Distributed Autonomous Virtual Resource Management in Datacenters Using Finite-Markov Decision Process

Haiying Shen, *Senior Member, IEEE, Member, ACM*, and Liuhua Chen

***Abstract*— To provide robust infrastructure as a service, clouds currently perform load balancing by migrating virtual machines (VMs) from heavily loaded physical machines (PMs) to lightly loaded PMs. Previous reactive load balancing algorithms migrate VMs upon the occurrence of load imbalance, while previous proactive load balancing algorithms predict PM overload to conduct VM migration. However, both methods cannot maintain long-term load balance and produce high overhead and delay due to migration VM selection and destination PM selection. To overcome these problems, in this paper, we propose a proactive Markov Decision Process (MDP)-based load balancing algorithm. We handle the challenges of allying MDP in virtual resource management in cloud datacenters, which allows a PM to proactively find an optimal action to transit to a lightly loaded state that will maintain for a longer period of time. We also apply the MDP to determine destination PMs to achieve long-term PM load balance state. Our algorithm reduces the numbers of service level agreement (SLA) violations by long-term load balance maintenance, and also reduces the load balancing overhead (e.g., CPU time and energy) and delay by quickly identifying VMs and destination PMs to migrate. We further propose enhancement methods for higher performance. First, we propose a cloud profit oriented reward system in the MDP model so that when the MDP tries to maximize the rewards for load balance, it concurrently improves the actual profit of the datacenter. Second, we propose a new MDP model, which considers the actions for both migrating a VM out of a PM and migrating a VM into a PM, in order to reduce the overhead and improve the effectiveness of load balancing. Our trace-driven experiments show that our algorithm outperforms both previous reactive and proactive load balancing algorithms in terms of SLA violation, load balancing efficiency, and long-term load balance maintenance. Our experimental results also show the effectiveness of our proposed enhancement methods.**

***Index Terms*— Markov decision process, cloud computing, resource management.**

I. Introduction

CLOUD computing is a new emerging IT service, which provides various services under one roof. Services such as storage, computing and web hosting, which used to be provided by different providers, are now provided by a single provider [1]–[3]. Many businesses move their services to clouds with their flexible "pay as you go" service model, in which a cloud customer only pays for the resources it has used. Such elasticity of the service model brings about cost saving for most businesses [4] by eliminating the need of developing, maintaining and scaling a large private infrastructure.

Clouds utilize hardware virtualization, which enables a physical machine (PM) to run multiple virtual machines (VMs) with different resource allocations. A cloud hosts multiple applications on the VMs. Since the load of each VM on a PM varies over time, a PM may become overloaded, i.e., the resource demand from its VMs is beyond its possessed resource. Such load imbalance in a PM adversely affects the performance of all the VMs (hence the applications) running on the PM. Insufficient resources provision to customer applications also violates the Service Level Agreement (SLA). An SLA is an agreement between a cloud customer and the cloud service provider that guarantees the application performance of the customer. In order to uphold the SLA, a cloud service provider must prevent load imbalance using load balancing algorithms, in which overloaded PMs migrate their VMs to underloaded PMs to release their excess loads.

Many load balancing algorithms [5]–[10] have been proposed that reactively perform VM migration upon the occurrence of load imbalance or when a PM's resource utilization reaches a threshold. However, these algorithms only consider the current state of the system. Fixing a load imbalance problem upon its occurrence not only generates a delay to achieve load balance but also cannot guarantee the subsequent long-term load balance state, which may lead to resource deficiency to cloud customer hence SLA violations. Also, the process of selecting migration VMs and destination PMs is complex and generates high delay and overhead.

Recently, some methods [11]–[16] have been proposed to predict VM resource demand in a short time for sufficient resources provision or load balancing. In the proactive load balancing, a PM can predict whether it will be overloaded by predicting its VMs' resource demands, and moves out VMs when necessary. However, this method has the following problems. First, a PM does not know which VMs to migrate out. Additional operations of identifying VMs to migrate bring about additional delay and overhead. Second, since it only achieves load balance at the predicted time spot, without considering long-term load balance, it generates high migration overhead. Third, some of the methods build a Markov chain

Manuscript received February 11, 2016; revised January 29, 2017 and May 6, 2017; accepted September 20, 2017; approved by IEEE/ACM Transactions on Networking Editor S. Sengupta. Date of publication October 26, 2017; date of current version December 15, 2017. This work was supported in part by the U.S. NSF under Grant OAC-1724845, Grant ACI-1719397, and Grant CNS-1733596, and in part by the Microsoft Research Faculty Fellowship under Grant 8300751. An early version of this work was presented in the Proceedings of SOCC, 2014 [32]. *(Corresponding author: Haiying Shen.)*

H. Shen is with the Computer Science Department, University of Virginia, Charlottesville, VA 22904-4740 USA (e-mail: hs6ms@virginia.edu).

L. Chen is with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634 USA.

Digital Object Identifier 10.1109/TNET.2017.2759276

model and calculate the transition probability matrix for each individual VM in the system, which generates prohibitive overhead especially in a system with a large number of VMs.

What's more, both reactive and proactive methods select the destination PMs simply based on their current available resources without considering their subsequent load status.

Effectively achieving the trade-off between the penalties associated with SLA violations and cloud resource utilization (hence revenue maximization) requires an algorithm that i) helps proactively handle the potential load imbalance problem by migrating VMs out of PMs that are about to be overloaded in advance and also maintains its load balance state for a long time, ii) generates low overhead and delay for load balancing, and iii) maintains a long-term load balance state for destination PMs after the VM migrations. However, as far as we know, there are no load balancing algorithms that can meet these requirements.

To meet this need, in this paper, we propose a proactive Markov Decision Process (MDP)-based [17] load balancing algorithm. However, there are two challenges in using the MDP for the load balancing purpose.

- First, the MDP components must be well designed for low overhead. An MDP consists of states (s), actions (a), transition probabilities (P) and rewards (R). After state s takes action a, it has probability $P_a(s, s')$ to transit to s' and then receives reward $R_a(s, s')$. If an MDP considers an action as moving out a specific VM, it needs to record the load state transitions of a PM for moving out each VM in the system, which generates a prohibitive cost and also is not accurate due to time-varying VM load. To handle this challenge, our designed MDP intelligently uses a PM load state as a state and records the transitions between PM load states by moving out a VM in a specific load state.
- Second, the transition probabilities in the MDP must be stable. Otherwise, the MDP cannot accurately provide guidance for VM migration or the MDP must be updated very frequently to keep the transition probabilities accurate. To handle this issue, we have studied VM migrations based on real traces, which confirms that the transition probabilities are stable in our MDP.

We also design the rewarding policies, which encourages a PM to transit to or maintain in the lightly loaded state and discourages a PM to stay at the heavily loaded state. Thus, when each PM attempts to maximize its rewards through performing VM migration actions, it can find an optimal action to transit to a lightly loaded state that will maintain for a longer period of time. A similar MDP is also built for determining destination PMs with the goal to not only maintain their load balance states for a long time but also fully utilize their resources.

Compared to previous reactive and proactive load balancing algorithms, our algorithm has several advantages. First, it reduces the numbers of SLA violations by proactive load balancing and long-term load balance maintenance. It also reduces the load balancing overhead and delay by quickly identifying VMs to migrate out based on MDP, which avoids the need of additional operations of the VM identification. In addition, it only needs to build one MDP that can be used by all PMs in the system. Unlike the previous proactive load balancing algorithms that focus on predicting VM or PM load, our work is the first that focuses on providing guidance on migration VM selection and destination PMs selection for long-term load balance state maintenance.

In order to improve the actual profit of the datacenter, we propose a cloud profit oriented reward system in the MDP model. It specifies the reward based on the datacenter's profit (calculated by revenue, energy consumption cost and live migration overhead) in the practical scenario. Thus, the datacenter's profit can be concurrently maximized when the MDP tries to maximize the rewards for load balance. Using such an improved reward system in the MDP model can improve the actual profit of the datacenter.

We further propose a new MDP model that considers the actions of both migrating a VM out of a PM and migrating a VM into a PM. Unlike the preliminary MDP-based load balancing algorithm that builds two MDP models (one MDP model is for selecting migration VMs from PMs to migrate out and the other MDP model is for selecting destination PMs for hosting the migration VMs), the new MDP model avoids the potential action conflictions and reduce the overhead brought about by using two separate MDP models.

The rest of the paper is organized as follows. Section II presents the related work. Section III presents the overview of our MDP-based load balancing algorithm. Section IV presents the detailed design of the MDP-based load balancing algorithm, together with our proposed enhancement methods. Section V presents the performance evaluation of our algorithm compared with other load balancing algorithms in trace driven simulations. Finally, Section VI concludes this paper with remarks on our future work.

II. Related Work

In recent years, many load balancing methods have been proposed to avoid overloaded PMs in the clouds [5]–[10]. These algorithms perform VM migration when a PM's resource utilization reaches a threshold. After migration VMs are selected, these methods select their destination PMs simply based on their available resources at the decision time without considering their subsequent load status.

Many methods [11]–[16] predict workloads of PMs or VMs in order to ensure the sufficient provision for the resource demands or for load balancing. They also select the destination PMs simply based on their current available resources.

However, the migration VM selection and destination PM selection in the previous reactive and proactive load balancing algorithms cannot maintain a long-term system load balance state, which otherwise reduces not only SLA violations (SLAV) but also the overhead and delay caused by load balancing execution. To overcome these problems, we propose a method that uses MDP to let each PM calculate the optimal action to perform with the goal of achieving long-term load balance state. Though our algorithm shares similarity with the previous algorithms in proactive prediction, those algorithms focus on predicting VM or PM load, while our algorithm focuses on providing PMs with guidance on migration VM selection for long-term load balance state maintenance. This work is non-trivial as it requires well-designed components of MDP to constrain the overhead of MDP creation and maintenance and ensure the MDP's stability. Our previous work in [18] considers VM workload patterns to consolidate VMs to fully utilize PM resources in the initial VM allocation phase. Our MDP model indicates a PM's state transitions when it migrates out a VM in different states. Since the state is the resource utilization of PMs, and the action is migrating out a VM in a state (a certain resource utilization level), so the MDP model is not affected by time-varying workload of VMs or the workload patterns. Many research works have been devoted to power management for VMs. For example, Laszewski *et al.* [19] used the technique of Dynamic Voltage Frequency Scaling (DVFS) for scheduling virtual machines in a compute cluster to reduce power consumption. Kansal *et al.* [20] presented a solution for VM power metering, named Joulemeter. They built power models to infer power consumption from resource usage at runtime and identify the challenges that arise when applying such models for VM power metering. Since DVFS can potentially decrease the system reliability, Xu *et al.* [21] proposed a data center management framework, DUAL, which consists of new VM power and reliability analysis tools in order to balance the dual needs of a data center: reducing energy consumption and providing high reliability. Unlike these works that mainly focus on energy consumption reduction, this work focuses on achieving the load balance state by VM migration with low overhead (e.g., energy consumption).

III. MDP-Based Load Balancing

A. Goals

The goal of our load balancing algorithm is to reduce SLAV and meanwhile reduce the load balancing overhead and delay. Usually SLAV comes from two parts: SLA Violation due to Overutilization (SLAVO) and SLA Violation due to Migrations (SLAVM) [22]. Thus, we need to guarantee sufficient resource provisioning to cloud VMs and reduce the number of VM migrations. To achieve the goals, we aim to prevent heavily loaded state for each PM and maintain the load balance state for a long time. In this way, we not only reduce SLAV but also reduce the times to execute the load balancing algorithm, hence reduce the number of VM migrations and overhead (energy, CPU time, etc.) caused by load balancing execution. Also, we aim to design a load balancing algorithm that generates low overhead and delay itself. Low load balancing delay can reduce the delay for the system to recover to the load balance state, hence also reduce SLAV. Low load balancing overhead saves the resources for applications, which increases the revenue of the cloud provider.

B. Low Overhead MDP Creation and Maintenance

To achieve the above-stated goals, we design an MDP model that provides guidance on migration VM and destination PM selections for long-term load balance state maintenance. An MDP [17] requires a 4-tuple input (States (S), Actions (A), Transition Probabilities (P), Rewards (R)). An MDP provides a general framework for finding an optimal action in a stochastic environment, which maximizes the rewards from the actions so that the outcomes follow the decision maker's desire. The overhead of both MDP creation and maintenance (determined by the update frequency) must be low in order to meet the low load balancing overhead requirement.

Unlike the previous VM load prediction models [11]–[13], [15], [16], we directly use the PM load state as the MDP state, which enables a PM to directly check whether it is heavily loaded or lightly loaded. The action set A should be a set of VM migrations that a PM in a certain state can perform. For an MDP, it is required that the set of actions A do not change; otherwise, MDP has to be updated upon a change. Declaring migration actions based on each individual VMs held by a PM will lead to the changes of action set A and their associated transition probabilities in the PM. This is because the VMs held by a PM may change and a PM could hold any VM in the system due to VM migration, hence the available actions of a PM may change. For example, if PM_1 migrates VM_1 to PM_2, the action of migrating out VM_1 needs to be deleted from PM_1's action set, and it needs to be added to PM_2's action set. When the resource utilization of VM_2 in PM_1 changes, the transition probabilities of the action of migrating out VM_2 from PM_1 to each transition state needs to be updated. To solve this problem, we can define the action set A as moving out each individual VM in the system. This solution however generates a prohibitive cost considering the huge number of VMs in the system. Also, the resource utilization of each VM dynamically changes, which also necessitates the frequent updates of the associated transition probabilities.

To achieve a stable and small action set and stable transition probabilities, we novelly define an action set as the migration of a VM with a specific load state (migration of VM-state in short). The load state is defined as a combination of the utilizations of different resources such as "CPU-high, Mem-high". We will explain the details of VM-state later on. Therefore, all PMs in the cloud have the same action set A, which includes the migrations of each VM-state. An MDP state has a transition probability to transit to another state after performing an action. As the total number of VM-states in the action set does not change regardless of a PM's actions, the action set A does not change. Also, each VM-state itself does not change, so the associated transition probability for migrating this VM-state does not change. Thus, MDP does not need to update with the migration of VM-states.

Our MDP model is continuously trained using the past actions. It indicates a PM's state transitions when it migrates out a VM in different states. Each PM refers to the established MDP to decide its optimal action (that leads to the maximum expected reward) based on its current state. Even though the workload trends of VMs are not reflected in the MDP model, the long-term expected reward has the same idea to keep the stability of the system (i.e., achieve and maintain long-term load balance).

It is required that the transition probabilities in an MDP must be stable. If the MDP creation approach cannot maintain stable transition probabilities, the MDP then cannot function well or it needs a very frequent update in order to provide correct guidance. To confirm whether our MDP is stable, we have conducted an experimental study on real traces. Before we present the results in Section III-D, we first introduce the definitions of the load states in Section III-C.

C. Load State of PMs and VMs

In our load balancing algorithm, each PM selects VMs in certain load states to migrate out in advance when they are about to be overloaded, so that it can maintain its load balance state for a long time. This algorithm proactively avoids overloading PMs in the cloud and continually maintains the system in a load balance state in a long term while limits the number of VM migrations. Therefore, a basic function of our algorithm is to determine the load state of PMs and VMs to represent PM-state and VM-state used in the MDP model. PM-state represents the load state of a PM in the MDP model, while VM-state is used to identify VMs with certain resource utilization degrees to migrate in the actions of PMs.

In a cloud environment, there are different types of resources (CPU, memory, I/O and network). Therefore, the workloads of PMs and VMs are multi-attribute in terms of different types of resources. In order to generalize our definitions, we use k to denote the number of resource types.

We assume there are N VMs running on M PMs in a cloud. We regard time period as a series of time intervals (τ) and use t_i to denote the specific time at the end of the i-th interval. We use $l_n^k(t_i)$ to denote the demanded resource amount (i.e., load) of the type-k resource in the n-th VM at time t_i. For time-sharing resources (e.g., CPU, bandwidth), $l_n^k(t_i)$ equals the average value from time t_{i-1} to time t_i. For space-sharing resources (memory, disk space), $l_n^k(t_i)$ equals the demand value at time t_i. We use $L_m^k(t_i)$ and $C_m^k(t_i)$ to denote the load and capacity of the type-k resource in the m-th PM at time t_i, respectively. Suppose the m-th PM has N_m number of VMs, then $L_m^k(t_i) = \sum_{j=1}^{N_m} l_j^k(t_i)$.

We define the *utilization* of the type-k resource in the n-th VM at time t_i as

$$u^k n_(t_i) = l_n^k(t_i)/c_n^k(t_i), \tag{1}$$

where $l_n^k(t_i)$ and $c_n^k(t_i)$ denote the load and assigned resource of the n-th VM at time t_i. We define the *utilization* of the type-k resource in the m-th PM at time t_i as

$$U_m^k(t_i) = L_m^k(t_i)/C_m^k(t_i) = \sum_{j=1}^{N_m} l_j^k(t_i)/C_m^k(t_i). \tag{2}$$

We use T_o^k to denote the threshold for the utilization of the type-k resource in a PM. The objective of our load balancing algorithms is to let each PM maintain $U_m^k(t_i) \leq T_o^k$ for each type of resources. For simplicity, we omit k in the notation unless we need to distinguish different types of resources.

In a PM, for a given resource, based on the resource utilization (i.e., load) of the PM, we determine the utilization level of this resource in this PM. We use three levels (high, medium and low) as an example to explain our algorithm in this paper, which can be easily extended to more levels. Specifically, to perform level determination for type-k resource, we use Equation (3), in which T_1^k and T_2^k are two thresholds used to distinguish low and medium, and medium and high levels, respectively.

$$\begin{cases} Low & \text{if } U_m^k < T_1^k \\ Medium & \text{if } U_m^k \geq T_1^k \text{ and } U_m^k < T_2^k \\ High & \text{if } U_m^k \geq T_2^k \end{cases} \tag{3}$$

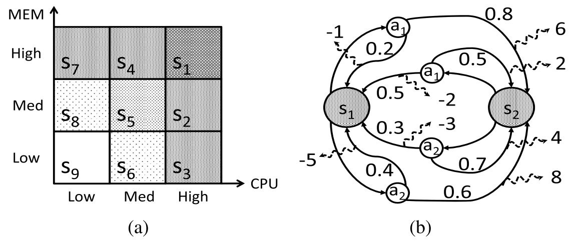


Fig. 1. Example of a simple MDP. (a) PM states. (b) MDP model.

To extend these three levels to fine-grained levels, we only need to add more thresholds. Accordingly, the number of states in the MDP model will be increased. The state determination of VMs is performed in the same manner by changing U_m^k in Equation (3) to u_n^k. We consider that a PM is *heavily loaded* when the utilization of one of its resources reaches the high-level threshold, and consider that a PM is *lightly loaded* when none of its resource utilizations reaches the high-level threshold.

Consider a set of K resources $\mathcal{R} = \{r_1, r_2, ...r_K\}$ in the cloud system and resource utilization levels $\mathcal{L} = \{$High, Medium, Low$\}$. The total number of states of VMs or PMs equals $|\mathcal{L}|^{|\mathcal{R}|}$; the Cartesian product of the two sets. The set of states is $S = \mathcal{R} \times \mathcal{L}$, where$\times$means the combination of r_k in different resource utilization levels. For example, if we consider two resources, $\mathcal{R} = \{$CPU, Mem$\}$, a PM's state can be represented by the utilization degree of each resource such as (CPU-high, Mem-high), (CPU-median, Mem-low), etc. Then, there are $3^2 = 9$ states for a VM or a PM as shown in Figure 1(a).

D. Trace Study on the Stability of Our MDP

State set S is a set of PM resource utilization levels based on Equation (3). As mentioned before, the transition probabilities of an MDP must be stable. To confirm whether our design of different MDP components can achieve the MDP stability, in this section, we conduct an experiment, which shows that the transition probability matrix remains stable even when we slightly change threshold T_i^k in Equation (3). Therefore, we can properly set approximate T_i^k to determine the resource utilization level in MDP construction.

In Equation (3), T_2^k is more important than T_1^k since T_2^k is a threshold to determine the high utilization level, which determines the heavily loaded state of a PM. Thus, we conducted experiments with varying T_2^k values and kept $T_1^k = 0.3$. We used CloudSim [23] for the experiments and compared the transition probability matrix obtained under varying threshold T_2^k values. The PMs are modeled from commercial product HP ProLiant ML110 G4 servers (1860 MIPS CPU, 4GB memory) and the VMs are modeled from EC2 micro instance (0.5 EC2 compute unit, 0.633 GB memory, which is equivalent to 500 MIPS CPU and 613 MB memory). We used two traces in the experiments: PlanetLab trace [23] and Google Cluster trace [24]. The PlanetLab trace contains the CPU utilization of VMs in PlanetLab every 5 minutes for 24 hours in 10 random days in March and April 2011. The Google Cluster trace records resource usage on a cluster of about 11000 machines from May 2011 for 29 days. As there are a very large number

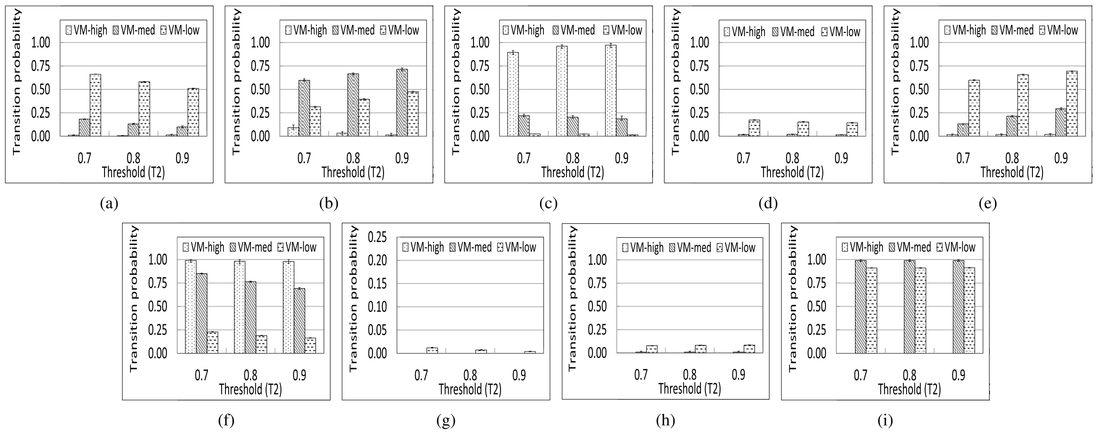


Fig. 2. Probability of state transitions of a PM using PlanetLab trace. (a) PM-state: high→high. (b) PM-state: high→med. (c) PM-state: high→low. (d) PM-state: med.→high. (e) PM-state: med.→med. (f) PM-state: med.→low. (g) PM-state: low→high. (h) PM-state: low→med. (i) PM-state: low→low.

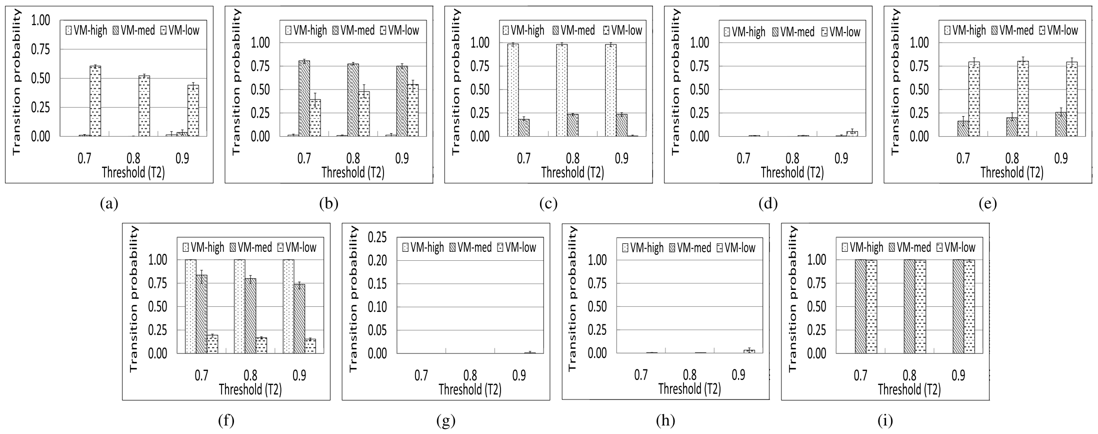


Fig. 3. Probability of state transitions of a PM using Google Cluster trace. (a) PM-state: high→high. (b) PM-state: high→med. (c) PM-state: high→low. (d) PM-state: med.→high. (e) PM-state: med.→med. (f) PM-state: low→low. (g) PM-state: low→high. (h) PM-state: low→med. (i) PM-state: low→low.

of states when considering multiple resources, we focus on the CPU resource in the experiments. In each test, we selected x VMs from the trace and assigned them to a PM, where x was randomly selected from [1, 20]. We then randomly selected a VM in the PM to migrate out. We measured the PM-state before and after VM migration based on the thresholds, and the load state of the migrating VM. In each experiment, we repeated this process for 100,000 times and calculated the transition probabilities for different PM-state changes when migrating different VM-states (e.g., the number of "high→medium" PM-state transitions when migrating a medium VM-state).

We repeated the experiment 100 times and calculated the transition probabilities. Figure 2 and Figure 3 show the transition probabilities of PM state changes when using the PlanetLab trace and the Google Cluster trace, respectively. The error bars show the 99th and 1st percentiles among the 100 experiments. Each figure shows the results with different T_2 threshold values from 0.7, 0.8 to 0.9. In these figures, VM-high, VM-medium and VM-low represent that the migration VM-state is high, medium and low, respectively. We use PM-high, PM-medium and PM-low to represent a PM in the high, medium and low state, respectively. For example, Figure 2(c) and Figure 3(c) indicate that a PM-high has a high probability (0.95-1 for PlanetLab trace and 1 for Google Cluster trace, respectively) to transit to state low when it migrates VM-high. In Figure 2(i) and Figure 3(i), a PM-low always (near 1 probability) transits to state low when it migrates VM-medium. It is interesting to see that in Figure 2(g) and Figure 3(g), the probability that a PM-low transits to state high when it migrates VM-low is not 0, which means that a PM-low can transit to state high even when it migrates out a VM, due to the fluctuation of workload. We can observe that in each of these figures, the probabilities are almost the same under varying threshold T_2 with different traces. The error bars indicate that the probabilities derived in different experiments have a very small variation. Compared to the transition probabilities derived from the PlanetLab trace in Figure 2, the absolute values of the transition probabilities derived from the Google Cluster trace in Figure 3 are slightly different, due to the difference of the workload characteristics of these two trace. We can still observe that in each of these three figures, the probabilities are similar under varying threshold T_2.

The results indicate that slightly varying threshold T_2 will not greatly affect the values of the probability transition matrix. As a result, we can tune the threshold for determining PM states as expected. In our MDP-based load balancing algorithm, we use $T_1 = 0.3$ and $T_2 = 0.8$, which are reasonable thresholds for the low and high resource utilization levels.

Note that the workload of VMs may change over time. A distinguishing feature of the MDP model is to achieve and maintain a long-term load balance. After a PM takes an action, it can maintain the lightly loaded state for a long term. The MDP model is built based on the historical data which embraces the workload changes. Therefore, the final established MDP model reflects the general case. If workload changes are normal situation in the system, they are caught in the MDP training and are reflected in the state transitions. Otherwise, they are just an occasional case and then upon a PM's action after the workload change, it is most likely to maintain a long time lightly load balanced state.

IV. Construction of MDP

A. Overview of The MDP Model

The previous two sections indicate the feasibility of our proposed MDP. Below, we present an overview of our MDP model in this section, and then present the details of the MDP components in the following sections. In our MDP-based load balancing algorithm for a cloud system, the resource utilization degree of a PM is classified to a number of levels. Unless otherwise specified, in this paper, we use three levels: {high, medium and low} and two resources {CPU, Mem} as an example for the MDP creation. Our method can be easily extended to more levels and more resources. Specifically, we define the 4 elements of MDP in our MDP-based load balancing algorithm as follows:

1) S is a finite set of states {(CPU-high, Mem-high), (CPU-medium, Mem-low),…}, which are multi-variate classified representation of current resource utilization of a PM (PM-state).
2) A is a set of actions. An action means a migration of VM in a certain state (VM-state) or no migration. VM-state is represented in the same manner as PM-state.
3) $P_a(s, s') = P_r(s_{t+1} = s'|s_t = s, a_t = a)$ is the probability that action $a \in A$ in state $s \in S$ at time t will lead to state $s' \in S$ at time $t + 1$. The transition probabilities are determined based on the trace of a given cloud system.
4) $R_a(s, s')$ is an immediate reward given after transition to state s' from state s with the transition probability $P_a(s, s')$ by taking action a.

Figure 1(b) illustrates the transition model of a simple MDP with two states and two actions. The 3×3 table in Figure 1(a) represents all possible PM states. The two circles with s_1 and s_2 indicate the two states of a PM. The four smaller circles with a_1 and a_2 mean an action of migrating out a VM in a certain VM-state or no migration. The fraction number along the arrow from state s_i to state s_j going through a_i means the probability that s_i will transit to s_j after taking action a_i ($P_a(s_i, s_j)$), and the number along the dashed arrow represents the reward associated with the state transition from s_i to s_j after taking action a_i ($R_a(s_i, s_j)$). As shown in the figure, for a PM in state s_1 (CPU-high, Mem-high), if it takes action a_1, it has a probability of 0.2 to stay in s_1 and receive reward -1, and has a probability of 0.8 to transit to s_2 (CPU-high, Mem-med) and receive reward 6.

The transition probability matrix for a given system is obtained by studying the trace information of the system. We will show in Section IV-B that the final constructed transition probability matrix remains stable during a certain period of time, hence does not require frequent recalculation of the probabilities in the MDP. In the set of states (S), some states mean that the PM is heavily loaded while others mean the PM is lightly loaded. In the MDP, a PM identifies the action with the highest expected reward and takes this action to maximize its earned reward, which enables it to transmit to or remain at the lightly loaded state for a long time.

For this purpose, we design the reward system in the MDP that assigns a positive reward for transiting to or maintaining at a lightly loaded state and a negative reward for maintaining a heavily loaded state. In Section IV-B, we present our reward system, which encourages a PM to find the optimal action to perform to attain and maintain a lightly loaded state for a longer time. As a result, each PM is in a lightly loaded state with high probability in a long term and the total number of VM migrations in the system is reduced.

B. Construction and Usage of MDP in a Cloud

In this section, we present the construction of an MDP in a cloud. As indicated earlier, the MDP needs 4-tuple variables: States S, Actions A, Transition Probabilities P and Rewards R. We explain each variable in the following.

States (S) and Actions (A): We explained "States" and "Actions" in Section III-C. As mentioned previously, $S = \mathcal{R} \times L$. The action set A consists of $(|L|^{|R|}) + 1$ elements and "1" represents "no action". In our MDP, no matter if incoming VM changes the state of a PM or the loads of VMs currently running on a PM change, the state set and action set will not change. The MDP is able to find an optimal action that achieves load balance state and sustains this state for a longer time period.

Using the state determination method introduced in Section III-C, a PM determines its own PM-state. It then identifies its position in the MDP and finds the actions it needs to take to transit to or remain at the lightly loaded state. To migrate out VMs to become or remain lightly loaded, a PM needs to determine the VM-state of each of its VM. Then, it chooses VMs in a certain VM-state to take the actions.

Transition Probabilities (P): For a PM in state $s_i \in S$, after it performs action $a \in A$, it will transit to another state $s_j \in S$ or remain in the same state. We need to determine the probability of transiting to each of other states after taking each action. The transition probability should be stable because a change in the transition probability would result in new transition policy if the change in value is too large.

The cloud uses the information from the trace of the state changes and VM migrations to determine the transition probability matrix. In the previous load balancing algorithms, a central server monitors the states of PMs and determines the VM migrations between PMs. We let this central server keep track of the VM-state of each migrated VM and the PM state changes upon the VM migration. Based on this information, the central server can calculate the transition probability from one state to another state upon an action. For example, in

TABLE I
PROBABILITIES WITH THRESHOLD $T_2 = 0.8$

	aH			aM			aL		
	vH	vM	vL	vH	vM	vL	vH	vM	vL
bH	0.01	0.13	0.59	0.03	0.65	0.39	0.96	0.22	0.02
bM	0.00	0.02	0.16	0.06	0.21	0.65	0.94	0.77	0.19
bL	0.00	0.00	0.01	0.00	0.00	0.08	1.00	1.00	0.91

the 1-resource environment, for action $a \in A$, if the transition high→high occurs 5 times, high→medium occurs 4 times, and high→low occurs 1 time, then the transition probability in performing action a when in state high is 0.5, 0.4, 0.1 to the high, medium, low state, respectively.

We conduct a similar experiment as in Section III-D. Table I shows the probabilities of PM state changes when $T_2 = 0.8$. bH, bM and bL represent the high, medium and low state *before* migration, respectively; aH, aM and aL represent the high, medium and low state *after* migration, respectively; and vH, vM and vL represent actions of migrating VM in state high, medium and low respectively. For a given "state" before migration and specific actions, the sum of the probabilities that transit to any states (aH, aM and aL) is 1. Notice that a PM in state low has a nearly zero probability to change to any states when taking action vH (migrating VM in state high). Table I will be used in our experiments in Section V.

Rewards (R): Rewards are incentives that are given to a PM after performing action $a \in A$. By encouraging each PM to maximize its received rewards, the reward system aims to constantly avoid heavily loaded state for each PM while minimizing the number of VM migrations; that is, maintain a system load balance state for a long time and minimize load balancing overhead. To achieve this goal, we need to carefully assign rewards for actions. For example, rewarding a PM for each migration might result in continuous migrations of a PM, which generates a high overhead. To achieve the load balance state, each overloaded PM should be encouraged to change to lightly loaded PM. Thus, the system rewards heavily loaded PM positively for performing actions that lead it to a lightly loaded state. Also, PMs should be rewarded to maintain their lightly loaded state. In order to prevent under-utilization of resources, the reward for maintaining the medium state is greater than maintaining the low state. We present the details of the reward policies for transiting from state s to state s' below. A PM receives a reward when the state of one of its resources is changed. Note that the rewards are for each type of resources. We consider the following two cases.

1) Reward for a resource utilization transiting from high state to another state (λ) by moving out a VM:
   a) Positive reward for a transition to a low (c) or medium (b) state.
   b) Negative reward for a transition to a high state (d).
   c) The reward for a transition to a medium state is higher than to a low state ($b > c$).
2) Reward for performing no action (γ):
   a) Reward for performing no action in a low (c') or medium state (b').
   b) Reward for no action in a low state is higher than in a medium state ($c' > b'$).
   c) Negative reward for performing no action in a high state (d').

Let $\mathcal{R}_H$ be the subset of resources in $\mathcal{R}$ of a PM whose resource utilizations are high after action a. Similarly, we let $\mathcal{R}_L$ and $\mathcal{R}_M$ be the resource subsets whose resource utilizations after action a are low and medium, respectively. Thus, we have,

$$\mathcal{R} = \mathcal{R}_L \cup \mathcal{R}_M \cup \mathcal{R}_H. \tag{4}$$

The first reward is λ, which is the reward for transiting to another state. This reward encourages each PM to transit each of the resources into a lower loaded state, thus helping to achieve load balance state. For a PM with R resources, after performing an action a, the reward λ equals:

$$\lambda = \sum_{r \in \mathcal{R}_H} d + \sum_{r \in \mathcal{R}_M} b + \sum_{r \in \mathcal{R}_L} c, \quad \forall r \in \mathcal{R}, \tag{5}$$

where d, b and c are non-negative reward and $d < c < b$.

Let's consider reward for no action γ. This reward encourages PM to maintain a low or medium state for a longer period of time. When a PM performs no action, it is rewarded for performing no action. The reward is dependent on the state of each of the PM's resources. The reward γ is calculated as follows and $c' > b' > d'$.

$$\gamma = \sum_{r \in \mathcal{R}_H} d' + \sum_{r \in \mathcal{R}_M} b' + \sum_{r \in \mathcal{R}_L} c', \quad \forall r \in \mathcal{R}$$

As a result, the total reward earned by a PM is the sum of the two rewards λ and γ.

$$R_a(s, s') = \lambda + \gamma$$

Note that if the action taken is "no action", γ is used and λ equals to 0, and if the action taken is moving out a VM, γ equals to 0 and λ is used. Each PM needs to find the optimal actions, denoted by $\pi(s)$ $(a \in A)$ to maximize its earned rewards, i.e., to reach or remain low or medium state for a long time period. In the next section, we explain how to obtain action set $\pi(s)$.

Optimal Action Determination Based on MDP:: The goal of the optimal action determination in an MDP is to find an action for each specific state that maximizes the cumulative function of expected rewards:

$$\sum_{t=0}^{\infty} R_{a_t}(s_t, s_{t+1}),$$

where t is a sequence number, and a_t is the action taken at t. The algorithm to calculate this optimal policy requires the storage for two arrays indexed by state: value $V(s)$, which contains the reward associated with a state, and policy $\Pi = \{\pi(s_1), \pi(s_2), \ldots, \pi(s_i, \ldots)\}$, which contains the action for each state that maximizes the cumulative expected rewards from the state. The algorithm outputs the optimal policy Π that contains the most suitable action for each state to take that will result in the maximum value $V(s)$ for the state The algorithm outputs the optimal policy Π that contains the most suitable action for each state to take that would result in the maximum value $V(s)$ for the state, and $V(s)$ contains the sum of the rewards to be earned (on average) by following the action from state s. The optimal policy for an MDP makes a PM attain a lightly loaded state and sustain for a longer period of time. The algorithm has the following two steps, which

Algorithm 1 The value-iteration algorithm.

```
Inputs: $T$, a transition probability matrix
        $R$, a reward matrix.
Output: Policy $\Pi$
1: $V(s_i) \leftarrow 0$, $V_{new}(s_i) \leftarrow R(s_i), i = 1, 2, ..., |S|$
2: while $max|V(s_i) - V_{new}(s_i)| \geq e, i = 1, 2, ..., |S|$ do
3:   $V \leftarrow V_{new}$
4:   for all state $i$ in $S$ do
5:     $V_{new}(s_i) \leftarrow R(s_i) + max_a \sum_j P(s_i, a, s_j)V(s_j)$
6:     $\pi(s_i) \leftarrow \arg\max_a\{\sum_j(P_a(s_i, s_j)(R(s_i, s_j) + V(s_j))\}$
7:   end for
8: end while
9: return $\Pi$
```

are repeated in some order for all the states until no further changes take place:

$$\pi(s_i) = \arg\max_a\{\sum_j(P_a(s_i, s_j)(R_a(s_i, s_j) + V(s_j))\} \quad (6)$$

$$V(s_i) = \sum_j P_{\pi(s_i)}(s_i, s_j)(R_{\pi(s_i)}(s_i, s_j) + V(s_j)) \quad (7)$$

Equation (6) obtains the optimal policy. In Equation (6), $V(s_j)$ is obtained by using Equation (7) for each state. Specifically, in order to determine the optimal policy, we apply the value-iteration algorithm [25], which is a dynamic algorithm. The aim of this algorithm is to find the max value $V(s_i)$ of each state and corresponding action $\pi(s_i)$, until it observes convergence in values for all states in successive iterations. Thus, using this algorithm, we can obtain the action for each state that can quickly lead to the maximum reward.

Algorithm 1 shows the pseudo code for the value-iteration algorithm. In the algorithm, $R(s_i)$ is calculated by

$$R(s_i) = \sum_j P_{\pi(s_i)}(s_i, s_j)R_{\pi(s_i)}(s_i, s_j), \quad (8)$$

where $\pi(s_i)$ is the optimal policy to maximize $V_{new}(s_i)$. The algorithm first initializes $V(s_i)$ and $V_{new}(s_i)$ (Line 1). It then repeatedly updates $V(s_i)$ based on Equation (7) and Equation (6) and the corresponding optimal policy $\pi(s_i)$ (Lines 2-7). When it observes convergence in values for all states, that is $max|V(s_i) - V_{new}(s_i)| < e$ (Line 2), it considers that $V(s_i)$ is close to its maximum value and the corresponding $\pi(s_i)$ is returned (Lines 9).

Analysis: In the following, we introduce a metric that evaluates the performance of an MDP in terms of the output optimal policy. The metric is called n-step transition probability, which is the probability that one state transits to another state after taking n actions. Recall that an MDP's policy is $\Pi = \{\pi(s_1), \pi(s_2), \ldots, \pi(s_i), ..., \pi(|S|)\}$, which contains the action for each state that maximizes the cumulative expected rewards from the state. That is, $\pi(s_i)$ is the action that a PM in state s_i should choose so that the cumulative expected rewards can be maximized. The n-step transition probability can be used to evaluate the policies of an MDP with different reward systems in order to find the best policy (or the best reward system). For example, suppose Π_1 and Π_2 are two policies corresponding to two reward systems. The n-step transition probabilities from state high to state medium of Π_1 and Π_2 are 95% and 90%, respectively. We prefer Π_1 as it has higher probability of transiting from state high to state medium, i.e., eliminating overloaded PMs.

For a fixed stationary policy Π, a transition probability matrix P, and a reword matrix R, action $a = \pi(s_i)$ is taken when a PM is in state s_i. The process of state transition $\{X_1, X_2, ..., X_k, \ldots, X_n\}$ is a Markov chain, in which the transition from $X_k = s_i$ to $X_{k+1} = s_j$ is an one-step transition with probability $P_{\pi(s_i)}(s_i, s_{i+1})$ when action $\pi(s_i)$ is taken based on Π. The n-step transition probabilities of this Markov chain can be represented by:

$$P^{(n)}_{\pi(s_i)}(s_i, s_j) = P\{X_n = s_j \mid X_0 = s_i\}. \quad (9)$$

Note that $P^{(1)}_{\pi(s_i)}(s_i, s_j) = P_{\pi(s_i)}(s_i, s_j)$, where $P_{\pi(s_i)}(s_i, s_j)$ is the one step transition probability that can be measured from the trace as introduced in Section III-D. By the Chapman-Kolmogorov equations [26], we get:

$$P^{(n)}_{\pi(s_i)}(s_i, s_j) = \sum_{s_k \in S} P^{(n-1)}_{\pi(s_i)}(s_i, s_k) \cdot P_{\pi(s_k)}(s_k, s_j), \quad (10)$$

where $P^{(0)}_{\pi(s_i)}(s_i, s_j) = 1$ for $j = i$ and $P^{(0)}_{\pi(s_i)}(s_i, s_j) = 0$ for $j \neq i$. By applying Equ. (10) to an arbitrary MDP policy, we can estimate the PM resource utilization state in long-term operation, and also can select the best MDP policy among different policies. Since the transition from state high to state medium is the most important transition in the MDP-based load balancing algorithm as it eliminates overloaded PMs, we only evaluate the probability of transiting from state high to state medium as an example. The MDP policy that has the highest probability is the best policy because it can elimiate overloaded PMs with the highest probability. For example, given a set of W policies $\{\Pi_1, \Pi_2, ..., \Pi_k, \ldots, \Pi_W\}$ and n = 50. We calculate $P^{(n)}_{\pi(s_i)}(s_i, s_j)$ based on Formula (10) for each policy Π_k, where s_i and s_j represent PM state high and state medium, respectively. We then select the policy Π_w that has the maximum $P^{(n)}_{\pi(s_i)}(s_i, s_j)$ as the best policy

C. A Cloud Profit Oriented Reward System

Recall that the MDP-based load balancing algorithm uses a reward system as one of its inputs and calculates the optimal policy for PMs that maximizes the cumulative expected rewards. Reward $R_a(s, s')$ is an immediate reward given after the transition to state s' from state s by taking action a. Different reward systems represent different preferences on PM state transitions from different actions. In this section, we improve the previous reward system considering its problems listed below.

1) It only aims to avoid overloaded PMs but is not closely related to the datacenter's profit, which is the ultimate goal of the cloud service provider. If the reward system is related to the datacenter's profit, the datacenter's profit can be concurrently maximized when the MDP tries to maximize the rewards.
2) It does not consider the actual VM migration cost or the power cost of PMs.
3) It only roughly gives guidance on how to set the reward values in terms of the relationship (e.g., d<c<b) instead of specifying the actual reward value for each state

transition by taking an action, which otherwise can more accurately reflect the reward.

Therefore, we propose a new reward system, called cloud profit oriented reward system, which is closely related to the cloud profit in the practical scenario. Using such an improved reward system in the MDP model will improve the actual profit of the datacenter.

Datacenter Profit: The prime motive of any datacenter operator is to make most of available resources to cash in as much profit as possible. In this section, we derive the formula for calculating the profit contributed by individual PMs. We denote the profit, the revenue, and the cost over a unit period of time of a PM at time t_i as P, I, and C, respectively. The equation to calculate profit is

$$P_m(t_i) = I_m(t_i) - C_m(t_i) \tag{11}$$

In the following, we explain how to calculate revenue $I_m(t_i)$ and cost $C_m(t_i)$, respectively.

Revenue Calculation: For each unit of time, a virtual machine VM$_n$ contributes E units to the total revenue of datacenter operator, if the resource requirement dictated in the SLA is satisfied. On the other hand, if resource requirement is not satisfied for VM$_n$, penalty of Y units is levied on datacenter operator.

For the revenue calculation, we assume that if a PM is unable to provision the aggregated demanded resource by its resident VMs, all VMs suffer from SLA violations. We detect such scenario by comparing the aggregate VM utilizations for each resource type with pre-defined utilization threshold for each resource type. We use $A_{nm}(t_i) = 1$ to indicate that VM$_n$ is placed on PM$_m$ at time t_i. In the following discussion, we remove t_i from $A_{nm}(t_i)$ for simplicity. Then $\sum_{n=1}^{N} A_{nm}$ is the number of VMs in PM$_m$ at time t_i. As a result, the revenue generated by a machine PM$_m$ at time t_i can be calculated by:

$$I_m(t_i) = \gamma(t_i) \sum_{n=1}^{N} A_{nm} \tag{12}$$

where $\gamma(t_i)$ is calculated by

$$\gamma(t_i) = \begin{cases} E & U_m^k(t_i) \le T_o^k \quad \forall\ k \quad (k = 1, 2, ..., K), \\ -Y & U_m^k(t_i) > T_o^k \quad \exists\ k \quad (k = 1, 2, ..., K) \end{cases} \tag{13}$$

where T_o^k is the threshold for type-k resource. Only when the utilization of all resource are smaller than the corresponding threshold, the revenue is positive; as long as there is one type of resource utilization greater than its threshold, the revenue is negative.

Cost Calculation: We consider power cost and VM live migration overhead for the cost calculation of a PM.

Power Cost: Each active PM consumes electricity and the power consumed is proportional to the CPU utilization level of the PM [27]. Each active PM, even though it is not being utilized, draws some minimal power called static power (denoted by C_{idle}). The power consumption increases with the CPU utilization of the PM and reaches the maximum when the PM has 100% CPU utilization. As proposed by Fan *et al.* [27], the power consumption of a PM, say PM$_m$, at time t_i follows a linear model

$$C_m(t_i) = C_{idle} + \alpha \times U_m^{cpu}(t_i) \tag{14}$$

where $U_m^{cpu}(t_i)$ represents the CPU utilization of PM$_m$ and α is a calibrated constant, which is determined by the commercial model of the server. Note that the power model we use is CPU utilization centric. The power usage of other types of resources such as memory can be assumed to be constant [28] and considered in C_{idle}.

Live Migration Overhead: Live migration of a VM consumes resources both on the source PM from which the VM is being migrated out and on the destination PM to which the VM is being migrated to. In our model, the live migration overhead caused by each VM is captured by extra CPU utilization, which is proportional to a factor β ($0<\beta\le 1$) as in [29]. The extra CPU utilizations introduced to both source and destination PMs vary linearly with the memory utilization of the migrating VM during migration. More specifically, if VM$_n$ requiring $u_n^{mem}(t_i)$ at time t_i is being migrated from PM$_s$ to PM$_d$, the migration overhead exerted on PM$_s$ and PM$_d$ (denoted by $\Delta U_{s,mig}^{cpu}(t_i)$ and $\Delta U_{d,mig}^{cpu}(t_i)$) are calculated by:

$$\Delta\ U_{s,mig}^{cpu}(t_i) = (1 + \beta) u_n^{mem}(t_i) \tag{15}$$
$$\Delta\ U_{d,mig}^{cpu}(t_i) = \beta u_n^{mem}(t_i) \tag{16}$$

Based on Equations (14)-(16), the total power consumption of PM$_m$ by migrating out VM$_p$ and migrating in VM$_q$ can be derived.

Reward Specification: If a PM, say PM$_m$, migrates out a VM and migrates in a VM, after obtaining the CPU utilization of PM$_m$, we can apply Equ. (14) to derive $C_m(t_i)$, and apply Equ. (12) to derive $I_m(t_i)$. Based on Equ. (11) the profit brought by this PM can be calculated.

In our cloud profit oriented reward system, this calculated profit is used to determine the reward, which is given to a transition from state s to state s' when taking action a. In the following, we first discuss the rewards for PM state changes by taking actions of migrating out a VM or no migration. The resulting policy will be used to guide migration VM selections from PMs. The rewards for PM state changes by taking actions of migrating in a VM (accepting a VM) and no migration can be derived similarly, and the corresponding resultant policy will be used to guide destination PM selection for selected migration VMs.

Suppose PM$_m$ is in state s and has CPU utilization $U_m^{cpu}(t_i)$ at time t_i. By taking action a (migrating out a VM in a certain VM-state or no migration), PM$_m$ transits to state s' and has CPU utilization $U_m^{cpu}(t_{i+1})$ at time t_{i+1}. We can calculate the profit from PM$_m$ at these two times, i.e., $P_m(t_i)$ and $P_m(t_{i+1})$, by Equ. (11). The corresponding reward equals the change of profits:

$$R_a(s, s') = P_m(t_{i+1}) - P_m(t_i) \tag{17}$$

D. Destination PM Selection

After a PM identifies the VMs to migrate out, the destination PMs need to be determined to host these migration VMs. In previous methods, a central server identifies the destination PMs where the identified VMs can migrate to [5], [6], and [30]. For example, Sandpiper [30] first defines volume for PMs as $volume = \ (1/(1 - U_{cpu})) * (1/(1 - U_{net}))* (1/(1 - U_{mem}))$, where U is resource utilization, and then selects the PM with the least volume as the destination. A PM can be a VM's destination PM if placing the VM at the PM does not violate the multidimensional capacities. Then, the central server identifies and distributes the PM destinations for each heavily loaded PM in the system. However, though such a method can ensure that the destination PM is not overloaded upon accepting the migration VM, it cannot ensure that this load balance status can sustain for a long time.

In order to maintain a long-term load balance states of these destination PMs while fully utilizing PM resources, we again develop a similar MDP-based model to determine the destination PMs. A central server runs the MDP and selects the PMs that are most suitable to accept migration VMs based on VM-states. We use MDP* to denote the method that uses an MDP model for determining the migration VMs and uses another MDP model for determining the destination PMs. Compared to the previous MDP model, MDP* has the same state set S. Its action set A is accepting a VM in a certain VM-state or not accepting any VM. Recall that by defining such an action set, we can ensure that A does not change, which is required by MDP. The transition probability $P_a(s_i, s_j)$ is defined as the probability of PM in state s_i transiting to state s_j after performing action $a \in A$. MDP* model uses the information from the trace of state changes when PM accepts VMs to build the transition probability matrix. The central server keeps track of the resource utilization status of the PMs when they accept VMs or take no action. The method introduced in Section IV-B is used for the probability calculation.

The rewards given to a PM after performing action $a \in A$ should encourage PMs to accept VMs while avoiding heavy state in a long term. Accordingly, the reward system is designed as follows for the state transition of each resource:

1) Positive reward for a transition to a low/medium state.
2) Negative reward for a transition to a high state.
3) The reward for a transition to a medium state is higher than to a low state.
4) The reward for actions of accepting a VM in different VM-states follows: high>medium>low. These rewards should be higher than the reward for "no action".

Similar as the migration VM selection, the optimal action set is created that leads to the highest expected reward. Then, this optimal action set is used to decide whether a PM should accept a VM or not. For a given migration VM, the central server can identify the most appropriate destination PMs based on the MDP that generate the highest expected reward. Better options from these PMs can be further identified based on additional consideration factors such as VM communication cost and migration distance [10].

E. An MDP with Extended Action Set

As indicated previously, two MDP models are needed to conduct the MDP-based load balancing; one MDP model is for selecting migration VMs from PMs to migrate out and the other MDP model is for selecting destination PMs for hosting the migration VMs. Building two MDP models brings about a high overhead. More importantly, the policies generated by these two MDP models may lead to contradiction of actions for a PM because the former MDP model uses the action set of migrating VMs out of PMs, while the latter MDP model uses the action set of migrating VMs into PMs. For example, the policy of the former model suggests migrating a VM out of a PM while the policy of the latter model suggests migrating a VM to the PM. The conflict could be solved by filtering out all PMs that want to migrate out VMs and only consider migrating in VMs for all the remaining PMs, if we give higher priority to migrating out VMs than accepting VMs. However, we still need to solve the high overhead problem caused by two MDP models. Therefore, we propose to develop a comprehensive MDP, we extend the action set to cover all possible migration actions (including migrating out VMs and migrating in VMs) of a PM. We introduce each component of this comprehensive MDP below.

State: Similar to the previous MDP model, the states are defined as the combination of different load levels of different types of resources (e.g., CPU-low, Mem-high). We adopt the same thresholds to distinguish different load levels as in Section III-C (i.e., $T_1 = 0.3$, $T_2 = 0.8$).

Action: We create a new action set by combining the actions in the two MDP models, i.e., actions for migrating out VMs in different VM-states and migrating in VMs in different VM-states. Unlike the previous MDP models, we now have two types of actions corresponding to every VM state. That is, migrating out a VM in this state and migrating in a VM in this state. Recall that there are $|L|^{|R|}$ VM states in total in the previous MDP model. Then, the extended action set consists of $2(|L|^{|R|}) + 1$ elements and the "1" represents "no action".

Probability: Since the probabilities are specified with respect to every action (e.g., the probability for a PM state transition when taking an action), we need to combine the probabilities of migrating VMs out (MDP) and the probabilities of accepting VMs (MDP*).

Reward: The rewards for migrating VM out or no migration are the same as in Section IV-C. The rewards for migrating VM in can be derived similarly as for migrating VM out.

V. Performance Evaluation

In this section, we conducted trace-driven experiments on CloudSim [23] to evaluate the performance of our proposed MDP-based load balancing algorithm in a two-resource environment (i.e., CPU and Mem). We used the VM utilization trace from PlanetLab [23] and Google Cluster [24] to generate VM workload to determine the transition probability matrix in our MDP model. We implement two versions of our MDP load balancing algorithm, represented by MDP and MDP*. In order to solely show the advantage of MDP on VM selection, MDP uses our MDP model for identifying VMs to migrate and adopts the PM selection algorithm as Sandpiper (Section IV-D). MDP* uses our MDP model for both VM selection and destination PM selection. We compared MDP and MDP* with Sandpiper [5] and CloudScale [11] in terms of the number of VM migrations, the number of overloaded PMs, and time and resource consumptions. We use Sandpiper to represent reactive load balancing algorithms and use CloudScale to represent proactive load balancing algorithms.

We simulated the cloud datacenter with 100 PMs hosting 1000 VMs. The PMs are modeled from commercial product HP ProLiant ML110 G4 servers (1860 MIPS CPU, 4GB

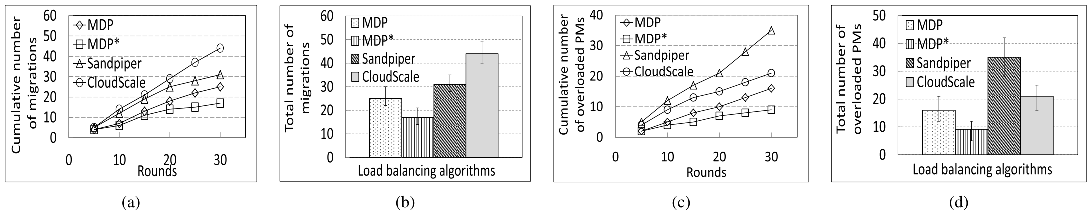


Fig. 4. Performance using the PlanetLab trace. (a) Cumulative # of VM migrations. (b) Total # of VM migrations. (c) Cumulative # of overloaded PMs. (d) Total # of overloaded PMs.

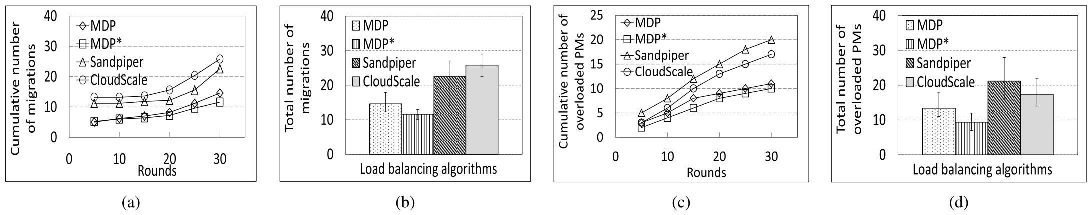


Fig. 5. Performance using the Google Cluster trace. (a) Cumulative # of VM migrations. (b) Total # of VM migrations. (c) Cumulative # of overloaded PMs. (d) Total # of overloaded PMs.

memory) and the VMs are modeled from EC2 micro instance (0.5 EC2 compute unit, 0.633 GB memory, which is equivalent to 500 MIPS CPU and 613 MB memory). The resource utilization trace from PlanetLab VMs and Google Cluster VMs are used to drive the VM resource utilizations in the simulation. We repeatedly carried out each experiment for 20 times and reported the results. At the beginning, the VMs are randomly allocated to the PMs. We used this VM-PM mapping for different load blanching algorithms in each experiment to have fair comparison. When the simulation is started, the simulator calculates the resource utilization status of all the PMs in the datacenter every 300 seconds, and records the number of VM migrations and the number of overloaded PMs (the occurrence of overloaded PMs) during that period. In each experiment round, each PM conducts load balancing once and waits for 300 seconds before the next load balancing execution. We used $T_1 = 0.3$ and $T_2 = 0.8$ as the resource utilization thresholds for both CPU and memory usage. Note that all methods conduct load balancing every 300 seconds. When a PM conducts load balancing, in Sandpiper and CloudScale, it performs VM migration if it detects that it is overloaded, while in MDP and MDP*, it chooses the action to perform that results in the maximal expected rewards in MDP and MDP*.

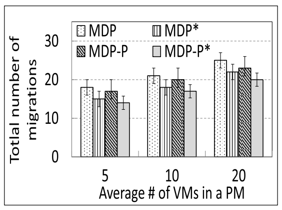
A. Performance of the Basic MDP

1) The Cumulative Number of Migrations: Figure 4 and Figure 5 show the performance of MDP, MDP*, Sandpiper and CloudScale with the PlanetLab trace and Google Cluster trace, respectively. Figure 4(a) and Figure 5(a) show the cumulative number of migrations over the rounds. Both results follow MDP*<MDP<Sandpiper<CloudScale. MDP and MDP* outperform Sandpiper and CloudScale because each PM can find the best actions to perform to keep a long-term load balance state while triggering a smaller number of VM migrations. Compared to MDP, MDP* further reduces the number of VM migrations due to the reason that it additionally selects the most suitable destination PMs for VM migrations based on MDP model, and hence results in a long-term load balance state, which helps reduce the number of VM migrations. CloudScale generates a larger number of VM migrations than Sandpiper in each round because CloudScale migrates VMs not only for a correctly predicted overloaded PM but also for an incorrectly predicted overloaded PM, but Sandpiper only migrates VMs for occurred overloaded PMs. Figure 4(b) and Figure 5(b) show the median, the 10th and 90th percentiles of the total number of VM migrations in the experiments. Due to the random VM to PM mapping at the beginning of simulations, the number of migrations varies in different simulations. Statistically, MDP* generates fewer VM migrations than MDP, MDP generates fewer VM migrations than Sandpiper, and Sandpiper generates fewer VM migrations than CloudScale due to the same reasons mentioned before. These results confirm that MDP and MDP* are advantageous in maintaining a long-term load balance state and minimizing the number of VM migrations, hence reducing load balancing overhead. Also, our MDP model is effective in both migration VM selection and destination PM selection to maintain a long-term load balance state.

2) The Number of Overloaded PMs: Next, we measure the number of overloaded PMs, which indicates the effectiveness of load balancing algorithms. Figure 4(c) and Figure 5(c) show the cumulative number of overloaded PMs over rounds. MDP and MDP* generate a smaller number of overloaded PMs in each round than CloudScale and Sandpiper. This is because the MDP algorithm incentivizes the PMs to perform optimal VM migration actions to maintain a system load balance state for a longer time. MDP* outperforms MDP with fewer overloaded PMs since it further uses the MDP model for the destination PM selection to maintain a long-term load balance state. CloudScale produces fewer overloaded PMs than Sandpiper because its predicted overloaded PMs migrate VMs out before they become overloaded, while Sandpiper conducts VM migrations upon the PM overload occurrence. Figure 4(d) and Figure 5(d) show the median, the 10th and 90th percentiles of the total number of overloaded PMs in the experiments. The results follow MDP*<MDP<CloudScale<Sandpiper due to the same reasons indicated previously.

B. Performance of the MDP with the Cloud Profit Oriented Reward System

We then study the performance of the MDP using the cloud profit oriented reward system introduced in Section IV-C.


(a)

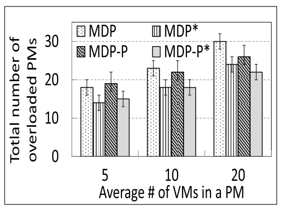
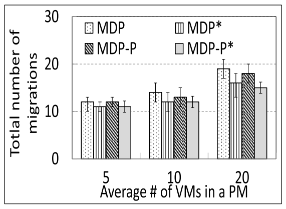

(b)

Fig. 6. Performance of cloud profit oriented reward system (PlanetLab trace). (a) The number of VM migrations. (b) The number of overloaded PMs.


(a)

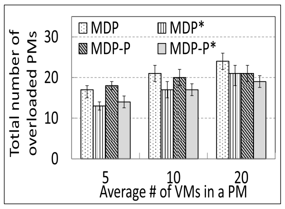
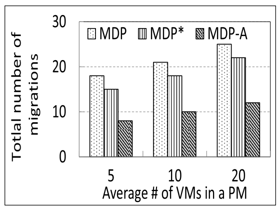

(b)

Fig. 7. Performance of cloud profit oriented reward system (Google Cluster trace). (a) The number of VM migrations. (b) The number of overloaded PMs.

We denote the MDP with this improved reward system as MDP-P and denote MDP* with the improved reward system as MDP-P*, and compare them with MDP and MDP*. As the improved reward system needs the average number of VMs in a PM as indicated in Equ. (12), we increased the average number of VMs in a PM from 5 to 20 to study the performance. When computing the rewards, we set $\alpha = 1$, $\beta = 1$, $E = 10$ for unit revenue and $Y = 10$ for penalty. For each average number of VMs, we apply Algorithm 1 to find the optimal policies. For each optimal policy, we applied it to CloudSim and repeated the simulation for 20 times.

1) The Number of VM Migrations: Figure 6 and Figure 7 show the experimental results with the PlanetLab trace and Google Cluster trace, respectively. Figure 6(a) and Figure 7(a) show the median, the 10th and 90th percentiles of the number of VM migrations of the four methods with different average number of VMs per PM. The number of VM migrations follows MDP-P<MDP and MDP-P*<MDP*. Compared to MDP, MDP-P reduces the number of VM migrations because MDP-P more focuses on the memory resource utilization of the migrating VM than the CPU resource utilization. For example, MDP uses Equ. (5) to construct the reward system, which does not explicitly reflect memory utilization of the migrating VMs. MDP-P relies on Equ. (17), which incorporates Equ. (15) and Equ. (16) to explicitly consider memory utilization of the migrating VM. Therefore, the reward system in MDP-P discourages migrating a VM with heavy memory resource utilization, a portion of the VM migrations in MDP are prevented. For example, when the cost of migrating a VM with intensive memory utilization surpasses the penalty of violating the SLA of this VM, this VM will not be migrate out. As a result, MDP-P produces fewer VM migrations. The result of MDP-P*<MDP* is caused by the same reasons. The number of VM migrations increases with the average number of VMs because the workload in the PMs increases.

Figure 6(b) and Figure 7(b) show the median, the 10th and 90th percentiles of the number of overloaded PMs of the four methods with different average number of VMs per PM.


(a)

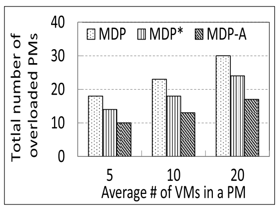

(b)

Fig. 8. Performance of MDP-A (Google Cluster trace). (a) The number of VM migrations. (b) The number of overloaded PMs.

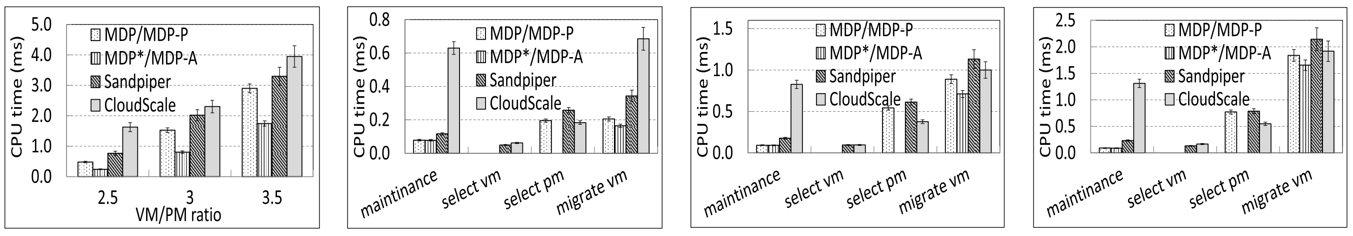
When the average number is 5, MDP has a smaller number of overload PMs than MDP-P. The reason is that the cost for violating SLA is relatively smaller as Equ. (12) indicates, i.e., violating SLA leads to a relatively smaller loss of revenue, than the cost of migration VMs. As a result, the MDP-P model tries to reduce the number of VM migrations at the cost of sacrificing SLA guarantees. When the average number is 10 and 20, MDP-P achieves a smaller number of overload PMs than MDP because violating SLA becomes more expensive and even a small number of the PMs in an overloaded status will lose a high amount of revenue. The relationship of MDP* and MDP-P* stays similar as the relationship of MDP and MDP-P due to the same reasons. The number of overloaded PMs increases with the average number of VMs because the workload of a PM increases with the number of VMs in the PM.

C. Performance of the MDP with Extended Action Set

We then study the performance of the MDP with extended action set denoted by MDP-A. Similar as previous experiments, we increased the average number of VMs in a PM from 5 to 20. For each average number of VMs, we applied the optimal policies corresponding to each algorithms (i.e., MDP, MDP* and MDP-A) to CloudSim and repeated the simulation for 20 times. Figure 8 presents the experimental results with the Google Cluster trace. Figure 8(a) shows the number of VM migrations of the algorithms. The number follows MDP-A<MDP*<MDP. MDP-A reduces the number of VM migrations because MDP-A considers both migrating VM out and accepting VM in the same MDP model and hence produces an optimal policy that avoids any conflicts of the actions as in MDP*. As a result, MDP-A tends to avoid unnecessary VM migrations. Figure 8(b) shows the number of overloaded PMs. The number follows MDP-A<MDP*<MDP because MDP-A reduces the number of overloaded PMs since PMs conduct actions according to a policy that is produced by one MDP model. In this case, the PMs are able to avoid being overloaded due to action conflicts (e.g., they migrate VMs out according to one MDP model and at the same time accept VMs according to the other MDP model).

D. Comparison of Different MDP Models

1) CPU Time for Load Balancing: The CPU time consumption for load balancing consists of the maintenance time spent on system monitoring, the time identifying VMs to migrate, the time to determine destination PMs for VMs and the time for VM migrations. The maintenance time refers to the CPU



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

Fig. 9. Comparison of CPU time consumption by different methods to achieve load balance. (a) Total time. (b) CPU time breakdown (ratio = 2.5). (c) CPU time breakdown (ratio = 3). (d) CPU time breakdown (ratio = 3.5).

time spent on checking whether there are overloaded PMs and determining whether VM migration is necessary in each round. MDP-P differs from MDP only in using a different reward system, while MDP-A differs from MDP* only in applying one MDP model to select VMs and PMs by using an extended action set, the CPU time consumption for load balancing and the time breakdowns of MDP-P and MDP-A are similar to MDP and MDP*, respectively. In the figures, we present the results of MDP-P together with MDP, and MDP-A together with MDP*. Figure 9(a) shows the median, the 10th and 90th percentiles of the CPU time consumption to achieve load balance in the four methods under different VM/PM ratios with 100 PMs. We see that the CPU time increases as the ratio increases for all four methods. As the ratio increases, the system needs more CPU resource to predict and monitor the workload status of more VMs. For each VM/PM ratio, the CPU time consumption follows MDP*<MDP<Sandpiper<CloudScale. CloudScale consumes more CPU time than the other methods due to two reasons. First, CloudScale needs to predict the load of each VM and hence needs more CPU time. Second, CloudScale has relatively more VM migrations, which consumes more VM migration CPU time. MDP consumes less time than Sandpiper since it can quickly make VM migration decisions and has a smaller number of VM migrations. MDP* consumes the least CPU time since it can quickly select both migration VMs and destination PMs.

In order to give a thorough comparison between the four methods, we broke down the CPU time to different parts as shown in Figure 9(b), Figure 9(c) and Figure 9(d) corresponding to three VM/PM ratios. MDP and MDP* consume the least maintenance time that is used to determine whether VM migrations are needed. In MDP and MDP*, each PM only needs to refer to the optimal policy Π and hence they require less CPU time. Sandpiper consumes more CPU time in maintenance than MDP and MDP* since it needs to calculate the $volume$ [5] of each PM to check the load status of the PMs. CloudScale consumes much more CPU time since it needs to predict the workload status of each VM and also predict the PM workload status to determine whether VM migrations are needed.

The time to identify VMs to migrate refers to the CPU time needed to determine which VMs to migrate when a PM is overloaded. MDP and MDP* refer to the optimal policy Π and quickly select VM to migrate in each round, and hence need little CPU time, while Sandpiper needs a relatively long CPU time to calculate the volume-to-size (VSR) ratio of each VM. Sandpiper consumes slightly less CPU time than CloudScale because Sandpiper does not need to predict each VM workload and it selects fewer VMs than CloudScale due to fewer VM migrations. The time to determine destination PMs is the CPU time for determining destination PMs where the selected VMs migrate to. MDP* quickly selects destination PMs by referring to the optimal policy Π derived from the MDP model and hence needs the least CPU time. MDP and Sandpiper use the same PM selection algorithm, so their CPU time is dominated by the number of VMs that need to migrate. MDP consumes a slightly less CPU time than Sandpiper due to fewer VM migrations. CloudScale uses a greedy algorithm to find the least loaded destination PM and hence consumes less CPU time than MDP and Sandpiper. The VM migration time depends on the number of VM migrations and it follows MDP*<MDP<Sandpiper<CloudScale.

VI. Conclusion

In this paper, we propose an MDP-based load balancing algorithm as an online decision making strategy to enhance the performance of cloud datacenters. Compared to the previous reactive load balancing algorithms, the MDP-based load balancing algorithm maintains the load balance state for a longer time (hence lower SLAV) and also produces low load balancing delay and overhead. Its advantages compared to previous proactive load balancing algorithms are three-fold. First, it aims to maintain a long-term load balance for both the source PM that performs VM migrations to release its workload and the destination PM that accommodated this VM, and hence prevents subsequent load imbalance. Second, it quickly determines which VMs to migrate out to achieve load balance, which eliminates the need of additional operations to identify migration VMs. Third, it quickly determines destination PMs with much less overhead and delay. We also develop a cloud profit oriented reward system to improve the actual profit of the datacenter. We further develop a new MDP model with an extended action set, which considers the actions of both migrating a VM out of a PM and migrating a VM into a PM. Our trace-driven experiments show that the MDP-based load balancing algorithm outperforms previous reactive and proactive algorithms. MDP is able to maintain the system in a relatively balanced state with a smaller number of PM overload occurrences in the system by triggering fewer number of VM migrations. Further, MDP consumes less CPU time and memory under different workload scenarios. The experimental results also show the effectiveness of the cloud profit oriented reward system and the new MDP model. In our future work, we aim to make our algorithm fully distributed to increase its scalability. In the future work, we can consider combine considering VM workload patterns to improve the performance of the MDP-based load balancing algorithm. For example, when selecting migration VMs or deciding accepting VMs, the MDP model can further consider the future resource utilization of VMs based on the workload patterns for achieving better long-term load balance. We will also study how to consider the time-sharing feature of some resources (e.g., CPU, bandwidth) in our algorithm to achieve better performance [31]. Recall that we did not consider the size of the VMs due to the equal size of VMs and also assume the same capacities of PMs. The sizes of VMs and the capacities of PMs may be different in practice. We will study this issue in our future work. Also, it may be true that considering too much of the future might affect the effectiveness of handling the present. We will study the short-term load balance performance of the MDP. Further, a PM overload does not necessarily mean a real SLA violation so we will study how the PM overloads affect the real SLA violation.

Acknowledgements

The authors would like to thank K. Sapra for his discussions.

References

[1] Amazon Web Service. (2016). [Online] http://aws.amazon.com/

[2] (2016). *Microsoft Azure*. [Online] http://www.windowsazure.com

[3] BEA System Inc. (2016). [Online] http://www.bea.com

[4] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson, “Cost-benefit analysis of cloud computing versus desktop grids,” in *Proc. IPDPS*, May 2009, pp. 1–12.

[5] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, “Sandpiper: Black-box and gray-box resource management for virtual machines,” *Comput. Netw.*, vol. 53, no. 17, pp. 2923–2938, 2009.

[6] M. Tarighi, S. A. Motamedi, and S. Sharifian, “A new model for virtual machine migration in virtualized cluster server based on fuzzy decision making,” *CoRR*, vol. 1, no. 1, pp. 40–51, Feb. 2010.

[7] E. Arzuaga and D. R. Kaeli, “Quantifying load imbalance on virtualized enterprise servers,” in *Proc. WOSP/SIPEW*, 2010, pp. 235–242.

[8] A. Singh, M. R. Korupolu, and D. Mohapatra, “Server-storage virtualization: Integration and load balancing in data centers,” in *Proc. SC*, 2008, Art. no. 53.

[9] A. Sallam and K. Li, “A multi-objective virtual machine migration policy in cloud systems,” *Comput. J.*, vol. 57, no. 2, pp. 195–204, 2013.

[10] L. Chen, H. Shen, and S. Sapra, “RIAL: Resource intensity aware load balancing in clouds,” in *Proc. INFOCOM*, Apr./May 2014, pp. 1294–1302.

[11] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, “CloudScale: Elastic resource scaling for multi-tenant cloud systems,” in *Proc. SOCC*, 2011, Art. no. 5.

[12] A. Beloglazov and R. Buyya, “Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 7, pp. 1366–1379, Jul. 2013.

[13] U. Sharma, P. J. Shenoy, S. Sahu, and A. Shaikh, “A cost-aware elasticity provisioning system for the cloud,” in *Proc. ICDCS*, Jun. 2011, pp. 559–570.

[14] N. Bobroff, A. Kochut, and K. Beaty, “Dynamic placement of virtual machines for managing SLA violations,” in *Proc. IM*, May 2007, pp. 119–128.

[15] Z. Gong, X. Gu, and J. Wilkes, “PRESS: Predictive elastic resource scaling for cloud systems,” in *Proc. CNSM*, Oct. 2010, pp. 9–16.

[16] A. Chandra, W. Gong, and P. J. Shenoy, “Dynamic resource allocation for shared data centers using online measurements,” in *Proc. SIGMETRICS*, 2003, pp. 300–301.

[17] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA, USA: MIT Press, 1960.

[18] L. Chen and H. Shen, “Consolidating complementary VMs with spatial/temporal-awareness in cloud datacenters,” in *Proc. INFOCOM*, Apr./May 2014, pp. 1033–1041.

[19] G. V. Laszewski, L. Wang, A. J. Younge, and X. He, “Power-aware scheduling of virtual machines in DVFS-enabled clusters,” in *Proc. CLUSTER*, Aug./Sep. 2009, pp. 1–10.

[20] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, “Virtual machine power metering and provisioning,” in *Proc. SOCC*, 2010, pp. 39–50.

[21] X. Xu, K. Teramoto, A. Morales, and H. H. Huang, “DUAL: Reliability-aware power management in data centers,” in *Proc. CCGrid*, May 2013, pp. 530–537.

[22] A. Beloglazov and R. Buyya, “Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers,” *Concurrency Comput., Pract. Exper.*, vol. 24, no. 13, pp. 1397–1420, 2012.

[23] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, “CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms,” *Softw., Pract. Exper.*, vol. 14, no. 1, pp. 23–50, 2011.

[24] (2016). *Google Cluster Data*. [Online]. Available: https://code.google.com/p/googleclusterdata/

[25] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.

[26] C. W. Gardiner *et al.*, *Handbook of Stochastic Methods*. vol. 4. Berlin, Germany: Springer, 1985.

[27] X. Fan, C. S. Ellis, and A. R. Lebeck, “The synergy between power-aware memory systems and processor voltage scaling,” in *Proc. PACS*, 2005, pp. 164–179.

[28] L. Minas and B. Ellison, *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. Mountain View, CA, USA: Intel Press, 2009.

[29] H. Liu, H. Jin, C.-Z. Xu, and X. Liao, “Performance and energy modeling for live migration of virtual machines,” *Cluster Comput.*, vol. 16, no. 2, pp. 249–264, 2013.

[30] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, “Black-box and gray-box strategies for virtual machine migration,” in *Proc. NSDI*, 2007, p. 17.

[31] L. Chen and H. Shen, “Considering resource demand misalignments to reduce resource over-provisioning in cloud datacenters,” in *Proc. INFOCOM*, 2016, pp. 1–9.

[32] L. Chen, H. Shen, and K. Sapra, “Distributed autonomous virtual resource management in datacenters using finite-Markov decision process,” in *Proc. SOCC*, 2014, pp. 1–13.

Haiying Shen (SM’13) received the B.S. degree in computer science and engineering from Tongji University, China, in 2000, and the M.S. and Ph.D. degrees in computer engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor with the Computer Science Department, University of Virginia. Her research interests include cloud computing and cyberphysical systems. She was the program co-chair for a number of international conferences and a member of the Program Committees of many leading conferences. She is a Microsoft Faculty Fellow of 2010 and a member of the ACM.

Liuhua Chen received the B.S. and M.S. degrees from Zhejiang University in 2008 and 2011, respectively. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Clemson University. His research interests include distributed and parallel computer systems, and cloud computing.