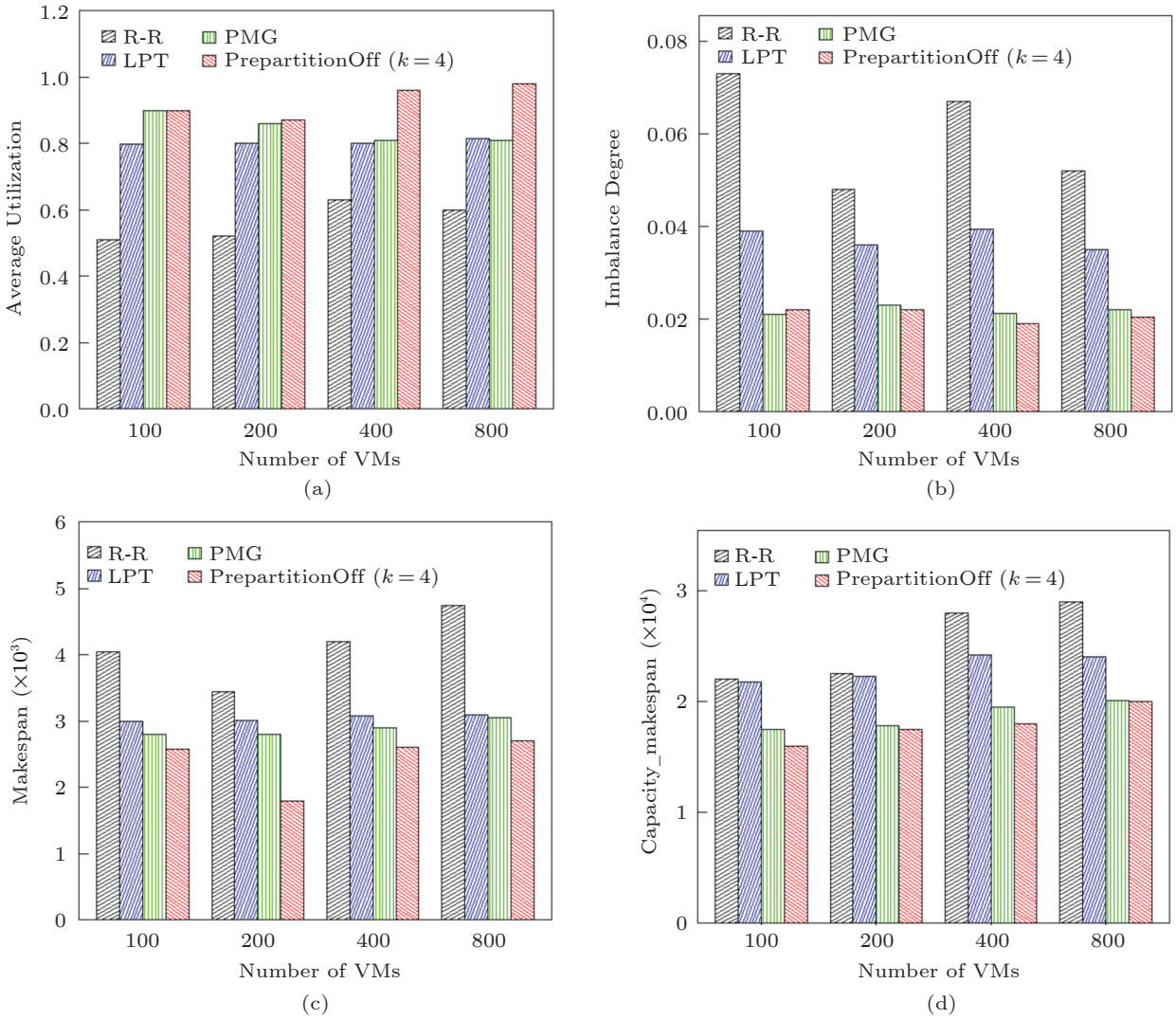

Fig.5. Comparison of offline algorithms with normal distribution. (a) Average utilization. (b) Imbalance degree. (c) Makespan. (d) Capacity_makespan.

PMG and LPT, and 40%–50% higher than R-R. As for makespan and Capacity_makespan, the PrepartitionOff algorithm is 8%–13% lower than PMG and LPT, and 40%–50% lower than R-R. We also note that the PMG algorithm can improve the performance of the LPT algorithm as it configures the up-threshold and low-threshold based on the Capacity_makespan value. The LPT algorithm is better than the R-R algorithm. Similar results are observed for the comparison of makespan.

5.2 PrepartitionOn1 Algorithm Performance Evaluation

We demonstrate the simulation results of the PrepartitionOn1 algorithm and the other three algorithms in this subsection. All VM requests are generated by following the normal distribution, and the online algorithms including Random, R-R, and Online Resource Scheduling Algorithm (OLRSA)[33] that has a good competitive ratio $(2-(1/m)$, where m is the number of PMs) are compared with PrepartitionOn1. OLRSA calculates the Capacity_makespan of all the PMs and sorts PM by Capacity_makespan in descending order, which assigns the VM request to the PM with the minimum Capacity_makespan and required resources.

5.2.1 *Replay with ESL Data Trace with PrepartitionOn1*

The ESL dataset aforementioned is also used in the experiments. Fig.6 illustrates the comparisons of the average utilization, imbalance degree, makespan,

and Capacity_makespan. According to Figs.5(a)–5(d), we can see that the PrepartitionOn1 algorithm demonstrates the highest average utilization, the minimum imbalance degree, and the minimum makespan. As for Capacity_makespan, OLRSA shows much better performance compared with the Random algorithm and the R-R algorithm, and the PrepartitionOn1 algorithm still out performs 10%–15% in average utilization, 20%–30% in imbalance degree, and 5%–20% in makespan than OLRSA.

5.2.2 *Results Comparison by Synthetic Data with PrepartitionOn1*

The requests are configured as the same as in Subsection 5.1 based on the normal distribution. We set that VMs with different types have equal probabilities, and we modify the requests generation approach to produce different sizes of requests to trace the tendency. From Fig.7, we can see that the PrepartitionOn1 algorithm has better performance in average utilization, imbalance degree, makespan, and Capacity_makespan. Compared with OLRSA, the PrepartitionOn1 algorithm still improves about 10% in average utilization, 30%–40% in imbalance degree, 10%–20% in makespan, and 10%–20% in Capacity_makespan.

LPT is one kind of the best approaches for offline load balance algorithms without migration, which has an approximation ratio of 4/3. Therefore, we suggest setting the value of k as 4, which can obtain an approximation ratio as $1+ 1/k= 5/4$. Under this configuration, a better approximation ratio could be obtained. With a higher k, better load balancing effects

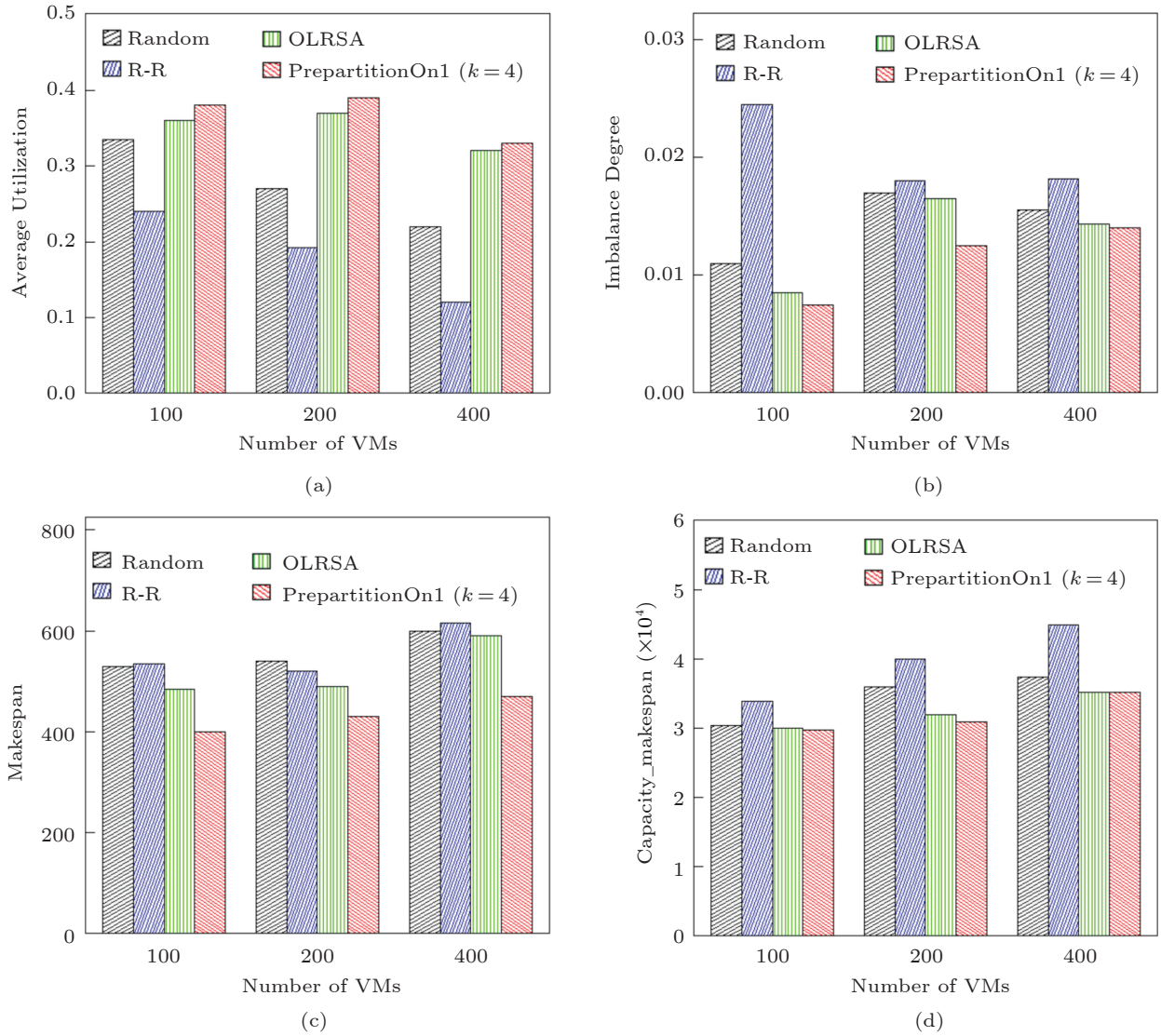

Fig.6. Comparison of online algorithms with ESL trace. (a) Average utilization. (b) Imbalance degree. (c) Makespan. (d) Capacity_makespan.

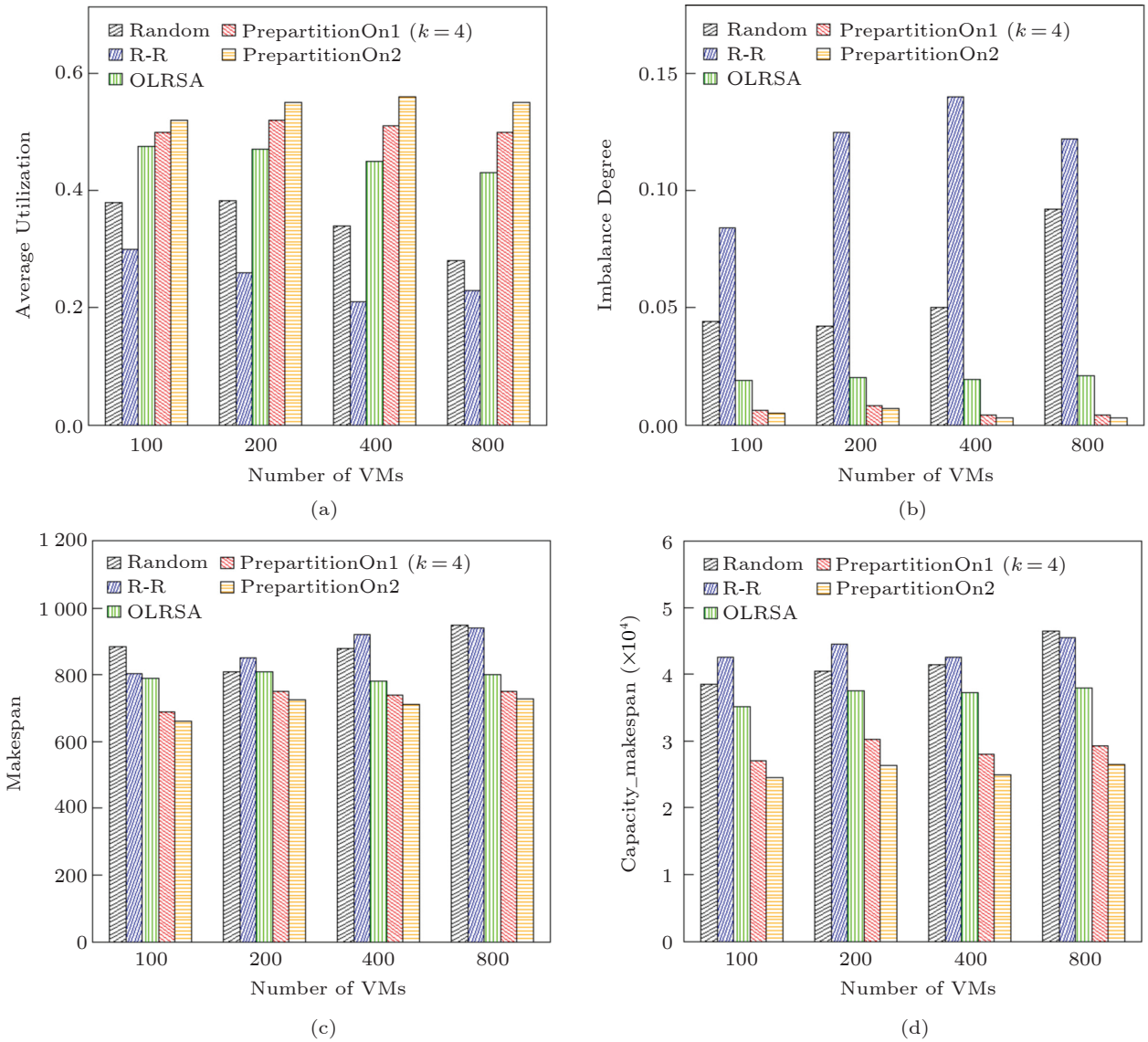

Fig.7. Comparison of online algorithms with normal distribution. (a) Average utilization. (b) Imbalance degree. (c) Makespan. (d) Capacity_makespan.

could be achieved. While there exist trade-offs between load balancing effect and time cost. For online load balance algorithms, we also suggest setting k as 4, and cloud service providers could reconfigure the value to be higher as suitable as the load balancing effects they desired.

Let us consider that we have $m = 100$ PMs and the value of k is set as 4, and then according to the analysis in [33], the complexity ratio of OLRSA is $2 - (1/m) = 2 - (1/100)$, and the complexity ratio of PrepartitionOn1 is $1 + (1/k) - (1/mk) = 1 + (1/4) - (1/400)$ based on Subsection 4.2. This proves that the PrepartitionOn1 algorithm can achieve better performance than OLRSA theoretically.

5.3 Performance Evaluation of the PrepartitionOn2 Algorithm

In this subsection, we display the simulation results of the PrepartitionOn2 algorithm and other three algorithms: Random, R-R, and OLRSA.

We still use the log data from ESL and normal distribution for experiments. Fig.7 and Fig.8 illustrate the comparisons of the average utilization, imbalance degree, makespan, and Capacity_makespan between the PrepartitionOn2 and the other online algorithms and the results show that PrepartitionOn2 performs the best in terms of compared metrics.

In Fig.9, we provide the consecutive imbalance degree comparison for four algorithms in online schedul-

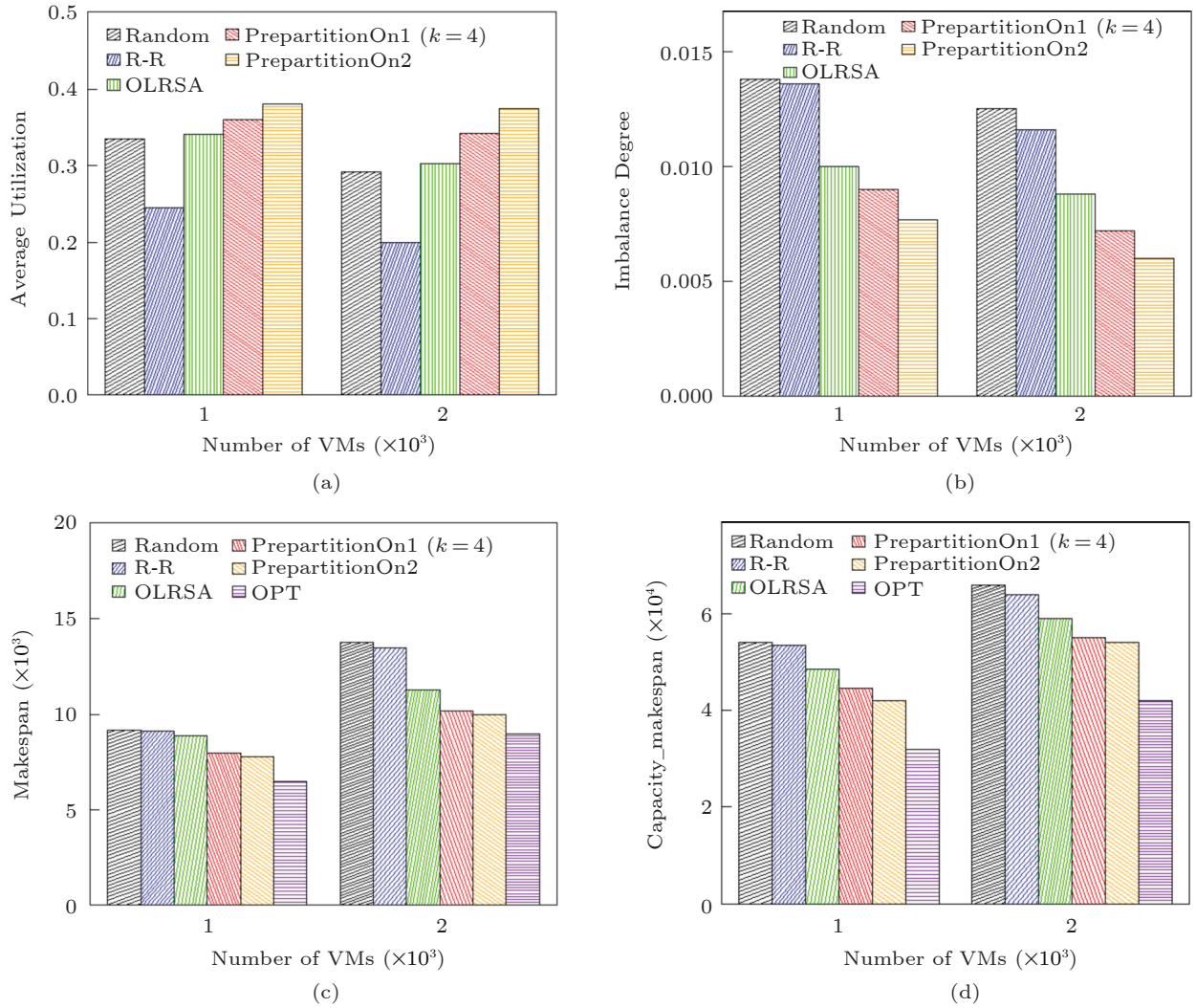

Fig.8. Comparison of online algorithms with ESL distribution. (a) Average utilization. (b) Imbalance degree. (c) Makespan. (d) Capacity_makespan.

ing with 1 000 VMs and 2 000 VMs respectively. In Fig.9(a) and Fig.9(b), the X-axis is for time and Y-axis is for imbalance degree. We can see that the PrepartitionOn2 algorithm has the minimum makespan and minimum imbalance degree most of the time during tests.

The large values of k may bring side effects since it will need more partitions. In Fig.10, we compare the time costs (simulated with ESL data and the time unit is millisecond) under different partition values of k. The PrepartitionOn1 algorithm with $k = 3$ takes about 10% less running time than that with $k = 4$, and PrepartitionOn1 with $k = 2$ takes 15% less running time than that with $k = 4$. A larger value of k will lead to a better load balance with a longer process time. We also observe that a larger value of k will induce a lower Capacity_makespan value. Similarly, with a larger value of k, a larger average utilization as well as a lower imbalance degree and a lower makespan can be obtained.

To evaluate the number of partitions triggered by different prepartition-based algorithms, Table 5 shows the number of partitions during our tests. Since the PrepartitionOff algorithm is offline, the number is much smaller than those of the online algorithms. And the partitions of PrepartitionOn2 are smaller than those of PrepartitionOn1, as PrepartitionOn2 has brought predefined parameters to avoid too many partitions as discussed in Subsection 4.3.

6 Conclusions

Load balancing for cloud administrators is a challenging problem in data centers. To address this issue,

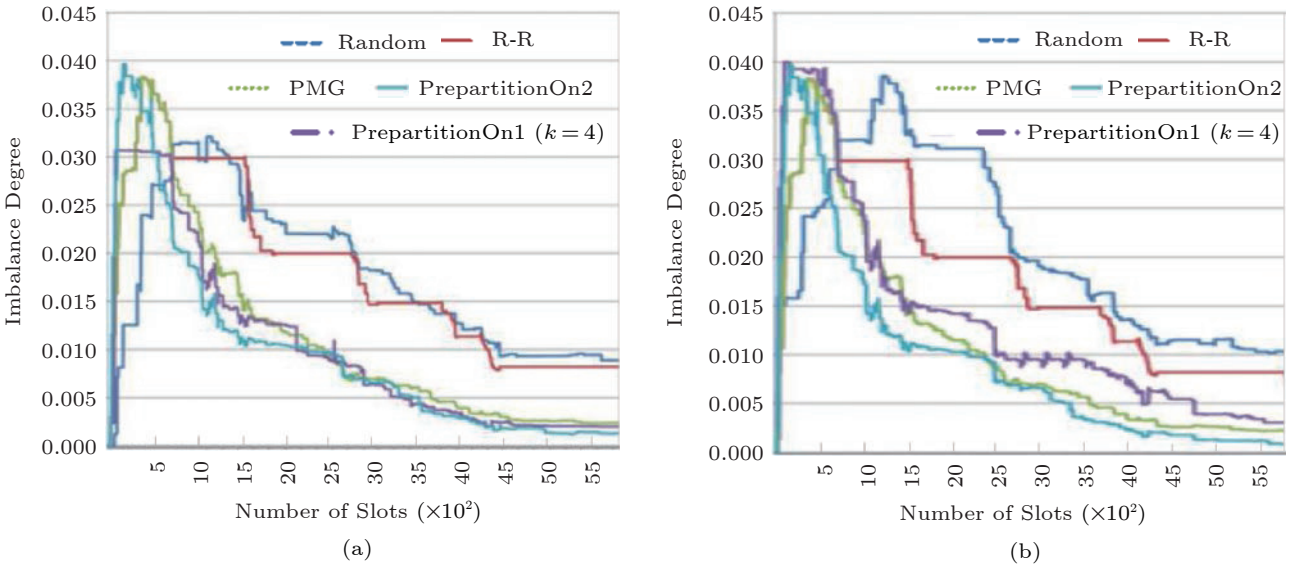

Fig.9. Consecutive imbalance degree under different numbers of VMs of five different online algorithms. (a) 1 000 VMs. (b) 2 000 VMs.

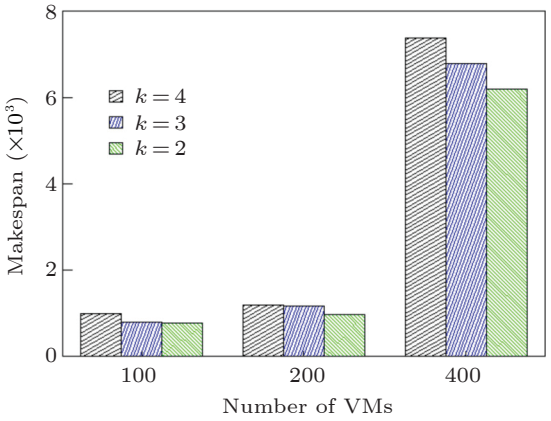

Fig.10. Comparison of time costs for PrepartitionOn1 by varying values of k.

Table 5. Number of Partitions in Different Algorithms

Algorithm	Number of Partitions	
	1 000 VMs	2000 VMs
PrepartitionOff	64	109
PrepartitionOn1	159	361
PrepartitionOn2	115	293

we proposed a novel VM reservation paradigm to balance the VM loads for PMs. Through prepartition operations before allocation for VMs, our algorithms can achieve better load balancing effects compared with well-known load balancing algorithms. In this paper, we presented both offline (PrepartitionOff) and online (PrepartitionOn1 and PrepartitionOn2) load balancing algorithms to reveal the feature of fixed interval constraints of VM scheduling and capacity sharing. Theoretically, we proved that PrepartitionOff is an algorithm with $1+\epsilon$ approximation ratio, where $\epsilon = 1/k$ and k is a positive integer. It is possible that the PrepartitionOff algorithm can be very close to the optimal solution via increasing the value of k, i.e., through setting up k, it is also attainable to achieve a desired load balancing goal defined in advance because PrepartitionOff is a $(1+(1/k))$-approximation. As for online algorithms, PrepartitionOn1 has competitive ratio $1+(1/k)-(1/mk)$ and PrepartitionOn2 has competitive ratio $1+f$ where f is a constant less than 0.5. Both the synthetic and trace-driven simulations validated theoretical observations and showed that the Prepartition algorithms can perform better than a few existing algorithms in terms of average utilization, imbalance degree, makespan, and Capacity_makespan. As such, other further research issues can be considered.

• *Appropriate Choice Between Load Balance and Total Partition Numbers.* The Prepartition-based algorithm can achieve desired load balance objective by setting a suitable value of k. It may need a large number of partitions so that the times of migration can be large depending on the characteristics of VM requests. For example, in Amazon EC2[26], the duration of VM reservations varies from a few hours to a few months; therefore we can classify different types of VMs based on their durations (Capacity_makespans) firstly, and then applying Prepartition will not have a large partition number for each type. In practice, we need to analyze traffic patterns to make the number of partitions (pre-migrations) reasonable so that the total costs, including running time and the times of migration, can be reduced.

• *Heterogeneous Configurations of PMs and VMs.* We mainly consider that a VM requires a portion of the total capacity from a PM. This is also applied in Amazon EC2 and [27]. When this is not true, multi-dimensional resources, such as CPU, memory, and bandwidth, have to be considered together or separately in the load balance.

• *Precedence Constraints Among Different VM Requests.* In reality, some VMs may be more important than others depending on applications running on them, and we would like to extend the current algorithm to consider this case.

• *Application Features Characterization with Multi-Tenancy and Resource Contention.* For instance, tightly coupled requests/applications can be partitioned on the same VM to reduce communication costs.

Acknowledgements The authors would like to thank the anonymous reviewers' valuable comments to improve the quality of our work.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Xu M X, Buyya R. Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions. *ACM Computing Surveys*, 2020, 52(1): Article No. 8. DOI: 10.1145/3234151.

[2] Xu F, Liu F M, Jin H, Vasilakos A V. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 2014, 102(1): 11–31. DOI: 10.1109/JPROC.2013.2287711.

[3] Gill S S, Tuli S, Toosi A N, Cuadrado F, Garraghan P, Bahsoon R, Lutfiyya H, Sakellariou R, Rana O, Dustdar S, Buyya R. ThermoSim: Deep learning based framework for modeling and simulation of thermal-aware resource management for cloud computing environments. *Journal of Systems and Software*, 2020, 166: 110596. DOI: 10.1016/j.jss.2020.110596.

[4] Xu M X, Buyya R. BrownoutCon: A software system based on brownout and containers for energy efficient cloud computing. *Journal of Systems and Software*, 2019, 155: 91–103. DOI: 10.1016/j.jss.2019.05.031.

[5] Zhang J, Yu F R, Wang S, Huang T, Liu Z Y, Liu Y J. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials*, 2018, 20(3): 2324–2352. DOI: 10.1109/COMST.2018.2816042.

[6] Rahman M, Iqbal S, Gao J. Load balancer as a service in cloud computing. In *Proc. the 8th International Symposium on Service Oriented System Engineering*, Apr. 2014, pp.204–211. DOI: 10.1109/SOSE.2014.31.

[7] Noshy M, Ibrahim A, Ali H A. Optimization of live virtual machine migration in cloud computing: A survey and future directions. *Journal of Network and Computer Applications*, 2018, 110: 1–10. DOI: 10.1016/j.jnca.2018.03.002.

[8] Song X, Ma Y F, Teng D. A load balancing scheme using federate migration based on virtual machines for cloud simulations. *Mathematical Problems in Engineering*, 2015, 2015: 506432. DOI: 10.1155/2015/506432.

[9] Xu M X, Tian W H, Buyya R. A survey on load balancing algorithms for virtual machines placement in cloud computing. *Concurrency and Computation: Practice and Experience*, 2017, 29(12): e4123. DOI: 10.1002/cpe.4123.

[10] Ghomi E J, Rahmani A M, Qader N N. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 2017, 88: 50–71. DOI: 10.1016/j.jnca.2017.04.007.

[11] Thakur A, Goraya M S. A taxonomic survey on load balancing in cloud. *Journal of Network and Computer Applications*, 2017, 98: 43–57. DOI: 10.1016/j.jnca.2017.08.020.

[12] Kumar P, Kumar R. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Computing Surveys*, 2019, 51(6): Article No. 120. DOI: 10.1145/3281010.

[13] Thiruvenkadam T, Kamalakkannan P. Energy efficient multi dimensional host load aware algorithm for virtual machine placement and optimization in cloud environment. *Indian Journal of Science and Technology*, 2015, 8(17): 1–11. DOI: 10.17485/ijst/2015/v8i17/59140.

[14] Cho K M, Tsai P W, Tsai C W, Yang C S. A hybrid meta-heuristic algorithm for VM scheduling with load balancing in cloud computing. *Neural Computing and Applications*, 2015, 26(6): 1297–1309. DOI: 10.1007/s00521-014-1804-9.

[15] Xu F, Liu F M, Liu L H, Jin H, Li B, Li B C. iAware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Trans. Computers*, 2014, 63(12): 3012–3025. DOI: 10.1109/TC.2013.185.

[16] Zhou Z, Liu F M, Zou R L, Liu J C, Xu H, Jin H. Carbon-aware online control of geo-distributed cloud services. *IEEE Trans. Parallel and Distributed Systems*, 2016, 27(9): 2506–2519. DOI: 10.1109/TPDS.2015.2504978.

[17] Liu F M, Zhou Z, Jin H, Li B, Li B C, Jiang H B. On arbitrating the power-performance tradeoff in SaaS clouds. *IEEE Trans. Parallel and Distributed Systems*, 2014, 25(10): 2648–2658. DOI: 10.1109/TPDS.2013.208.

[18] Tian W H, Xu M X, Chen Y, Zhao Y. Prepartition: A new paradigm for the load balance of virtual machine reservations in data centers. In *Proc. the 2014 IEEE International Conference on Communications*, Jun. 2014, pp.4017–4022. DOI: 10.1109/ICC.2014.6883949.

[19] Wen W T, Wang C D, Wu D S, Xie Y Y. An ACO-based scheduling strategy on load balancing in cloud computing environment. In *Proc. the 9th International Conference*

on Frontier of Computer Science and Technology, Aug. 2015, pp.364–369. DOI: 10.1109/FCST.2015.41.

[20] Chhabra S, Singh A K. Optimal VM placement model for load balancing in cloud data centers. In *Proc. the 7th International Conference on Smart Computing & Communications*, Jun. 2019. DOI: 10.1109/ICSCC.2019.8843607.

[21] Bala A, Chana I. Prediction-based proactive load balancing approach through VM migration. *Engineering with Computers*, 2016, 32(4): 581–592. DOI: 10.1007/s00366-016-0434-5.

[22] Ebadifard F, Babamir S M. A PSO-based task scheduling algorithm improved using a load-balancing technique for the cloud computing environment. *Concurrency and Computation: Practice and Experience*, 2018, 30(12): e4368. DOI: 10.1002/cpe.4368.

[23] Ray K, Bose S, Mukherjee N. A load balancing approach to resource provisioning in cloud infrastructure with a grouping genetic algorithm. In *Proc. the 2018 International Conference on Current Trends Towards Converging Technologies*, Mar. 2018. DOI: 10.1109/ICCTCT.2018.8550885.

[24] Deng W, Liu F M, Jin H, Liao X F, Liu H K. Reliability-aware server consolidation for balancing energy-lifetime tradeoff in virtualized cloud datacenters. *International Journal of Communication Systems*, 2014, 27(4): 623–642. DOI: 10.1002/dac.2687.

[25] Kleinberg J, Tardos É. Algorithm Design. Pearson/Addison-Wesley, 2006.

[26] Emeras J, Varrette S, Plugaru V, Bouvry P. Amazon Elastic Compute Cloud (EC2) versus in-house HPC platforms: A cost analysis. *IEEE Transaction on Cloud Computing*, 2019, 7(2): 456–468. DOI: 10.1109/TCC.2016.2628371.

[27] Knauth T, Fetzer C. Energy-aware scheduling for infrastructure clouds. In *Proc. the 4th IEEE International Conference on Cloud Computing Technology and Science*, Dec. 2012, pp.58–65. DOI: 10.1109/CloudCom.2012.6427569.

[28] Graham R L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 1969, 17(2): 416–429. DOI: 10.1137/0117039.

[29] Tian W H, Zhao Y, Zhong Y L, Xu M X, Jing C. A dynamic and integrated load-balancing scheduling algorithm for Cloud datacenters. In *Proc. the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, Sept. 2011, pp.311–315. DOI: 10.1109/CCIS.2011.6045081.

[30] Tian W H, Zhao Y, Xu M X, Zhong Y L, Sun X S. A toolkit for modeling and simulation of real-time virtual machine allocation in a cloud data center. *IEEE Trans. Automation Science and Engineering*, 2015, 12(1): 153–161. DOI: 10.1109/TASE.2013.2266338.

[31] Gulati A, Shanmuganathan G, Holler A, Ahmad I. Cloud-scale resource management: Challenges and techniques. In *Proc. the 3rd USENIX Conference on Hot Topics in Cloud Computing*, Jun. 2011, Article No. 3. DOI: 10.5555/2170444.2170447.

[32] Feitelson D, Tsafrir D, Krakov, D. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 2014, 74(10): 2967–2982. DOI: 10.1016/j.jpdc.2014.06.013.

[33] Xu M X, Tian W H. An online load balancing scheduling algorithm for cloud data centers considering real-time multi-dimensional resource. In *Proc. the 2nd International Conference on Cloud Computing and Intelligence Systems*, Oct. 30–Nov. 1, 2012, pp.264–268. DOI: 10.1109/CCIS.2012.6664409.

Wen-Hong Tian received his Ph.D. degree in computer science from the Department of Computer Science, North Carolina State University, Raleigh, in 2007. He is now a professor at the University of Electronic Science and Technology of China, Chengdu. His research interests include scheduling in cloud computing and big data platforms, image recognition by deep learning, and parallel training of large-scale model and evolution algorithms. He has more than 110 journal/conference publications and five books in related areas. He is a senior member of CCF and a member of ACM and IEEE.

Min-Xian Xu is currently an associate professor at Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen. He received his B.Sc. degree in 2012 and his M.Sc. degree in 2015, both in software engineering from University of Electronic Science and Technology of China, Chengdu. He obtained his Ph.D. degree in computer science from the University of Melbourne, Melbourne, in 2019. His research interests include resource scheduling and optimization in cloud computing. He has co-authored about 40 peer-reviewed papers published in prominent international journals and conferences. His Ph.D. thesis was awarded the 2019 IEEE TCSC Outstanding Ph.D. Dissertation Award.

Guang-Yao Zhou received his Bachelor's degree in 2014 and his Master's degree in 2017 from School of Architectural Engineering, Tianjin University, Tianjin. He is now a Ph.D. candidate at School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, majoring in software engineering. His current research interests include resource scheduling of cloud computing, facial expressions recognition, and theory of deep reinforcement learning.

Kui Wu entered the Special Class for the Gifted Young of Wuhan University, Wuhan, in 1985. He received his B.Sc. and his M.Sc. degrees in computer science from Wuhan University, Wuhan, in 1990 and 1993, respectively, and his Ph.D. degree in computing science from the University of Alberta, Edmonton, in 2002. He worked as a project manager in Haven Software Inc. (Hong Kong) from 1995 to 1998. He joined the Department of Computer Science at the University of Victoria, Victoria, in 2002, and is currently a professor there. He was a JSPS Fellow at the University of Tsukuba, Tsukuba, in 2009, a visiting professor at City University of Hong Kong, Hong Kong, in 2009 and 2019, and a visiting professor at Norwegian University of Science and Technology, Trondheim, in 2008 and 2019. His current research interests include network performance analysis, online social networks, Internet of Things, and parallel and distributed algorithms.

Cheng-Zhong Xu is the dean of Faculty of Science and Technology and the director of Institute of Collaborative Innovation, University of Macau, Macau, and a chair professor of computer and information science. Dr. Xu's main research interests lie in parallel and distributed computing and cloud computing, in particular, with an emphasis on resource management for system's performance, reliability, availability, power efficiency, and security, and in big data and data-driven intelligence applications in smart city and self-driving vehicles. He published two research monographs and more than 300 peer-reviewed papers in journals and conference proceedings. His papers received about 15 000 citations with an H-index of 66. He obtained his B.Sc. and M.Sc. degrees from Nanjing University, Nanjing, in 1986 and 1989 respectively, and his Ph.D. degree from the University of Hong Kong, Hong Kong, in 1993, all in computer science and engineering.

Rajkumar Buyya is a Redmond Barry Distinguished Professor of School of Computing and Information Systems and the director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Melbourne. He has authored over 625 publications and seven text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=160, g-index=350, 140 400+ citations). Dr. Buyya is recognized as a "Web of Science Highly Cited Researcher" for four consecutive years since 2016, a fellow of IEEE, and Scopus Researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to cloud computing. (For further information on Dr. Buyya, please visit his cyberhome: www.buyya.com.)