W. Tian, M. Xu, G. Zhou *et al.* Journal of computer science and technology: Instruction for authors. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 1–21 September 2021. DOI 10.1007/s11390-015-0000-0

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

Wenhong Tian[1], Minxian Xu[2,*], Guangyao Zhou[1], Kui Wu[3], Chengzhong Xu[4] and Rajkumar Buyya[1,5]

[1] *University of Electronic Science and Technology of China, Chengdu 610054, China*

[2] *Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China*

[3] *University of Victoria, Victoria, BC, V8W 3P6, Canada*

[4] *State Key Lab of IOTSC, University of Macau, Macau 999078, China*

[5] *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 July 15, 2018 [Month Day, Year]; accepted October 14, 2018 [Month Day, Year].

Abstract Load balancing is vital for the efficient and long-term operation of cloud data centers. With virtualization, post (reactive) migration of virtual machines after allocation is the traditional way for load balancing and consolidation. However, reactive migration is not easy to obtain predefined load balance objectives and 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 Prepartition for offline (PrepartitionOff) scheduling has 10%-20% better performance than the existing load balancing algorithms 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 existing online algorithms.

Keywords Cloud Computing, Physical Machines, Virtual Machines, 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 is always faced with a difficult tradeoff 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 expenses on 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 tradeoff is datacenter load balancing.

Due to the importance of data center load balanc-

Regular Paper

This work is supported by Key-Area Research and Development Program of Guangdong Province (NO. 2020B010164003), National Natural Science Foundation of China (NO. 62102408) and SIAT Innovation Program for Excellent Young Researchers.

*Corresponding Author

① https://jcst.ict.ac.cn/EN/column/column107.shtml, May 2020.

ing, 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:

1. initializing requests: user submits a VM request through a providers' web portal.

2. matching suitable resources: based on the user's feature (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 resource is allocated to the request.

3. Sending feedbacks (e.g., whether or not the request has been satisfied) to users.

4. Scheduling tasks: determine 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.

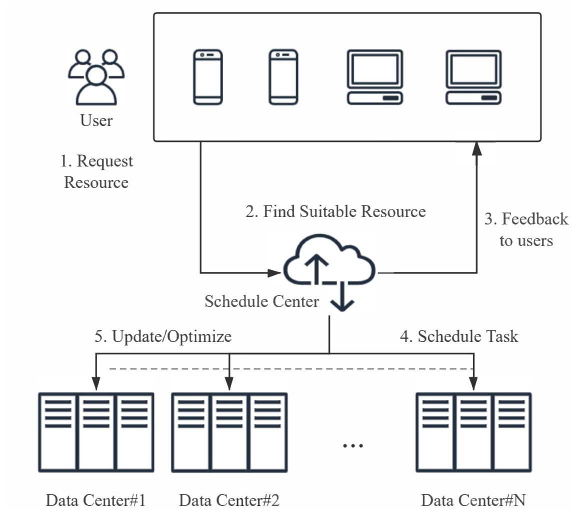


Fig.1. A high-level view on resource scheduling/load balancing in cloud data centers.

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 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 the problem of load balancing from a different angle. Starkly different from previous methods, it handles the VM load balance in a more proactive way.

Fig. 2 shows the illustrative example based on the

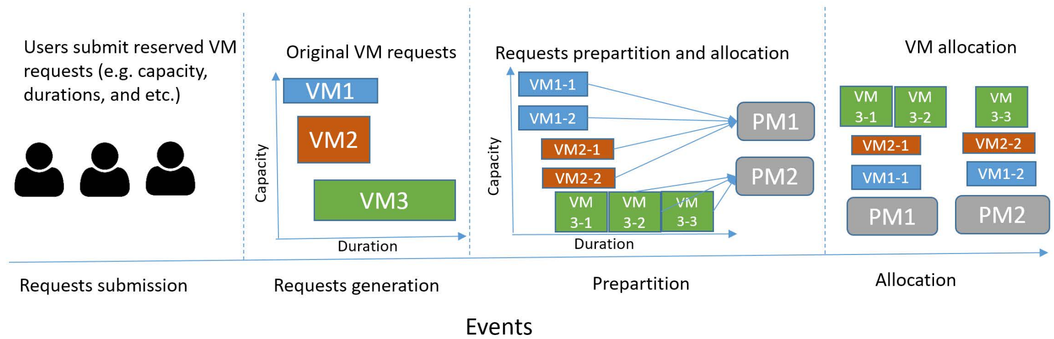


Fig.2. Illustrative example for Prepartition

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 that the original VM requests can be partitioned into sub-requests and allocated to PMs before the VM migration stage, for instance, VM1 is partitioned as VM1-1 and VM1-2, and allocated to PM1. And finally, to further optimize VM locations, VM migrations can be further applied.

As the prepartition process happens before the final requests generation stage, and it does not need to execute the job, therefore, the costs are rather low compared with the overall job execution. The prepartition costs 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. seconds), which are 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 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 PMs[1]. 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 algorithms for both offline and online scheduling which can prepare migration in advance and set process time bound

[1]Note 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 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.

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.
- Carrying out performance evaluation in terms of average utilization, imbalance degree, makespan, time costs as well as Capacity_makespan (a metric to represent loads, more 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 the related work on load balancing in cloud data centers, and Section 3 introduces problem formulation. Section 4 presents Prepartition 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 in most works. Xu et al. [20] had a survey for the state-of-art VM placement algorithms. Ghomi et al. [21] 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 et al. [22]. Noshy et al. [23] reviewed the latest optimization technology dedicated to developing live VM migration. They also emphasized a further investigation, which aims to optimize the virtual machines migration process. Kumar et al. [24] conducted a survey to discuss 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 loading balancing: Song et al. [7] proposed a VM migration method to dynamically balance VM loads for high-level application federations. Thiruvenkadam et al. [8] showed a hybrid genetic VMs balancing algorithm, which aims to minimize the number of migrations. Cho et al. [10] tried to maximize the balance of loads in cloud computing by combing ant colony with particle swarm optimization. Xu et al. [16] proposed iWare, which is a lightweight interference model for VM migration. The iWare can capture the relationship between VM performance interference and the important factors. Zhou et al. [18] 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. [19] 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. [11] 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. [9] proposed a distributed VM load balancing strategy with the goals of utilizing resources in a balanced manager and minimizing the number of migrations. By estimating resource usage, Chhabra et al. [12] developed a virtual machines placement method for loading balancing according to maximum likelihood estimation for parallel and distributed applications. Bala et al. [13] presented an approach to improving proactive load balancing by predicting multiple resource types in the cloud. In Ebadifard et al. [14], a task scheduling approach derived from

Table 1. The comparison of closely related work

Approach	**Algorithm Type**		**VM Type**		**Resource Type**		**Theoretical Analysis**	**Metrics**					
	Online	Offline	Heterogeneous	Homogeneous	Single	Multiple		Utilization	Imblance Degree	Makespan	Time Cost	Capacity_makespan	SLA violations
Song et al. [7]	✓			✓	✓			✓					
Thiruvenkadam et al. [8]	✓		✓			✓	✓	✓					
Wen et al. [9]		✓	✓			✓							✓
Cho et al. [10]		✓	✓			✓		✓					
Tian et al. [11]		✓	✓			✓	✓		✓	✓			
Chhabra et al. [12]	✓		✓		✓			✓	✓				
Bala et al. [13]		✓	✓			✓		✓	✓				✓
Ebadfard et al. [14]		✓	✓			✓	✓	✓		✓			
Ray et al. [15]		✓		✓		✓			✓				
Xu et al. [16]	✓		✓			✓		✓			✓		
Deng et al. [17]		✓	✓			✓					✓		✓
Zhou et al. [18]	✓			✓	✓		✓						✓
Liu et al. [19]	✓		✓		✓		✓				✓		✓
Our Approach Prepartition	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	

particle swarm optimization algorithm has been proposed. In their work, tasks are independent and non-preemptive. Ray et al. [15] presented a genetic-based load balancing approach to distribute VM requests uniformly among the physical machines. Deng et al. [17] introduced a server consolidation approach to achieve energy efficient server consolidation in a reliable and profitable manner.

Different from all the above methods, 1) we investigated the reservation model that makespan and VM capacity are considered together for optimization rather than only considering the makespan or capacity separately. 2) Our approach can be applied to both online and offline scenarios rather than for a single scenario. 3) We also performed theoretical analysis for the proposed approach, and 4) evaluated more performance metrics. A qualitative comparison between our method and others is listed in Table 1.

3 Problem description and formulation

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

3.1 Assumptions

The key assumptions are:

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 end time f_i are integer numbers of one slot. Then the interval of demand can be expressed in slot fashion with (start time, end time). For instance, if sl_0=10 minutes, an interval (5, 12) represents that it starts at the 5th time slot and finishes at the 12th time slot. The duration of this demand is (12-5)×10=70 minutes.

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

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

Table 2. Key notations in our models

Notations	Definitions	Notations	Definitions
T	The whole observation time period	sl_0	The length of each time slot
n	The maximum number of requests	s_i	The start time of request i
f_i	The finishing time of request i	$A(i)$	The set VMs requests scheduled to PM i
d_j	The capacity demand of VM j	CM_j^r	The capacity_makespan of VM request j
CM_i	The capacity_makespan of PM i	$PCPU_i$	The CPU capacity of PM i
$PMem_i$	The memory capacity of PM i	$PSto_i$	The storage capacity of PM i
$VCPU_j$	The CPU demand of VM j	$VMem_j$	The memory demand of VM j
$VSto_j$	The storage demand of VM j	T_j^{start}	The start time of VM request j
T_j^{end}	The finishing time of VM request j	T_r	The time span between time slot t_{r-1} and t_r
CM^p	The maximum capacity_makespan of all PMs	IMD	The imbalance degree
k	The partition value	P_0	The lower bound of the optimal solution OPT
B_d	The dynamic balance value based on capacity_makespan	L	The amount of VM requests that already arrived
m	The number of PMs in use	I	A set of VM requests
CM_b	The predefined capacity_makespan threshold for partition	f	Constant value to avoid too frequent partitions

3.2 Key Definitions

A few key definitions are given here:

Definition 1. *Traditional interval scheduling problem (TISP) with fixed processing time*: A batch of demands $\{1, 2, \ldots, n\}$ where the i-th demand refers to an interval of time starting at s_i and finishing at f_i ($\forall i, s_i < f_i$), each one requires a capacity of 100%, i.e. utilizing the full capacity of a server during that interval.

Definition 2. *Interval scheduling with capacity sharing (ISWCS)*: Difference 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, for short, 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, *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. *The Capacity_makespan of a PM i*: In the schedule of VM requests to PMs, denote $A(i)$ as the set of VM requests scheduled to PM_i. With this scheduling, PM_i will have load as the sum of the product of each requested capacity and its duration, called Capacity_makespan, abbreviated as CM, as follows:

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

in which 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, it can be simplified as a capacity based on Assumption 3), and t_j for the span of demand j, being the length of processing time of 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 Eq. (2). Considering m PMs are in the data center, we can form the problem as:

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

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

in which d_j is the capacity demand of VM j and the whole capacity of a PM i is normalized to 1. The condition (3) shows the sharing capacity constraint that in any time interval, the shared resources should not use up all the provisioned resources (100%).

From this form, we see that lifecycle and capacity sharing are key differences from traditional metrics like makespan which focuses on process time. Traditionally Longest Process Time first (LPT) [28] is widely adopted to load balance offline multi-processor scheduling. Reactive migration of VMs is another way to compensate after allocation.

3.4 Metrics for ISWCS load balancing

A few key metrics for ISWCS load balancing are given in the following. Other metrics are the same as given in [29].

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

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

3) Discrete time: Considering a time span be partitioned into equal length of slots. The s slots can be considered as $[(t_0, t_1), (t_1, t_2), \ldots, (t_{s-1}, t_s)]$, 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 is defined as:

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

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 $PCPU_i^{T_r}$ can be obtained by monitoring CPU utilization in slot T_r. Average memory utilization ($PMem_i^U$) and storage utilization ($PSto_i^U$) of PMs can be calculated similarly. Similarly, the average CPU (memory and storage) utilization of a VM can be calculated.

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

6) The maximum Capacity_makespan (CM^p) of all PMs: is calculated as:

$$CM^p = \max_i (CM_i) \tag{5}$$

where we can apply CPU, memory and storage utilization too.

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

$$\frac{\sum_{i=0}^{m}(\frac{(Avg_i-CPU_u)^2}{3}+\frac{(Avg_i-Mem_u)^2}{3}+\frac{(Avg_i-Sto_u)^2}{3})}{m} \tag{6}$$

where

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

and CPU_u, Mem_u,Sto_u is respectively the average utilization of CPU, memory and storage in a Cloud data center during consideration 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.
The proof was provided in our previous work [29] and is omitted here. Our model in this paper differs from the previous one in several perspectives: 1). we model

[2] in this paper, we interchange demands and requests, both of them are referred to VM requests

that the multiple VM requests are allowed to be executed on the same host simultaneously rather than a single VM request; 2) Our objective is minimizing the Capacity_makespan rather than the longest processing time; 3) we extend our model to be suitable for both offline and online scenarios.

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

4 Prepartition Algorithms

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

4.1 PrepartitionOff Algorithm

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. In this way, 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 denote OPT as the optimal solution with regard to minimizing the Capacity_makespan. Firstly define

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

where J denotes the total number of allocated VMs, and P_0 denotes the lower bound for the OPT.

Algorithm 1 gives the pseudocodes of PrepartitionOff algorithm which measures the ideal load balancing among m PMs. The algorithm firstly calculates the balancing value by formula (9), sets a partition value (k) and computes the length of each partition, i.e. $\lceil P_0/k \rceil$, representing the maximum CM that a VM can continuously 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 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 Capacity_makespan and enough available resources (lines 6-8), thus achieving the load balancing objective. The Capacity_makespan of PM will also be updated accordingly (line 9). Finally, the algorithm calculates the Capacity_makespan of each PM when all requests are assigned and finds total partition numbers (line 10). In practice, the scheduler records all possible subintervals and their hosting PMs so that migrations of VMs can be prepared beforehand to alleviate overheads.

Theorem 2. Applying priority queue data structure, the PrepartitionOff algorithm has a computational complexity of $O(nlogm)$, 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 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 $O(logn)$ 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 $O(logm)$ time. In total, the time complexity of the PrepartitionOff algorithm is $O(nlogm)$ for n demands.

Theorem 3. PrepartitionOff algorithm has approximation ratio of $(1 + \epsilon)$ regarding the capac-

Input: m:total number of PMs; n: total number of VM requests, requests are indicated by their (demanded VM IDs, start times, finishing times, demanded capacity), CM_j^r is the Capacity_makespan of request j, CM_i for the Capacity_makespan of PM i;
Output: Assign PM IDs to all requests and their partitions

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

Algorithm 1: The pseudo codes of PrepartitionOff algorithm

ity_makespan where $\epsilon = \frac{1}{k}$ 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. Think the last job (later than all other jobs) to complete and assume the start time of this job is T_0. We also assume that all other servers are allocated with VM requests and denote the maximum Capacity_makespan as CM_m, this means $CM_m \leq$OPT. Since, for all requests $i \in J$, we have $CM_i \leq \epsilon$OPT (by the setting of PrepartitionOff algorithm in formula (9)), this job finishes with load (CM_m+ϵOPT). Therefore, the schedule with Capacity_makespan (CM_m+ϵOPT) $\leq$ (1+ϵ)OPT, this ends the proof.

4.2 PrepartitionOn1 Algorithm

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 offline Prepartition algorithm to the online scenario as PrepartitionOn1, 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. Firstly define

$$B_d = min(\max_{1 \leq j \leq L}(CM_j^r)/2, \sum_{j=1}^{L}(CM_j^r)/m) \qquad (9)$$

B_d is called dynamic balance value, which is one-half of the max Capacity_makespan of all current PMs or the ideal load balance value of all current PMs in the system, where L is the number of VMs requests already arrived. Notice that the reason to set B_d as one half of the max Capacity_makespan of all current PMs is to avoid that a large number of partitions may cause extra management costs.

Algorithm 2 shows the pseudocodes of the PrepartitionOn1 algorithm. Since in an online algorithm, the requests come one by one, the system can only capture the information of arrived requests. The algorithm firstly predefines the prepartition value 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 equation (10) (lines 3-4). After B_d is computed, if Capacity_maskespan of VM request is too large (larger than $\lceil(B_d/k)\rceil$), 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 5-6). The VM requests with small Capacity_maskespan after partition would be executed when their start time begins, which will be assigned to the PM that has the lowest value of Capacity_makespan (lines 7-9). Once all demands are allocated, PrepartitionOn1 calculates the Capacity_makespan value of all the PMs and outputs all the partition values for n demands (line 10). Since the number 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.

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. PrepartitionOn1 has a competitive ratio of $(1 + \frac{1}{k} - \frac{1}{mk})$ with regarding to the Capacity_makespan.

Proof: Without loss of generality, we label PMs in order of non-decreasing final loads (CM) in PrepartitionOn1. Denote OPT and $PrepartitionOn1(I)$ as the optimal load balance value of corresponding offline scheduling and load balance value of PrepartitionOn1 for a given set of jobs I, respectively. Then the load of PM_m defines the Capacity_makespan. The first (m-1) PMs each process a subset of the jobs and then experience an (possibly none) idle period. All PMs together finish a total Capacity_makespan $\sum_{i=1}^{n} CM_i$ during their busy periods. Consider the allocation of the last job j to PM_m. By the scheduling rule of PrepartitionOn1, PM_m had the lowest 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., $\frac{\max_{1\leq i\leq n} CM_i}{k}$. Based on Equation (10), then we have

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

which is equivalent to

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

which is

$$PrepartitionOn1(I) \leq (OPT + (\frac{1}{k} - \frac{1}{mk})OPT) \tag{12}$$

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

Theorem 5. By using the priority queue, the computational complexity of PrepartitionOn1 is $O(nlogm)$, where n is the number of VM requests after the prepartition operations and m is the total number of used PMs.

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

Input: m: total number of PMs; n: total number of VM requests, requests are indicated by their (demanded VM IDs, start times, finishing times, demanded capacity), CM_j^r is the Capacity_makespan of request j, CM_i for the Capacity_makespan of PM i;
Output: Assign PM IDs to all requests and their partitions

Set the partition value k, total partition number P=0;
for each arrived job j **do**
  Pick up the VM with the start time equals to system time in the VM queue to schedule; Compute CM_j^r of VM j, and B_d using Eq. (10);
  if $CM_j^r > \lceil (B_d/k) \rceil$ **then**
    partition VM_j into multiple $\lceil (B_d/k) \rceil$ equal subintervals, treat each subinterval as a new demand and add them into VM queue, $P = P + \lceil \frac{CM_j^r}{B_d/k} \rceil$, Update load ($CM$) of the PM;
  else
    Allocate j to PM with the smallest load and enough capacity;
    Update load (CM) of the PM;
Output total number of partitions P

Algorithm 2: PrepartitionOn1 Algorithm

4.3 PrepartitionOn2 Algorithm

Observing that the PrepartitionOn1 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 from PrepartitionOn1 are the followings:

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

2) Setting a CM bound for each PM, for instance, each PM has a CM=1×24 in each day within 24 hours, 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_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 multiple 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 smallest Capacity_makespan value, and the Capacity_makespan of PM as well as the PM with the smallest Capacity_makespan are calculated with that supposition (lines 5-7). If the increased Capacity_makespan of 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 directed to the PM with the smallest loads (lines 10-11). Finally, the scheduling results and number of partitions can be obtained (line 12).

Theorem 6. PrepartitionOn2 has a computational complexity of $O(nlogm)$ 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, we therefore omit it.

Theorem 7. The competitive ratio of PrepartitionOn2 is at most $(1+f)$ and each PM has CM at most CM_B with regard to the Capacity_makespan.

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 PrepartitionOn2 is at most $(1+f)$. This completes the proof.

5 Performance Evaluation

Notice that there are three types of PMs in Table 4 and 8 types VMs in Table 3, 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 future, we will extend to other cases. In the following, the simulation results of 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

All simulations ran on a computer configured with an Intel i5 processor at 2.5GHz and 4GB memory. All VM requests are generated by following Normal distribution. In offline algorithms, Round-Robin (RR) algorithm, Longest Process Time (LPT) algorithm and Post Migration Algorithm (PMG) are implemented and compared.

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

2) **Longest Processing Time first (LPT)**: LPT is one of the best practices for offline scheduling algorithms without migration, which has an approximate 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 smallest load in the sorted order. The smallest load indicates the smallest Capacity_makespan among all the PMs.

3) **Post Migration algorithm (PMG)**: PMG algorithm comes from the VMware DRS algorithm [31], which adopts migrations to achieve load balance regarding makespan. In the beginning, it allocates the demands 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 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 (can be configured dynamically to meet the demands), which represents up-threshold is 1.1 times of the average Capacity_makespan and the low-threshold is 0.9 multiples the average Capacity_makespan. The algorithm also maintains a migration list containing the VMs on the PMs with higher Capacity_makespan value than the low-threshold. The VM migrations are triggered to make the PM 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 smallest Capacity_makespan in sequence to balance the loads.

VMs and PMs have the same configuration with Amazon EC2. The configurations are shown in Table

Input: m: total number of PMs; n: total number of VM requests, requests are indicated by their (demanded VM IDs, start times, finishing times, demanded capacity), CM_j^r is the Capacity_makespan of request j, CM_i for the Capacity_makespan of PM i;
Output: Assign PM IDs to all requests and their partitions

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

Algorithm 3: PrepartitionOn2 Algorithm

Table 3. 8 types of VMs derived from Amazon EC2

Compute Capacity (Units)	Memory (GB)	Storage (GB)	VM Type
1	1.875	211.25	1-1(1)
4	7.5	845	1-2(2)
8	15	1690	1-3(3)
6.5	17.1	422.5	2-1(4)
13	34.2	845	2-2(5)
26	68.4	1690	2-3(6)
5	1.875	422.5	3-1(7)
20	7	1690	3-2(8)

3 and Table 4, in which one unit of compute capacity equals to around 1.0 - 1.2 GHz 2007 Xeon or 2007 Opteron processors [26].

Remarks: We adopted the typical recommended VM types suggested by Amazon EC2. EC2 has a variety of VM types, and it classifies them as General Purpose, Compute Optimized (computational intensive VMs), Memory Optimized (memory-intensive VMs), Storage Optimized (storage-intensive VMs). Although we adopted EC2 classification, our approach can still be extended to other classifications.

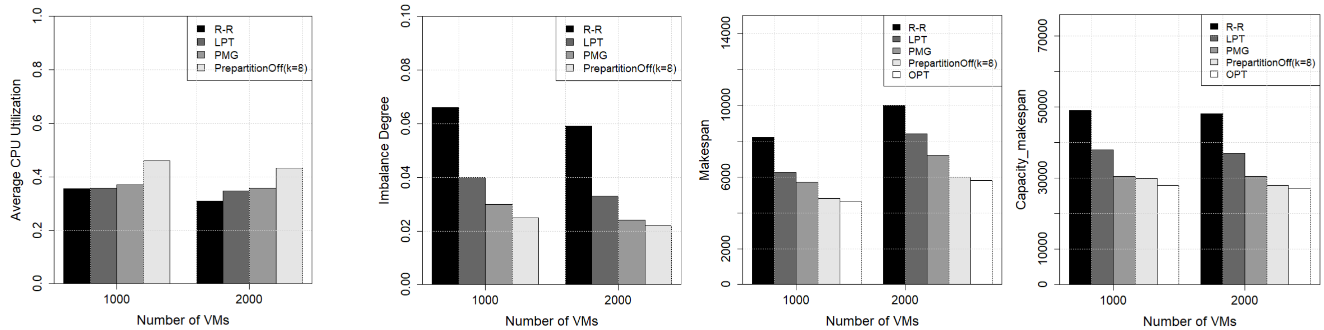
5.1.1 *Replay with ESL Data Trace*

To reflect realistic data generation, we utilize the data derived from 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 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 minutes, the 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

Table 4. 3 types of suggested PMs specification

PM Type	Compute Capacity (Units)	Memory (GB)	Storage (GB)
1	16	30	3380
2	52	136.8	3380
3	40	14	3380



(a) (b) (c) (d)

Fig.3. The offline algorithm comparison of (a) average utilization with ESL trace; (b) imbalance degree with ESL trace; (c) makespan with ESL trace; (d) Capacity_makespan with ESL trace

Capacity_makespan. According to the results, we can observe that the PrepartitionOff algorithm can achieve better performance than other algorithms in four aspects. For average utilization, the PrepartitionOff algorithm is 10%-20% higher than PMG, LPT, and Random-Robin (RR). 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 number of total PMs. For makespan and Capacity_makespan, the PrepartitionOff algorithm is 10%-20% lower than PMG and LPI, 30%-40% lower than Random-Robin (RR). 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 methods to balance loads, however, it can not assure a bounded or predefined target.

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

The main reason is PrepartitionOff takes actions in a much more refined and desired scale by pre-partition based on reservation data while PMG is just a best-effort trial by migration. The reason is that PrepartitionOff is much more precise and desired with the aid of pre-partition while PMG is just a trial to balance load as much as possible. To compare imbalance degree (IMD) change as time goes, we also did the tests about consecutive imbalance degree using 1000 and 2000 VMs among 4 different offline algorithms. In Fig 4, we provide the consecutive imbalance degree comparison for four algorithms in offline scheduling with 1000 VMs and 2000VMs respectively. In these two figures, the X-axis is for makespan and Y-axis is for imbalance degree. We can see that PrepartitionOff (with k=8) has the smallest makespan and smallest 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.

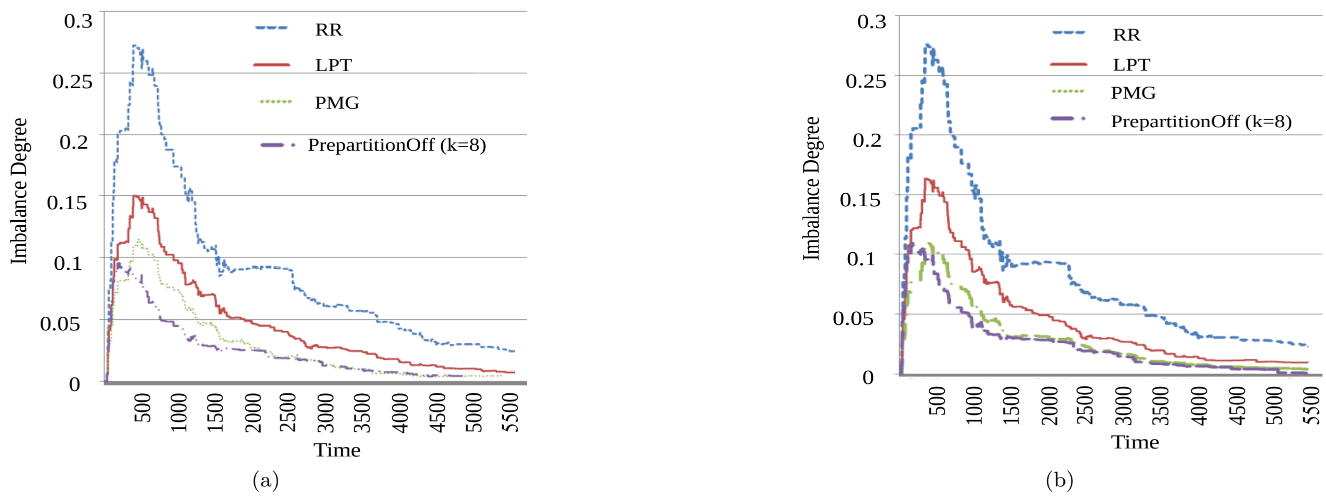


Fig.4. The consecutive imbalance degree under 1000 VMs among 4 different offline algorithms, where x-axis is for time and y-axis for imbalance degrees (a) 1000 VMs; (b) 2000 VMs.

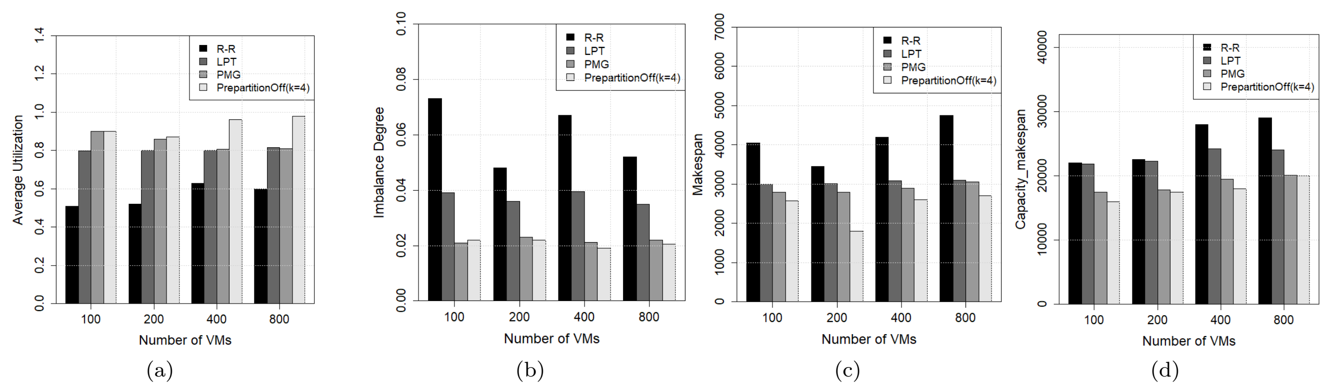


Fig.5. The comparison of offline algorithm with (a) average utilization; (b) imbalance degree; (c) makespan; (d) capacity_makespan with Normal distribution

5.1.2 *Results Comparison by Synthetic Data*

We configure the time slot to be 5 minutes as mentioned before, so 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 comparison results are collected. For data collection, first we set k of PrepartitionOff algorithm as 4 (we configure the value as 4 because in previous research [11], this value has been validated to be an effective value to improve performance). And the different types of VM are with equal probabilities. We also vary the number of VMs from 100, 200, 400 and 800 to analyze the trend. Each data set is an average of 10-runs.

Fig. 5 displays the comparison of different algorithms in average utilization, makespan and Capacity_makespan. From these figures, we can know that the PrepartitionOff algorithm is 10%-20% higher than PMG and LPT for average utilization, 40%-50% higher than Random-Robin (RR). As for makespan and Capacity_Makespan, the PrepartitionOff algorithm is 8%-13% lower than PMG and LPT, 40%-50% lower than Round-Robin (RR). We also note that the PMG algorithm can improve the performance of the LPT algorithm as it configures up-threshold and low-threshold based on Capacity_makespan value. LPT algorithm is better than the RR algorithm. Similar results are observed for the comparison of makespan.

5.2 PrepartitionOn1 Algorithm

We demonstrate the simulation results of the PrepartitionOn1 algorithm and the other three algorithms in this section. All VM requests are generated by following normal distribution, and Random, RR, Online Resource Scheduling Algorithm (OLRSA) [33] that has a good competitive ratio ($2-\frac{1}{m}$, where m is the number of PMs) for online algorithm has been implemented to compare 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 lowest Capacity_makespan and required resources.

5.2.1 Replay with ESL Data Trace

The ESL dataset aforementioned is also used in the experiments. Fig. 6 illustrates the comparisons of the average utilization, imbalance degree, makespan, Capacity_makespan. According to these figures, we can see that PrepartitionOn1 demonstrates the highest average utilization, the lowest imbalance degree, and the lowest makespan. As for Capacity_makespan, OLRSA has been proved much better performance compared with random and round-robin algorithms, and PrepartitionOn1 still improves 10%-15% in average utilization, 20%-30% in imbalance degree, and 5% to 20% in makespan than OLRSA.

5.2.2 Results Comparison by Synthetic Data

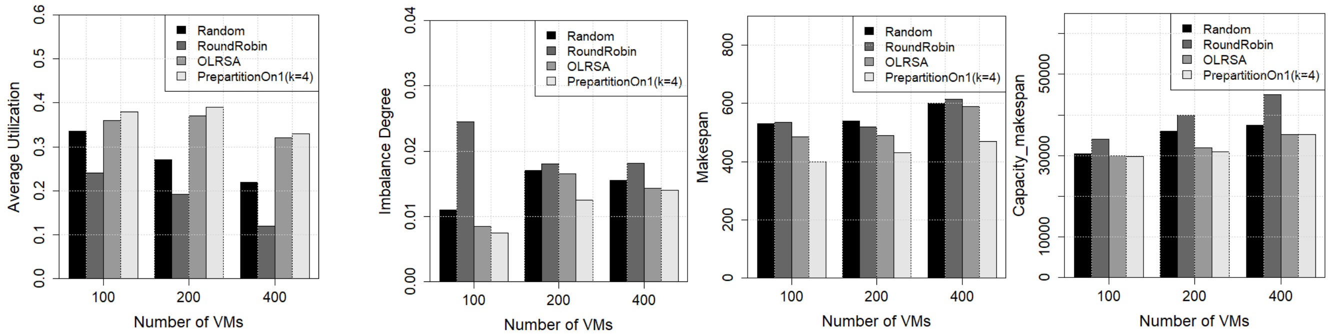
The requests are configured as same as in Section 5.1 based on the normal distribution. We set that VMs with different types have equal probabilities, and we modify the requests generation approach to produce different sizes of requests to trace the tendency. From Fig. 7, we can see that PrepartitionOn1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

LPT is one kind of the best methods for offline load balance algorithms without migration, which has an approximate ratio of 4/3. So we suggest setting the k value as 4, which can obtain an approximate ratio as $1+\ 1/k=\ 5/4$. Under this configuration, a better approximate ratio could be obtained. With higher k value, better load balancing effects could be achieved. While there exist tradeoffs between load balancing effect and time cost. For online load balance algorithms, we also suggest setting the k value as 4, and cloud service providers could reconfigure that value to be higher as suitable as the load balancing effects they desired.

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

5.3 PrepartitionOn2 Algorithm

In this part, we display the simulation results of the PrepartitionOn2 algorithm and the other three algorithms. Random, Round-Robin, Online Resource Scheduling Algorithm (OLRSA) [33] and PrepartitionOn2 Algorithm are implemented for comparison.



(a) (b) (c) (d)

Fig.6. The comparison of online algorithm with (a) average utilization; (b) imbalance degree; (c) makespan; (d) capacity_makespan with ESL trace

5.3.1 Replay with ESL Data Trace and Synthetic Trace

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, Capacity_makespan between PrepartitionOn2 and other online algorithms and the results show that PrepartitionOn2 performances best in terms of mentioned metrics.

In Fig. 9, we provide the consecutive imbalance degree comparison for four algorithms in online scheduling with 1000 VMs and 2000VMs respectively. In these two figures, the X-axis is for makespan and Y-axis is for imbalance degree. We can see that PrepartitionOn2 has the smallest makespan and smallest imbalance degree most of the time during tests.

The large k values 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 value k, PrepartitionOn1 algorithm with $k = 3$ takes about 10% less running time than that with $k = 4$, and $k = 2$ takes 15% less running time than that with $k = 4$. A larger k value will lead to a better load balance with a longer process time. We also observe that a larger k value will induce a lower Capacity_makespan value. Similarly, with a larger k value, larger average utilization, lower imbalance degree, and makespan are obtained.

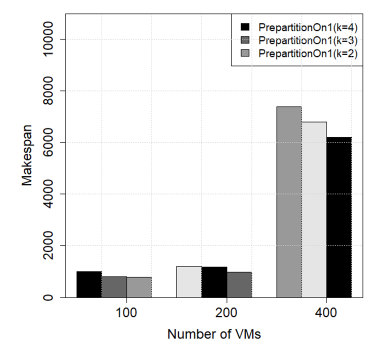
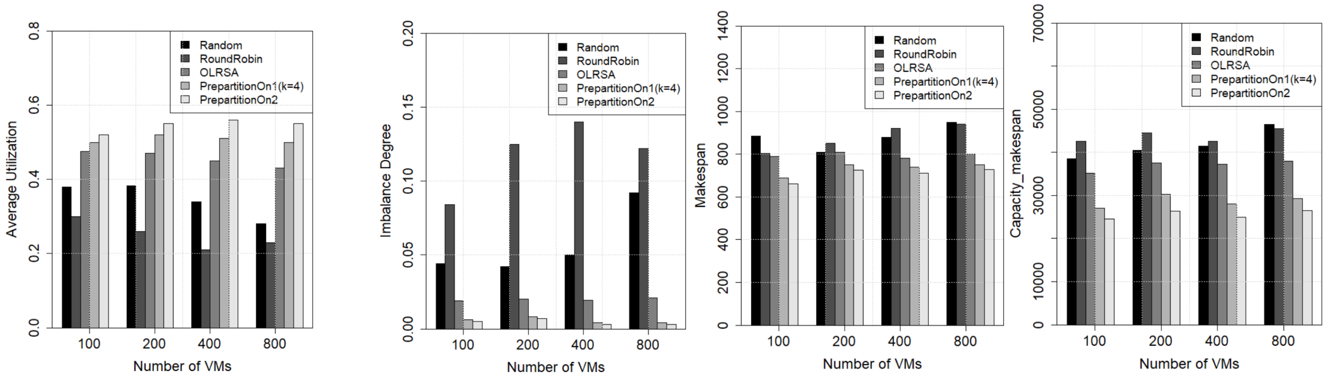


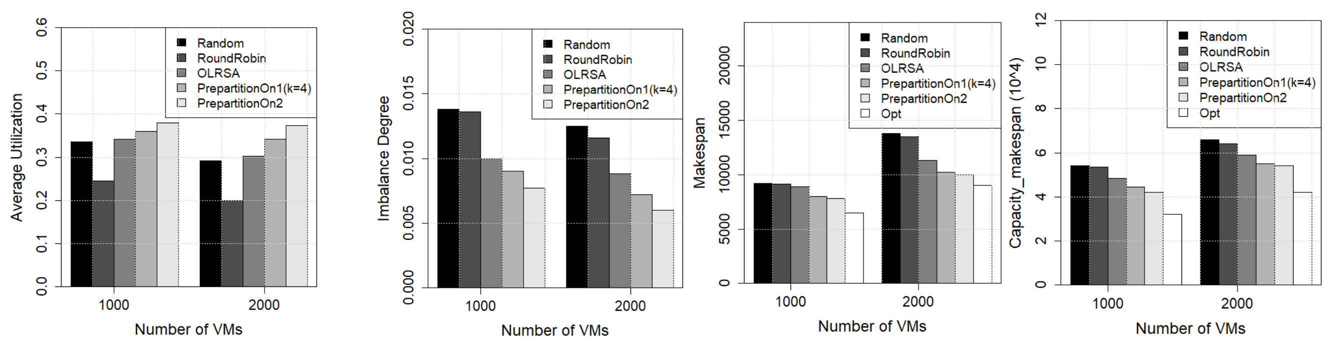
Fig.10. The comparison of time costs for PrepartitionOn1 by varying k values

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



(a) (b) (c) (d)

Fig.7. The online algorithm comparison of (a) average utilization; (b) imbalance degree; (c) makespan; (d) capacity_makespan with normal distribution



(a) (b) (c) (d)

Fig.8. The online algorithms and Prepartition algorithm comparison of (a) average utilization; (b) imbalance degree; (c) makespan; (d) capacity_makespan with ESL distribution

6 Conclusions and Future Work

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

- Making an appropriate choice between load bal-

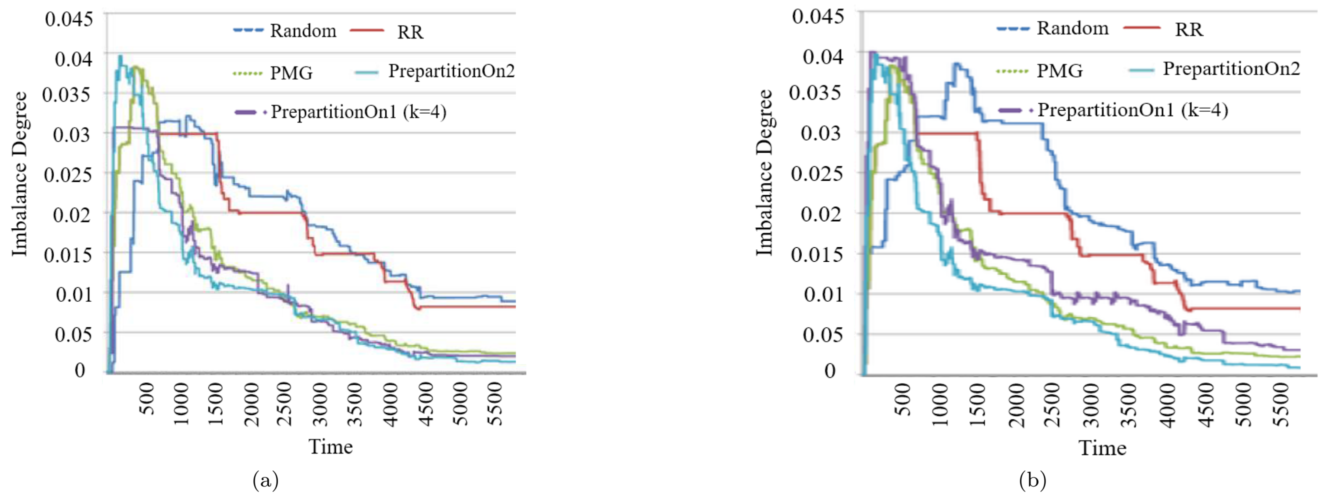


Fig.9. The consecutive imbalance degree under 1000 VMs among 5 different online algorithms, where x-axis is for time and y-axis for imbalance degrees (a) 1000 VMs; (b) 2000 VMs.

Table 5. Number of partitions in different algorithms

Algorithm	Number of partitions	
	1000 VMs	2000 VMs
PrepartitionOff	64	109
PrepartitionOn1	159	361
PrepartitionOn2	115	293

ance and total partition numbers. Prepartition algorithm can achieve desired load balance objective by setting suitable k value. It may need a large number of partitions so that the number of migrations can be large depending on the characteristics of VM requests. For example in EC2 [26], the duration of VM reservations varies from a few hours to a few months, we can classify different types of VMs based on their durations (Capacity_makespans) firstly, 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 number migration can be reduced.

- Considering the heterogeneous configuration 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 EC2 and Knauth et al. [27]. When this is not true, multi-dimensional resources, such as CPU, memory, and bandwidth, etc. have to be considered together or separately in the load balance, see [34] and [35] for a detailed discussion about considering multi-dimensional resources.

- Considering precedence constraints among different VM requests. In reality, some VMs may be more important than others depending on applications running on them, we would like to extend current algorithm to consider this case.

- Considering the multi-tenancy and resource contention when making prepartitions, which can be investigated by characterizing application features. 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 editors and anonymous reviewers' valuable comments to improve the quality of our work. This work is supported by Key-Area Research and Development Program of Guangdong Province (NO. 2020B010164003), National Natural Science Foundation of China (NO. 62102408) and SIAT Innovation Program for Excellent Young Researchers.

References

[1] Minxian Xu and Rajkumar Buyya. Brownout approach for adaptive management of resources and applications in cloud computing systems: a taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 52(1):1–27, 2019.

[2] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31, 2013.

[3] Sukhpal Singh Gill, Shreshth Tuli, Adel Nadjaran Toosi, Felix Cuadrado, Peter Garraghan, Rami Bahsoon, Hanan Lutfiyya, Rizos Sakellariou, Omer Rana, Schahram Dustdar, and Rajkumar Buyya. Thermosim: Deep learning based framework for modeling and simulation of thermal-aware resource management for cloud computing environments. *Journal of Systems and Software*, 166:110596, 2020.

[4] Minxian Xu and Rajkumar Buyya. Brownoutcon: A software system based on brownout and containers for energy-efficient cloud computing. *Journal of Systems and Software*, 155:91 – 103, 2019.

[5] Jiao Zhang, F Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials*, 20(3):2324–2352, 2018.

[6] Mazedur Rahman, Samira Iqbal, and Jerry Gao. Load balancer as a service in cloud computing. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 204–211. IEEE, 2014.

[7] Xiao Song, Yaofei Ma, and Da Teng. A load balancing scheme using federate migration based on virtual machines for cloud simulations. *Mathematical Problems in Engineering*, 2015, 2015.

[8] T Thiruvenkadam and P Kamalakkannan. Energy efficient multi dimensional host load aware algorithm for virtual machine placement and optimization in cloud environment. *Indian journal of science and technology*, 8(17):1–11, 2015.

[9] Wei-Tao Wen, Chang-Dong Wang, De-Shen Wu, and Ying-Yan Xie. An aco-based scheduling strategy on load balancing in cloud computing environment. In *2015 Ninth International Conference on Frontier of Computer Science and Technology*, pages 364–369. IEEE, 2015.

[10] Keng-Mao Cho, Pang-Wei Tsai, Chun-Wei Tsai, and Chu-Sing Yang. A hybrid meta-heuristic algorithm for vm scheduling with load balancing in cloud computing. *Neural Computing and Applications*, 26(6):1297–1309, 2015.

[11] Wenhong Tian, Minxian Xu, Yu Chen, and Yong Zhao. Prepartition: A new paradigm for the load balance of virtual machine reservations in data centers. In *2014 IEEE International Conference on Communications (ICC)*, pages 4017–4022. IEEE, 2014.

[12] Sakshi Chhabra and Ashutosh Kumar Singh. Optimal vm placement model for load balancing in cloud data centers. In *2019 7th International Conference on Smart Computing & Communications (ICSCC)*, pages 1–5. IEEE, 2019.

[13] Anju Bala and Inderveer Chana. Prediction-based proactive load balancing approach through vm migration. *Engineering with Computers*, 32(4):581–592, 2016.

[14] Fatemeh Ebadifard and Seyed Morteza Babamir. A pso-based task scheduling algorithm improved using a load-balancing technique for the cloud computing environment. *Concurrency and Computation: Practice and Experience*, 30(12):e4368, 2018.

[15] K. Ray, S. Bose, and N. Mukherjee. A load balancing approach to resource provisioning in cloud infrastructure with a grouping genetic algorithm. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, pages 1–6, 2018.

[16] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. iaware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Transactions on Computers*, 63(12):3012–3025, 2013.

[17] Wei Deng, Fangming Liu, Hai Jin, Xiaofei Liao, and Haikun Liu. Reliability-aware server consolidation for balancing energy-lifetime tradeoff in virtualized cloud datacenters. *International Journal of Communication Systems*, 27(4):623–642, 2014.

[18] Zhi Zhou, Fangming Liu, Ruolan Zou, Jiangchuan Liu, Hong Xu, and Hai Jin. Carbon-aware online control of geo-distributed cloud services. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2506–2519, 2015.

[19] Fangming Liu, Zhi Zhou, Hai Jin, Bo Li, Baochun Li, and Hongbo Jiang. On arbitrating the power-performance tradeoff in saas clouds. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2648–2658, 2013.

[20] Minxian Xu, Wenhong Tian, and Rajkumar Buyya. A survey on load balancing algorithms for virtual machines placement in cloud computing. *Concurrency and Computation: Practice and Experience*, 29(12):e4123, 2017.

[21] Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 88:50–71, 2017.

[22] Avnish Thakur and Major Singh Goraya. A taxonomic survey on load balancing in cloud. *Journal of Network and Computer Applications*, 98:43–57, 2017.

[23] Mostafa Noshy, Abdelhameed Ibrahim, and Hesham Arafat Ali. Optimization of live virtual machine migration in cloud computing: A survey and future directions. *Journal of Network and Computer Applications*, 110:1–10, 2018.

[24] Pawan Kumar and Rakesh Kumar. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Comput. Surv.*, 51(6), February 2019.

[25] E. Tardos J. Kleinberg. *Algorithm Design*. 2005.

[26] Amazon. Amazon elastic compute cloud, 2013.

[27] Thomas Knauth and Christof Fetzer. Energy-aware scheduling for infrastructure clouds. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 58–65. IEEE, 2012.

[28] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[29] Wenhong Tian, Yong Zhao, Yuanliang Zhong, Minxian Xu, and Chen Jing. A dynamic and integrated load-balancing scheduling algorithm for cloud datacenters. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 311–315. IEEE, 2011.

[30] Wenhong Tian, Yong Zhao, Minxian Xu, Yuanliang Zhong, and Xiashuang Sun. A toolkit for modeling and simulation of real-time virtual machine allocation in a cloud data center. *IEEE Transactions on Automation Science and Engineering*, 12(1):153–161, 2013.

[31] Ajay Gulati, Ganesha Shanmuganathan, Anne M Holler, and Irfan Ahmad. Cloud scale resource management: Challenges and techniques. *HotCloud*, 11:3–3, 2011.

[32] Hebrew University, 2013.

[33] Minxian Xu and Wenhong Tian. An online load balancing scheduling algorithm for cloud data centers considering real-time multi-dimensional resource. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, volume 1, pages 264–268. IEEE, 2012.

[34] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[35] Sun X, Xu P, and Shuang Kai. Multi-dimensional aware scheduling for co-optimizing utilization in data center. *China Communications*, pages 19–27, 2011.

Wenhong Tian (Senior Member, CCF) has a PhD from Computer Science Department of North Carolina State University. He is a full professor at University of Electronic Science and Technology of China (UESTC). His research interests include resource scheduling in Cloud computing and Bigdata processing, image identification and classification, and automatic quality analysis of natural language with deep learning. He published about 50 journal and conference papers, and 3 English books in related areas. He is a member of ACM, IEEE and CCF.

Minxian Xu (Member, IEEE) is currently an assistant professor at Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. He received the BSc degree in 2012 and the MSc degree in 2015, both in software engineering from University of Electronic Science and Technology of China. He obtained his PhD degree from the University of Melbourne in 2019. His research interests include resource scheduling and optimization in cloud computing. He has co-authored 28 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.

Guangyao Zhou received Bachelor's degree and Master's degree from School of architectural engineering, Tianjin University, China. He is now a Ph.D candidate at School of information and software engineering, University of Electronic Science and Technology of China, majoring in Software Engineering. His current research interests include Cloud Computing and image recognition.

Kui Wu (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science from Wuhan University, Wuhan, China, in 1990 and 1993, respectively, and the Ph.D. degree in computing science from the University of Alberta, Edmonton, AB, Canada, in 2002. In 2002, he joined the Department of Computer Science, University of Victoria, Victoria, BC, Canada, where he is currently a professor. His current research interests include network performance analysis, online social networks, Internet of Things, and parallel and distributed algorithms.

Chengzhong Xu (Fellow, IEEE) is the Dean of Faculty of Science and Technology and the Interim Director of Institute of Collaborative Innovation, University of 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 10K citations with an H-index of 52. He obtained BSc and MSc degrees from Nanjing University in 1986 and 1989 respectively, and a PhD degree from the University of Hong Kong in 1993, all in Computer Science and Engineering.

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. 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=136, g-index=300, 98,800+ 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