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

Liuhua Chen, Haiying Shen and Karan Sapra
Department of Electrical and Computer Engineering
Clemson University, Clemson, South Carolina 29634
{liuhuac, shenh, ksapra}@clemson.edu

Abstract

To provide robust infrastructure as a service (IaaS), 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, energy) and delay by quickly identifying VMs and destination PMs to migrate. 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.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Design studies
General Terms Algorithms, Design, Performance
Keywords MDP, Cloud computing, Resource management

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2671003

1. 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 [15] 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 [4, 11, 17, 20, 21, 23] 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 [7, 8, 10, 12, 18, 19] 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, it cannot maintain a long-term load balance because it only achieves load balance at the predicted time spot. Third, it needs to build a Markov chain 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 [14] 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.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the overview and the detailed design of our MDP-based load balancing algorithm. Section 4 presents the performance evaluation of our algorithm compared with other load balancing algorithms in trace driven simulations. Finally, Section 5 concludes this paper with remarks on our future work.

2. Related Work

In recent years, many load balancing methods have been proposed to avoid overloaded PMs in the clouds [4, 11, 17, 20, 21, 23]. 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. Sandpiper [23] carries out dynamic monitoring and hotspot probing on PMs. It identifies overloaded PM, in which the resource usage exceeds a certain threshold. For each overloaded PM, Sandpiper calculates the volume for each VM in the PM, which is defined as the inverse reciprocate product of the resources. Each VM has a volume-to-size ratio (VSR), where the size is the memory footprint of the VM. When selecting VMs to migrate from a hotpot PM, Sandpiper considers VMs in decreasing order of VSRs, and attempts to migrate the VM with the maximum VSR to the PM with the least volume. Tarighi *et al.* [21] first formed a decision matrix with different weights assigned to each specific resource of the VMs (or PMs). They then determined the ideal solution by using maximum value for the benefit criteria and minimum value for the cost criteria. Finally, they calculated the Euclidean distance of each VMs (or PMs) from the ideal solution to select the most suitable candidates. Arzuaga *et al.* [4] used predetermined weights for resource

usage to measure virtualized server load. For VM migration, they proposed to select the VM that yields the greatest improvement of the imbalance metric (determined by the ratio of standard deviation over the mean) at its present state. VectorDot [20] utilizes vectors to represent the multidimensional resource requirements of VMs for migration. In order to ensure successful migration and reduce migration cost, it migrates VMs from an overloaded PM to a PM that has the lowest vector product of item path vector and node path vector. Sallam *et al.* [17] proposed a migration policy consolidated by a new elastic multi-objective optimization strategy to evaluate different objectives (including migration cost) simultaneously. Chen *et al.* [11] proposed a resource intensity aware load balancing method to dynamically assign different weights to different resources according to their usage intensity in the PM, which reduces the time and cost to achieve load balance.

Many methods [7, 8, 10, 12, 18, 19] 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. Gong *et al.* [12] proposed an online resource demand prediction model, which uses a hybrid approach that employs signature-driven and state-driven prediction algorithms. Based on this work, CloudScale [19] performs prediction error handling to achieve adaptive resource allocation. Using the predicted load, the system migrates VMs to prevent overloading PMs before they become overloaded. Sharma *et al.* [18] presented a replication and migration scheme and focused on optimizing customer cost using linear programming. Bobroff *et al.* [8] introduced a dynamic server migration and consolidation algorithm, which collects resource demand data and uses a sliding window of this data to predict resource demand in the next interval. Chandra *et al.* [10] proposed a workload prediction algorithm using auto-regression and histogram based methods. Beloglazov *et al.* [7] introduced a Markov model for host overload detection, which is used to decide when a VM should be migrated from the host to satisfy a quality-of-service requirement, while maximizing the time between VM migrations.

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.

3. MDP-based Load Balancing

3.1 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) [6]. 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.

3.2 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 [14] 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 [7, 10, 12, 18, 19], 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. 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 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.

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 3.4, we first introduce the definitions of the load states in Section 3.3.

3.3 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 $l_{t_i}^{nk}$ to denote the demanded resource amount (i.e., load) of the type-k resource in the n-th VM at time t_i. We use $L_{t_i}^{mk}$ and $C_{t_i}^{mk}$ 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_i number of VMs, then $L_{t_i}^{mk} = \sum_{j=1}^{n_i} l_{t_i}^{jk}$.

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

$$u_{t_i}^{nk} = l_{t_i}^{nk} / c_{t_i}^{nk}, \tag{1}$$

where $l_{t_i}^{nk}$ and $c_{t_i}^{nk}$ 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

$$U_{t_i}^{mk} = L_{t_i}^{mk} / C_{t_i}^{mk} = \sum_{j=1}^{n_i} l_{t_i}^{jk} / C_{t_i}^{mk}. \tag{2}$$

We use T^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 $U_{t_i}^{mk} \leq T^k$ 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.

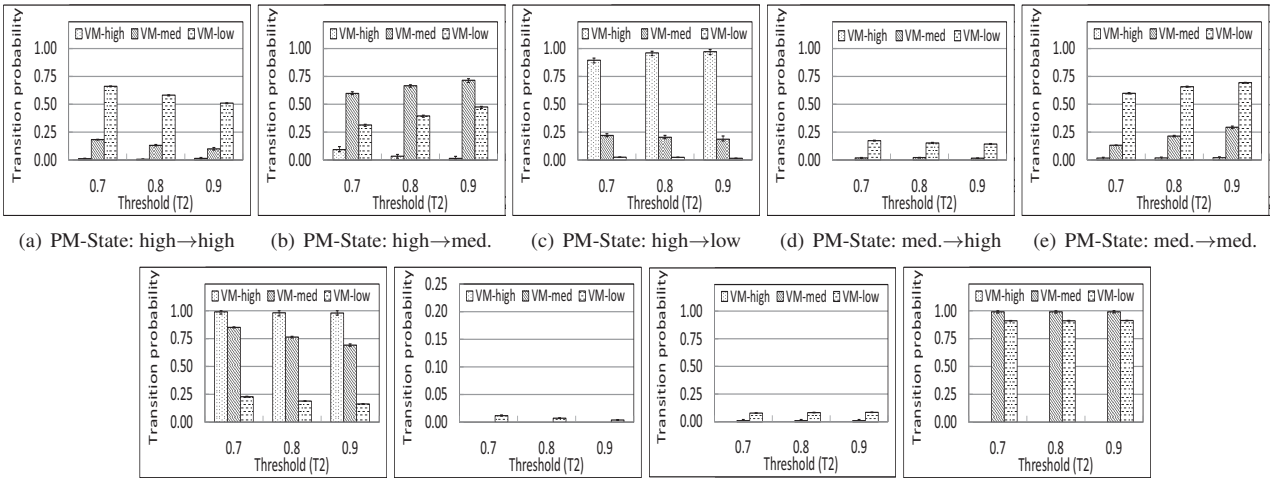
$$\begin{cases} Low & \text{if } U^k < T_1^k \\ Medium & \text{if } U^k \geq T_1^k \text{ and } U^k < T_2^k \\ High & \text{if } U^k \geq T_2^k \end{cases} \tag{3}$$

The state determination of VMs is performed in the same manner by changing U^k in Equation (3) to u^k. If the utilization of at least one resource in a PM reaches the heavily loaded threshold, this PM is heavily loaded. Only when the utilizations of all resources in a PM do not reach the heavily loaded threshold, this PM is lightly loaded.

Consider a set of n resources $\mathcal{R}$=$\{r_1, r_2, ...r_n\}$ in the cloud system and resource utilization levels L=$\{$High, Medium, Low$\}$. The total number of states of VMs or PMs equals $|L|^{|\mathcal{R}|}$; the Cartesian product of the two sets. The set of states is S=$\mathcal{R}\times L$, where$\times$means the combination of r_i 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 3(a).

3.4 Trace Study on the Stability of Our MDP

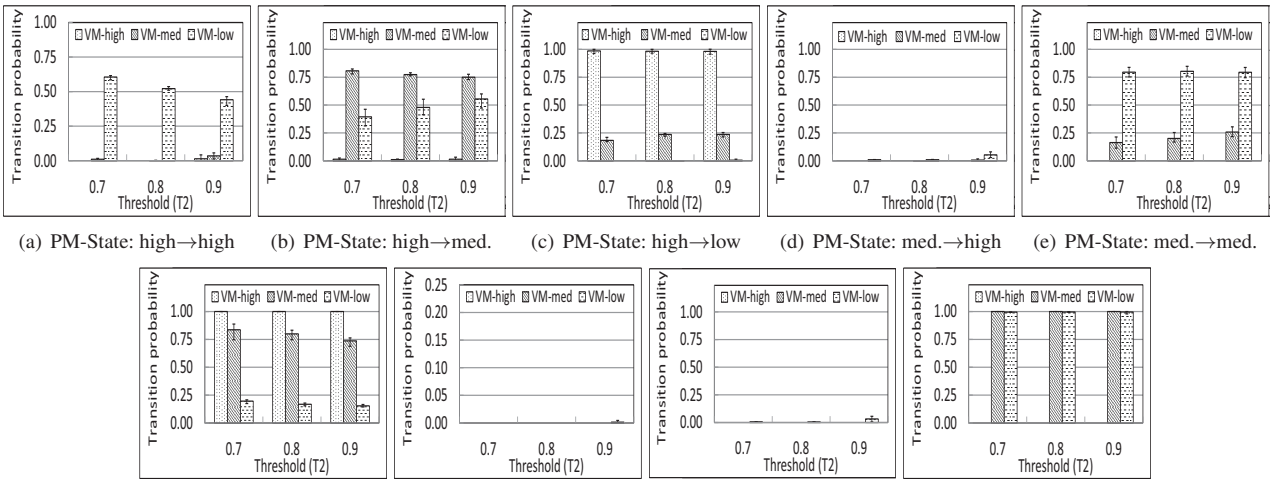
State set S is a set of PM resource utilization levels based on Equation (3). As mentioned before, the transition probabilities of an MDP must be stable. To confirm whether our design of different MDP components can achieve the MDP stability, in this section, we conduct an experiment, which shows that 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.



(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

Figure 1. Probability of state transitions of PM-high using PlanetLab trace.



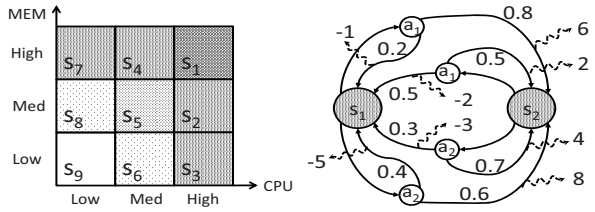
(a) PM-State: high→high (b) PM-State: high→med. (c) PM-State: high→low (d) PM-State: med.→high (e) PM-State: med.→med.

(f) PM-State: med.→low (g) PM-State: low→high (h) PM-State: low→med. (i) PM-State: low→low

Figure 2. Probability of state transitions of PM-high using Google Cluster trace.

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 [9] for the experiments and compared the transition probability matrix obtained under varying threshold T_2^k values. We used two traces in the experiments: PlanetLab trace [9] and Google Cluster trace [13]. 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 of states when considering multiple resources, we focus on the CPU resource in the experiments. In each test, we first generated a random number of VMs ranging from 1 to 20, and assigned these VMs to a PM. 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. 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 1 and Figure 2 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 1(c) and Figure 2(c) indicate that a PM-



(a) PM states. (b) MDP model.

Figure 3. Example of a simple MDP.

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 1(i) and Figure 2(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 1(g) and Figure 2(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 1, the absolute values of the transition probabilities derived from the Google Cluster trace in Figure 2 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.

3.5 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 3(b) illustrates the transition model of a simple MDP with two states and two actions. The 3×3 table in Figure 3(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 3.6 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.

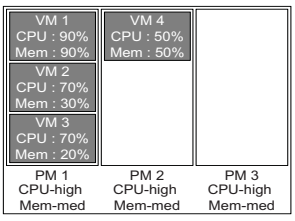


Figure 4. PM and VM state determination in a cloud.

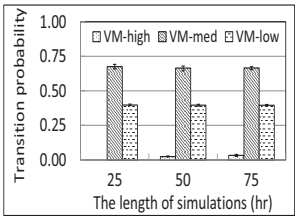


Figure 5. Transition probability vs. simulation time.

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 3.6, 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.

	aH			aH			aH		
	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	0.00	1.00	0.91

Table 1. Probabilities with threshold $T_2 = 0.8$.

3.6 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 3.3. 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 3.3, 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.

Consider the example shown in Figure 4. Based on Equation (3) with T_1=0.3 and T_2=0.8, we can determine the states and actions. PM_1's state is (CPU-high, Mem-medium), PM_2's state is (CPU-medium, Mem-medium) and PM_3's state is (CPU-low, Mem-low). VM_1's state is (CPU-high, Mem-high), the state of VM_2 and VM_3 is (CPU-high, Mem-low), and VM_4's state is (CPU-medium, Mem-medium). Moving out a VM in one of these VM-states is an available action that this PM can carry out. With the reward system, the MDP-based load balancing algorithm motivates PMs to choose actions to become or remain lightly loaded while reducing VM migrations and maintaining load balance state for a longer time.

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 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 3.4. Table 3.5 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 3.5 will be used in our experiments in Section 4.

We also carried out the experiment with varying simulation time to study the stability of the transition probabilities. Figure 5 shows the results of the experiments with trace length of 25, 50 and 75 hours. We selected the transition probabilities of PM state changes of high→medium after migrating VM-high, VM-medium and VM-low as representative examples to show in the figure. We see that the transition probability is stable regardless of the length of trace. The probability of PM high→medium transitions when migrating VM-high in the 25 hour simulation is zero. Though this probability is nonzero in the longer simulation experiments, its magnitude is still very close to zero, which means that the probability matrix is relatively stable.

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):
   (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,

$$\mathcal{R} = \mathcal{R}_L \cup \mathcal{R}_M \cup \mathcal{R}_H. \quad (4)$$

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:

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

where d is a negative reward and 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 $c' > b' > d'$.

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

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

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

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, which contains the reward associated with a state, and policy π which contains actions for the states. The optimal policy for an MDP is a collection of actions performed in each state that maximize the reward, which in return makes a PM attain a lightly loaded state and sustains for a longer period of time. At the end of the algorithm, π contains the most suitable action for each state to take that would result in the maximum value V for that specific state, and $V(s)$ will contain the sum of the rewards to be earned (on average) by following the action from state s. The algorithm has the following two steps, which are repeated in some order for all the states until no further changes take place:

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

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

Algorithm 1 The iterative value iteration algorithm.

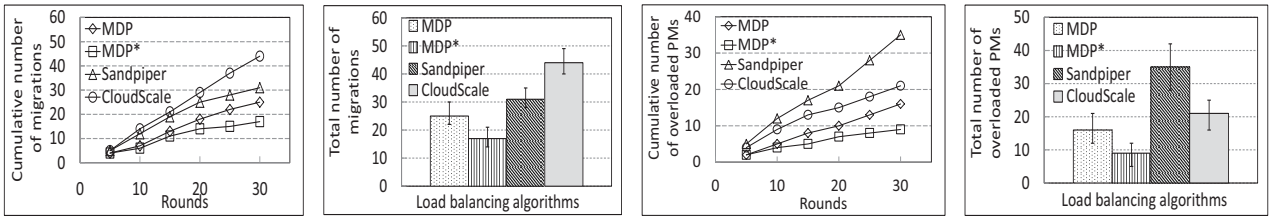
Require: T, a transition probability matrix
Require: R, a reward matrix.
Ensure: Policy π
1: $V \leftarrow 0, V_{new} \leftarrow R$
2: **while** $max|V(s_i) - V_{new}(s_i)| \geq e$ **do**
3: $V \leftarrow V_{new}$
4: **for all** state i in S **do**
5: $V_{new}(s_i) \leftarrow R(s_i) + max_a \sum_j P(s_i,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

Equation (6) obtains the optimal policy. In Equation (6), V is obtained by using Equation (7) for each state $s_i \in S$. Specifically, in order to determine the optimal policy, we apply the value-iteration algorithm [5], 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 we observe convergence in values for all states for each successive iterations. Thus, using this value-iteration algorithm, we can obtain actions for the states that result in the maximum rewards for that state. Using the pair of action and value of each state, we can calculate the optimal policy π using Equation (6).

Algorithm 1 shows the pseudo code for the value-iteration algorithm. The algorithm first assigns zero reward to $V(s_i)$ for all s_i (Line 1) and repeatedly update $V(s_i)$ based on Equation (7) and Equation (6) (Line 2 to Line 7), and the corresponding optimal policy $\pi(s_i)$ (Line 8 to Line 11). In the algorithm, $R(s_i)$ is calculated by

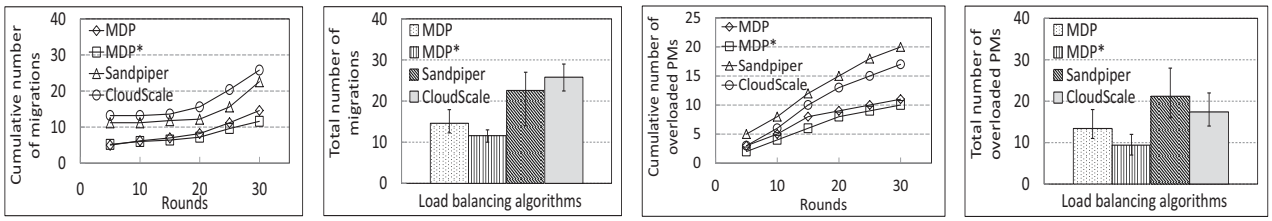
$$R(s_i) = \sum_j P_{\pi(s_i)}(s_i, s_j)R_{\pi(s_i)}(s_i, s_j). \quad (8)$$

When we observe convergence in values for all states, that is $max|V(s_i) - V_{new}(s_i)| \geq e$ (Line 3), we can assume that $V(s_i)$ is close to its maximum value and the corresponding $\pi(s_i)$ is the optimal policy we want.



(a) Cumulative # of VM migrations. (b) Total # of VM migrations. (c) Cumulative # of overloaded PMs. (d) Total # of overloaded PMs.

Figure 6. 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.

Figure 7. Performance using the Google Cluster trace.

3.7 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 [21–23]. For example, Sandpiper [22] 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. Compared to the previous MDP model, this new MDP model 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 $a \in A$. This 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 3.6 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>no action.

For a given migration VM, the central server can identify the most appropriate destination PMs based on the MDP. Better options from these PMs can be further identified based on additional consideration factors such as VM communication cost and migration distance [11].

4. Performance Evaluation

In this section, we conducted trace-driven experiments on CloudSim [9] 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 [9] and Google Cluster [13] 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 3.7). MDP* uses our MDP model for both VM selection and destination PM selection. We compared MDP and MDP* with Sandpiper [23] and CloudScale [19] 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 algo-

Server CPU uti.	0%	20%	40%	60%	80%	100%
HP ProLiant G4	86	92.6	99.5	106	112	117

Table 2. Energy consumption for different CPU utilizations [6].

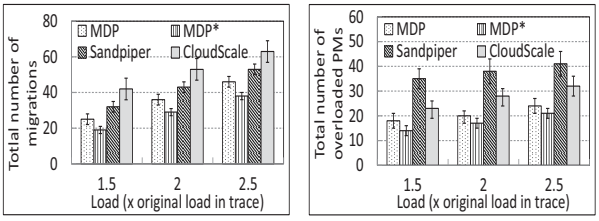
rithms and use CloudScale to represent proactive load balancing algorithms.

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 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 updates 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. Sandpiper and CloudScale perform VM migrations whenever a PM is detected overloaded (i.e., either CPU or memory utilization exceeds 0.8) and select the destination PM based on their corresponding PM selection algorithms. In MDP and MDP*, each PM chooses the action to perform that results in the maximal expected rewards.

4.1 Introduction of Two Metrics

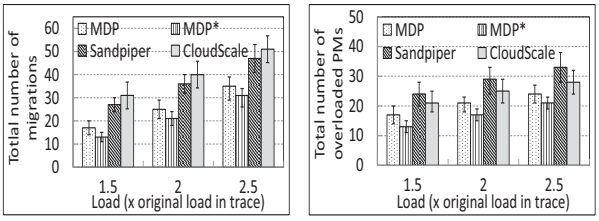
We first introduce two performance metrics. One metric is the total energy consumption of the PMs of a cloud datacenter caused by the application workloads. Energy consumption by PMs in datacenters is mostly determined by the CPU, memory, disk storage, power supplies and cooling systems [16], and the work in [6] gave the total energy consumption amount based on the CPU utilization. The configuration and power consumption characteristics of our used servers, HP ProLiant ML110 G4 (Intel Xeon 3040, 2 cores$\times$1860 MHz, 4 GB), is shown in Table 4. Using this table, we calculate and compare the energy consumption of different algorithms.

The other important metric to evaluate the performance of the load balancing algorithms in cloud datacenters is SLAs [8]. SLA violation (SLAV) is determined by the percentage of time during which the active hosts have experienced a CPU utilization of 100% (SLAVO) and performance degradation due to VM migration (SLAVM) [6]. SLAVO=$\frac{1}{N}\sum_{i=1}^{N}\frac{T_{s_i}}{T_{a_i}}$, and SLAVM=$\frac{1}{M}\sum_{j=1}^{M}\frac{C_{d_j}}{C_{r_j}}$. Here, N is the number of active PMs, T_{s_i} is the total time during which PM i has experienced CPU utilization of 100%, and T_{a_i} is the total time during which PM i is serving VMs. M is the number of VMs, C_{d_j} is an estimate of the performance degradation of the VM j caused by migration (we use 10% as in [6]), and C_{r_j} is the total CPU capacity requested by VM j during its lifetime. We measured SLAV as the product of SLAVO and SLAVM to evaluate the algorithms.



(a) The number of VM migrations with increasing workload ratio. (b) The number of overloaded PMs with increasing workload ratio.

Figure 8. Performance with the PlanetLab trace.

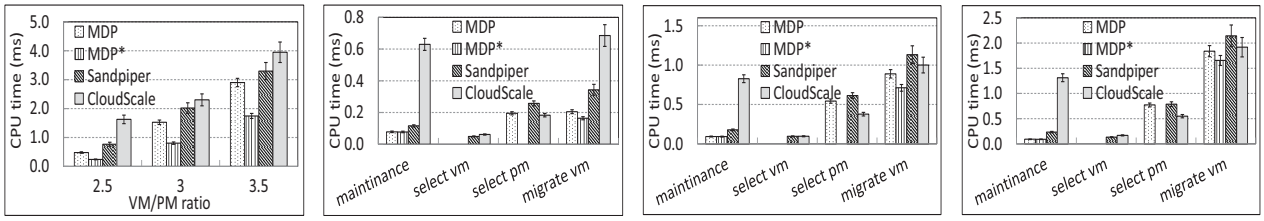


(a) The number of VM migrations with increasing workload ratio. (b) The number of overloaded PMs with increasing workload ratio.

Figure 9. Performance with the Google Cluster trace.

4.2 Experimental Results

Figure 6 and Figure 7 show the performance of MDP, MDP*, Sandpiper and CloudScale with the PlanetLab trace and Google Cluster trace, respectively. Figure 6(a) and Figure 7(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 6(b) and Figure 7(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



(a) Total time. (b) CPU time breakdown (ratio=2.5). (c) CPU time breakdown (ratio=3). (d) CPU time breakdown (ratio=3.5).

Figure 10. Comparison of CPU time consumption by different methods to achieve load balance.

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.

Next, we measure the number of overloaded PMs, which indicates the effectiveness of load balancing algorithms. Figure 6(c) and Figure 7(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 6(d) and Figure 7(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.

We then increased the VM's workload to 1.5, 2 and 2.5 times of its original workload in the trace to study the performance under various workloads. For each workload level, we repeated the simulation for 20 times. Figure 8 and Figure 9 show the experimental results with the PlanetLab trace and Google Cluster trace, respectively. Figure 8(a) and Figure 9(a) show the median, the 10th and 90th percentiles of the number of VM migrations of the three methods under different workload ratios. The number of VM migrations increases as the workload ratio increases. Within each workload ratio, the number of VM migrations follows MDP*<MDP<Sandpiper<CloudScale, which is consistent with the results in Figure 6(b) and Figure 7(b) due to the same reasons as explained before. Figure 8(b) and Figure 9(b) show the median, the 10th and 90th percentiles of the number of overloaded PMs of the three methods with different workload ratios. The number of overloaded PMs increases with workload ratio, and follows MDP*<MDP<CloudScale<Sandpiper within each workload ratio. The results are consistent with Figure 6(d) and Figure 7(d) due to the same reasons. Thus, MDP and MDP* perform better then Sandpiper and CloudScale in terms of the number of VM migrations and the number of overloaded PMs in different workloads.

The CPU time consumption for load balancing consists of the 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 system monitoring time refers to the CPU time spent on checking whether there are overloaded PMs and determining whether VM migration is necessary in each round. Figure 10(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. 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 10(b), Figure 10(c) and Figure 10(d) corresponding to three VM/PM ratios. The maintenance time refers to the CPU time needed to determine when to perform VM migrations. 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 $volume$ [23] 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.

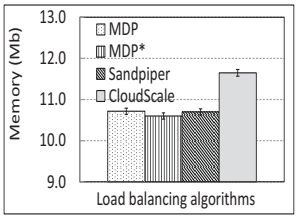


Figure 11. Memory consumption (ratio=3).

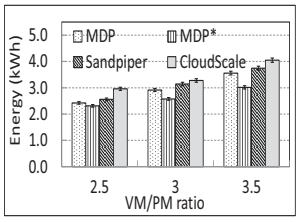


Figure 12. Energy consumption in algorithms.

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 refers to the CPU time needed to determine destination PMs where the selected VMs can 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 have 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 its 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.

Figure 11 shows the median, the 10th and 90th percentiles of the memory utilization of the four methods when the VM/PM ratio equals 3. We see that MDP, MDP* and Sandpiper consume similar amount of memory resource. CloudScale consumes much more memory since it needs to store a 40×40 probability transition matrix as indicated in [12] for each VM for workload prediction and it also has a higher number of VM migrations.

We then compare energy consumption of the four different load balancing algorithms. We ran each experiment for one hour and measured the total energy consumption of different algorithms. Figure 12 shows the median, the 10th and 90th percentiles of the total amount of the energy consumption among total 10 experiments. The idle energy consumption is measured when the PM is idle and stays at its lowest power state, which has a value about 2.2kWh. The energy consumption follows MDP*<MDP<Sandpiper<CloudScale for three reasons. First, MDP* and MDP can maintain the system in a long-term load balance state and hence free the PMs from busily calculating (i.e., determining VMs to migrate and selecting destination PMs). Second, MDP* and MDP reduce the number of VM migrations and hence avoid additional energy consumption of the system. Third, MDP* and MDP can more quickly select migration VMs and destination PMs, hence consume less CPU time than the other two algorithms. The result that MDP* consumes less energy than MDP verifies the effectiveness of our MDP-base algorithm in destination PM selection.

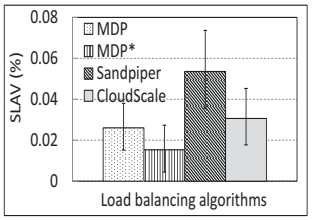


Figure 13. The SLAV metric.

Figure 13 shows the median, the 10th and 90th percentiles of the total SLAV values in the experiments. The results follow MDP*<MDP<CloudScale<Sandpiper, which is correlated with the number of overloaded PMs, since SLAV is affected by the resource utilization of the PMs. MDP and MDP* generate a lower SLAV value because they are able to maintain the system in a load balance state for a long time. The result that MDP* generates a lower SLAV value than MDP shows that the MDP-based algorithm is effective in reducing SLAV by effective migration VM selection and destination PM selection. CloudScale outperforms Sandpiper since it has less overloadeds due to its proactive prediction.

5. 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. 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. In our future work, we aim to make our algorithm fully distributed to increase its scalability.

Acknowledgments

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, CNS-1249603, Microsoft Research Faculty Fellowship 8300751.

References

[1] Microsoft Azure. http://www.windowsazure.com.

[2] BEA System Inc. http://www.bea.com.

[3] Amazon. Amazon Web Service. http://aws.amazon.com/.

[4] E. Arzuaga and D. R. Kaeli. Quantifying load imbalance on virtualized enterprise servers. In *Proc. of WOSP/SIPEW*, 2010.

[5] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[6] 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. *CCPE*, 24(13):1397–1420, 2011.

[7] A. Beloglazov and R. Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *TPDS*, 24(7): 1366–1379, 2013.

[8] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proc. of IM*, 2007.

[9] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *SPE*, 41(1):23–50, 2011.

[10] A. Chandra, W. Gong, and P. J. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proc. of SIGMETRICS*, 2003.

[11] L. Chen, H. Shen, and S. Sapra. RIAL: Resource intensity aware load balancing in clouds. In *Proc. of INFOCOM*, 2014.

[12] Z. Gong, X. Gu, and J. Wilkes. PRESS: Predictive elastic resource scaling for cloud systems. In *Proc. of CNSM*, 2010.

[13] GoogleTraceWebsite. Google cluster data. https://code.google.com/p/googleclusterdata/.

[14] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[15] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *Proc. of IPDPS*, 2009.

[16] M. Lauri and E. Brad. *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. Intel Press, 2009.

[17] A. Sallam and K. Li. A multi-objective virtual machine migration policy in cloud systems. *The Computer Journal*, 57(2):195–204, 2013.

[18] U. Sharma, P. J. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proc. of ICDCS*, 2011.

[19] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. of SOCC*, 2011.

[20] A. Singh, M. R. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proc. of SC*, 2008.

[21] 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*, 1(1):40–51, 2010.

[22] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. of NSDI*, 2007.

[23] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.