8-2016

Optimizing Virtual Resource Management in Cloud Datacenters

Liuhua Chen
Clemson University, liuhuac@g.clemson.edu

Optimizing Virtual Resource Management in Cloud Datacenters

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

by
Liuhua Chen
August 2016

Abstract

Datacenter clouds (e.g., Microsoft's Azure, Google's App Engine, and Amazon's EC2) are emerging as a popular infrastructure for computing and storage due to their high scalability and elasticity. More and more companies and organizations shift their services (e.g., online social networks, Dropbox file hosting) to clouds to avoid large capital expenditures. Cloud systems employ virtualization technology to provide resources in physical machines (PMs) in the form of virtual machines (VMs). Users create VMs deployed on the cloud and each VM consumes resources (e.g., CPU, memory and bandwidth) from its host PM. Cloud providers supply services by signing Service Level Agreement (SLA) with cloud customers that serves as both the blueprint and the warranty for cloud computing. Under-provisioning of resources leads to SLA violations while over-provisioning of resources leads to resource underutilization and then revenue decrease for the cloud providers. Thus, a formidable challenge is effective management of virtual resource to maximize energy efficiency and resource utilization while satisfying the SLA.

This proposal is devoted to tackle this challenge by addressing three fundamental and essential issues: i) initial VM allocation, ii) VM migration for load balance, and iii) proactive VM migration for long-term load balance. Accordingly, this proposal consists of three innovative components:

(1) Initial Complementary VM Consolidation. Previous resource provisioning strategies either allocate physical resources to virtual machines (VMs) based on static VM resource demands or dynamically handle the variations in VM resource requirements through live VM migrations. However, the former fail to maximize energy efficiency and resource utilization while the latter produce high migration overhead. To handle these problems, we propose an initial VM allocation mechanism that consolidates complementary VMs with spatial/temporal-awareness. Complementary VMs are the VMs whose total demand of each resource dimension (in the spatial space) nearly reaches

their host's capacity during VM lifetime period (in the temporal space). Based on our observation of the existence of VM resource utilization patterns, the mechanism predicts the lifetime resource utilization patterns of short-term VMs or periodical resource utilization patterns of long-term VMs. Based on the predicted patterns, it coordinates the requirements of different resources and consolidates complementary VMs in the same physical machine (PM). This mechanism reduces the number of PMs needed to provide VM service hence increases energy efficiency and resource utilization and also reduces the number of VM migrations and SLA violations.

(2) Resource Intensity Aware VM Migration for Load Balance. The unique features of clouds pose formidable challenges to achieving effective and efficient load balancing. First, VMs in clouds use different resources (e.g., CPU, bandwidth, memory) to serve a variety of services (e.g., high performance computing, web services, file services), resulting in different overutilized resources in different PMs. Also, the overutilized resources in a PM may vary over time due to the time-varying heterogenous service requests. Second, there is intensive network communication between VMs. However, previous load balancing methods statically assign equal or predefined weights to different resources, which leads to degraded performance in terms of speed and cost to achieve load balance. Also, they do not strive to minimize the VM communications between PMs. This proposed mechanism dynamically assigns different weights to different resources according to their usage intensity in the PM, which significantly reduces the time and cost to achieve load balance and avoids future load imbalance. It also tries to keep frequently communicating VMs in the same PM to reduce bandwidth cost, and migrate VMs to PMs with minimum VM performance degradation.

(3) Proactive VM Migration for Long-Term Load Balance. 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, 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 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.

Finally, we conducted extensive experiments to evaluate the proposed three mechanisms. i) We conducted simulation experiments based on two real traces and real-world testbed experiments to show that the initial complementary VM consolidation mechanism significantly reduces the number of PMs used, SLA violations and VM migrations of the previous resource provisioning strategies. ii) We conducted trace-driven simulation and real-world testbed experiments to show that RIAL outperforms other load balancing approaches in regards to the number of VM migrations, VM performance degradation and VM communication cost. iii) We conducted trace-driven experiments to show that the MDP-based load balancing algorithm outperforms previous reactive and proactive load balancing algorithms in terms of SLA violation, load balancing efficiency and long-term load balance maintenance.

Acknowledgments

I would like to thank many people who helped me during my Ph.D. study at Clemson University. I would like to express my sincere appreciation and thanks to my advisor Dr. Haiying Shen, whose passion and earnest manner in research have transformed me. Her guidance helped me overcome numerous difficulties and finish my Ph.D study. Without her help, I cannot finish my Ph.D. study, not even to mention a productive research. Her passion for scientific research and hard working will always set an example to me, which will benefit me for my whole life.

I would also like to thank my committee members: Dr. Walter B. Ligon III, Dr. Adam Hoover, and Dr. Amy W. Apon, for their valuable comments and suggestions.

My life at Clemson University is full of happiness and friendship. I thank my colleagues in the Pervasive Communications Laboratory: Kang Chen, Guoxin Liu, Chenxi Qiu, Bo Wu, Yuhua Lin, Zhuozhao Li, Jinwei Liu, Li Yan, Ankur Sarker, Haoyu Wang, for helping me on my research and also daily life. You guys are wonderful companies, and together we went through all good times and hard times.

I especially thank my beloved parents and brother for their constant support and unconditional love. I will forever be thankful to all my families who are the sources of all my happiness.

Table of Contents

List of Tables

List of Figures

Chapter 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, 2, 4]. Many businesses move their services to clouds due to their high scalability and flexible "pay as you go" service model, in which a cloud customer only pays for used resources. Such elasticity of the service model brings about cost saving for businesses [26] by eliminating the need of developing, maintaining and scaling a large private infrastructure. Cloud systems, such as Amazon EC2, Google App Engine, and Microsoft Azure, employ virtualization technology to provide resources in physical machines (PMs) in the form of virtual machines (VMs). Users create VMs deployed on the cloud on demand. Each VM runs its own operating system and consumes resources (e.g., CPU, memory and bandwidth) from its host PM.

Cloud providers supply services by signing Service Level Agreement (SLA) with cloud customers that serves as both the blueprint and the warranty for cloud computing. Under-provisioning of resources leads to SLA violations while over-provisioning of resources leads to resource underutilization and then revenue decrease for the cloud providers. The scale of modern cloud datacenters has been growing and current cloud datacenters contain tens to hundreds of thousands of computing and storage devices running complex applications. Energy consumption thus becomes critical concerns. Therefore, a formidable challenge is effective management of virtual resource to maximize energy efficiency and resource utilization while satisfying the SLA.

This project is devoted to tackle this challenge by addressing three fundamental and essential issues: *i) initial VM allocation, ii) VM migration for load balance, and iii) proactive VM migration for long-term load balance.* Initial VM allocation methods allocate PM physical resources to VMs when they are created. When a PM is overloaded, VM migration methods select VMs in the overloaded PM to migrate to underloaded PMs to release its load. Proactive VM migration methods migrate out VMs from a PM before it is about to be overloaded to keep it lightly loaded. By addressing these issues, we expect to avoid resource over-provisioning and under-provisioning in the cloud, and thus improve the profit of cloud providers, the application Quality of Service (QoS) of cloud users and green computing.

1.1 Problem Statement

1.1.1 Initial VM allocation.

Previous VM allocation strategies can be classified to two categories: static methods and dynamic methods [51]. Static VM allocation methods [7, 9, 39, 46, 62] allocate physical resources to VMs only once based on static VM peak resource demands, which can be reduced to a bin-packing problem. However, reserving VM peak resource requirement for the entire execution time cannot fully utilize resources as cloud applications consume varying amount of resources in different phases. In order to fully utilize cloud resources, dynamic VM allocation methods [5, 25, 45, 48, 49, 58] have been proposed, which first consolidate VMs using a simple bin-packing heuristic and then handle the variations in VM resource requirements through live VM migrations [58]. However, VM migration generates high migration overhead and also degrades the VM performance [52]. In addition, all previous VM allocation strategies only consider resource demands at one or each time point. Therefore, they fail to coordinate the resource requirements in different resource dimensions (in the spatial space) for a period of time (in the temporal space); that is, they are spatial/temporal-unaware, which fails to continuously fully utilize different resources.

1.1.2 VM migration for load balance.

Clouds currently perform load balancing by migrating VMs from heavily loaded PMs to lightly loaded PMs so that the utilizations of PMs' resources (defined as the ratio between actual requested resource amount and the resource capacity) are below a threshold. Previously proposed load balancing methods [5, 25, 45, 48, 57] combine the utilizations of different resources in selecting VMs to migrate and finding the most suitable destination PMs. They predefine a weight (or give equal weight) for each resource, calculate the weighted product of different resource utilizations to represent the load of PMs and the weighted product of owned amount of each resource to represent the capacity of PMs. They then migrate VMs from the most heavily loaded PMs to the most lightly loaded PMs. However, predetermined or equal resource weight cannot adapt to the heterogeneous resource intensities (i.e., degree of resource demand) among PMs and time-varying resource intensity in one PM. Also, previous load balancing methods do not consider the communication between VMs and VM performance (i.e., response time) degradation due to migration. There may be intensive network communication between two VMs, so separating such two VMs to two different PMs would increase the network bandwidth consumption. Moving a VM to a distant PM would lead to high VM performance degradation. Therefore, the previous methods are not efficient for cloud tasks where VM communication is intensive and delayed VM response time is highly undesirable.

1.1.3 Proactive VM migration for long-term load balance.

Many load balancing methods [5,16,41,45,48,56] 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 methods 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 customers hence SLA violations. Also, the process of selecting migration VMs and destination PMs is complex and generates high delay and overhead. Recently, a number of proactive load balancing methods [11, 12, 14, 20, 42, 43] have been proposed to predict VM resource demand within a short time for sufficient resources provision or load balancing. In this method, 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 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.

1.2 Research Approaches

In this project, we propose novel techniques to handle these issues inherent in managing virtual resources to achieve the efficient utilization of resources in cloud datacenters while ensuring the SLA requirements. As shown in Figure 1.1, we propose three mechanisms: i) initial complementary VM consolidation, ii) resource intensity aware VM migration for load balance, and iii) proactive VM migration for long-term load balance.

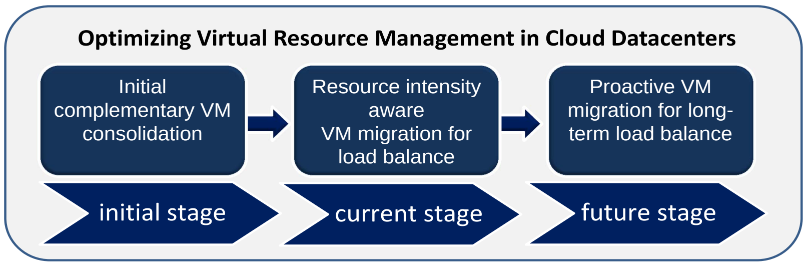

Optimizing Virtual Resource Management in Cloud Datacenters
Initial complementary VM consolidation
Resource intensity aware VM migration for load balance
Proactive VM migration for long-term load balance
initial stage
current stage
future stage

Figure 1.1: Optimizing Virtual Resource Management.

(1) **Initial Complementary VM Consolidation.** Previous VM allocation methods either allocate VMs based on VM peak resource requirement which cannot fully utilize resources or rely on VM migrations which generate high overhead. This proposed mechanism predicts VM resource utilization patterns and consolidates complementary VMs, whose total demand of each resource dimension (in the spatial space) nearly reaches their host PM's capacity during VM lifetime period (in the temporal space). This mechanism maximizes energy efficiency and resource utilization while reducing migration overhead compared to the previous VM allocation mechanisms.

(2) **Resource Intensity Aware VM Migration for Load Balance.** Unlike the previous VM migration methods that statically assign equal or predefined weights to different resources, this proposed mechanism dynamically assigns different weights to different resources according to their usage intensity in the PM, which significantly reduces the time and cost to achieve load

balance and avoids future load imbalance. It also tries to keep frequently communicating VMs in the same PM to reduce bandwidth cost, and migrate VMs to PMs with minimum VM performance degradation.

(3) **Proactive VM Migration for Long-Term Load Balance.** Previous VM migration methods cannot maintain long-term load balance and produce high overhead and delay. We propose a Markov Decision Process (MDP)-based load balancing mechanism, which proactively and directly provides guidance on migration VM selection and destination PMs selection for long-term load balance.

The research is innovative because it proposes enhanced techniques to optimize virtual resource management in cloud datacenters, which help fully utilize cloud resources while upholding SLAs, thus improving the profit of cloud providers, the application QoS for cloud users and green computing. This research will blend formal development, analysis, implementation, deployment, experimentations, and evaluation of the mechanisms. Success of this research will advance our understanding of inherent problems that prohibit efficient resource utilization in current cloud datacenters, promote new techniques, and ultimately contribute to building cost-effective infrastructures as a service (IaaS) for cloud services. Both the distributed system and cyberinfrastructure communities that require highly efficient use of geographically distributed resources may also find our proposed mechanisms quite useful. Finally, there will be significant opportunities for technology transfer to industrial research partners.

1.3 Contributions

We summarize our expected contributions of the dissertation proposal below:

- We propose an initial complementary VM consolidation mechanism that consolidates complementary VMs with spatial/temporal-awareness. Complementary VMs are the VMs whose total demand of each resource dimension (in the spatial space) nearly reaches their host's capacity during VM lifetime period (in the temporal space). Based on our observation of the existence of VM resource utilization patterns, the mechanism predicts the lifetime resource utilization patterns of short-term VMs or periodical resource utilization patterns of long-term VMs. Based on the predicted patterns, it coordinates the requirements of different resources and consolidates complementary VMs in the same physical machine (PM). This mechanism reduces the number of PMs needed to provide VM service hence increases energy efficiency and resource utilization and also reduces the number of VM migrations and SLA violations.

- We propose a resource intensity aware VM migration mechanism for load balance that dynamically assigns different weights to different resources according to their usage intensity in the PM, which significantly reduces the time and cost to achieve load balance and avoids future load imbalance. It also tries to keep frequently communicating VMs in the same PM to reduce bandwidth cost, and migrate VMs to PMs with minimum VM performance degradation.

- We propose a proactive 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 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.

1.4 Dissertation Organization

The rest of this proposed is structured as follows. Chapter 2 introduces the related works. Chapter 3 details of the proposed initial complementary VM consolidation mechanism. Chapter 4 presents RIAL, economical and deadline-driven video flow scheduling system. Chapter 5 introduces RIAL, a Resource Intensity Aware Load balancing method. Finally, Chapter 6 concludes this dissertation with remarks on our future work.

Chapter 2

Related Work

2.1 Initial VM Allocation

Recently, many static and dynamic VM allocation strategies have been proposed [51]. Static provisioning [7,9,39,46,62] allocates physical resources to VMs only once based on static VM resource demands. For example, Srikantaiah *et al.* [46] proposed to use Euclidean distance between VM resource demands and residual capacity as a metric for consolidation. However, static provisioning cannot fully utilize resources because of time-varying resource demands of VMs. To fully utilize cloud resources, dynamic provisioning [5,25,45,48,49,58] first consolidates VMs using a simple bin-packing heuristic and handles the variations in VM resource requirements through live VM migrations, which however results in high migration overhead. Sandpiper [58] uses the product of CPU, network and memory load to represent the load of a VM and a PM, and migrates the most loaded VM from an overloaded PM to the least loaded PM. TOPSIS [48] determines the ideal solution consisting of resource utilizations, and migrates the VM with the shortest Euclidean distance with the ideal solution to the PM with the longest Euclidean distance with the ideal solution. Arzuaga *et al.* [5] selects the VM that yields the greatest improvement of the server load imbalance metric to migrate. Khanna *et al.* [25] proposed to select the VM with the lowest utilization from the overloaded PM and migrate it to the PM that has the least residual capacity big enough to hold this VM. Verma *et al.* [49] applied a first-fit decreasing heuristic to optimize VM placement to minimize power consumption and maximize performance. These strategies consider the current state of resource demand and available capacity at a time point rather than the trend state during a time period for VM migration, which is insufficient for maintaining a continuous load balanced state. Our idea of consolidating complementary VMs for a certain time period can help these migration strategies maintain the load balanced state for a longer time period.

Some works [12,43,55,61] predict resource demands for VM migration to avoid SLA violation in the future. Bobroff *et al.* [12] proposed an algorithm to predict VM resource requirement based on the recent history of resource demands in order to allocate minimum resources to VMs such that the overall SLA violations will not be more than p-percentile. Shen *et al.* [43] proposed an online resource demand prediction model for proactive VM migration to avoid PM overload. Our mechanism is different from these two methods in several aspects. First, our mechanism predicts the resource demand patterns during a certain time, while these two methods predict resource demands at one time point, which cannot help retain a continuous load balanced state. Second, these prediction methods are used for VM migration during VM running, while our mechanism is for initial allocation, and hence reduces VM migration overhead. Third, these two methods use historical record of a running VM to predict its future demand, while our mechanism predicts a VM's resource utilization pattern in initial VM allocation based on profiles of previous VMs executing similar job tasks. All previous VM allocation strategies consider the current or future state of resource demand and available capacity at a time point rather than during a time period, which is insufficient for maintaining a continuous load balanced state. Though our work focuses on initial VM allocation rather than subsequent VM migration, our idea of consolidating complementary VMs for a certain time period can help the migration strategies maintain the load balanced state for a longer time period.

Recently, some works focus on allocating network bandwidth resources to tenant VMs [7,29, 39,60]. Oktopus [7] provides static bandwidth reservations throughout the network. Popa *et al.* [39] proposed a set of properties to navigate the tradeoff space of requirements-payment proportionality and minimum guarantees when sharing cloud network bandwidth. PROTEUS *et al.* [60] provides bandwidth provisioning using predicted bandwidth utilization profile. Lin *et al.* [29] propose and an economical and deadline-driven video flow scheduling system called EcoFlow to transmit videos in the order of their deadline tightness to reduce bandwidth cost. Different from these works, we focus on consolidating VMs that have demands on multi-resources rather than a single resource.

2.2 VM Migration for Load Balance

Many load balancing methods have been proposed to deal with the PM overload problem using VM migration [5, 25, 45, 48, 57]. Sandpiper [57] tries to move load from the most overloaded servers to the most underloaded servers. It defines volume for VMs and PMs: $volume$=(1/(1-u_{cpu}))*(1/(1-u_{net}))*(1/(1-u_{mem})), where u is resource utilization. It also defines a volume-to-size ratio (VSR) for each VM: VSR=$volume/size$, where $size$ is the memory footprint of the VM. It then migrates the VM with the maximum VSR to the PM with the least volume. TOPSIS [48] predetermines weights for different criteria (e.g., CPU, memory, bandwidth, PM temperature). To select VMs to migrate (or select destination PM), it first forms a weighted normalized decision matrix with the utilizations of VMs of a PM (or PMs) with respect to each criterion. It then determines the ideal solution by using the maximum utilization for the benefit criteria and the minimum utilization for the cost criteria. Khanna *et al.* [25] treated different resources equally. They proposed to select the VM with the lowest product of resource utilizations from the overloaded PM and migrate it to the PM that has the least residual capacity big enough to hold this VM. Arzuaga *et al.* [5] used predetermined resource weights to calculate the product of weighted utilizations of different resources of a PM or a VM as its load. It then chooses the VM with the highest load from an overloaded PM to migrate to a selected PM that yields the greatest improvement of the system imbalance metric. Tang *et al.* [47] proposed a load balancing algorithm that strives to maximize the total satisfied application demand and balance the load across PMs. They define load-memory ratio of an instance as its CPU load divided by its memory consumption to measure its resource utilization. However, all previous methods statically assume equal or predefined weights for different resources, which may not be correct due to the different time-varying demands on different resources in each PM. RIAL is distinguished from these methods in that it dynamically determines the resource weight based on the demand on the resource in each PM, which leads to fast and constant convergence to the load balanced state.

Some works deal with load balancing on one resource such as storage [24] and bandwidth [7, 39, 60]. Hsiao *et al.* [24] proposed a load balancing algorithm for distributed file systems in clouds by moving file chunks from overloaded servers to lightly loaded servers. Oktopus [7] provides static reservations throughout the network to implement bandwidth guarantees. Popa *et al.* [39] navigated the tradeoff space of requirements-payment proportionality, resource minimum guarantee and system utilization when sharing cloud network bandwidth. Xie *et al.* [60] proposed PROTEUS for bandwidth provisioning using predicted bandwidth utilization profile in order to increase the system bandwidth utilization and reduce the cost to the tenants. However, by focusing on only one resource, these approaches cannot be directly used for PM load balancing where VMs use different types of resources.

Many other works for resource management in clouds deal with scheduling incoming workload requests or initial placement of VMs with the concern of cost and energy efficiency [28,33,34,44]. Lin *et al.* [28] proposed an algorithm to achieve dynamic right-sizing in datacenters in order to save energy. It uses a prediction window of future arrivals to decide when to turn off an idle server. Maguluri *et al.* [33] focused on resource allocation that balances the load among servers to achieve throughput optimization. Meng *et al.* [34] used traffic patterns among VMs to determine VM placement in order to improve network scalability. Shrivastava *et al.* [44] proposed AppAware that considers inter-VM dependencies and the underlying network topology to place VMs with intensive mutual communication in the same PM to reduce network traffic. Shen *et al.* [43] proposed an online resource demand prediction method to achieve adaptive resource allocation.

2.3 Proactive VM Migration for Long-term Load Balance

In recent years, many load balancing methods have been proposed to avoid overloaded PMs in the clouds [5, 16, 41, 45, 48, 56]. 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 [11,12,14,20,42,43] 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.

Chapter 3

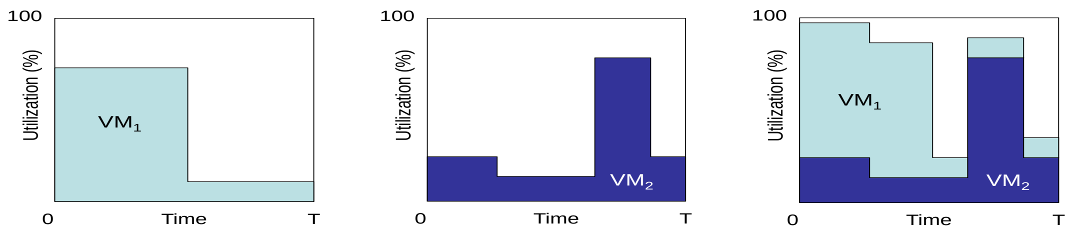
Initial Complementary VM Consolidation

3.1 VM Resource Utilization Pattern Detection

3.1.1 Basic Rationale

Our primary goal in designing the initial VM allocation mechanism is to minimize the number of PMs used and the number of VM migrations, and maximize resource utilization, while ensuring SLA guarantees. Figure 3.1 shows a simple 1-dimensional example to explain the idea of our mechanism. VM1 has a high resource utilization at an early phase but low resource utilization at a later phase, while VM2 has a low resource utilization at an early phase but a high resource utilization at a later phase. Our mechanism predicts the VM resource utilization pattern and places such complementary VMs in the same PM to achieve the goal.

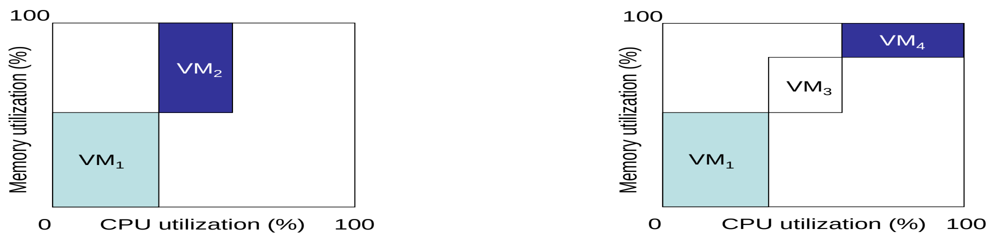
The initial VM allocation mechanism must consider resource demands across every resource dimension such as CPU, memory and bandwidth. Consolidating complementary VMs in a multi-dimensional space is a non-trivial task. For example, we should avoid placing VMs that intensively use the same resource in a PM, which otherwise prevents the PM from accepting other VMs due to lack of this resource. Placing VMs that intensively use different resources (e.g., a high-CPU-utilization VM and a high-memory-utilization VM) in a PM can fully utilize PM multi-dimensional resources while increases the number of VMs that can reside in one PM. Figure 3.2 demonstrates



(a) VM1 (b) VM2 (c) VM1 and VM2

Figure 3.1: Consolidating complementary VMs in one PM.

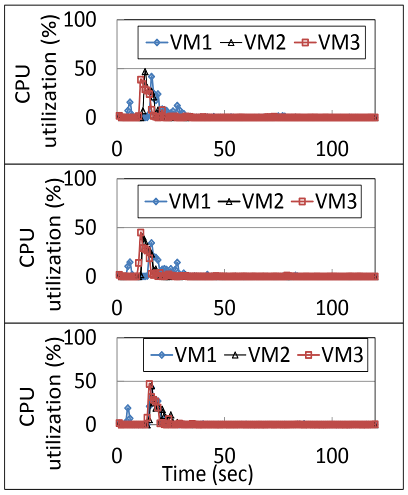
an example in a 2-dimensional resource space. In Figure 3.2(a), VM1 and VM2 have high memory utilizations and they use up the memory resource of the host PM. Though this PM still has spare CPU resource, it cannot host any more VMs due to the shortage of memory. In Figure 3.2(b), by consolidating VM3 and high-CPU-utilization VM4 with VM1, the CPU and memory resources of this PM are fully utilized. This example implies that when initially allocating a VM, it is desirable to choose the PM that makes the load sum point move towards the top right corner of the PM in the figure; that is, the resource in each dimension tends to be equally fully utilized.



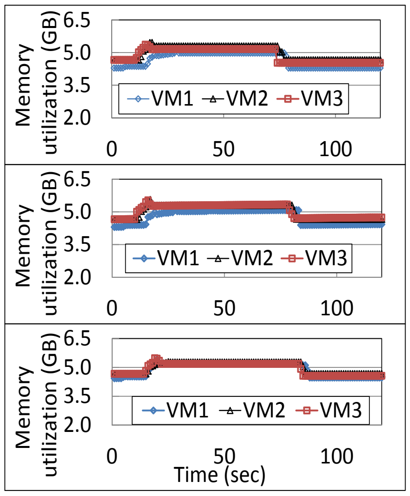
(a) CPU is not fully used (b) Both resources are fully used

Figure 3.2: Consolidating complementary VMs to fully utilize multi-dimensional resources in a PM.

In the following sections, we first conduct a measurement study on VM resource utilizations for both short-term and long-term applications to verify the existence of utilization patterns (Section 3.1.2). Second, we discuss how to detect the patterns of a group of VMs running the same job (e.g., WordCount) (Section 3.1.3). Third, we present how to coordinate the resource requirements of different dimensions of the VMs based on predicted utilization patterns to consolidate complementary VMs (Section 3.2.1).

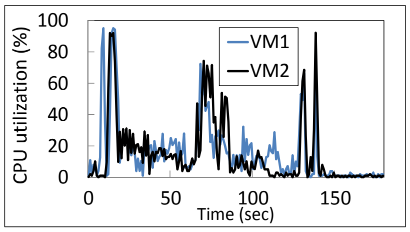


(a) CPU utilization

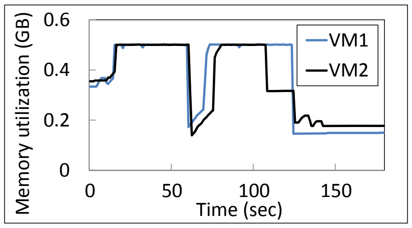


(b) Memory utilization

Figure 3.3: VM resource utilization for *TeraSort* on three datasets.



(a) CPU utilization



(b) Memory utilization

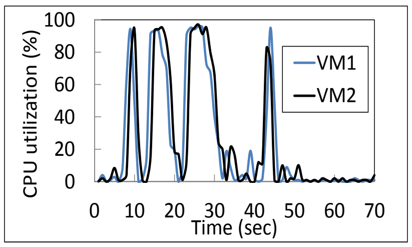
Figure 3.4: VM resource utilization for *TestDFSIO write*.

3.1.2 Profiling VM Resource Demands

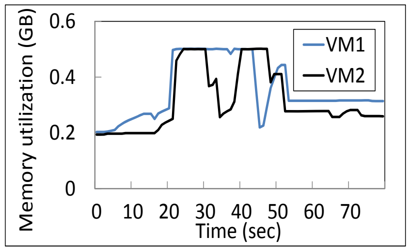
In order to predict the resource demand profiles of cloud VMs, we conducted a measurement study on VM resource utilizations. Workload arrives at the virtual cluster of a tenant in the form of jobs. Usually all tasks in a job execute the same program with the same options. Also, application user activities have daily patterns. Thus, different VMs running the same job tend to have similar resource utilization patterns. To confirm this, we conducted a measurement study on both short-term jobs and long-term jobs.

3.1.2.1 Utilization Patterns of VMs for Short-Term Jobs

MapReduce jobs represent an important class of applications in cloud datacenters. We profile the CPU and memory utilization patterns of typical MapReduce jobs. We conducted the profiling

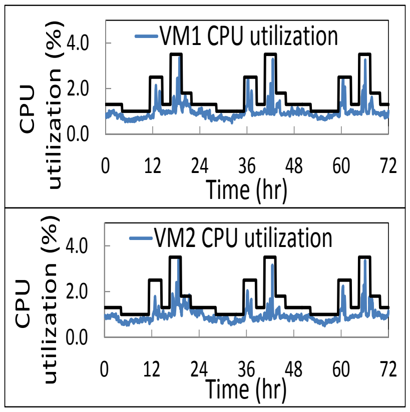


(a) CPU utilization

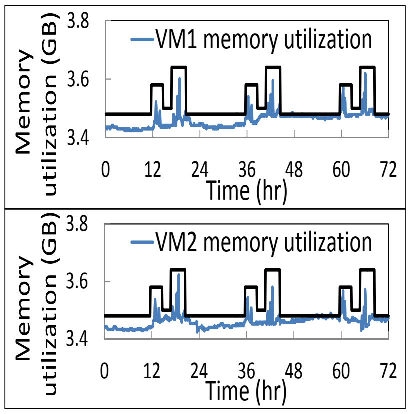


(b) Memory utilization

Figure 3.5: VM resource utilization for *TestDFSIO read*.



(a) CPU utilization



(b) Memory utilization

Figure 3.6: VM resource utilization from Google Cluster trace.

experiments on our cluster consisting of 15 machines (3.4GHz Intel(R) i7 CPU, 8GB memory) running Ubuntu 12.04. We constructed a virtual cluster of a tenant with 11 VMs; each VM instance runs Hadoop 1.0.4. We recorded the CPU and memory utilization of each VM every 1 second.

We used *Teragen* to randomly generate 1G data, then ran *TeraSort* to sort the data in the virtual cluster. Figures 3.3(a) and 3.3(b) display the resource utilization results of three VMs for different generated datasets. Figure 3.4 displays the resource utilizations of two VMs running *TestDFSIO write*, which generates 10 output files with each file having 0.1GB. Figure 3.5 displays the resource utilizations of two VMs running *TestDFSIO read*, that reads 10 input files generated by *TestDFSIO write*. From the figures, we can find that the VMs collaboratively running the same job have similar resource utilization patterns. The VMs running the same job on different datasets also have similar resource utilization patterns. We repeatedly ran each experiment several times and got similar resource utilization patterns for the VMs, which indicates that VMs running the same job task at different times also have similar resource utilization patterns.

3.1.2.2 Utilization Patterns of VMs for Long-Term Jobs

To study the utilization patterns of VMs for long-term jobs, we used publicly available Google Cluster trace [21] and the PlanetLab trace [13]. The Google Cluster trace records resource usage on a cluster of about 11000 machines from May 2011 for 29 days. The PlanetLab trace contains the CPU utilization of each VM in PlanetLab every 5 minutes for 24 hours in 10 random days in March and April 2011. In the Google Cluster trace, we analyzed 700 VMs and found that different VMs running the same job tend to have similar utilization patterns. Also, for a long-term VM, daily periodical patterns can be observed from the VM trace. We randomly chose two VMs running the same job as an example to show our observations. Figure 3.6(a) shows the CPU utilizations of two VMs every five minutes during three days and Figure 3.6(b) shows their memory utilizations. We see that both CPU and memory resource demands exhibit periodicity approximately every 24 hours. Also, the two VMs exhibit similar resource utilization patterns since they collaboratively ran the same job. In the PlanetLab trace, we analyzed 900 VMs and also found that they exhibit daily periodical patterns. Figure 3.7 shows the CPU utilization of a randomly selected VM to show their periodical patterns.

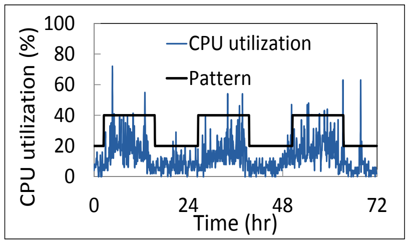
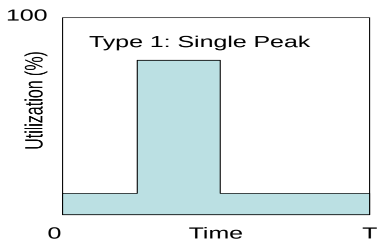


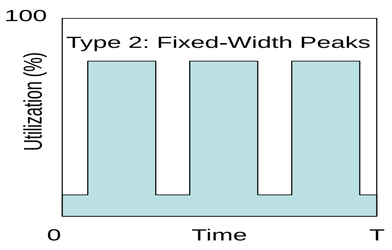
Figure 3.7: VM resource utilization from PlanetLab trace.

3.1.3 VM Resource Utilization Pattern Detection

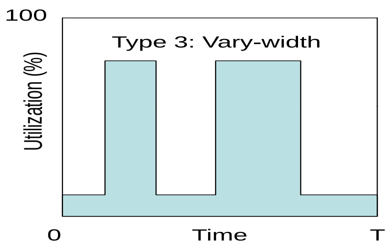
The previous section shows the existence of similar resource utilization patterns of VMs running the same job. Given the resource requirement pattern of VMs in an application, we can potentially derive some complicated functions (e.g., high-order polynomials) to precisely model the changing requirement over time. However, such smooth functions significantly complicate the process of VM allocation due to the complexity of model formulation. Also, very accurate pattern



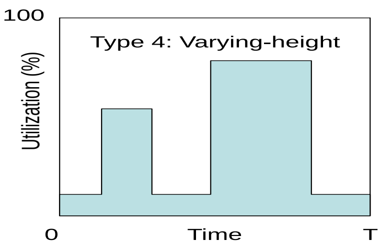
(a) Type 1: Single peak



(b) Type 2: Fixed-width peaks



(c) Type 3: Varying-width peaks



(d) Type 4: Varying height peaks

Figure 3.8: Time-varying resource utilization classification.

modeling of an individual VM cannot represent the general patterns of a group of VMs for similar applications. To achieve a balance between modeling simplicity and modeling precision, we choose to model the resource requirement as simple pulse functions introduced in [60] as shown in Figure 3.8. These four models sufficiently capture the resource demands of the applications. An actual VM resource demand that is much more complicated usually exhibits a pattern which is a combination of these simple types.

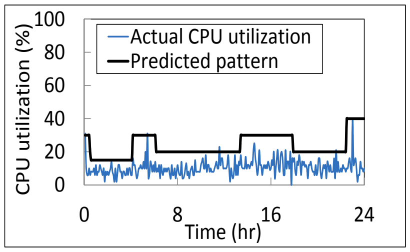
Next, we introduce how to detect the resource utilization pattern for a VM. The cloud records the resource utilizations of the VMs of a tenant. If the job on a VM is a short-term job (e.g., MapReduce job), the cloud records the entire lifetime of the job. If the job on a VM is a long-term job (e.g. Web server VM), the cloud records several periods that show a regular periodical pattern. From the log, the cloud can obtain the resource utilization of VMs of a tenant running the same application. When a tenant issues a VM request to the cloud, based on the resource utilization pattern of previous VMs from this tenant running the same application, the cloud can estimate the resource utilization pattern of this requested VM.

Let $\mathcal{D}_i(t)$=$(D_i^1(t), .., D_i^d(t))$ be the actual d dimension resource demands of VM i at time t. Given the resource demands of a set of VMs running the same job from a tenant, denoted by $\mathcal{D}_i(t)$ (t=$T_0, ..., T_0$+T, i=$1, 2, ..., N$), our pattern detection algorithm finds a pattern $\mathcal{P}(t)$=$(P^1(t), ..,$

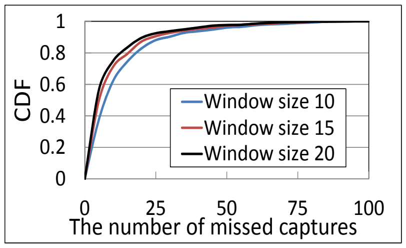
Algorithm 1 VM resource demand pattern detection.

```
1:  Input: $\mathcal{D}_i(t)$: Resource demands of a set of VMs
2:  Output: $\mathcal{P}(t)$: VM resource demand pattern
3:      /* Find the maximum demand at each time */
4:      $\mathcal{E}(t_j) = \max_{i \in N}\{\mathcal{D}^i(t_j)\}$ for each time $t_j$
5:      /* Smooth the maximum resource demand series */
6:      $\mathcal{E}(t_j) \leftarrow \text{LowPassFilter}(\mathcal{E}(t_j))$ for each time $t_j$
7:      /* Use sliding window $W$ to derive pattern */
8:      $\mathcal{P}(t_j) = \max_{t_j \in [t_j, t_j+W]}\{\mathcal{E}(t_j)\}$ for each time $t_j$
9:      /* Round the resource demand values */
10:     $\mathcal{P}(t_j) \leftarrow \text{Round}(\mathcal{P}(t_j))$ for each time $t_j$
11:     return $\mathcal{P}(t)$ $(t = T_0, ..., T_0 + T)$
```

$P^d(t)$) (t=$T_0, ..., T_0 + T$) to cover the future resource demand profile of a requested VM from the tenant.



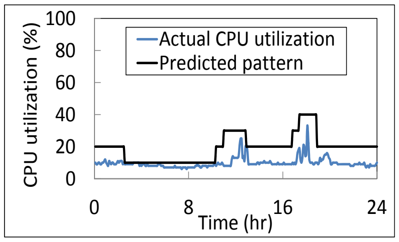
(a) Actual and predicted CPU utilizations



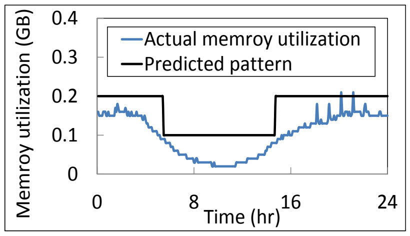
(b) CDF of # of missed captures using the PlanetLab trace.

Figure 3.9: Pattern detection using the PlanetLab trace.

Algorithm 1 shows how to generate the resource demand pattern for a requested VM. The algorithm first finds the maximum demand $\mathcal{E}(t)$ among the set of $\mathcal{D}_i(t)$ $(i = 1, 2, ..., N)$ at each time t (Line 4). Then, it passes $\mathcal{E}(t)$ through a low pass filter (Line 6) to remove high frequency components to smooth $\mathcal{E}(t)$. The algorithm then utilizes a sliding window of size W to find the envelop of $\mathcal{E}(t)$ (Line 8). Finally, it rounds the demand values (Line 10).



(a) Actual and predicted CPU utilizations



(b) Actual and predicted memory utilizations

Figure 3.10: Pattern detection using the Google Cluster trace.

To evaluate the accuracy of our pattern detection algorithm, we conducted an experiment on predicting VM resource request pattern based on resource utilization records of a group of VMs running the same application from the PlanetLab trace and the Google Cluster trace. We randomly selected 700 jobs and predicted the CPU utilization of a VM in each job during 24 hours. Specifically, in the PlanetLab trace, we used the CPU utilizations of three VMs of a job on March 3rd, 6th and 9th in 2011 to predict the CPU utilization of a VM and compared it with the actual utilization of a VM of the job on March 22nd, 2011. In the Google Cluster trace, we used the CPU and memory utilizations of two VMs of a job on May 1st and 2nd in 2011 to predict the CPU and memory utilizations of a VM and compared them with the real utilizations of a VM of the job on May 3rd, 2011.

Figure 3.9(a) displays the actual VM CPU utilization and the predicted pattern generated by our pattern detection algorithm using the PlanetLab trace. Figure 3.10(a) and Figure Figure 3.10(b) display the actual VM CPU and memory utilizations and the predicted pattern using the Google Cluster trace. We see that the pattern can capture the utilization most of the time except for a few burst peaks. Most of these burst peaks are only slightly greater than the pattern cap and are single bursts. This means that the resources provisioned according to the pattern can ensure the SLA guarantees most of the time, i.e., before and after the burst points.

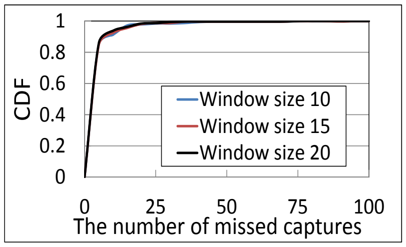


Figure 3.11: CDF of # of missed captures using the Google Clusster trace.

When the real VM CPU request from the trace is greater than the predicted value, we say that a *missed capture* occurs. Figure 3.9(b) and Figure 3.11 show the cumulated distributed function (CDF) of the number of missed captures from our 700 predictions using the PlanetLab trace and the Google Cluster trace, respectively. The three curves in the figure correspond to the pattern detection algorithm with different window sizes. We see that up to 90% of the detected patterns have missed captures fewer than 25 during the 24 hours in PlanetLab trace, and up to 90% of the detected patterns have missed captures fewer than 10 in Google Cluster trace. We also see that the patterns generated by a bigger window size generates fewer missed captures compared to a small window size because a larger window size leads to more resource provisioning. As the previous dynamic provisioning strategies, VM migration upon SLA violation is a solution for these missed captures. Our initial VM allocation mechanism helps reduce a large number of VM migrations in the previous dynamic provisioning strategies.

3.2 Initial VM allocation Mechanism

3.2.1 Initial VM Allocation Policy Based on Resource Efficiency

The goal of our initial VM allocation mechanism is to place all VMs in as few hosts as possible, ensuring that the aggregated demand of VMs placed in a host does not exceed its capacity across each resource dimension. We consider the VM consolidation as a classical d-dimensional vector bin-packing problem [17], where the hosts are conceived as bins and the VMs as objects that need to be packed into the bins. This problem is an NP-hard problem [17]. We then use a dimension-aware heuristic algorithm to solve this problem, which takes advantage of cross dimensional complimentary requirements for different resources as illustrated in Figures 3.1 and 3.2 in Section 3.1.2.

Each host j is characterized by a d-dimensional vector to represent its capacities $\mathcal{H}_j = (H_j^1, H_j^2, ..., H_j^d)$. Each dimension represents the host's capacity corresponding to a different resource such as CPU, memory, and disk bandwidth. Recall that $\mathcal{D}_i(t) = (D_i^1(t), D_i^2(t), .., D_i^d(t))$ denotes the actual resource demands of VM i. We define the fractional VM demand vector of VM i on PM j as

$$\mathcal{F}_{ij}(t) = (F_{ij}^1(t), F_{ij}^2(t), ...F_{ij}^d(t)) = (\frac{D_i^1(t)}{H_j^1}, \frac{D_i^2(t)}{H_j^2}, .., \frac{D_i^d(t)}{H_j^d}). \tag{3.1}$$

The resource utilization of PM j with N VMs on resource k at time t is calculated by $U_j^k(t) = \frac{1}{H_j^k}\sum_{i=1}^{N} D_i^k(t)$.

In order to measure whether a PM has available resource for a VM in a future period of time, we define the normalized residual resource capacity of a host as $\mathcal{R}_j(t) = (R_j^1(t), R_j^2(t), ..., R_j^d(t))$, in which

$$R_j^k(t) = 1 - U_j^k(t) = 1 - \frac{1}{H_j^k}\sum_{i=1}^{N} D_i^k(t). \tag{3.2}$$

When a VM is allocated to a PM, the VM's fractional VM demand F_{ij}^k and the PM's normalized residual resource capacity R_j^k must satisfy the capacity constraint below at each time t

Algorithm 2 Pseudocode for initial VM allocation.

```
1: Input: $\mathcal{P}_i(t)$: Predicted resource demands
          $\mathcal{R}_j(t)$: Residual resource capacity of candidates
2: Output: Allocated host of the VM
3:     $M$=Double.MAX_VALUE //initialize the distance
4:     for $j = 1$ to $m$ do
5:         if CheckValid($\mathcal{P}(t)$,$\mathcal{R}_j(t)$)==false then
6:             continue
7:         else
8:             for $k = 1$ to $d$ do
9:                 $E_j^k = E_j^k + \frac{1}{T \cdot H_j^k} \int_{T_0}^{T_0+T} P^k(t)dt$
10:                $M_j+ = \{w_k(1 - E_j^k)\}^2$
11:            end for
12:            if $M_j$<$M$ then
13:                $M$=$M_j$
14:                AllocatedHost=host $j$
15:    end for
16:    return AllocatedHost
17:
18: function CheckValid($\mathcal{P}(t)$,$\mathcal{R}_j(t)$):
19:    for $k = 1$ to $d$ do
20:        for $t = T_0$ to $T_0 + T$ do
21:            if $F_{ij}^k(t) > R_j^k(t)$ (Eq.(3.3)==false)
22:            return false
23:        end for
24:    end for
25:    return true
```

and for each resource k:

$$F_{ij}^k(t) \leq R_j^k(t),\ t = T_0, ..., T_0 + T,\ k = 1, 2..., d. \tag{3.3}$$

in order to guarantee that the host has available resource to host the VM resource request for the time period $[T_0, T_0 + T]$.

For each resource k, we hope that a PM j's $U_j^k(t)$ at each time t is close to 1, that is, its each resource is fully utilized. To jointly measure a PM's resource utilization across different resources at each time, we define the *resource efficiency* during time period $[T_0, T_0 + T]$ as the ratio of the aggregated resource demand over the total resource capacity:

$$E_j^k = \frac{1}{T \cdot H_j^k} \int_{T_0}^{T_0+T} \sum_{i=1}^{N} D_i^k(t)dt. \tag{3.4}$$

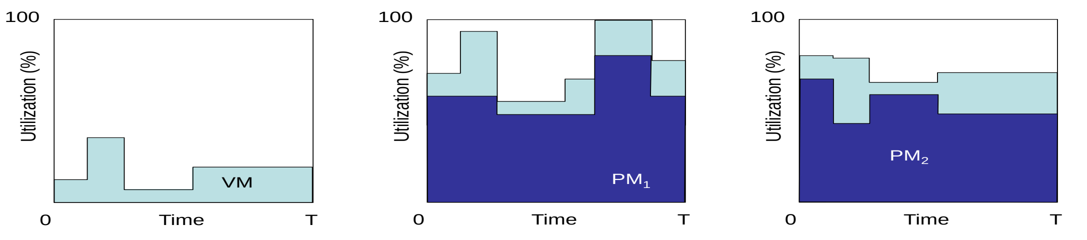
We use a norm-based greedy algorithm [36] to capture the distance between the average resource demand vector and the capacity vector of a PM (e.g., the top right corner of the rectangle in the 2-dimensional space):

$$M_j = \sum_{k=1}^{d} \{w_k(1 - E_j^k)\}^2, \tag{3.5}$$

where w_k is the assigned weight to resource k, which can be determined by resource insensitive aware algorithms [15]. For simplicity, we can make all weights the same and set $w_k = 1$. This distance metric coordinately measures the closeness of each resource's utilization to 1.

To identify the PM from a group PMs to allocate a requested VM i, our initial VM allocation mechanism first identifies the PMs that do not violate the capacity constraint of Equ. (3.3). It then places the VM i to a PM that minimizes the distance M_j, that is, this VM can more fully utilize each resource in this PM.

Algorithm 2 shows the pseudocode for our initial VM allocation mechanism. This mechanism refers to the resource demand pattern $\mathcal{P}_i(t)$ from the library that approximately predicts the resource demands of VMs from the same tenant for the same job. Based on $\mathcal{P}_i(t)$ and the host capacity vector $\mathcal{H}_j$, we can derive predicted $\mathcal{F}_{ij}(t)$. For each candidate host, we first check whether it has enough resource for hosting the VM at each time $t = T_0, ..., T_0 + T$ for each resource by comparing $\mathcal{F}_{ij}(t)$ and $\mathcal{R}_j(t)$ (Line 5 and Lines 18-25) in order to ensure that $F_{ij}^k(t) \leq R_j^k(t)$ (Eq.(3.3)) during the VM lifetime or periodical interval $[T_0, T_0 + T]$. If the host has sufficient residual resource capacity to host this VM, then we calculate the resource efficiency (Lines 8-11) after allocating this VM during time period $[T_0, T_0 + T]$ using Eq. (3.4). Finally, we choose the PM that leads to the minimum distance based on resource efficiency (Lines 12-16). It means this VM can make this PM most fully utilize its different resources among the PM candidates. In this way, the complementary VMs are allocated to the same PM, thus fully utilizing its different resources.



(a) VM (b) VM+PM1 (c) VM+PM2

Figure 3.12: Placing VM to PM1 and PM2.

3.2.2 Enhanced Initial VM Allocation Mechanisms

3.2.2.1 Basic Rationale

In the previous section, the VM allocation mechanism tries to maximize the *resource efficiency* during the monitoring time period based on Equ. (3.4). However, E_j^k is the average utilization of PM j during the monitoring time period, and it cannot reflect the deviation of the resource utilization during this period. For example, the time period consists of epochs t_1 and t_2. A PM with a resource usage of 10 units at epoch t_1 and a usage of 20 units at epoch t_2 has the same *resource efficiency* as a PM with usages of 15 units at both t_1 and t_2. Let's say we are selecting a PM for hosting a VM from two candidates. The VM demands 10 unites of resource at epoch t_1 and 20 units of resource at epoch t_2. The first PM's available capacity is 100 units and 20 units for the two epochs, respectively. The second PM's total capacity is 60 for both epochs. Both candidate PMs have the same *resource efficiency.* If we choose the first PM, the capacity is used up at epoch t_2. It cannot host more VMs though it has available capacity at epoch t_1. Choosing the second PM is preferred as it can still host extra VMs after accepting the VM.

Figure 3.12 shows another example. We need to select a PM from PM1 and PM2 for a VM with resource utilization indicated in Figure 3.12(a). Figure 3.12(b) and Figure 3.12(c) show the resource utilization of PM1 and PM2 after allocating the VM to them, respectively. They have the same average *resource efficiency* during T after the VM allocation. Then, PM1 and PM2 are equivalent selection options according to Algorithm 2. However, we can see that PM2 is a better option because PM2 still has available resource for hosting more VMs, while allocating the VM to PM1 will use up its resource for a small period of time and hence make it unable to host more VMs. In other words, allocating the VM to PM1 is not a clever choice as it will lead to resource wastage in PM1. In order to solve this problem and further improve resource utilization efficiency, we can further consider the deviation during different epochs to distinguish the above two PMs instead of only calculating the average E_j^k. In this example, since PM1 has less deviation between different epochs after hosting the VM, it will be selected as the destination PM to host the VM. In the following, we will introduce three methods to improve the initial VM allocation mechanism.

3.2.2.2 Utilization Variation Based Mechanism

In order to further distinguish PMs, we should measure other metrics instead of only calculating the average E_j^k. We can exam the utilization variation of the estimated utilization curve of a PM j after accepting the VM. We define the variance of a PM j with residual resource $R_j^k(t_i)$ as

$$\sigma^2 = \frac{\sum_{i=1}^{N}[R_j^k(t_i) - \overline{R_j^k(t_i)}]}{N} \tag{3.6}$$

where $R_j^k(t_i)$ is the residual type-k resource at time t_i, and $\overline{R_j^k(t_i)}$ is the average residual type-k resource. σ^2 is the utilization variation, which measures how far a set of numbers is spread out. A variance of zero indicates that all the values are identical. A small variance indicates that the data points tend to be very close to the mean and hence close to each other, while a high variance indicates that the data points are very spread out around the mean and dispersed from each other. We can select PMs that will have identical resource utilization between time epochs after accepting the VM based on the utilization variation of the resulting utilization of the PM.

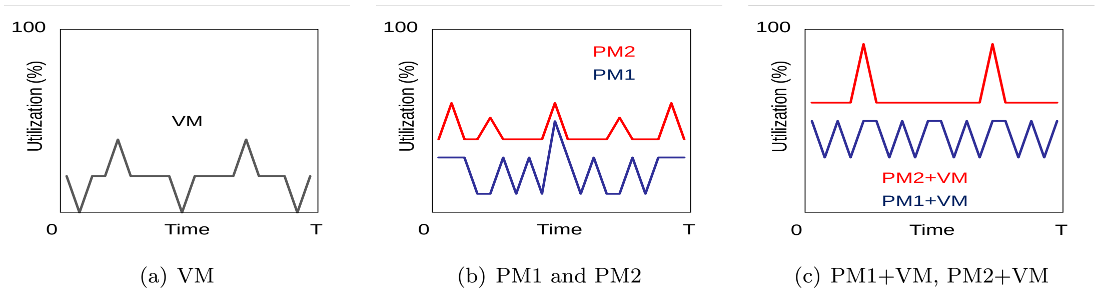
Algorithm 3 Pseudocode for the utilization variation based VM allocation mechanism.

```
1: Input: $\mathcal{P}_i(t)$: Predicted resource demands
          $\mathcal{R}_j(t)$: Residual resource capacity of candidates
2: Output: Allocated host of the VM
3:    $Var$=Double.MAX_VALUE //utilization variation
4:    for $j = 1$ to $m$ do
5:        if CheckValid($\mathcal{P}(t)$,$\mathcal{R}_j(t)$)==false then
6:            continue
7:        else
8:            for $k = 1$ to $d$ do
9:                $\sigma^2 = \frac{\sum_{i=1}^{N}[(R_j^k(t_i)-P^k(t_i))-\overline{R_j^k(t_i)-P^k(t_i)}]}{N}$
10:               $Var_j = Var_j + \sigma^2$
11:           end for
12:           if $Var_j$<$Var$ then
13:               $Var$=$Var_j$
14:               AllocatedHost=host $j$
15:   end for
16:   return AllocatedHost
```

Algorithm 3 shows the pseudocode for the utilization variation based VM allocation mechanism. Similar to Algorithm 2, this mechanism refers to the resource demand pattern $\mathcal{P}_i(t)$ of VM i and the residual resource capacity $\mathcal{R}_j(t)$ of candidate PM j. For each candidate host, the algorithm first checks whether it has enough resource for hosting the VM for each resource by calling CheckValid($\mathcal{P}(t)$,$\mathcal{R}_j(t)$) (Lines 5-7). If the host has sufficient residual resource capacity to host this VM, then we calculate the utilization variation of the utilization curve after allocating this VM during time period $[T_0, T_0 + T]$ using Eq. (3.6) (Line 9). Finally, we choose the PM that leads to the minimum utilization variation (Lines 12-14). It means this VM can make this PM have similar resource utilization between time epochs, and hence have the potential to host more VMs in the future and fully utilizes its resources.

3.2.2.3 Correlation Coefficient Based Mechanism

The utilization variation based algorithm ensures that the PM resource utilization does not spread out around the mean. However, it cannot fully reflect the complementariness of the VM utilization and PM utilization during the time period. In the example shown in Figure 3.13, we need to select a PM from two PMs to allocate a VM. The utilizations of the VM and PMs are shown in Figure 3.13(a) and Figure 3.13(b), respectively. Figure 3.13(c) shows the resource utilization after allocating the VM to each PM. Both curves have the same utilization variation value, so the two PMs are equivalent in PM selection since selecting either one will result in the same utilization variation according to Algorithm 3. However, PM1 is a better choice because it is more complementary to the VM, and will result in a more flat resource utilization, which enables to allocation more VMs in the PM. In order to further take advantage of such complementariness between VMs to be allocated and PMs, we further propose a correlation coefficient based initial VM allocation mechanism.



(a) VM (b) PM1 and PM2 (c) PM1+VM, PM2+VM

Figure 3.13: Placing VM to PM1 and PM2.

The statistical correlation coefficient (denoted by c_r) for a VM with predicted resource demands $\mathcal{P}(t)$ and a PM j with residual resource capacity $\mathcal{R}(t)$ are calculated by

$$c_r = \frac{\sum_{i=1}^{N}(P^k(t_i) - \overline{P^k(t_i)})(R_j^k(t_i) - \overline{R_j^k(t_i)})}{\sqrt{\sum_{i=1}^{N}(P^k(t_i) - \overline{P^k(t_i)})^2 \cdot \sum_{i=1}^{N}(R_j^k(t_i) - \overline{R_j^k(t_i)})^2}} \tag{3.7}$$

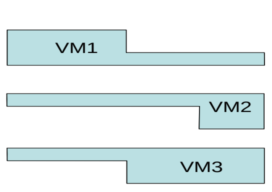
where $P^k(t_i)$ is predicted type-k resource demand at time t_i and $R_j^k(t_i)$ is residual type-k resource at time t_i, $\overline{P^k(t_i)}$ and $\overline{R_j^k(t_i)}$ are the average value and N is the total number of time periods of measurement. The correlation coefficient illustrates a quantitative measure of the correlation (i.e., statistical relationships) between the two utilization traces. It ranges from -1 to 1. A correlation coefficient closer to 1 means that the two traces are more similar, a correlation coefficient closer to -1 indicates a more perfect negative correlation, that is, the two traces are opposite to each other in terms of magnitude, and a correlation coefficient closer to 0 means less relationship between the two traces. For example, if VM1 has 100% and 0% utilization while VM2 has 0% and 100% utilization in the first and second halves of time period T, then VM1 and VM2 have a -1 correlation coefficient. Therefore, for a VM, we aim to find a PM that has a correlation coefficient most close to -1 (i.e., the smallest correlation coefficient) with the VM as the destination PM to allocate this VM. Accordingly, we propose to select PMs based on the correlation coefficient of the VM utilization and PM utilization. As the VMs consume multiple types of resources, the algorithm first calculates the correlation coefficient for each resource and then calculates the average of all the correlation coefficients of different resources. The algorithm finds the PM that has the smallest average correlation coefficient (i.e., most close to -1) with the VM to be allocated as the VM's host. A PM with the smallest average correlation coefficient with the VM means that this VM allocation will result in resource utilization that does not fluctuate severely and hence has higher probability to accommodate more VMs.

Algorithm 4 Pseudocode for the correlation coefficient based VM allocation mechanism.

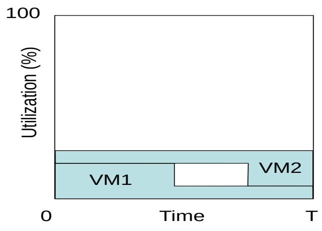
```
1: **Input:** $\mathcal{P}_i(t)$: Predicted resource demands
           $\mathcal{R}_j(t)$: Residual resource capacity of candidates
2: **Output:** Allocated host of the VM
3:     $Cor$=Double.MAX_VALUE
4:     for $j = 1$ to $m$ do
5:         if CheckValid($\mathcal{P}(t)$,$\mathcal{R}_j(t)$)==false then
6:             continue
7:         else
8:             $Avg_j$=0 //average correlation coefficient
9:             for $k = 1$ to $d$ do
10:                $c_r = \frac{\sum_{i=1}^{N}(P^k(t_i)-\overline{P^k(t_i)})(R^k(t_i)-\overline{R^k(t_i)})}{\sqrt{\sum_{i=1}^{N}(P^k(t_i)-\overline{P^k(t_i)})^2 \cdot \sum_{i=1}^{N}(R^k(t_i)-\overline{R^k(t_i)})^2}}$
11:            $Avg_j = Avg_j + \frac{c_r}{N}$
12:            end for
13:            if $Avg_j < Cor$ then
14:              $Cor=Avg_j$
15:              AllocatedHost=host $j$
16:    end for
17:    return AllocatedHost
```

Algorithm 4 shows the pseudocode for the correlation coefficient based VM allocation mechanism. Similar to Algorithm 2, this mechanism refers to the resource demand pattern $\mathcal{P}_i(t)$ and the residual resource capacity of candidates $\mathcal{R}_j(t)$. The algorithm first checks whether the candidate host has enough resource (Lines 5-7). It then calculates the correlation coefficient of the VM utilization and the residual resource capacity of the candidates for each type of resource based on Equ.

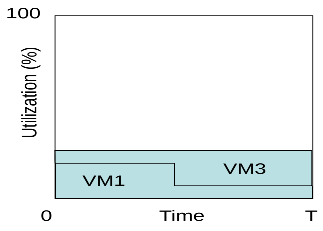
(3.7) (Line 10). It averages the correlation coefficient across different types of resource (Line 11). Specifically, the algorithm calculates the average Avg_j of the correlation coefficient values that are obtained from the utilization trace of each resource, and then selects the PM that has the smallest Avg_j (Lines 13-15). It means that this PM is the most complimentary to the VM across different resource, and allocating the VM to this PM can make this PM have similar resource utilization between time epochs during time period T. In this way, the algorithm is actually allocating complementary VMs (e.g., the VM is complementary with the existing VMs in the PM) to the same PM. As a result, the PM will have similar resource utilization between time epochs, and hence has potential to host more VMs in the future and fully utilizes its resources (i.e., more accommodating).



(a) VMs



(b) VM1+VM2



(c) VM1+VM3

Figure 3.14: Placing VMs vs. placing the VM group to PM.

3.2.2.4 VM Group Based Mechanism

Rather than considering one VM, in this section, we try to place complementary VMs together by combining complementary VMs into a group first and then assigning the whole group to a PM. Compared to allocating VMs individually, combining complementary VMs into a group first for allocation has the advantage of extensively exploring the complementariness of the VMs and maximally consolidating complementary VMs, and hence can reduce the number of PMs needed. For example, suppose we allocate three VMs in the sequence of VM1, VM2 and VM3, as shown in Figure 3.14(a). Since the allocation result depends on the allocating order of the VMs, Algorithm 4 will end up with placing VM1 and VM2 together as shown in Figure 3.14(b). However, VM3 is more complementary than VM2 to VM1. Placing VM3 and VM1 togather is more preferred because it will result in similar resource utilization between time epochs in the PM, and hence make the PM more accommodating to other VMs. If we group complementary VMs together and then do the allocation, we can place VM1 and VM3 in one PM as shown in Figure 3.14(c) and hence make the PM more accommodating.

A question in grouping complementary VMs is which VM we should start with. In online VM allocation algorithms, it is difficult to find a PM to place a VM with high resource utilization variations, especially when such VMs are allocated later with less residual resources in PMs. Therefore, we give higher priorities to the VMs with higher utilization variation to start with in VM grouping, so that they will have more chances in finding complementary VMs. Specifically, in order to group complementary VMs together, we first sort the VMs based on the utilization variation in descending order. Then, we start from the first VM for VM grouping.

We can combine arbitrary number of VMs into one group, as long as the group resource demand does not exceed the PM resource capacity. We define the group resource demand as the combined resource demands of each type of resource of the VMs in the group. There is a tradeoff between the number of VMs that are selected to form a group and the complexity of the algorithm. In order to demonstrate the effectiveness of the VM group based mechanism and also achieve time efficiency of the mechanism, we combine two VMs in a group without the loss of generosity. The procedure of combining VMs to groups is as follows. For each VM, we select the VM that is most complementary to it, and then combine these two VMs. For example, we calculate the correlation coefficients of this VM with all remaining VMs and select the one with the smallest correlation coefficient value. After that, we denote the VM groups as $G_n (n = 1, 2, ...)$, and sort the groups based on the group resource demand. Similar to the predicted resource demand pattern $\mathcal{P}_i(t)$ of a VM, the group resource demand is a d-dimension vector with each dimension representing its demands in one resource type. Suppose a group G_n comprises of m VMs, the combined resource demand of this group is:

$$\mathcal{P}_{G_n}(t) = (\sum_{i=1}^{m} P_i^1(t), \sum_{i=1}^{m} P_i^2(t), ..., \sum_{i=1}^{m} P_i^d(t)) \tag{3.8}$$

where $P_i^k(t)$ is the type-k resource utilization of VM i, and $\sum_{i=1}^{m} P_i^k(t)$ is the combined type-k resource demands of the m VMs in the group. The group resource demand can be calculated by

$$S_{G_n} = \sum_{k=1}^{d} \left\{ w_k \frac{1}{T} \int_{T_0}^{T_0+T} [\sum_{i=1}^{m} P_i^k(t)] dt \right\}^2, \tag{3.9}$$

where $\frac{1}{T}\int_{T_0}^{T_0+T}[\sum_{i=1}^{m}P_i^k(t)]dt$ is the demand of type-k resource of group G_n, and w_k is the weight associated to type-k resource as in Equ. (3.5).

The reason for sorting the groups is that it is more difficult to find destination PMs to allocate the groups with large group resource demands, especially if such a group is allocated later after many other VM groups with few PM options left. Similar as the first-fit decreasing algorithm [6] that allocates large demand VM first, this algorithm can lead to fewer PMs used by allocating the groups with larger group resource demands first.

Similarly, we define the residual resource capacity of a PM based on the normalized residual resource capacity of the PM $\mathcal{R}_j(t) = (R_j^1(t), R_j^2(t), ..., R_j^d(t))$. The residual resource capacity of PM j is a positive scalar value representing the magnitude of the resource utilization in multiple dimensions, which can be calculated by

$$S_j = \sum_{k=1}^{d}\{w_k\frac{1}{T}\int_{T_0}^{T_0+T}R_i^k(t)dt\}^2, \tag{3.10}$$

where $\frac{1}{T}\int_{T_0}^{T_0+T}R_i^k(t)dt$ is the residual resource capacity of type-k resource in the PM j; w_k is the assigned weight to resource k (the same with Equ. (3.5)).

Algorithm 5 shows the pseudocode for the VM group based allocation mechanism, that is used to derive the decisions of assigning VM groups to PMs, based on the residual resource capacities of PMs and group resource demands of VM groups. Given a list of VMs $\mathbb{L}_{VM}$ with their predicted resource demands $\mathcal{P}_i(t)$, and a list of PMs $\mathbb{L}_{PM}$ with their residual resource capacities $\mathcal{R}_j(t)$ (Line 1), the algorithm sorts the VMs based on their utilization variations calculated by Equ. (3.6) (Line 3). For each VM in the list $\mathbb{L}_{VM}$, the algorithm finds a VM that is the most complementary to the first VM (Lines 6-10), combines them into a group (Line 11), and then adds to the group list $\mathbb{L}_G$ (Line 12). The algorithm computes and sorts the groups based on their group resource demands and sort the PMs based on their residual resource capacities (Line 13), and then allocates the group with the biggest group resource demand to a feasible PM with the smallest residual resource capacity (Line 15). If a feasible PM cannot be found, the algorithm returns false (Lines 16-17), otherwise, it returns the VM-to-PM mapping after all the groups are allocated to the PMs (Line 18).

Algorithm 5 Pseudocode for the VM group based allocation mechanism.

```
1: Input: $\mathbb{L}_{VM}$: list of VMs with predicted resource demands $\mathcal{P}_i(t)$
          $\mathbb{L}_{PM}$: list of PMs with residual resource capacities $\mathcal{R}_j(t)$
2: Output: VM to PM mapping
3:     Arrays.sort($\mathbb{L}_{VM}$)
4:     $\mathbb{L}_G$ = new Array()
5:     while $\mathbb{L}_{VM}$ not empty do
6:         VM1=$\mathbb{L}_{VM}$.remove() // Removed VM from list
7:         for VM2 in $\mathbb{L}_{VM}$
8:             Compute correlation coefficient of VM1 and VM2
9:         VM2=VM that has lowest correlation coefficient with VM1
10:        Remove VM2 from $\mathbb{L}_{VM}$
11:        Create group $G$ that comprises VM1 and VM2
12:        $\mathbb{L}_G$.add($G$)
13:    Compute group resource demands and residual resource capacities based on Equ. (3.9) and (3.10)
14:    while $\mathbb{L}_G$ not empty do
15:        The biggest group $G \rightarrow$ the smallest feasible PM
16:        if cannot find feasible PM then
17:            return False
18:    return VM to PM mapping
```

3.3 Trace-Driven Simulation Performance Evaluation

In this section, we conducted the simulation experiments to evaluate the performance of our proposed complementary VM allocation mechanism (denoted by CompVM) using VM utilization trace from PlanetLab [13] and Google Cluster [21]. We used workload records of three days from the trace to generate VM resource request patterns and then executed CompVM for the fourth day's resource requests. The window size was set to 15 in the pattern detection in CompVM. We compared CompVM with Wrasse [40] and CloudScale [43], which are dynamic VM allocation methods. All three methods first conduct initial VM allocation and then periodically execute VM migration by migrating VMs from overloaded PMs to first-fit PMs every 5 minutes. In the initial VM allocation, Wrasse and CloudScale place each VM to the first-fit PM based on the expected VM resource demands. In migration, CloudScale first predicts future demands and then migrates VMs to achieve load balance in a future time point.

In the default setup, we configured the PMs in the system with capacities of 1.5GHz CPU and 1536 MB memory and configured VMs with capacities of 0.5GHz CPU and 512 MB memory. With our experiment settings, the bandwidth consumption did not overload PMs due to their high network bandwidth capacities, so we focus on CPU and memory utilization. Unless otherwise specified, the number of VMs was set to 2000 and each VM's workload is twice of its original workload in the trace. We measured the following metrics after the simulation was run for 24 hours to report.

- *The number of PMs used.* This metric measures the energy efficiency of VM allocation mechanisms.

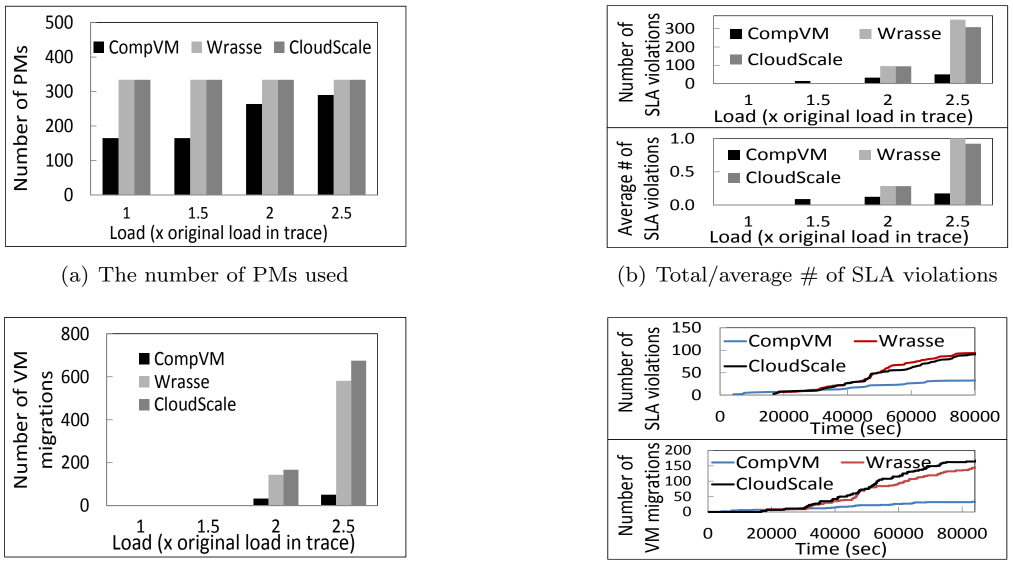
- *The number of SLA violations.* This is the number of occurrences that a VM cannot receive the required amount of resource from its host PM.

- *Average number of SLA violations.* This is the average number of SLA violations per PM. It reflects the effect of consolidating VMs into relatively fewer PMs.

- *The number of VM migrations.* This metric presents the cost of the allocation mechanisms that required satisfying VM demands and avoiding SLA violations.

3.3.1 Performance with Varying Workload

Figure 3.15 and Figure 3.16 show the performance of the three methods under different VM workloads using the PlanetLab trace and Google Cluster trace, respectively. We varied the workload of the VMs through increasing the original workload in the trace by 1.5, 2 and 2.5 times.



(a) The number of PMs used

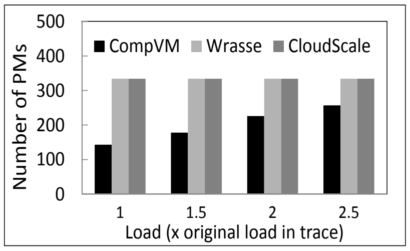
(b) Total/average # of SLA violations

(c) The number of VM migrations

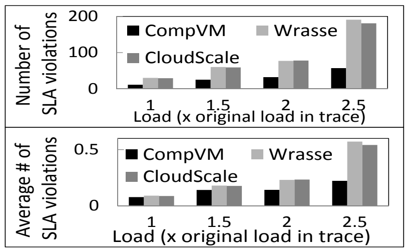
(d) # of SLA violations and migrations

Figure 3.15: Performance under different workloads using the PlanetLab Trace.

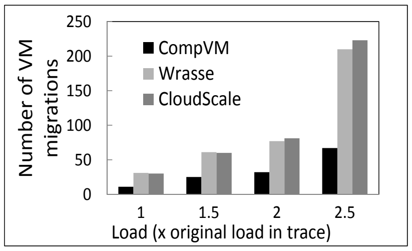
Figure 3.15(a) and Figure 3.16(a) show the total number of PMs used, which follows CompVM< CloudScale=Wrasse. CloudScale and Wrasse aim to avoid overloading each PM in initial VM placement and subsequent VM migration at each time point. This may result in some PMs that fully utilize one resource but under-utilize other resources, failing to fully utilize all resources. In contrast, in initial VM placement, CompVM consolidates complementary VMs in different resource dimensions, thus fully utilizing each resource in each PM. Since it considers the resource periodical utilization patterns during a certain time period, it reduces the VM migrations and constrains the number of PMs used. Both figures also show that as the workload increases, the number of PMs of CompVM increases, while those of Wrasse and CloudScale remain the same. This is because as the actual workload increases, CompVM's predicted resource demands increase in initial VM placement, while CloudScale and Wrasse still allocate VM according to the labeled VM capacities. The result further confirms that CompVM uses PM resources based on actual usage, while CloudScale and Wrasse under-utilize some resources by provisioning PM resources more than needed. As a result, CompVM needs much fewer PMs than CloudScale and Wrasse, hence achieves higher energy efficiency.



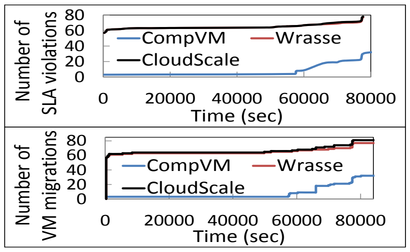
(a) The number of PMs used



(b) Total/average # of SLA violations



(c) The number of VM migrations



(d) # of SLA violations and migrations

Figure 3.16: Performance under different workloads using the Google Cluster Trace.

Figure 3.15(b) and Figure 3.16(b) show the total number of SLA violations and the average number of SLA violations. We see that with the PlanetLab trace, when the workload is low, all three methods can provide service without violating SLAs. Both figures show that as the workload increases, both metric results increase and they exhibit CompVM<CloudScale<Wrasse. CompVM has fewer SLA violations because its predicted patterns can capture the time-varying VM resource

demands and hence guarantee the resource provisioning. CloudScale has fewer SLA violations than Wrasse since CloudScale iteratively predicts VM resource demands and proactively migrates VMs before SLA violations occur. These results illustrate that CompVM maintains a smaller average number of SLA violations per PM even though it uses fewer PMs than CloudScale and Wrasse, which confirms CompVM's higher performance in energy efficiency and SLA guarantees.

Figure 3.15(c) and Figure 3.16(c) show the total number of VM migrations in the three methods. Since the workload in the PlanetLab trace is relatively low compared to the Google Trace trace, when the workload is low, there are no SLA violations hence no VM migrations. Both figures show that as the workload increases, the number of VM migrations increases due to the increase of SLA violations as shown in Figure 3.15(b) and Figure 3.16(b). CompVM always triggers significantly fewer VM migrations than CloudScale and Wrasse due to its much fewer SLA violations. This experimental result confirms the effectiveness of CompVM in reducing VM migrations.

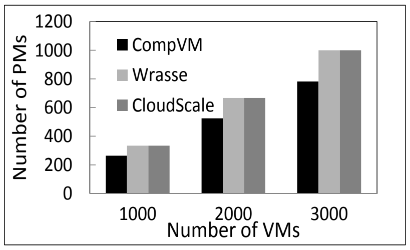
Figure 3.15(d) and Figure 3.16(d) show the accumulated number of SLA violations and VM migrations over time, respectively. In Figure 3.15(d), as the workload is low relative to PM capacity initially in the PlanetLab trace, all three methods have similar number VM violations and migrations at the early stage of simulation. As time goes on, due to the awareness of future resource demand pattern of the VMs during initial VM allocation, CompVM produces fewer VM violations and migrations than Wrasse and CloudScale during the experiment.

In the Google Cluster trace, the workload is high relative to PM capacity initially. Therefore, in Figure 3.16(d), due to the unawareness of future VM resource demands, the initial VM placement of Wrasse and CloudScale leads to around 60 VM migrations to guarantee enough resource provisioning. In contrast, CompVM generates 0 SLA violations and 0 migrations until at 6000s when the workload becomes higher. We also observe that when the workload is high relative to PM capacity, most of the migrations are caused by inappropriate initial VM placement. Therefore, our initial VM allocation mechanism is significant in helping greatly reduce the SLA violations and VM migrations.

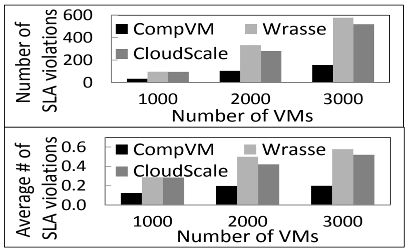
3.3.2 Performance with Varying Number of VMs

Figure 3.17 and Figure 3.18 present the performance of the three methods when the number of VMs was varied from 1000 to 3000 using the PlanetLab trace and the Google Cluster trace, respectively.

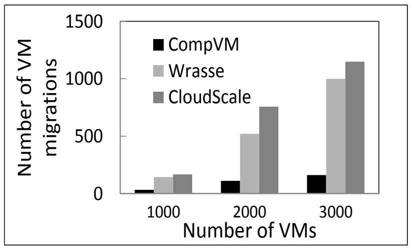
Figure 3.17(a) and Figure 3.18(a) show the total number of PMs used to provide service for



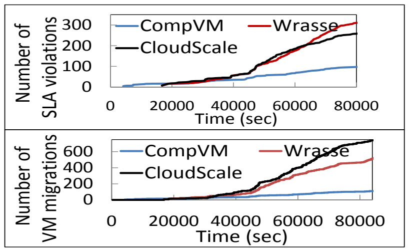
(a) The number of PMs used



(b) Total/average # of SLA violations



(c) The number of VM migrations



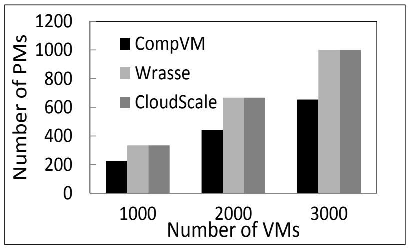
(d) # of SLA violations and migrations

Figure 3.17: Performance with different number of VMs using the PlanetLab Trace.

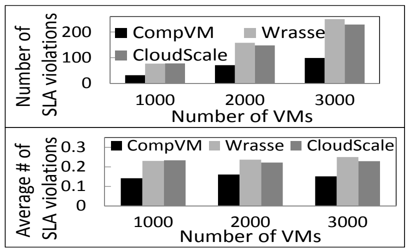
the corresponding number of VMs. We see the result follows CompVM<CloudScale=Wrasse due to the same reasons as in Figure 3.15(a) and Figure 3.16(a). Also, as the number of VMs increases, the number of PMs used increases in each method since more PMs are needed to host more VMs. These experimental results confirm the advantage of CompVM in reducing the number of PMs used hence achieving higher energy efficiency.

Figure 3.17(b) and Figure 3.18(b) show the number of SLA violations and the average number of SLA violations per PM. We see both metric results follow CompVM < CloudScale<Wrasse due to the same reasons in Figure 3.15(b) and Figure 3.16(b). Also, as the number of VMs increases, both metric values in each method increase since more resource demands from more VMs lead to more SLA violations.

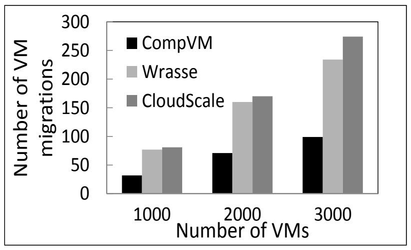
Figure 3.17(c) and Figure 3.18(c) show the total number of VM migrations in the three methods. As the number of VMs increases, the number of VM migrations increases due to the increase of SLA violations. CompVM always triggers significantly fewer VM migrations than CloudScale and Wrasse due to its much fewer SLA violations as shown in Figure 3.17(b) and Figure 3.18(b). CloudScale has slightly more migrations than Wrasse because it triggers VM migrations upon a predicted SLA violation, which may not actually occur. These experimental results confirm the effectiveness of CompVM in reducing VM migrations.



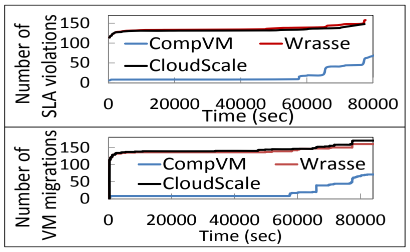
(a) The number of PMs used



(b) Total/average # of SLA violations



(c) The number of VM migrations



(d) # of SLA violations and migrations

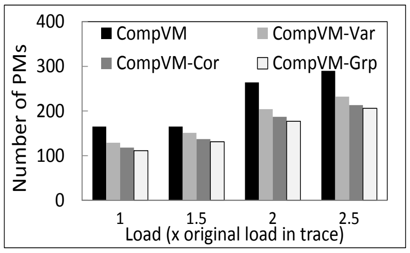
Figure 3.18: Performance with different number of VMs using the Google Cluster Trace.

Figure 3.17(d) and Figure 3.18(d) show the number of migrations and SLA violations over time. The figures show similar trends of the three method as those shown in Figure 3.15(d) and Figure 3.16(d) due to the same reasons.

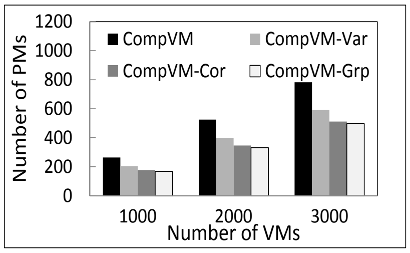
3.3.3 Performance of Enhancement Mechanisms

We implemented the improved initial VM allocation mechanisms described in Section 3.2.2, and then compared them with CompVM, the original heuristic algorithm based on the average resource efficiency. We denote the utilization variation based mechanism as CompVM-Var (Algorithm 3), denote the correlation coefficient based mechanism as CompVM-Cor (Algorithm 4), and denote the VM group based mechanism as CompVM-Grp (Algorithm 5).

Figure 3.19 compares the performance of the improved initial VM allocation mechanisms with CompVM using the PlanetLab trace. Figure 3.19(a) and Figure 3.19(b) show the total number of PMs used, with varying workloads and varying number of VMs, respectively. In both figures, the number of PMs follows CompVM-Grp≈CompVM-Cor<CompVM-Var<CompVM. CompVM-Var consolidates VMs to PMs based on the variation of resource utilization of PM after accommodating the VM. Compared to CompVM that is based on the average resource efficiency, CompVM-Var reduces the number of used PMs due to the reason that CompVM-Var tries to improve resource



(a) Varying workload.



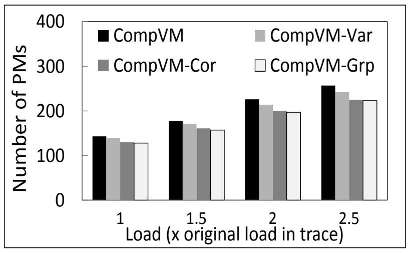
(b) Varying number of VMs.

Figure 3.19: The number of PMs used with PlanetLab trace.

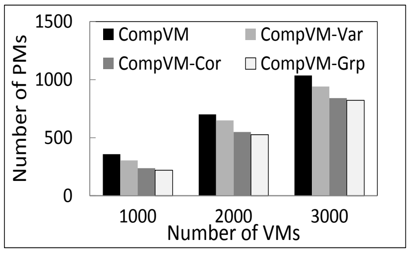
utilization by ensuring that the PM resource utilization does not spread out around the mean, and hence is able to consolidate more VMs, which leads to fewer PMs needed to host the VMs. CompVM-Cor further reduces the number of used PMs because the correlation coefficient based method ensures that the selected PM has the most complementary resource utilization to the VM and hence results in a high resource utilization during every time epoch, thus enabling a PM to host more VMs. CompVM-Grp has similar number of PMs to CompVM-Cor since both of them rely on correlation coefficient when selecting PMs for VMs.

Compared to CompVM-Cor, CompVM-Grp has a slightly fewer PMs because it considers a group of VMs rather than a single VM when assigning the VMs to the PMs. Combining complementary VMs into groups before allocating them to PMs has the advantage of extensively exploring the complementariness of the VMs and maximally consolidating complementary VMs, and hence enables a PM to host more VMs, which further reduces the total number of PMs needed to host all VMs. We also see that Figure 3.19(a) shows that as the workload increases, the number of PMs of all methods increases. This is because as the actual workload increases, the predicted resource demands increase in initial VM placement. Figure 3.19(b) shows that as the number of VMs increases, the number of PMs used increases in each method since more PMs are needed to host more VMs. These experimental results confirm the effectiveness of the improved initial VM allocation mechanisms in reducing the number of used PMs.

Similarly, Figure 3.20 compares the performance of the improved initial VM allocation mechanisms with CompVM using the Google Cluster trace. Figure 3.20(a) and Figure 3.20(b) show the total number of PMs used, with varying workloads and varying number of VMs, respectively. We see that the number of PMs follows CompVM-Grp≈CompVM-Cor<CompVM-Var<CompVM, which is consistent with previous result using the PlanetLab trace due to the same reasons mentioned



(a) Varying workload.



(b) Varying number of VMs.

Figure 3.20: The number of PMs used with Google Cluster trace.

before. Using the Google Cluster trace, the improved initial VM allocation mechanisms do not reduce as many PMs as using the PlanetLab trace in the previous experiment. This is because the resource utilization from the Google Cluster trace does not fluctuate as severely as the resource utilization in the PlanetLab trace, hence the average resource efficiency mechanism performs well in guiding initial VM allocation. These experimental results again confirm effectiveness of the improved initial VM allocation mechanisms. The results also indicate that these mechanisms are more effective when the resource utilizations exhibit greater fluctuation.

3.4 Real-World Testbed Experiments

We deployed a real-world testbed to conduct experiments to validate the performance of CompVM in comparison with Wrasse and CloudScale. The testbed consists of 7 PMs (2.00GHz Intel(R) Core(TM)2 CPU, 2GB memory, 60GB HDD) and an NFS (Network File System) server with storage capacity of 80GB. We implemented CompVM, Wrasse and CloudScale in Java using the XenAPI library [59] running in a management PM (3.00GHz Intel(R) Core(TM)2 CPU, 4GB memory). We used the VM template of XenServer to create VMs (1VCPU, 256MB memory, 8.0G-B virtual disk, running Debian Squeeze 6.0) in the cluster. We used publicly available workload generator *lookbusy* [31] to generate VM workloads.

Figure 3.21 shows the number of PMs used to provide the service of different number of VMs. Since Wrasse and CloudScale are unable to predict workload at the beginning, they both use the maximum request resource of the VMs for allocation and hence have similar results. We also monitored the number of SLA violations during the experiment period, and found that were no SLA violations in all three methods during the experiment. These experimental results confirm that

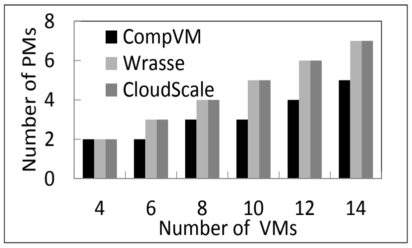


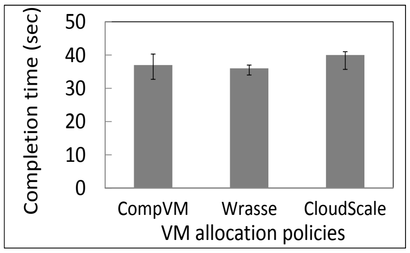
Figure 3.21: # of PMs in testbed.

CompVM is able to provide service with fewer number of PMs than Wrasse and CloudScale while ensures SLA guarantees.

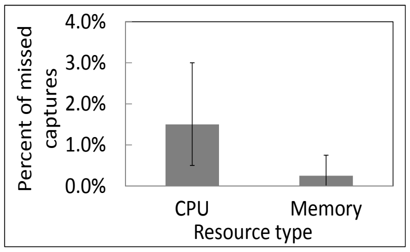
Table 3.1: VM allocation mapping.

PM	CompVM	Wrasse	CloudScale
PM1	VM1, VM2	VM1, VM2	VM1, VM2
PM2	VM3, VM4, VM5	VM3, VM4	VM3, VM4
PM3	-	VM5	VM5

We then deployed a virtual cluster with 5 VMs collaboratively running the WordCount Hadoop benchmark job. We first conducted a profiling run of such MapReduce job and used the collected resource utilization to generate patterns for initial VM allocation in CompVM. The 5 VMs were initially allocated to different PMs by different methods. The initial VM to PM mapping is shown in Table 3.1. We see that CompVM uses fewer PMs than Wrasse and CloudScale. During the experiment, no SLA violations were detected in all three methods. Figure 3.22(a) shows the median, 10th percentile and 90th percentile of the job completion time in ten experiments. We see that though CompVM uses few PMs, it has a similar completion time as Wrasse and CloudScale. This result verifies the advantage of CmpaVM in requiring fewer PMs without sacrificing the performance quality of the VMs.



(a) Job completion time



(b) % of missed captures in CompVM

Figure 3.22: Performance of running WordCount job on the real-world testbed.

Figure 3.22(b) shows the median, 10th and 90th percentiles of the percent of missed captures of CompVM during the experiment. We see that CompVM produces very few missed captures relative to the total number of predictions at each time point, which verifies the effectiveness of CompVM in resource demand pattern detection. We also see that the percent of missed captures of CPU and its variance are relatively larger than those of memory. This is due to the reason that the memory utilizations of the VMs exhibit more obvious patterns and hence are easier to be captured in pattern detection.

Chapter 4

Resource Intensity Aware VM Migration for Load Balance

4.1 Objectives and Problem Statement

4.1.1 Notations and Final Objective

We consider a scenario in which a total of N PMs serve as a resource pool in the cloud. Let P_i denote PM i ($i = 1, 2, ..., N$), and n_i be the number of VMs hosted by P_i, denoted by V_{ij} ($j = 0, 1, ..., n_i$). Let C_{ik} ($k \in K$) denote the capacity (total amount) of type-k resource owned by P_i, where K is the set of resources.

Let $L_{ijk}(t)$ denote the type-k resource requested by V_{ij} in P_i at time t. It is a time varying function. To avoid small transient spikes of $L_{ijk}(t)$ measurements that trigger needless VM migrations, we use the average of $L_{ijk}(t)$ during time period Δt, denoted by $\overline{L_{ijk}}$.

$$\overline{L_{ijk}} = \frac{1}{\Delta t} \int_{t-\Delta t}^{t} L_{ijk}(t)\mathrm{d}t \tag{4.1}$$

Δt is an adaptive value depending on how fine grained we want to monitor the resource demands.

The usage of type-k resource in P_i is the sum of type-k resource requested by its VMs:

$$L_{ik} = \sum_{j=1}^{n_i} \overline{L_{ijk}} \tag{4.2}$$

Taking into account the heterogeneity of server capacities, we define the utilization rate of type-k resource in P_i (denoted by u_{ik}) as the ratio between actual requested resource amount of all VMs in P_i and the capacity of type-k resource of P_i.

$$u_{ik} = \frac{L_{ik}}{C_{ik}}. \tag{4.3}$$

We use Θ_k to denote the predetermined utilization threshold for the type-k resource in a PM in the cloud. The final objective of RIAL is to let each P_i maintain $u_{ik} < \Theta_k$ for each of its type-k resource (i.e., lightly loaded status). We call a PM with $u_{ik} > \Theta_k$ *overloaded PM*, and call this type-k resource *overutilized resource*.

Cloud customers buy VMs from cloud provider with predefined capabilities. For example, a small VM instance in Amazon EC2 is specified by 1.7GB of memory, 1 EC2 compute unit, 160G-B of local instance storage, and a 32-bit platform. We use C_{ijk} to denote label capacity of V_{ij} corresponding to type-k resource. The utilization of V_{ij} is defined as

$$u_{ijk} = \frac{\overline{L_{ijk}}}{C_{ijk}} \tag{4.4}$$

In order to deal with heterogeneity, where the VM capacities are not the same, u_{ijk} can be defined in a new way: $\hat{u}_{ijk} = \frac{u_{ijk} \cdot C_{ijk}}{C_{ik}}$ or $\hat{u}_{ijk} = \frac{\overline{L_{ijk}}}{C_{ik}}$.

Like the load balancing methods in [48, 57], RIAL can use a centralized server(s) to collect node load information and conduct load balancing. It can also use a decentralized method as in [24] to conduct the load balancing. In this paper, we focus on how to select VMs and destination PMs to achieve a fast and constant convergence while minimize the adverse effect of VM migration on the cloud services.

4.1.2 Reducing VM Communications between PMs

The VMs belonging to the same customer are likely to communicate with each other much more frequently than with other VMs. Placing VMs with high communication frequency in different PMs will consume considerable network bandwidth. To save bandwidth consumption and hence increase cloud service quality, we try to keep VMs with frequent communication in the same PM. Thus, we try not to select VMs with a high communication rate with local VMs (residing in the same PM) to migrate to other PMs. We use T_{ijpq} to denote the communication rate between V_{ij} and V_{pq}, and use T_{ij} to denote the communication rate of V_{ij} with local VMs:

$$T_{ij} = \sum_{q=1}^{n_i} T_{ijiq} \tag{4.5}$$

Also, we try to choose the destination PM with the highest communication rate with migration VM V_{ij}. We denote the communication rate between V_{ij} and PM P_p as

$$T_{ijp} = \sum_{q=1}^{n_p} T_{ijpq} \tag{4.6}$$

where n_p is the number of VMs in P_p.

4.1.3 Reducing VM Performance Degradation

When a VM is being migrated to another PM, its performance (response time) is degraded [52]. We also aim to minimize the VM performance degradation caused by migrations. We calculate the performance degradation of VM V_{ij} migrating to PM P_p based on a method introduced in [10, 52]:

$$D_{ijp} = d_{ip} \cdot \int_{t}^{t+\frac{M_{ij}}{B_{ip}}} u_{ij}(t)\mathrm{d}t \tag{4.7}$$

where t is the time when migration starts, M_{ij} is the amount of memory used by V_{ij}, B_{ip} is the available network bandwidth, $\frac{M_{ij}}{B_{ip}}$ indicates the time to complete the migration, $u_{ij}(t)$ is the CPU utilization of V_{ij}, and d_{ip} is the migration distance from P_i to P_p. The distance between PMs can be determined by the cloud architecture and the number of switches across the communication path [34, 39].

4.1.4 Problem Statement

In a cloud system, we denote the set of all overload PMs by $\mathcal{O}$ and the set of all lightly loaded PMs by $\mathcal{L}$. Given $\mathcal{O}$ and $\mathcal{L}$, our objective is to select V_{ij} from $P_i \in \mathcal{O}$ and then select the destination $P_p \in \mathcal{L}$ to migrate V_{ij} to in order to eliminate overloaded PMs and meanwhile minimize the number of VM migrations, the total communications between the migration VMs and PMs and the total performance degradation of all migration VMs. We use $\mathcal{S}_i$ to denote the set of selected migration VMs in P_i, and use $|\cdot|$ to represent the size of a set. Then, our problem can be expressed as:

$$\min |\{V_{ij}|V_{ij} \in \mathcal{S}_i,\ P_i \in \mathcal{O}\}| \tag{4.8}$$

$$\min \sum T_{ijp} \tag{4.9}$$

$$\min \sum D_{ijp} \tag{4.10}$$

$$subject\ to:\ u_{ik} \leq \Theta_k,\ \forall\ i,k \tag{4.11}$$

Our problem of VM migration is a variant of the multiple knapsack problem, which is NP-complete [32]. A simpler formulation of our problem has been shown to be NP-complete in [34, 44]. Our problem differs from them mainly in that it minimizes the number of VM migrations. We can construct a special instance of our problem that is similar to them and hence prove that our VM migration problem is NP-complete. We will present a method for solving this problem below.

4.2 The Design of RIAL

Like all previous load balancing methods, RIAL periodically finds overloaded PMs, identifies the VMs in overloaded PMs to migrate out and identifies the destination PMs to migrate the VMs to. In RIAL, each PM P_i periodically checks its utilization for each of its type-k ($k \in K$) resources to see if it is overloaded. We use L and O ($L \cup O = K$) to denote the set of resource types in the PM that are non-overutilized and overutilized, respectively. An overloaded PM triggers VM migration to migrate its VMs to other PMs until its $u_{ik} \leq \Theta_k$ ($k \in K$). Below, we present the methods for selecting VMs to migrate and for selecting destination PMs with the objectives listed in Section 4.1.4.

4.2.1 Selecting VMs to Migrate

We first introduce a method to determine the weight of each type of resource based on resource intensity. We aim to find VMs to migrate out of each overloaded P_i to quickly reduce its workload. If P_i is overutilized in CPU, then we hope to select the VM with the highest CPU utilization in order to quickly relieve P_i's load. Since non-overutilized resources do not overload P_i, we do not need to reduce the utilization of these resources in P_i. Therefore, we also aim to select the VM with the lowest utilization in non-overutilized resources in order to fully utilize resources. To jointly consider these two factors, we determine the weight for each type-k resource according to its overload status in P_i.

To achieve the above mentioned objective, we give overutilized resources relatively higher weights than non-overutilized resources. Among the non-overutilized resources, we assign lower weights to the resources that have higher utilizations in order to more fully utilize resources in the PM. Therefore, the weight for a non-overutilized resource with resource utilization u_{ik} is determined by

$$w_{ik} = 1 - u_{ik}.$$

A resource with utilization zero receives a weight of 1. The weight decreases as the utilization increases. The resource with a utilization closest to the threshold Θ_k (i.e., $u_{ik} < \Theta_k$ and $u_{ik} \approx \Theta_k$) receives the lowest weight $1 - \Theta_k$. Thus, this resource has the lowest probability to be migrate out.

Among the overutilized resources, the resources that have higher utilizations should receive higher weights than those with relatively lower utilizations. For the overutilized resources that have similar but different utilization values, we hope to assign much higher weights to the resources with higher utilizations and assign much lower weights to the resources with lower utilization. That is, we exaggerate the difference between the weights of resources based on the difference between their utilization. Thus, we use a power function with a basic form to determine the weight for an overutilized resource with resource utilization u_{ik}:

$$w_{ik} = \frac{1}{a u_{ik}^{\alpha} + b},$$

where a and b are constant coefficients, and α is an integer exponent. In order to simplify the above equation and at the same time meet the design requirements as discussed previously, we let $\alpha = 1$.

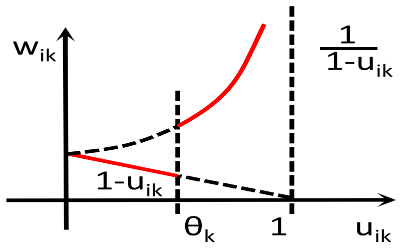


Figure 4.1: Weight vs. utilization.

To satisfy the monotonically increasing property (i.e., higher utilization receives higher weight), we set $a = -1$. Considering that the domain of the function should cover [Θ_k,1) (i.e., for an overutilized resource, $\Theta_k \leq u_{ik} < 1$), so $b = 1$. As a result, the weight given to a resource can be determined by

$$w_{ik} = \begin{cases} \frac{1}{1-u_{ik}}, & \text{if } k \in O, \\ 1 - u_{ik}, & \text{if } k \in L. \end{cases} \tag{4.12}$$

The weight of resource k (w_{ik}) means the priority of migrating this resource out. The function in Equation 4.12 is shown in Figure 4.1. That is, for an overutilized resource $k \in O$ ($u_{ik} \geq \Theta_k$), a higher utilization leads to a higher weight. For a non-overutilized resource $k \in L$ ($u_{ik} < \Theta_k$), a higher utilization leads to a lower weight. Note that $w_{ik} > 1$ for a resource $k \in O$ always has a higher weight than $w_{ik} < 1$ for a resource $k \in L$, which means that overutilized resources always have higher priority to migrate out than underutilized resources. The figure shows that, determining resource weight w_{ik} based on Equ. (4.12) satisfies all the requirements discussed before. For example, when $u_{ik} < \Theta_k$, $w_{ik} = 1 - u_{ik}$ is a decreasing function with a constant slope (left red curve) of -1. When $u_{ik} \geq \Theta_k$, $w_{ik} = \frac{1}{1-u_{ik}}$ is an increasing function with increasing slopes (right red curve). $w_{ik} > 1$ for an overutilized resource ($u_{ik} \geq \Theta_k$) while $w_{ik} < 1$ for a non-overutilized resource ($u_{ik} < \Theta_k$). The resource with a utilization smaller than and close to the threshold has the lowest weight.

We use the Mullti-Criteria Decision Making (MCDM) [54] method to select the VM to migrate. Basically, the MCDM method calculates the weighted distances of all the candidates from the ideal solution, and selects the one with shortest distance. Recall that u_{ijk} is the type-k resource utilization rate of VM V_{ij}. Using the MCDM method, we establish a $|K| \times n_i$ decision matrix D_i for PM P_i with n_i VMs as

$$D_i = \begin{pmatrix} u_{i11} & \cdots & u_{in_i1} \\ \vdots & \ddots & \vdots \\ u_{i1|K|} & \cdots & u_{in_i|K|} \end{pmatrix} \tag{4.13}$$

in which each row represents one type of resource and each column represents each VM in P_i. In the case of heterogenous VM types, we use the normalized VM utilizations and simply replace u_{ijk} with $\hat{u}_{ijk}$ in Equ. (4.13).

We then normalize the decision matrix:

$$X_i = \begin{pmatrix} x_{i11} & \cdots & x_{in_i1} \\ \vdots & \ddots & \vdots \\ x_{i1|K|} & \cdots & x_{in_i|K|} \end{pmatrix} \tag{4.14}$$

where

$$x_{ijk} = \frac{u_{ijk}}{\sqrt{\sum_{j=1}^{n_i} u_{ijk}^2}} \tag{4.15}$$

Next, we determine the ideal migration VM (denoted by R_{VM}) which has the highest usage of overutilized resources and has the lowest usage of non-overloaded resources. That is,

$$R_{VM} = \{r_{i1}, ..., r_{i|K|}\} = \{(\max_j x_{ijk} | k \in O), (\min_j x_{ijk} | k \in L)\}; \tag{4.16}$$

for each type-k resource, if it is overutilized, its r_{ik} is the largest element from $(x_{i1k} \cdots x_{ijk} \cdots x_{in_ik})$ in X_i; otherwise, r_k is the smallest element.

As indicated in Section 4.1.2, we also hope to select the VM with the lowest communication rate to other VMs in the same PM (i.e., T_{ij}) in order to reduce subsequent VM communication cost after migration. Therefore, we set the ideal value of T_{ij} to 0. We then calculate the Euclidean distance of each candidate V_{ij} in P_i with the ideal VM and ideal T_{ij}.

$$l_{ij} = \sqrt{\sum_{k=1}^{|K|} [w_{ik}(x_{ijk} - r_{ik})]^2 + [w_t T_{ij}]^2}, \tag{4.17}$$

where w_t is the weight of the communication rate and it can be adaptively adjusted based on the tradeoff between the convergence speed/cost and the network bandwidth cost for VM communication. The migration VM is the VM with the shortest Euclidean distance (l_{ij}), i.e., the most similar resource utilizations as the ideal VM. After selecting a VM V_{ij}, RIAL checks if V_{ij}'s $u_{ijk} (k \in K)$

is in R_{VM}. If so, RIAL replaces V_{ij}'s u_{ijk} in R_{VM} with the updated value. RIAL then continues to choose the VM with the second shortest l_{ij}. Using the above method, RIAL keeps selecting migration VMs from P_i until P_i is no longer overloaded.

4.2.2 Selecting Destination PMs

When selecting destination PMs to migrate the selected VMs from P_i, we consider resource intensity, VM communication rate and performance degradation as indicated in Section 4.1. We use J to denote the set of lightly loaded PMs. We also use the MCDM method for destination PM selection. We build the $|K| \times |J|$ decision matrix D' as

$$D' = \begin{pmatrix} u_{11} & \cdots & u_{|J|1} \\ \vdots & \ddots & \vdots \\ u_{1|K|} & \cdots & u_{|J||K|} \end{pmatrix} \tag{4.18}$$

in which each row represents one type of resource and each column represents each lightly loaded PM.

We then normalize the decision matrix:

$$X' = \begin{pmatrix} x'_{11} & \cdots & x'_{|J|1} \\ \vdots & \ddots & \vdots \\ x'_{1|K|} & \cdots & x'_{|J||K|} \end{pmatrix} \tag{4.19}$$

where

$$x'_{jk} = \frac{u_{jk}}{\sqrt{\sum_{j=1}^{|J|} u_{jk}^2}} \tag{4.20}$$

Recall that the weight of type-k resource (w_{ik}) represents the priority of migrating this resource out from overloaded PM P_i. Hence, it also indicates the priority of considering available resource in selecting destination PMs. Therefore, we also use these weights for different resources in candidate PMs in order to find the most suitable destination PMs that will not be overloaded by hosting the migration VMs. We represent the ideal destination PM as

$$R'_{PM} = \{r'_1, ..., r'_k, ..., r'_{|K|}\} = \{\min_j x'_{jk} | k \in K\}. \tag{4.21}$$

consisting of the lowest utilization of each resource from the candidate PMs.

When choosing destination PMs, we also hope that the VMs in the selected destination PM P_p have higher communication rate with the migration VM V_{ij} (i.e., T_{ijp}) in order to reduce network bandwidth consumption. Thus, we set the ideal T_{ijp} to be the maximum communication rate between V_{ij} and all candidate PMs, $T_{max} = \max T_{ijp}$ $(p \in J)$. Further, the performance degradation of the migrated VMs should be minimized.

By considering the above three factors, we calculate the Euclidean distance of each candidate PM P_p from the ideal PM.

$$l_{p,ij} = \sqrt{\sum_{k=1}^{|K|}[w_{ik}(x'_{pk} - r'_k)]^2 + [w_t(T_{ijp} - T_{max})]^2 + [w_d D_{ijp}]^2} \tag{4.22}$$

where w_d is the weight of performance degradation consideration that can be adaptively adjusted like w_t. We then select the PM with the lowest $l_{p,ij}$ value as the migration destination of selected VMs. If the selected PM does not have sufficient available resources to hold all VMs, the PM with the second lowest $l_{p,ij}$ is selected using the same method as selecting migration VMs. This process is repeated until the selected PMs can hold all selected migration VMs of P_i. Note that the magnitudes of w_t and w_d should be properly determined based on the practical requirements of the cloud on the tradeoff of the number of VM migrations, bandwidth cost and VM performance degradation. Higher w_t and w_d lead to more VM migrations, while lower w_t generates higher bandwidth cost for VM communications and lower w_d generates higher VM performance degradation. How to determine these magnitudes for an optimal tradeoff is left as our future work.

4.2.3 Parameter Determination

Our load balancing algorithm selects VMs to be migrated out from each overloaded PM and selects the destination PM to host each migrated VM in order to quickly reach the load balanced state in the system (i.e., quick convergence). Equ. (4.17) is used to select VMs that should be migrated out from an overloaded PM considering the weights for resources (w_{ik}) and for communication cost (w_t). Equ. (4.22) is used to select the destination PM considering the weights for resources (w_{ik}), for communication cost (w_t) and for performance degradation due to migration (w_d). The values of these weight parameters have a direct impact on the performance of our proposed load balancing algorithm. In this section, we present how to determine these parameters to achieve better performance.

As indicated in Equ. (4.17), in order to calculate the Euclidean distance of candidate VM V_{ij} when selecting a VM to migrate, we must determine w_{ik} and w_t. Recall that w_{ik} is determined by Formula 4.12. Then, we must first determine the value of w_t before we calculate the Euclidean distance The importance of considering the communication cost (w_t) should not overtake the importance of relieving overutilized resources (w_{ik}), which is the primary objective of our load balancing algorithm. A high w_t may lead to the failure of mitigating the load of overloaded resources, while a low w_t may lead to the unawareness of the communication rate in migration VM selection. Thus, in load balancing, we give the highest priority to offloading the excess load in an overloaded PM, and paying as much attention as possible to communication rates between VMs in order to maximize the VM communications within a PM.

Therefore, we determine w_t so that one of the VMs that are the most similar to the ideal VM without considering the communication cost is selected and at the same time w_t is maximized. Suppose V_{is} is the selected VM in the VM selection algorithm without considering the communication rate of the VMs (i.e., $w_t = 0$ in Equ. (4.17)):

$$V_{is} = \arg\min_{V_{ij}} l_{V_{ij}} \tag{4.23}$$

and

$$l_{V_{ij}} = \sqrt{\sum_{k=1}^{|K|} [w_{ik}(x_{ijk} - r_{ik})]^2} \tag{4.24}$$

A VM V_{ij} is regarded as one of the most similar VMs to the ideal VM, if

$$l_{V_{im}} \leq \; l_{V_{is}} + \delta_v, \tag{4.25}$$

where δ_v is a positive constant. By selecting a similar VM rather than the most similar VM without considering the communication cost (i.e., V_{is}), we slightly sacrifice the priority of offloading the excess load to reducing communication cost. The value of δ_v determines the extent of the sacrifice. We denote the set of VMs that satisfy Equ. (4.25) as $\mathcal{S}_v$. With our determined w_t, the VM in $\mathcal{S}_v$ that can maximally reduce the communication cost will be selected to migrate out. In the following, we explain how to determine the value of w_t based on δ_v and w_{ik} for the aforementioned objective.

The problem of finding the maximum w_t with the constrain of δ_v can be expressed as follows. Given the normalized decision matrix X_i of P_i and the ideal migration VM R_{VM}, the problem is to maximize w_t, subject to:

$$l_{im} \leq l_{ij}, \quad \forall\, V_{im} \in \mathcal{S}_v,\ V_{ij} \notin \mathcal{S}_v \tag{4.26}$$

where l_{im} and l_{ij} are calculated by Equ. (4.17). It means that V_{im} will always be selected to migrate out even with the maximized w_t. It is to ensure that the selected VM without considering the communication rate (Equ. (4.25)) will not change when taking into account the communication rate (Equ. (4.26)).

In order to solve this problem, we can combine Equ. (4.17) and Equ. (4.26), and then derive Equ. (4.27) below:

$$\sum_{k=1}^{|K|} w_{ik}^2[(x_{imk} - r_{ik})^2 - (x_{ijk} - r_{ik})^2] \leq \ w_t^2(T_{ij}^2 - T_{im}^2) \tag{4.27}$$

Since x_{ijk} is known, we can find x_{isk} and hence x_{imk} based on Equ. (4.23) and Equ. (4.25). Since T_{ij} and T_{im} are also known, we can solve Equ. (4.27). Equ. (4.27) can be solved based on the values of $T_{ij}^2 - T_{im}^2$ and $(x_{imk} - r_{ik})^2 - (x_{ijk} - r_{ik})^2$, which can be either positive or negative. We ignore useless constraints of these two values that are derived from the condition in Equ. (4.27). For example, if $T_{ij}^2 - T_{im}^2 >$ and $(x_{imk} - r_{ik})^2 - (x_{ijk} - r_{ik})^2 < 0$, we derive that w_t is greater than a negative value, which is always true and thus useless. Then, we derived that when $(x_{imk} - r_{ik})^2 - (x_{ijk} - r_{ik})^2 < 0$ and $T_{ij}^2 - T_{im}^2 < 0$,

$$w_t \leq \ \sqrt{\frac{\sum_{k=1}^{|K|} w_{ik}^2[(x_{imk} - r_{ik})^2 - (x_{ijk} - r_{ik})^2]}{T_{ij}^2 - T_{im}^2}} \tag{4.28}$$

Finally, we solve Equ. (4.28) and select the maximum value for w_t.

Solving Equ. (4.28) involves complicated calculations including determining weights for resources based on Equ. (4.12), finding V_{is} based on Equ. (4.23) and solving Equ. (4.27). In the following, we try to simplify the process of determining w_t. Since we consider mitigating the load of the overutilized resources and at the same time maximizing the VM communications within a PM (by selecting the VM that has minimal communications with the co-locating VMs to migrate out), we can further loose Equ. (4.28) to simplify the process of w_t determination. Specifically, we only consider the most sensitive weight, which is defined as the minimum weight of the overutilized resources:

$$w_m = \min\{w_k | k \in O\}. \tag{4.29}$$

Because w_m is the minimum weight of the overutilized resources, by ensuring that w_t does not overtake w_m, we can satisfy the condition that w_t does not overtake all the weights of the overutilized resources with a high probability. We then determine w_t based on w_m in order to prevent w_t from overtaking the minimum weight of the overloaded resources. We aim to maximize w_t while guaranteeing that the most similar VM should be selected. To simplify the process, we can only consider the VM that is the most similar to the ideal VM and the VM that is the second similar. Because every VM together with the VM that is the most similar to the ideal VM can specify a range for the value of w_t, and the constraints placed on w_t by other VMs are relatively looser compared to the second similar VM. Suppose there is only one resource overutilized, and the weight is w_m; the VM (VM$_0$) which is the most similar to the ideal VM has normalized utilization x_0 and communication rate T_0; the VM (VM$_1$) which is the second similar has utilization x_1 and communication rate T_1. We use a linear function $l = w_m x + w_t T$ to represent Equ. (4.17). Then, the above problem can be expressed as: to maximize w_t, subject to

$$w_m x_0 + w_t T_0 \leq w_m x_1 + w_t T_1, \quad x_0, x_1, T_0, T_1 > 0. \tag{4.30}$$

Finally, we can find the maximum w_t as

$$w_t = -\frac{x_0 - x_1}{T_0 - T_1} w_m \tag{4.31}$$

In order to further make the determination of w_t easier, we derive a constant weight. As a rule of thumb, w_t is greater than 1, which is the maximal weight for a non-overutilized resource, because considering communication rate is more important than considering the non-overutilized resources. Also, weight w_t should be lower than the weight of overutilized resources, because mitigating the load on overload resources has the highest priority. That is, $w_t < \frac{1}{1-\Theta_k}$ based on Equ. (4.12). For example, for a threshold $\Theta_k = 0.75$, the weight for communication rate $w_t < 4$. Then, w_t can be set to constant 3, which is the maximum value that satisfies < 4. In our experiment in Section 4.3, with $\Theta_k = 0.75$, we set a constant to w_t, i.e., $w_t = 3$.

Next, we discuss how to determine the weight for communication cost (w_t) and for performance degradation due to migration (w_d) in Equ. (4.22) for the destination PM for a migrated VM. Different from VM selection, here, we need to determine two parameters. However, a formulated problem can only be used for optimizing one object. We then combine w_t and w_d to one optimization object. Then, similar to what has been discussed before, we can derive both w_t and w_d together for PM selection by altering the object function of the aforementioned problem for VM selection. That is, we place equal importance on the two weights since both weights are important (i.e., $w_d = w_t$) and try to maximize w_d. Suppose P_s is the selected destination PM in PM selection algorithm without considering the communication rate or performance degradation of the VMs (i.e., $w_t = 0$ and $w_d = 0$ in Equ. (4.22)):

$$P_s = \arg\min_{P_p} l_{P_p} \tag{4.32}$$

and

$$l_{P_p} = \sqrt{\sum_{k=1}^{|K|}[w_{ik}(x'_{pk} - r'_k)]^2} \tag{4.33}$$

A PM P_p is regarded as one of the most similar PMs to the ideal PM, if

$$l_{P_m} \leq \ l_{P_s} + \delta_p, \tag{4.34}$$

where δ_p is a positive constant. Similarly, δ_p represents the extent of the sacrifice of the priority of offloading overloaded resource to reducing communication cost and performance degradation due to VM migrations. We denote the set of PMs that satisfy Equ. (4.34) as $\mathcal{S}_p$. Then, the problem can be transformed to maximize w_t, subject to:

$$l_{m,ij} \leq l_{p,ij}, \quad \forall\ P_m \in \mathcal{S}_p,\ P_p \notin \mathcal{S}_p \tag{4.35}$$

Similarly, we can derive

$$w_t \leq \ \sqrt{\frac{\sum_{k=1}^{|K|} w_{ik}^2[(x'_{mk} - r'_k)^2 - (x'_{pk} - r'_k)^2]}{[(T_{ijp} - T_{max})^2 - (T_{imp} - T_{max})^2] + (D_{ijp}^2 - D_{imp}^2)}} \tag{4.36}$$

For more simplified w_t and w_d, we adopt $w_t = 3$ and $w_d = 3$ as the constant values for these weights. Similar as previous, for a threshold $\Theta_k = 0.75$, the weight for communication rate $w_t < 4$, the weight for performance degradation $w_d < 4$. Then, both w_t and w_d can be set to constant 3. We will show the experiment results with varying w_t and w_d in Section 4.3.

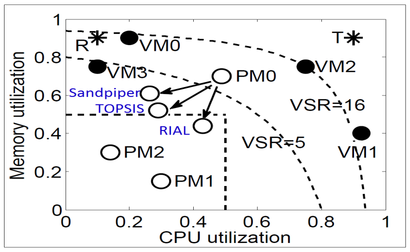


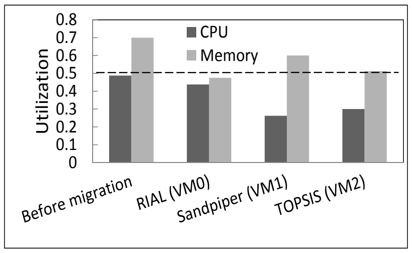
Figure 4.2: VM and PM selection process.

4.2.4 Performance Comparison Analysis

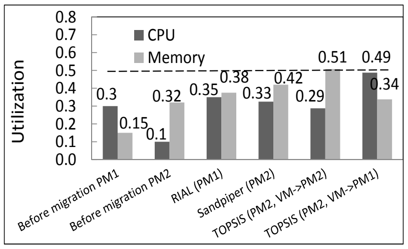
Compared to Sandpiper [57] and TOPSIS [48], RIAL produces fewer migrations. Because RIAL determines the resource weight based on resource intensity, it can quickly relieve overloaded PMs by migrating out fewer VMs with high usage of high-intensity resources. Also, the migration VMs have low usage of low-intensity resources, which helps fully utilize resources and avoids overloading other PMs. In addition, the migration destination has a lower probability of being overloaded subsequently as it has sufficient capacity to handle the high-intensity resources. Finally, RIAL leads to fewer VM migrations in a long term.

We use an example with 3 PMs (PM0, PM1, PM2) to demonstrate the advantage of RIAL. In practice, the overloaded threshold should be close to 1. To make the example simple with few VMs, we set the threshold to 0.5, and only consider the CPU and memory resources. We assume that PM0 has 4 VMs (VM0, VM1, VM2, VM3) with the same capacity and PM0's capacity is four times of the VM's. PMs have the same capacity. As in [48], the weight of CPU and memory in TOPSIS is 9 and 4, respectively. Figure 4.2 shows the CPU and memory utilizations of the 4 VMs, VM0(0.2,0.9), VM1(0.9,0.4), VM2(0.75,0.75), VM3(0.1,0.75) and the 3 PMs, PM0(0.49,0.7), PM1(0.3,0.15), PM2(0.1,0.32). PM0 is overloaded in memory resource usage since 0.7>0.5.

Sandpiper attempts to migrate the VM with maximum VSR=$volume/size$, where $volume$= $(1/(1-u_{cpu}))*(1/(1-u_{net}))*(1/(1-u_{mem}))$. Based on this formula, we draw two dash curves in Figure 4.2 to indicate the points whose VSR equals to 5 and 16, respectively. We see that among the 4 VMs, VM1 located beyond the curve of VSR=16 has the highest VSR. Therefore, Sandpiper selects VM1 to migrate out of PM1. TOPSIS first determines its ideal VM (T* in Figure 4.2) with the maximum CPU and memory utilizations from the 4 candidate VMs (i.e., (0.9, 0.9)), then compares the weighted distances of the 4 VMs to the ideal VM, and finally chooses VM2 that has



(a) Utilizations of PM0 with different selected migration VMs.



(b) Utilizations of different selected destination PMs.

Figure 4.3: Advantage of RIAL in reducing migrations.

the shortest distance. In RIAL, according to Equ. (4.16), the CPU and memory utilizations of the ideal VM (R* in Figure 4.2) are 0.1 and 0.9. Base on Equ. (4.12), the weights for memory and CPU are 3.33 and 0.51, respectively. Unlike TOPSIS, RIAL gives a weight to CPU smaller than memory, since CPU is not so intensively used as memory. RIAL finally chooses VM0 which has the shortest weighted distance to the ideal VM.

Table 4.1: Number of migrations needed for load balance

	Sandpiper	TOPSIS	RIAL
# selected migration VMs	2	2	1
# of overload destination PMs after VM migrations	0	1	0
Total # of migrations	2	3	1

Figure 4.3(a) shows the CPU and memory utilizations of PM0 before VM migration and after migrating VM0, VM1 and VM2 by RIAL, Sandpiper and TOPSIS, respectively. The arrows in Figure 4.2 indicate the resource utilizations of PM0 after migration in each method, respectively. We see that neither migrating VM1 (by Sandpiper) nor migrating VM2 (by TOPSIS) can eliminate memory overload in PM0. Hence, these two methods require another migration. RIAL reduces both CPU and memory utilizations below the threshold.

For destination PM selection, PM1(0.3,0.15) and PM2(0.1,0.32) are two candidates for the VM from PM0. Sandpiper selects the PM that has the least *volume* as the destination, which is PM2. TOPSIS determines the ideal PM with the least CPU and memory utilization of all candidate PMs (i.e., (0.15, 0.1)), and selects the one with the shortest weighted distance to the ideal PM, which is PM2. However, after migrating VM2 to PM2, the memory utilization of PM2 increases to 0.51, higher than the threshold. Then, TOPSIS has to execute another migration and chooses PM1 to migrate VM2 to. RIAL determines the same ideal PM as TOPSIS, but assigns higher weight to

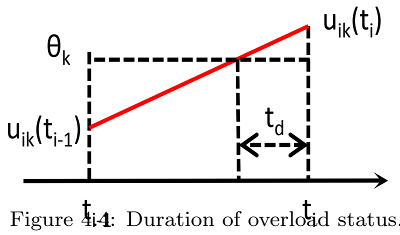


Figure 4.4: Duration of overload status.

memory, so it chooses PM1 as the destination that has the shortest weighted distance.

Figure 4.3(b) shows the CPU and memory utilizations of the destination PMs before and after migrations. TOPSIS overloads the destination PM2 in memory and needs another migration (VM2→PM1) to relieve its memory load. Though all three methods finally eliminate the memory overload in PM0, RIAL generates a more balanced state since resource utilizations after balancing are relatively lower than those in Sandpiper and TOPSIS, which reduces the probability of overloading PMs, and hence helps maintain the system load balanced state for a longer time period.

Table 4.1 lists the number of selected VMs to relieve overloaded PM0, the number of overloaded destination PMs after the VM migrations, and the total number of migrations to achieve the load balanced state in one load balancing operation. We see that RIAL generates the least number of migrations due to its advantages mentioned previously.

4.2.5 When to Trigger VM Migration

In today's cloud datacenter, VMs may generate transient workload spikes in PMs [50], which are sharp rises in the resource utilization that immediately followed by decreases. A PM may become overloaded (i.e., $u_{ik} > \Theta_k$) during a spike, and becomes underloaded after the spike. In this case, simply triggering the VM migration upon the observation that the resource utilization of a PM exceeds a threshold (i.e., $u_{ik} > \Theta_k$) generates unnecessary VM migration operations and overhead, and also fail to fully utilize resources. The occurrence of $u_{ik} > \Theta_k$ at a certain time does not necessarily mean that the resource utilization of this PM will continually exceed the threshold for a certain period of time in the future. Furthermore, Xen live migration is CPU intensive, which may degrade the performance of both the source and destination PMs. Without sufficient resource, a VM migration will take a long time to finish, which will increase the service latency of tasks running on the PMs and may result in SLO violations. Therefore, we need to avoid triggering unnecessary migrations.

For this purpose, we specify that a migration is triggered only if the overload status of the PM (i.e., $u_{ik} > \Theta_k$) will last continuously for at least T_{t_d} time units, where T_{t_d} is the duration of overload status of the PM from time t_{i-1} to time t_i. The value of T_{t_d} determines the balance between offloading the overload resource and avoiding migrations due to transient overload status, and it can be tuned by the cloud provider. We can either set T_{t_d} to a constant time or make T_{t_d} a function of the VM migration time based on the requirement of guaranteeing SLO. We will first demonstrate when to trigger VM migration based on T_{t_d} and then discuss how to determine T_{t_d} so that the number of migrations is minimized without increasing the number of SLO violations.

Suppose the monitoring interval is Δt time units. That is, a PM checks its resource utilization every Δt time units, i.e., $\Delta t = t_i - t_{i-1}$, where t_i is current time and t_{i-1} is the time of last monitoring. As in [13], we assume that the resource utilization linearly increases from $u_{ik}(t_{i-1})$ to $u_{ik}(t_i)$ during time interval Δ t. As shown in Figure 4.4, suppose a PM detects that its resource utilization exceeds the threshold ($u_{ik}(t_i) > \Theta_k$) at time t_i. According to historical record, it has resource utilization $u_{ik}(t_{i-1})$ at time t_{i-1}. The duration of overload status of the PM, t_d, can be calculated by

$$t_d = \frac{u_{ik}(t_i) - \Theta_k}{u_{ik}(t_i) - u_{ik}(t_{i-1})}\Delta t \tag{4.37}$$

Then, VM migration will be triggered if and only if $t_d > T_{t_d}$.

We then discuss how to determine the value of T_{t_d} with the consideration of the SLO requirement. In this paper, we define SLO as the requirement that ε (in percentage) of resource demands of a VM must be satisfied during its lifetime [53]. We use t_s to denote the start time of a VM, and use t_v to denote the cumulated time that this VM has experienced resource overload since t_s until last monitoring time t_{i-1}. Considering that the time to complete VM migration is $\frac{M_{ij}}{B_{ip}}$, the total time that the VM will experience overload before migration is completed equals $T_{t_d} + \frac{M_{ij}}{B_{ip}} + t_v$. We delay the migration as much as possible by fully taking advantage of SLO that allows $1 - \varepsilon$ violations. That is, if current time t_i is the migration start time of a VM in the PM, the VM should satisfy:

$$\frac{T_{t_d} + \frac{M_{ij}}{B_{ip}} + t_v}{t_i - t_s + \frac{M_{ij}}{B_{ip}}} = \varepsilon \tag{4.38}$$

Finally, we can get $T_{t_d} = \varepsilon(t_i - t_s + \frac{M_{ij}}{B_{ip}}) - \frac{M_{ij}}{B_{ip}} - t_v$. Therefore, in order to determine T_{t_d}, we need to record the start time t_s of each VM in the PM, and have the variable t_v to keep track of the cumulated SLO violation time of each VM.

4.2.6 Decentralized Destination PM Selection

Recall that the VM selection is conducted in each PM in a distributed manner, but the destination PM is selected in a central server because it needs to be chosen from all PMs. The centralized approach for destination PM selection is not efficient for a large scale cloud datacenter, because the amount of information required for this algorithm increases and may overburden the centralized server. In order to relieve the load of the centralized server, we develop a distributed version of the PM selection algorithm. The topology of a cloud datacenter can be abstracted by a graph with its nodes indicating PMs and switches and edges indicating physical links that connect PMs and switches. In this pater, we focus on tree-like topologies [3, 22], which are typical topologies of today's datacenters. As shown in Figure 4.5, we partition all the nodes in cloud datacenter into small clusters and each cluster consists of the nodes in one rack. Then, the load balancing is conducted within each cluster. That is, the VMs are migrated between physically close nodes. This way, the performance degradation due to VM migration can be reduced.

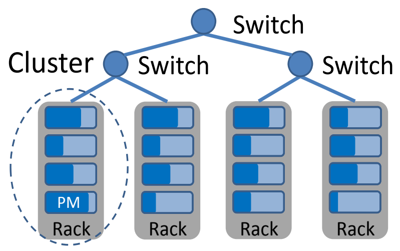
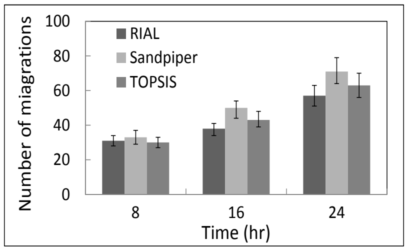
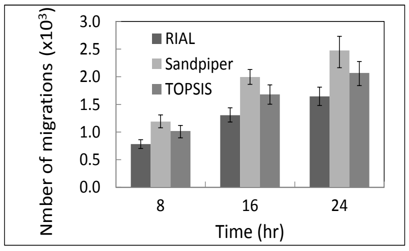


Figure 4.5: Datacenter network.

Within each cluster, the nodes select the cluster master, who is responsible for selecting PMs for VM migrations in this cluster. This selected PM should not be overutilized and at the same time has the least probability to be selected as the destination PM, that is, it has the highest resource utilization. Unlike the centralized algorithm, in which a centralized server collects the information required in Equ. (4.22) from all the PMs in the datacenter, in the decentralized algorithm, the information is sent from every PM to its cluster master. For example, every PM in the cluster reports its status (i.e., resource utilization, communication rate with other PMs in this cluster) periodically (i.e., 5 minutes). The VM selection is conducted distributively in each PM. When a PM detects that it is overloaded, it selects its migration VMs and submits VM migration requests to its cluster master. Upon receiving the VM migration requests from the PMs, the cluster master then determines the ideal destination PM in its cluster, and selects PMs for the migration VMs based on Equ. (4.22). By limiting the PM selection within a small cluster (as opposed to the whole datacenter), we can increase the scalability of the PM selection algorithm. We will compare the the distributed algorithm and the centralized algorithm in Section 4.3.



(a) 100 PMs and 250 VMs

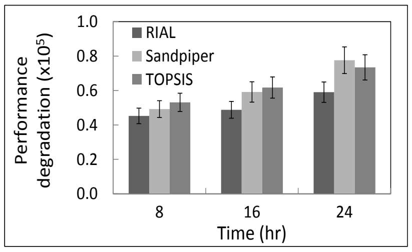


(b) 1000 PMs and 5000 VMs

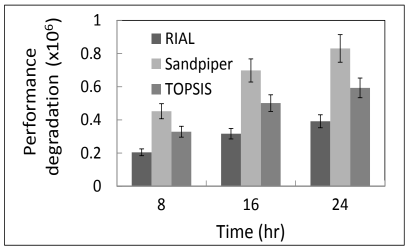
Figure 4.6: Total number of VM migrations.

4.3 Performance Evaluation

We used the CloudSim [13] simulator and our deployed small-scale real-world testbed to evaluate the performance of RIAL in comparison to Sandpiper [57] and TOPSIS [48]. We used the real workload trace available in CloudSim to generate each VM's CPU resource consumption [10,37]. To simulate memory and bandwidth usage, as in [44], we generated 5 different groups of (mean, variance range) for resource utilization, (0.2,0.05),(0.2,0.15),(0.3,0.05),(0.6,0.10),(0.6,0.15), and set each VM's memory/bandwidth utilization to a value generated by a randomly chosen group. Each PM has 1GHz 2-core CPU, 1536MB memory, and 1GB/s network bandwidth. Each VM has 500Hz CPU, 512MB memory, and 100Mbit/s bandwidth. With our experiment settings, the bandwidth consumption will not overload PMs due to their high network bandwidth. In CloudSim, we conducted experiments for two cloud scales. In the small scale experiment, we simulated 250 VMs running on 100 PMs. In the large scale experiment, we simulated 5000 VMs running on 1000 PMs. We generated a tree-like topology to connect the PMs, and measured the transmission delay between PMs based on the number of switches between them [34]. At the beginning of experiments, we randomly and evenly mapped the VMs to PMs. The overload threshold was set to 0.75. The weights for different resource are the same for Sandpiper or set to predefined ratio (e.g., 9:4 for CPU:MEM) as adopted in their papers. The load balancing algorithm was executed every 5 minutes. As in [44], we generated a random graph $G(n, p = 0.3)$ to simulate the VM communication topology, where n is the number of VMs and p is the probability that a VM communicates with another VM. The weight of each edge was randomly selected from [0,1] to represent the communication rate between two VMs. Unless otherwise specified, we repeated each test 20 times with a 24 hour trace and recorded the median, the 90th and 10th percentiles of the results.



(a) 100 PMs and 250 VMs



(b) 1000 PMs and 5000 VMs

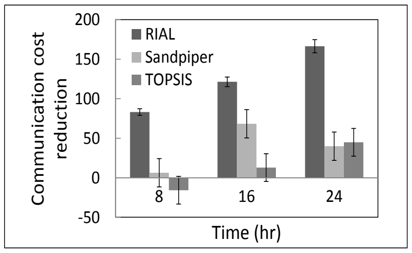
Figure 4.7: Total VM performance degradation.

4.3.1 The Number of Migrations

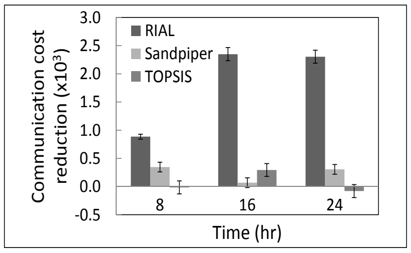
Figure 4.6(a) and Figure 4.6(b) show the median, 10th percentile and 90th percentile of the total number of VM migrations by the time $t = 8h, 16h, 24h$ of the three methods in the small-scale and large-scale tests, respectively. We see that RIAL generates fewer migrations than Sandpiper and TOPSIS. Since RIAL considers resource intensity of different resources, it migrates fewer VMs from a PM to relieve its extra load. Also, RIAL proactively avoids overloading the destination PMs in the future. Thus, it keeps the system in a balanced state for a relatively longer period of time, resulting in fewer VM migrations than Sandpiper and TOPSIS within the same period of time. We also see that TOPSIS produces fewer VM migrations than Sandpiper because TOPSIS gives different weights to different resources while Sandpiper treats different resource equally. Additionally, we see that the three methods exhibit similar variances due to the initial random VM assignment to PMs.

4.3.2 VM Performance Degradation due to Migrations

We measured the total performance degradation of all migration VMs based on Equ. (4.7). Figure 4.7(a) and Figure 4.7(b) show the median, 90th and 10th percentiles of the total performance degradation (Formula (4.7)) in the small-scale and large-scale tests, respectively. We see that the

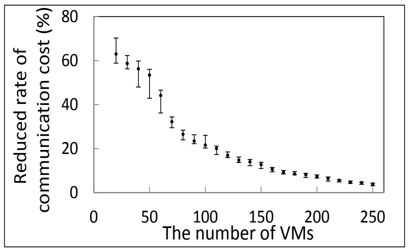


(a) 100 PMs and 250 VMs

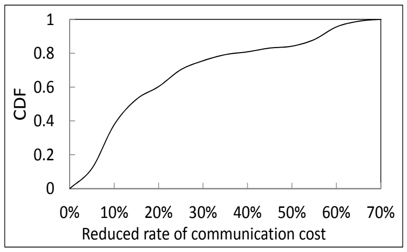


(b) 1000 PMs and 5000 VMs

Figure 4.8: Total VM communication cost reduction.



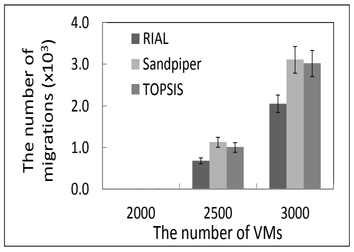
(a) Reduced rate of communication cost



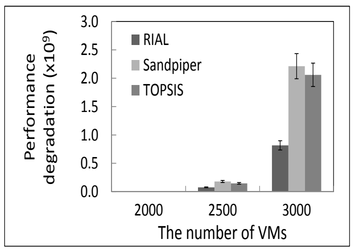
(b) CDF over all experiments

Figure 4.9: Communication cost reduction of RIAL over Sandpiper/TOPSIS.

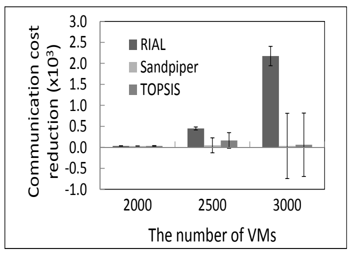
total performance degradation of RIAL is lower than those of TOPSIS and Sandpiper in both small and large scale tests. This is caused by the distinguishing features of RIAL. First, RIAL triggers fewer VM migrations. Second, RIAL tries to minimize performance degradation in destination PM selection. Third, RIAL chooses VMs with lower utilizations of the non-intensive resources. TOPSIS generates lower performance degradation than Sandpiper because it generates fewer VM migrations as shown in Figure 4.6. We also see that in both the small-scale and large-scale tests, the performance degradation variance of the three methods follows RIAL<TOPSIS<Sandpiper though the difference is small in the small-scale test.



(a) The number of VM migrations

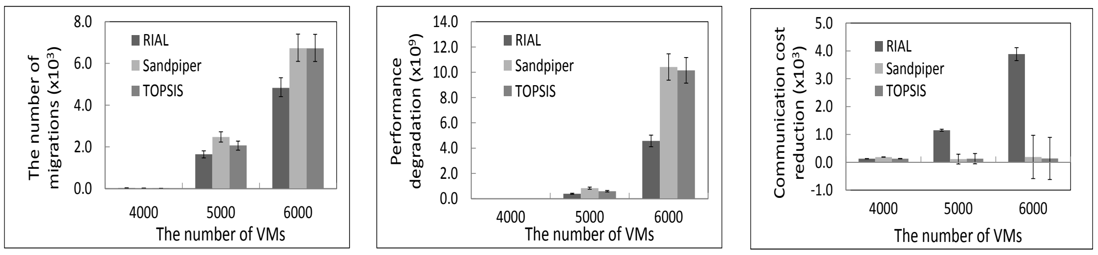


(b) Performance degradation



(c) Communication cost reduction

Figure 4.10: Performance with varying VM to PM ratio (500 PMs).



(a) The number of VM migrations (b) Performance degradation (c) Communication cost reduction

Figure 4.11: Performance with varying VM to PM ratio (1000 PMs).

4.3.3 VM Communication Cost Reduction

The communication cost between a pair of VMs was measured by the product of their communication rate and transmission delay. We calculated the *communication cost reduction* by subtracting the total communication cost observed at a certain time point from the initial total communication cost of all VMs. Figure 4.8(a) and Figure 4.8(b) show the median, the 90th and 10th percentiles of total communication cost reduction at different time points in the small-scale and large-scale tests, respectively. We see that RIAL's migrations reduce much more communication cost than TOPSIS and Sandpiper, which may even increase the communication cost by migrations (shown by the negative results). RIAL exhibits smaller variance because RIAL tries to reduce VM communication rate between PMs caused by VM migration, while the other two methods do not consider it.

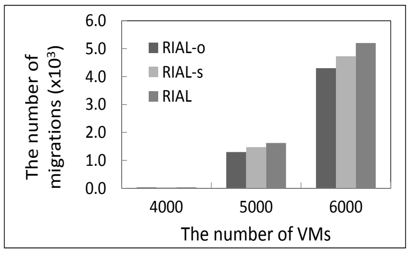
We then directly compare the communication costs after the migrations between different methods. We measured the communication costs of RIAL (x) and Sandpiper/TOPSIS (y) at the end of simulation and calculated the *reduced rate of communication cost* by $(y - x)/y$. We varied the number of VMs from 20 to 250 with an increment of 10, and mapped the VMs to 50 PMs. Each experiment is run for 30 times. As the reduced rates of RIAL over Sandpiper and TOPSIS are similar, we only show one result to make the figures clear.

Figure 4.9(a) shows the median, 10th percentile and 90th percentile of the reduced rate of communication cost with different numbers of VMs. We see that a smaller number of VMs lead to higher reduced rate of communication cost, which implies that RIAL can reduce more communication cost with fewer VMs relative to PMs. This is due to the fact that fewer VMs lead to fewer overloaded PMs hence more PM choices for a VM migration, which helps RIAL reduce more communication costs. Figure 4.9(b) plots the cumulative distribution function (CDF) of all 30*24 experiments versus the reduced rate of communication cost. We see that RIAL consistently outperforms Sandpiper and TOPSIS with lower communication cost in all experiments, and decreases the communication cost by up to 70%.

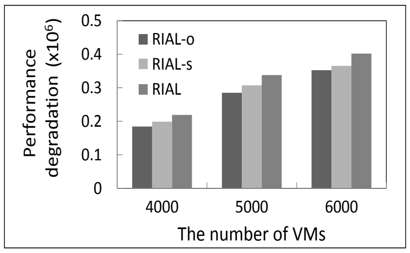
4.3.4 Performance of Varying Number of VMs and PMs

We then study the impact of different ratios of the number of VMs to the number of PMs on performance. Accordingly, we conducted two sets of tests. One test has 500 PMs with the number of VM varying from 2000 to 3000, and the other test has 1000 PMs with the number of VM varying from 4000 to 6000.

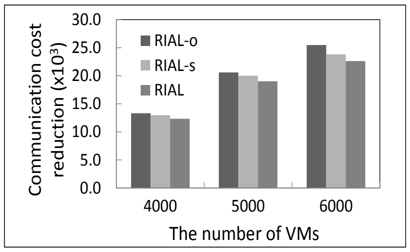
Figure 4.10(a) and Figure 4.11(a) show the median, 10th percentile and 90th percentile of the total number of migrations in the two tests, respectively. As the number of VMs increases, the total load on the cloud increases, resulting in more overloaded PMs and hence more VM migrations. When the number of VMs is 1000, the resource requests by VMs in the cloud is not intensive and only a few migrations are needed. When there are more VMs, the result of number of VM migrations follows RIAL<TOPSIS<Sandpiper, which is consistent with Figure 4.6 due to the same reasons.



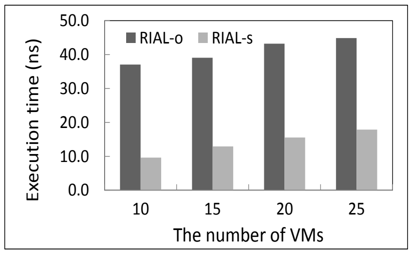
(a) The number of VM migrations



(b) Performance degradation



(c) Communication cost reduction

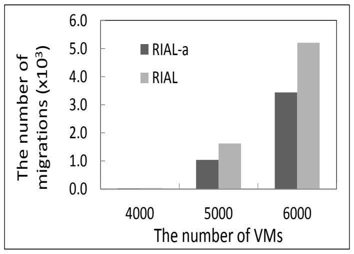


(d) Execution time

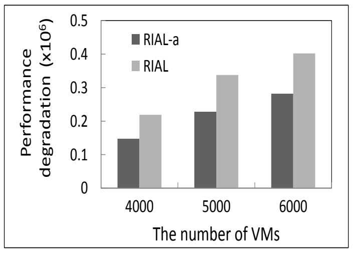
Figure 4.12: Performance of weight determination algorithms with varying VM to PM ratio (1000 PMs).

Figure 4.10(b) and Figure 4.11(b) show the results of the total VM performance degradation in the two tests, respectively. As the number of VM increases, the performance degradation increases in each method, mainly because of more triggered VM migrations. RIAL generates lower performance degradation than Sandpiper and TOPSIS, especially with a higher number of VMs. We also see that the relative performance on the median, 10th percentile and 90th percentile between the three methods is aligned with that in Figure 4.7 due to the same reasons.

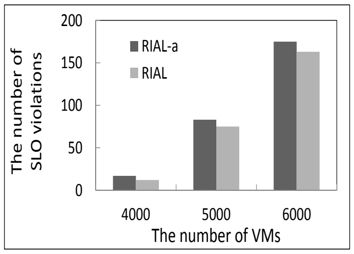
Figure 4.10(c) and Figure 4.11(c) show the results of the total communication cost reduction in the two tests, respectively. When the VM number is small, there is only a few VM migrations, resulting in small cost reduction and small variance for all methods. As the number of VMs grows, RIAL achieves a higher cost reduction than Sandpiper and TOPSIS. Also, RIAL has much smaller variance than Sandpiper and TOPSIS as the error bars indicate. Both Sandpiper and TOPSIS performs similarly since neither of them considers the VM communications when selecting VMs and PMs. The relative performance between the three methods is consistent with that in Figure 4.8 due to the same reasons.



(a) The number of VM migrations.



(b) Performance degradation.



(c) The number of SLO violations.

Figure 4.13: Performance of migration triggering algorithm.

Comparing Figure 4.10 and Figure 4.11, we see that the results in Figure 4.11 have higher absolute values than those in Figure 4.10 because the workload and the scale of the cloud are doubled. We can conclude from Figure 4.10 and Figure 4.11 that RIAL outperforms Sandpiper and TOPSIS under varying ratios of the number of VMs to PMs in terms of the number of VM migrations, VM performance degradation and communication cost.

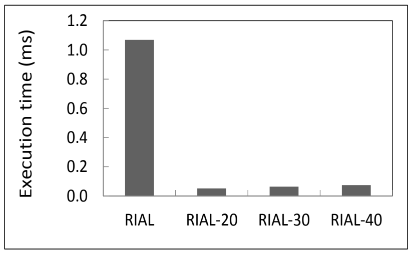
4.3.5 Performance of Weight Determination Algorithms

We study the performance of different weight determination algorithms introduced in Section 4.2.3. In the following sections, we adopt the same setting for the large scale (1000 PMs) and vary the number of VMs from 4000 to 6000 with an increment of 2000 at each step, unless otherwise specified. We use RIAL-o to denote the optimal weight determination algorithm based on Equ.

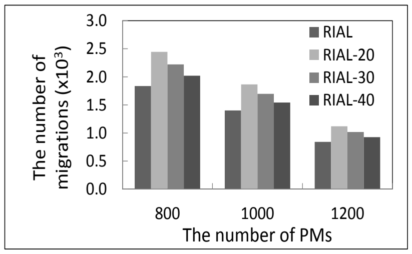
(4.28) and Equ. (4.36), use RIAL-s to denote the simplified algorithm based on Equ. (4.31), and use RIAL to denote the algorithm with constant weights (i.e., $w_t = 3$, $w_d = 3$). Figure 4.12(a) shows the number of VM migrations, which follows RIAL-o<RIAL-s<RIAL. This is because RIAL-o guarantees that the weights of overutilized resource are not overtaken by the weights for communication rate and performance degradation, while RIAL-s and RIAL cannot. Therefore, RIAL-o needs fewer migrations to offload extra load of overutilized resources. RIAL-s outperforms RIAL because RIAL-s determines w_t and w_d based on the weights for resources while RIAL uses constant w_t and w_d, which may make w_t and w_d overtake the resource weights. The number of migrations increases with the number of VMs because more VMs imposes more workload on the same number of PMs (i.e., 1000 PMs). Figure 4.12(b) shows the performance degradation, which follows RIAL-o<RIAL-s<RIAL because fewer migrations lead to less performance degradation. The performance degradation increases with the number of VMs due to the same reason as Figure 4.12(a). Figure 4.12(c) shows the communication cost reduction, which follows RIAL-o>RIAL-s>RIAL. This is because the amount of sacrifice on the priority of offloading the excess load to reducing communication cost follows RIAL-o<RIAL-s<RIAL in weight determination. The communication cost reduction increases with the number of VMs due to the same reason mentioned before. We also measured the execution time of the wight determination algorithms by varying the number of VMs in a PM from 10 to 25 with an increment of 5 at each step. Figure 4.12(d) shows the execution time of the different algorithms with different number of VMs in a PM. We see that RIAL-s is faster than RIAL-o due to the simpleness of Equ. (4.31) compared to Equ. (4.28). We do not present RIAL here because it has zero execution time (constant complexity). This result confirms the feasibility of RIAL-s as it can achieve similar performance as RIAL-o while consuming less time.

4.3.6 Performance of Migration Triggering Algorithm

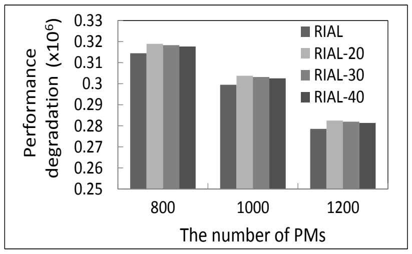
We use RIAL-a to denote RIAL that avoids unnecessary migrations using the migration triggering policy. We set $\varepsilon = 0.95$ and determine T_{t_d} based on Equ. (4.38). The number of SLO violations is the number of VMs that have experienced overload status for a duration more than 1-ε percent of their lifetimes. Figure 4.13(a) shows the number of VM migrations. We see that RIAL-a triggers a fewer number of VM migrations than RIAL since it avoids unnecessary VM migrations and meanwhile avoids violating SLO requirements. The number of VM migrations increases as the number of VMs increases since more VMs aggravate the load in the datacenter. Figure 4.13(b) shows the total performance degradation. We see that the total performance degradation of RIAL-a is lower than RIAL. This is mainly because that RIAL-a avoids unnecessary VM migrations and triggers fewer VM migrations. The performance degradation increases with the number of VMs since move VMs generate more workload and hence more VM migrations. Figure 4.13(c) shows the number of SLO violations. We see that RIAL-a produces a similar number of SLO violations as RIAL although RIAL-a does not immediately trigger VM migration upon detecting $u_{ik} > \Theta_k$. Also, the number of SLO violations increases with the number of VMs due to higher workloads on PMs. This result confirms that triggering VM migration only when the overload status of a PM lasts continuously for at least T_{t_d} time can improve the performance of RIAL without significantly affecting SLO.



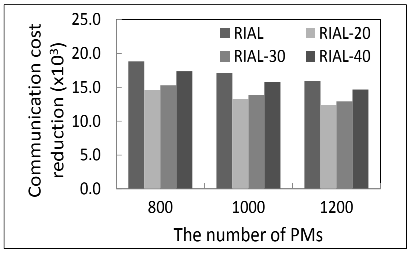
(a) Execution time



(b) The number of VM migrations



(c) Performance degradation



(d) Communication cost reduction

Figure 4.14: Performance of decentralized destination PM selection algorithm with varying VM to PM ratio (1000 PMs).

4.3.7 Performance of Decentralized Destination PM Selection

We then compare the performance of decentralized destination PM selection algorithm introduced in Section 4.2.6 with the centralized algorithm. We denote the centralized algorithm as RIAL, and the decentralized algorithms with cluster size c as RIAL-c, where c was set to 20, 30 and 40, respectively. Figure 4.14(a) shows the execution time of different algorithms. We see that the execution time follows RIAL-20<RIAL-30<RIAL-40<RIAL. This is because a cluster with a smaller number of PMs has a smaller problem size and all cluster masters conduct the destination PM selection simultaneously. RIAL has a higher time than the others because it must rank all the PMs based on Equ. (4.22) in the datacenter for PM selection. Figure 4.14(b) shows the number of VM migrations, which follows RIAL<RIAL-40<RIAL-30<RIAL-20. This is because the selected destination PM in a smaller cluster is not the ideal destination PM in the system scope with high probability and is more likely to become overloaded later on, which leads to more VM migrations. Figure 4.14(c) shows the performance degradation, which follows RIAL<RIAL-40<RIAL-30<RIAL-20. Although selecting PM nearby (in a smaller cluster) can reduce the distance of migration, but the large number of VM migrations (as indicated in Figure 4.14(b)) offsets the benefit, resulting in higher performance degradation. Figure 4.14(d) shows the communication cost reduction, which follows RIAL-20<RIAL-30<RIAL-40<RIAL due to the reason that a larger cluster has more opportunities or options for reducing communication cost. The best PM selected within a cluster reduces less communication cost compared to the best PM selected within the whole datacenter. These results confirms that the decentralized algorithm does not degrade the performance greatly while significantly reduces the algorithm execution time.

4.3.8 Real-World Testbed Experiments

For real-world testbed experiments of RIAL, we deployed a cluster with 7 PMs (2.00GHz Intel(R) Core(TM)2 CPU, 2GB memory, 60GB HDD) and two NFS (Network File System) servers with a combined capacity of 80GB. We then implemented the various load balancing algorithms in Python 2.7.2 using the XenAPI library [59] running in a management node (3.00GHz Intel(R) Core(TM)2 CPU, 4GB memory, running Ubuntu 11.04). We created 15 VMs (1VCPU, 256MB memory, 8.0GB virtual disk, running Debian Squeeze 6.0) in the cluster; each with Apache2 Web Server installed. We used the publicly available workload generator *lookbusy* [31] to generate both CPU and memory workloads.

The communication delay between two PMs is determined by the number of switches across the communication paths in the testbed architecture. We created latency between machines such that all traffic from machine is in the ratio of 1:4:10 to follow the network hierarchical setup [38]. That is, if the communication path between two PMs comes across one switch, two switches, and three switches, respectively, the latency between VMs in the two PMs was set to be 1, 4 and 10, respectively. We run each test for 20 times; each lasts for approximately 60m.

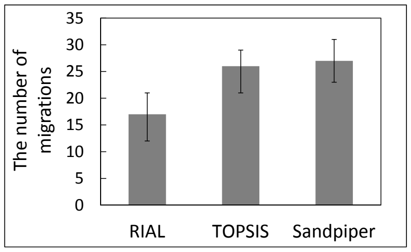


Figure 4.15: Number of migrations.

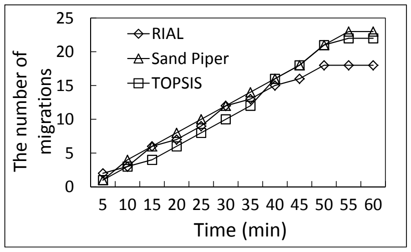


Figure 4.16: Accumulated # of migrations.

4.3.8.1 The Number of Migrations

Figure 4.15 shows the median, 10th percentile and 90th percentile of the total number of migrations in different methods. We can see that RIAL triggers fewer VM migrations than the other two methods to achieve a load balanced state, while TOPSIS generates fewer VM migrations than Sandpiper. Figure 4.16 shows the accumulated number of migrations over time. We see that before 40m, RIAL generates a similar number of migrations as Sandpiper and TOPSIS, since all methods begin from a similar load unbalanced state at the beginning of the experiment. After around 40m, RIAL produces much fewer migrations and after 50m, it produces no migrations and reaches the load balanced state, while TOPSIS and Sandpiper continue to trigger VM migrations. This result confirms that RIAL generates fewer migrations and achieves the load balanced state faster due to its consideration of resource intensity.

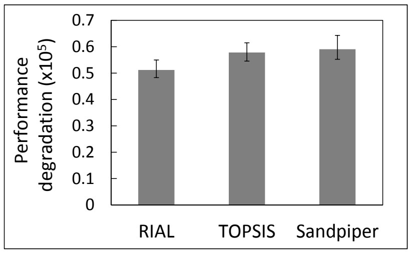


Figure 4.17: Performance degradation.

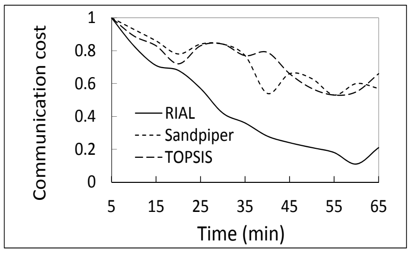


Figure 4.18: Communication cost.

4.3.8.2 Performance Degradation

Figure 4.17 shows the median, 10th percentile and 90th percentile of the total VM performance degradation of the three methods. We measured the real migration time to replace $\frac{M_{ij}}{B_{ip}}$ in

Formula (4.7) to calculate the performance degradation. The figure shows that the VM performance degradation of RIAL is lower than those of Sandpiper and TOPSIS since it tries to reduce VM performance degradation when selecting destination PMs. TOPSIS has a slightly lower VM performance degradation than Sandpiper. As in the simulation, the variance of the results also follows RIAL<TOPSIS<Sandpiper though it is not obvious due to the small scale. These experimental results confirm the advantage of RIAL with the consideration of VM performance degradation in load balancing.

4.3.8.3 Communication Cost

We generated a random graph $G(n = 15, p = 0.2)$ to represent the VM communication topology. Initially, we manually placed intensively communicating VMs in PMs with higher network delay for testing.

We measured the sum of the communication cost of each pair of communicating VMs at the initial stage as the base and measured the total communication cost at every 5 minutes during the experiment. Figure 4.18 shows the normalized communication cost according to the base. We see that as time goes on, the communication cost of all methods decreases. This is because we initially placed intensively communicating VMs in PMs with higher network delay and VM migration can reduce the communication cost. Our method can reduce the communication cost much more and faster than the other methods, reaching 20% of the base communication cost. TOPSIS and Sandpiper have similar curves since they neglect VM communication cost in load balancing.

Chapter 5

Proactive VM Migration for Long-Term Load Balance

5.1 MDP-based Load Balancing

5.1.1 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) [10]. 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.

5.1.2 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 [23] 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, 14, 20, 42, 43], 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.

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 5.1.4, we first introduce the definitions of the load states in Section 5.1.3.

5.1.3 Load State of PMs and VMs

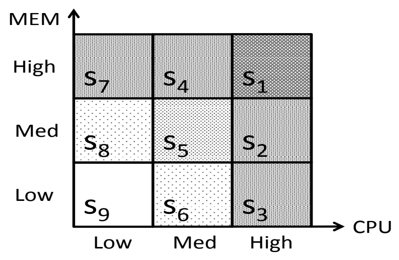
In our load balancing algorithm, each PM selects VMs in certain load states to migrate out in advance when they are about to be overloaded, so that it can maintain its load balance state for a long time. This algorithm proactively avoids overloading PMs in the cloud and continually maintains the system 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.

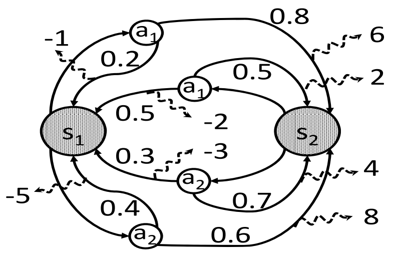
We assume there are N VMs running on M PMs in a cloud. We regard time period as a series of time intervals (τ) and use t_i to denote the specific time at the end of the i-th interval. We use $l_n^k(t_i)$ to denote the demanded resource amount (i.e., load) of the type-k resource in the n-th VM at time t_i. 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

$$u^k n_(t_i) = l_n^k(t_i)/c_n^k(t_i), \tag{5.1}$$



(a) PM states.



(b) MDP model.

Figure 5.1: Example of a simple MDP.

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

$$U_m^k(t_i) = L_m^k(t_i)/C_m^k(t_i) = \sum_{j=1}^{N_m} l_j^k(t_i)/C_m^k(t_i). \tag{5.2}$$

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 $U_m^k(t_i) \le T_o^k$ for each type of resources. For simplicity, we omit k in the notation unless we need to distinguish different types of resources.

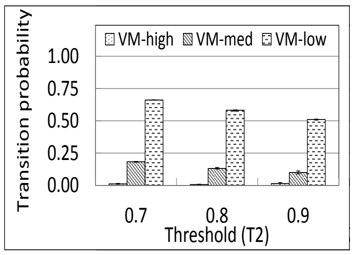
In a PM, for a given resource, based on the resource utilization (i.e., load) of the PM, we determine the utilization level of this resource in this PM. We use three levels (high, medium and low) as an example to explain our algorithm in this paper, which can be easily extended to more levels. Specifically, to perform level determination for type-k resource, we use Equation (5.3), in which T_1^k and T_2^k are two thresholds used to distinguish low and medium, and medium and high levels, respectively.

$$\begin{cases} Low & \text{if } U_m^k < T_1^k \\ Medium & \text{if } U_m^k \ge T_1^k \text{ and } U_m^k < T_2^k \\ High & \text{if } U_m^k \ge T_2^k \end{cases} \tag{5.3}$$

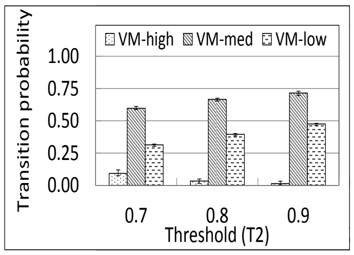
The state determination of VMs is performed in the same manner by changing U_m^k in Equation (5.3) to u_n^k. If the utilization of at least one resource in a PM reaches the heavily loaded threshold, this PM is heavily loaded. Only when the utilizations of all resources in a PM do not reach the heavily loaded threshold, this PM is lightly loaded.

Consider a set of 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

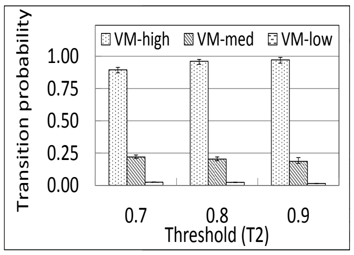
r_k in different resource utilization levels. For example, if we consider two resources, $\mathcal{R} = \{\text{CPU}, \text{Mem}\}$, a PM's state can be represented by the utilization degree of each resource such as (CPU-high, Mem-high), (CPU-median, Mem-low), etc. Then, there are 3^2=9 states for a VM or a PM as shown in Figure 5.1(a).



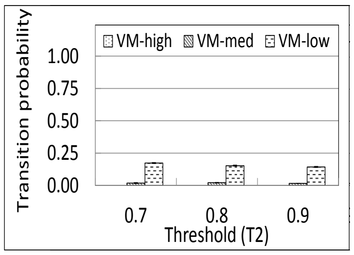
(a) PM-State: high→high



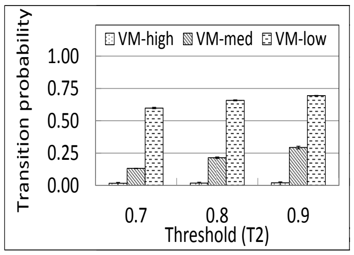
(b) PM-State: high→med.



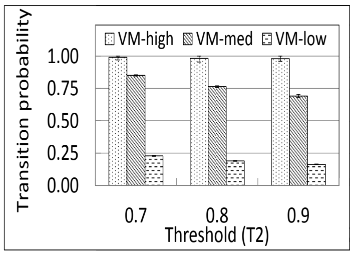
(c) PM-State: high→low



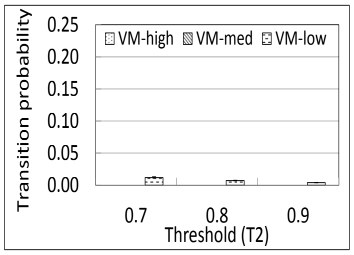
(d) PM-State: med.→high



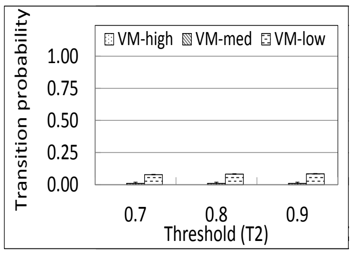
(e) PM-State: med.→med.



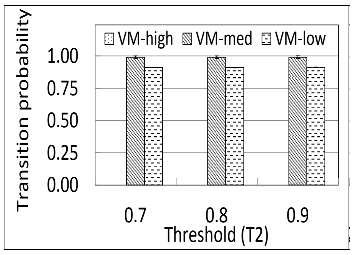
(f) PM-State: med.→low



(g) PM-State: low→high



(h) PM-State: low→med.



(i) PM-State: low→low

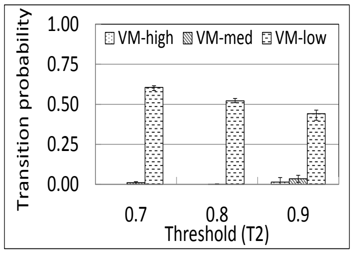
Figure 5.2: Probability of state transitions of PM-high using PlanetLab trace.

5.1.4 Trace Study on the Stability of Our MDP

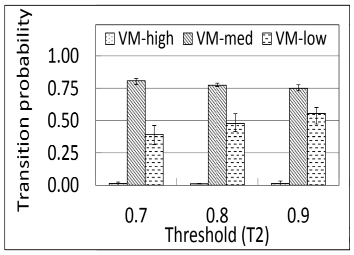
State set S is a set of PM resource utilization levels based on Equation (5.3). As mentioned before, the transition probabilities of an MDP must be stable. To confirm whether our design of different MDP components can achieve the MDP stability, in this section, we conduct an experiment, which shows that the transition probability matrix remains stable even when we slightly change threshold T_i^k in Equation (5.3). Therefore, we can properly set approximate T_i^k to determine the resource utilization level in MDP construction.

In Equation (5.3), 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 [13] 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 [13] and Google Cluster trace [21]. 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 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$\rightarrow$medium" PM-state transitions when migrating a medium VM-state).

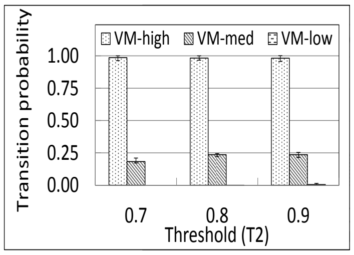
We repeated the experiment 100 times and calculated the transition probabilities. Figure 5.2 and Figure 5.3 show the transition probabilities of PM state changes when using the PlanetLab trace and the Google Cluster trace, respectively. The error bars show the 99th and 1st percentiles among the 100 experiments. 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 5.2(c) and Figure 5.3(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 5.2(i) and Figure 5.3(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 5.2(g) and Figure 5.3(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



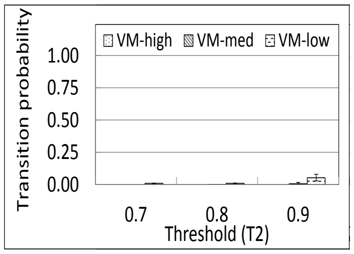
(a) PM-State: high→high



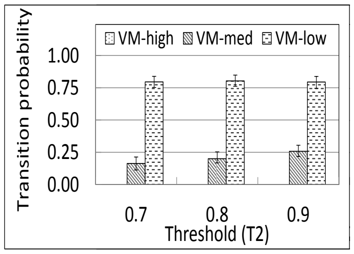
(b) PM-State: high→med.



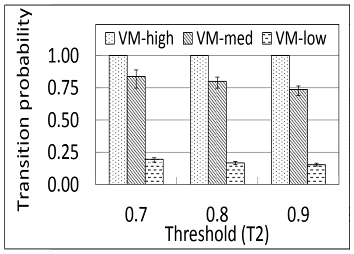
(c) PM-State: high→low



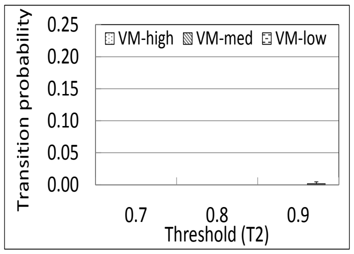
(d) PM-State: med.→high



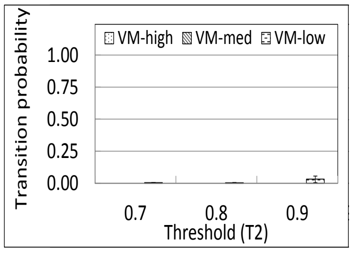
(e) PM-State: med.→med.



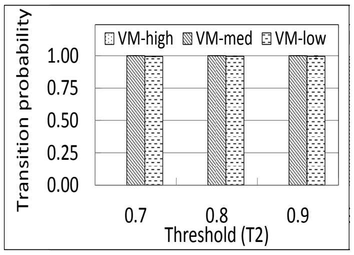
(f) PM-State: med.→low



(g) PM-State: low→high



(h) PM-State: low→med.



(i) PM-State: low→low

Figure 5.3: Probability of state transitions of PM-high using Google Cluster trace.

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 5.2, the absolute values of the transition probabilities derived from the Google Cluster trace in Figure 5.3 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.

5.2 Construction of MDP

5.2.1 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 5.1(b) illustrates the transition model of a simple MDP with two states and two actions. The 3×3 table in Figure 5.1(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.

	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	0.00	1.00	0.91

Table 5.1: Probabilities with threshold $T_2 = 0.8$.

The transition probability matrix for a given system is obtained by studying the trace information of the system. We will show in Section 5.2.2 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 5.2.2, 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.

5.2.2 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 5.1.3. 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 5.1.3, 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.

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$\rightarrow$high occurs 5 times, high$\rightarrow$medium occurs 4 times, and high$\rightarrow$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 5.1.4. Table 5.1 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 5.1 will be used in our experiments in Section 5.3.

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,

$$\mathcal{R} = \mathcal{R}_L \cup \mathcal{R}_M \cup \mathcal{R}_H. \tag{5.4}$$

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:

$$\lambda = -\prod_{r \in \mathcal{R}_H} d + \prod_{r \in \mathcal{R}_M} b + \prod_{r \in \mathcal{R}_L} c, \quad \forall r \in \mathcal{R}, \tag{5.5}$$

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 $c' > b' > d'$.

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

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 $V(s)$, which contains the reward associated with a state, and policy $\Pi = \{\pi(s_1), \pi(s_2), ..., \pi(s_i, ...)\}$, 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 are repeated in some order for all the states until no further changes take place:

$$\pi(s_i) = \arg\max_a \{\sum_j (P_a(s_i, s_j)(R_a(s_i, s_j) + V(s_j))\} \tag{5.6}$$

$$V(s_i) = \sum_j P_{\pi(s_i)}(s_i, s_j)(R_{\pi(s_i)}(s_i, s_j) + V(s_j)) \tag{5.7}$$

Equation (5.6) obtains the optimal policy. In Equation (5.6), $V(s_j)$ is obtained by using Equation (5.7) for each state. Specifically, in order to determine the optimal policy, we apply the value-iteration algorithm [8], 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 6 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 $\Pi$
```

Algorithm 6 shows the pseudo code for the value-iteration algorithm. In the algorithm, $R(s_i)$ is calculated by

$$R(s_i) = \sum_j P_{\pi(s_i)}(s_i, s_j)R_{\pi(s_i)}(s_i, s_j), \tag{5.8}$$

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 (5.7) and Equation (5.6) 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 $\Pi = \{\pi(s_1), \pi(s_2), ..., \pi(s_i), ..., \pi(|S|)\}$, 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 $\{X_1, X_2, ..., X_k, ..., X_n\}$ 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:

$$P^{(n)}_{\pi(s_i)}(s_i, s_j) = P\{X_n = s_j \mid X_0 = s_i\}. \tag{5.9}$$

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 5.1.4. By the Chapman-Kolmogorov equations [19], we get:

$$P^{(n)}_{\pi(s_i)}(s_i, s_j) = \sum_{s_k \in S} P^{(n-1)}_{\pi(s_i)}(s_i, s_k) \cdot P_{\pi(s_k)}(s_k, s_j), \tag{5.10}$$

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. (5.10) 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 $\{\Pi_1, \Pi_2, ..., \Pi_k, ..., \Pi_W\}$ and n=50. We calculate $P^{(n)}_{\pi(s_i)}(s_i, s_j)$ based on Formula (5.10) 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.

5.2.3 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

$$P_m(t_i) = I_m(t_i) - C_m(t_i) \tag{5.11}$$

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. Recall that $\sum_{n=1}^{N} A_{nm}$ 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:

$$I_m(t_i) = \gamma(t_i) \sum_{n=1}^{N} A_{nm} \tag{5.12}$$

where $\gamma(t_i)$ is calculated by

$$\gamma(t_i) = \begin{cases} E & U_m^k(t_i) \leq T_o^k \;\; \forall\, k \;\; (k = 1, 2, ..., K), \\ -Y & U_m^k(t_i) > T_o^k \;\; \exists\, k \;\; (k = 1, 2, ..., K) \end{cases} \tag{5.13}$$

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 [18]. 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.* [18], the power consumption of a PM, say PM$_m$, at time t_i follows a linear model

$$C_m(t_i) = C_{idle} + \alpha \times U_m^{cpu}(t_i) \tag{5.14}$$

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 [35] 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 ($0<\beta\leq 1$) as in [30]. 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:

$$\Delta\ U_{s,mig}^{cpu}(t_i) = (1+\beta)u_n^{mem}(t_i) \tag{5.15}$$

$$\Delta\ U_{d,mig}^{cpu}(t_i) = \beta u_n^{mem}(t_i) \tag{5.16}$$

Based on Equations (5.14)-(5.16), the total power consumption of PM$_m$ by migrating out VM$_p$ and migrating in VM$_q$ can be derived. equals:

$$C_m(t_i) = C_{idle} + \alpha\Big[(1+\beta)u_p^{mem}(t_i) + \beta\ u_q^{mem}(t_i) + \sum_{n=1}^{N} A_{nm}\frac{u_n^{cpu}(t_i)c_n^{cpu}}{C_m^{cpu}}\Big] \tag{5.17}$$

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. (5.14) to derive $C_m(t_i)$, and apply Equ. (5.12) to derive $I_m(t_i)$. Based on Equ. (5.11) 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. (5.11). The corresponding reward equals the change of profits:

$$R_a(s,s') = P_m(t_{i+1}) - P_m(t_i) \tag{5.18}$$

To simplify the calculation of the above equation, let $N_m(t_i) = \sum_{n=1}^{N} A_{nm}(t_i)$ in Equ. (5.12). We consider two cases: i) there is no VM migration at time t_i , and ii) one VM is migrated out and no other VMs are migrated in at time t_i.

In case i), we have $N_m(t_{i+1}) = N_m(t_i)$. Based on Equ. (5.12), we can derive

$$\begin{aligned} \Delta\ I_m(t_{i+1}) &= I_m(t_{i+1}) - I_m(t_i) \\ &= [\gamma(t_{i+1}) - \gamma(t_i)]N_m(t_i) \\ &= \Delta\gamma(t_{i+1})N_m(t_i) \end{aligned} \tag{5.19}$$

Based on Equ. (5.17), we have

$$\begin{aligned} \Delta\ C_m(t_{i+1}) &= C_m(t_{i+1}) - C_m(t_i) \\ &= \alpha[U_m^{cpu}(t_{i+1}) - U_m^{cpu}(t_i)] \\ &= \alpha\sum_{n=1}^{N} A_{nm}\frac{c_n^{cpu}}{C_m^{cpu}}[u_n^{cpu}(t+1) - u_n^{cpu}(t)] \end{aligned} \tag{5.20}$$

In case ii), we have $N_m(t_i) - N_m(t_{i+1}) = 1$. Based on Equ. (5.12), we can derive:

$$\Delta\ I_m(t_{i+1}) = I_m(t_{i+1}) - I_m(t_i) = \Delta\gamma(t_{i+1})N_m(t_{i+1}) - \gamma(t_i) \tag{5.21}$$

Suppose VM$_x$ is migrated out. Based on Equ. (5.17), we have

$$\Delta\ C_m(t_{i+1}) = \alpha\sum_{n=1}^{N} A_{nm}\frac{c_n^{cpu}}{C_m^{cpu}}[u_n^{cpu}(t+1) - u_n^{cpu}(t)] + \alpha(1+\beta)u_x^{mem}(t_i) \tag{5.22}$$

We estimate $N_m(t_i)$ as a constant $= N_m = \frac{N}{M}$. Finally, based on Equations (5.18)-(5.22), we can derive:

$$R_a(s,s') = \begin{cases} \Delta\gamma(t_{i+1})\frac{N}{M} - \alpha\Delta\ U_m^{cpu}(t_{i+1}) & \text{no migration,} \\ (\Delta\gamma(t_{i+1})(\frac{N}{M} - 1) - \gamma(t_i)) - \alpha(\Delta\ U_m^{cpu}(t_{i+1}) + (1+\beta)u_x^{mem}(t_i)) & \text{otherwise} \end{cases} \tag{5.23}$$

where

$$\Delta\gamma(t_{i+1}) = \begin{cases} 0, & U_m^k(t_i) \leq T_o^k \ \ \forall\, k,\ U_m^k(t_{i+1}) \leq T_o^k \ \ \forall\, k \\ E+Y, & U_m^k(t_i) > T_o^k \ \ \exists\, k,\ U_m^k(t_{i+1}) \leq T_o^k \ \ \forall\, k \\ -E-Y, & U_m^k(t_i) \leq T_o^k \ \ \forall\, k,\ U_m^k(t_{i+1}) > T_o^k \ \ \exists\, k \\ 0, & U_m^k(t_i) > T_o^k \ \ \exists\, k,\ U_m^k(t_{i+1}) > T_o^k \ \ \exists\, k \end{cases} \tag{5.24}$$

and

$$\Delta U_m^{cpu}(t_{i+1}) = \sum_{n=1}^{N} A_{nm} \frac{C_n^{cpu}}{C_m^{cpu}} [u_n^{cpu}(t+1) - u_n^{cpu}(t)] \tag{5.25}$$

is the amount difference of CPU utilization of PM$_m$ between the PM states s and s'.

Given a PM state s, suppose $U_{s,L}^k$ and $U_{s,U}^k$ are the lower bound and the upper bound of type-k resource utilization of this state, respectively. We can estimate the CPU utilization of this state by $U_s^{cpu} = U_{s,L}^{cpu} + \frac{1}{2}(U_{s,U}^{cpu} - U_{s,L}^{cpu})$. Similarly, we can estimate the CPU utilization of this PM state $U_m^{cpu}(t_{i+1})$, and the memory utilization of the migrating VM $u_x^{mem}(t_i)$. Finally, using these these values, we can determine the reward value based on Equations (5.23)-(5.25).

As mentioned previously, the rewards for PM state changes with actions of migrating VM in (accepting VM) and no migration can be derived in a similar way. In the above calculation, we replace the CPU utilization overhead $(1+\beta)u_x^{mem}(t_i)$ by $\beta\, u_x^{mem}(t_i)$, and let $N_m(t_i) - N_m(t_{i+1}) = -1$ since the PM is accepting the migrating VM.

5.2.4 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 [48, 56, 57]. For example, Sandpiper [57] 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 5.2.2 is used for the probability calculation.

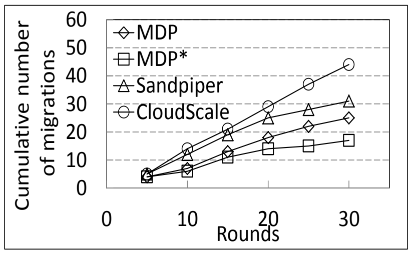
The rewards given to a PM after performing action $a \in A$ should encourage PMs to accept VMs while avoiding heavy state in a long term. Accordingly, the reward system is designed as follows for the state transition of each resource:

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.

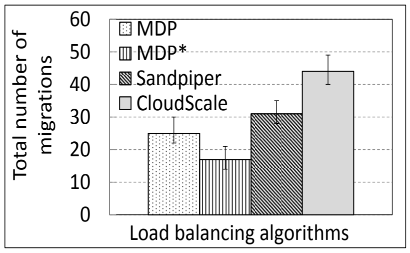
For a given migration VM, the central server can identify the most appropriate destination PMs based on the MDP. Better options from these PMs can be further identified based on additional consideration factors such as VM communication cost and migration distance [16].

5.2.5 An MDP with Extended Action Set

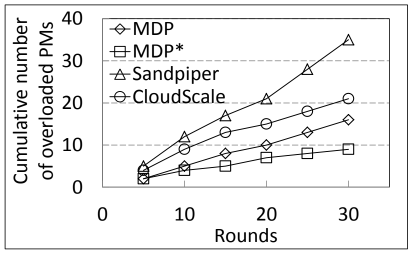
As indicated previously, two MDP models are needed to conduct the MDP-based load balancing; one MDP model is for selecting migration VMs from PMs to migrate out and the other MDP model is for selecting destination PMs for hosting the migration VMs. Building two MDP models brings about a high overhead. More importantly, the policies generated by these two MDP



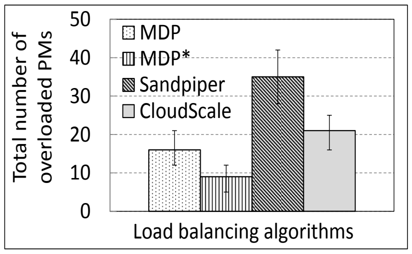
(a) Cumulative # of VM migrations.



(b) Total # of VM migrations.



(c) Cumulative # of overloaded PMs.



(d) Total # of overloaded PMs.

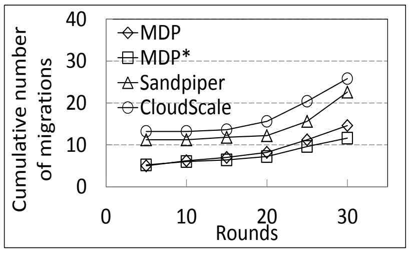
Figure 5.4: Performance using the PlanetLab trace.

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. In order to develop a comprehensive MDP to avoid such conflictions, 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.

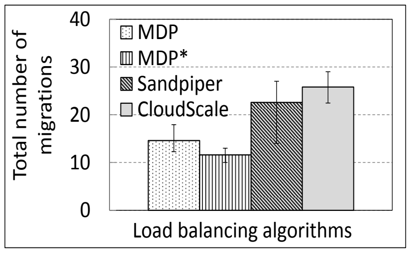
State. Similar to the previous MDP model, the states are defined as the combination of different load levels of different types of resources (e.g., CPU-low, Mem-high). We adopt the same thresholds to distinguish different load levels as in Section 5.1.3 (i.e., $T_1 = 0.3$, $T_2 = 0.8$).

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".

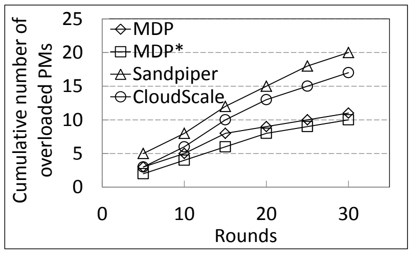
Probability. Since the probabilities are specified with respect to every action (e.g., the



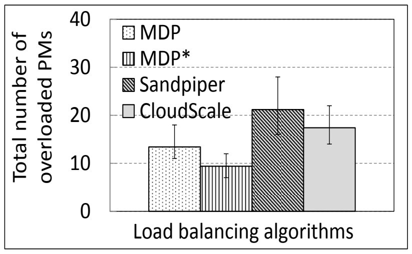
(a) Cumulative # of VM migrations.



(b) Total # of VM migrations.



(c) Cumulative # of overloaded PMs.



(d) Total # of overloaded PMs.

Figure 5.5: Performance using the Google Cluster trace.

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 5.2.3. The rewards for migrating VM in can be derived similarly as for migrating VM out.

5.3 Performance Evaluation

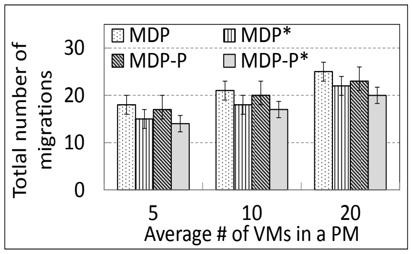
In this section, we conducted trace-driven experiments on CloudSim [13] to evaluate the performance of our proposed MDP-based load balancing algorithm in a two-resource environment (i.e., CPU and Mem). We used the VM utilization trace from PlanetLab [13] and Google Cluster [21] to generate VM workload 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 5.2.4). MDP* uses our MDP model for both VM selection and destination PM selection. We compared MDP and MDP* with Sandpiper [56] and CloudScale [43] 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 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. 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.

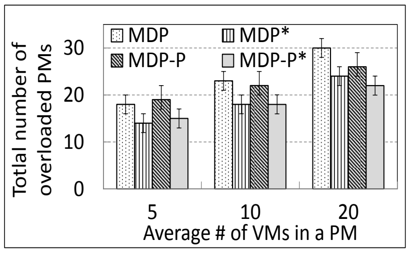
5.3.1 Performance of the Basic MDP

5.3.1.1 The Cumulative Number of Migrations

Figure 5.4 and Figure 5.5 show the performance of MDP, MDP*, Sandpiper and CloudScale with the PlanetLab trace and Google Cluster trace, respectively. Figure 5.4(a) and Figure 5.5(a) show the cumulative number of migrations over the rounds. Both results follow MDP*<MDP< Sandpiper<Clo-udScale. 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

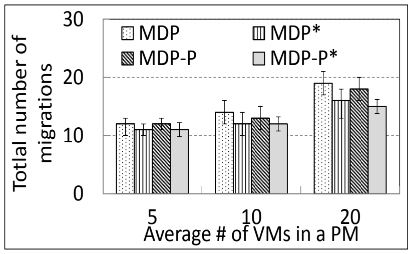


(a) The number of VM migrations.

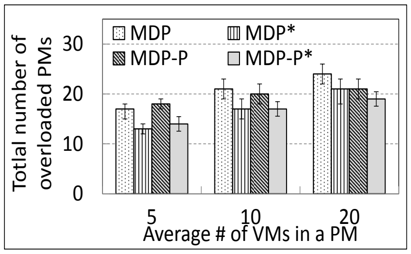


(b) The number of overloaded PMs.

Figure 5.6: Performance of cloud profit oriented reward system (PlanetLab trace).



(a) The number of VM migrations.



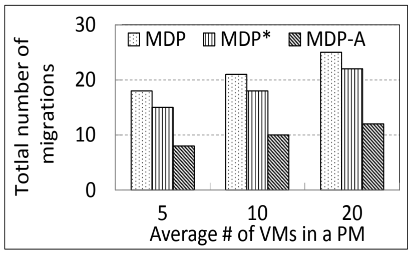
(b) The number of overloaded PMs.

Figure 5.7: Performance of cloud profit oriented reward system (Google Cluster trace).

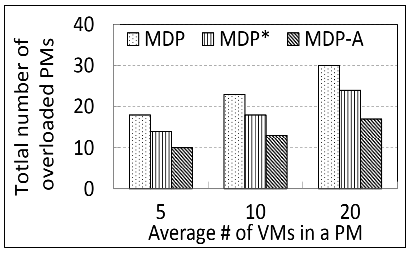
reduce the number of VM migrations. CloudScale generates a larger number of VM migrations than Sandpiper in each round because CloudScale migrates VMs not only for a correctly predicted overloaded PM but also for an incorrectly predicted overloaded PM, but Sandpiper only migrates VMs for occurred overloaded PMs. Figure 5.4(b) and Figure 5.5(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.

5.3.1.2 The Number of Overloaded PMs

Next, we measure the number of overloaded PMs, which indicates the effectiveness of load balancing algorithms. Figure 5.4(c) and Figure 5.5(c) show the cumulative number of overloaded



(a) The number of VM migrations.



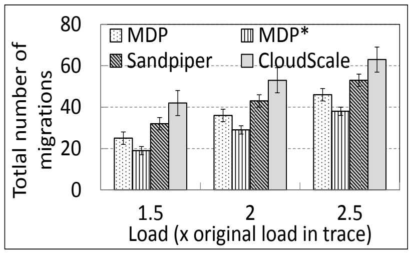
(b) The number of overloaded PMs.

Figure 5.8: Performance of MDP-A (Google Cluster trace).

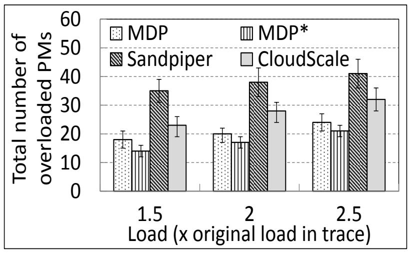
PMs over rounds. MDP and MDP* generate a smaller number of overloaded PMs in each round than CloudScale and Sandpiper. This is because the MDP algorithm incentivizes the PMs to perform optimal VM migration actions to maintain a system load balance state for a longer time. MDP* outperforms MDP with fewer 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 5.4(d) and Figure 5.5(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.

5.3.1.3 The Number of VM Migrations

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 5.9 and Figure 5.10 show the experimental results with the PlanetLab trace and Google Cluster trace, respectively. Figure 5.9(a) and Figure 5.10(a) show the median, the 10th and 90th percentiles of the number of VM migrations of the four 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 5.4(b) and Figure 5.5(b) due to the same reasons as explained before. Figure 5.9(b) and Figure 5.10(b) show the median, the 10th and 90th percentiles of the number of overloaded PMs of the four methods with different workload ratios. The number of overloaded PMs increases with workload ratio, and follows MDP*<MDP<CloudScale<Sandpiper

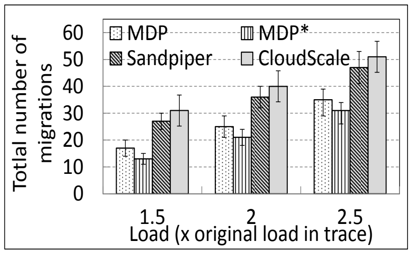


(a) The number of VM migrations.

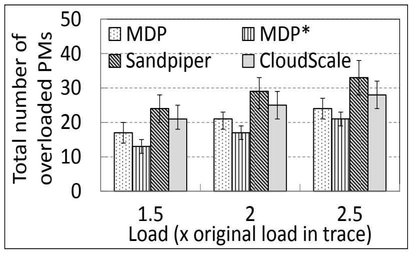


(b) The number of overloaded PMs.

Figure 5.9: Performance of basic MDP and MDP* with increasing workload ratio (PlanetLab trace).



(a) The number of VM migrations.



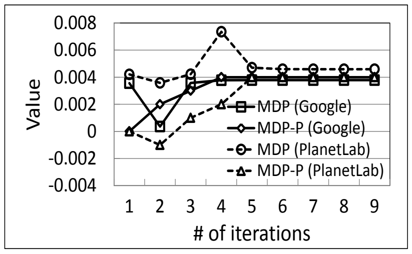
(b) The number of overloaded PMs.

Figure 5.10: Performance of basic MDP and MDP* with increasing workload ratio (Google Cluster trace).

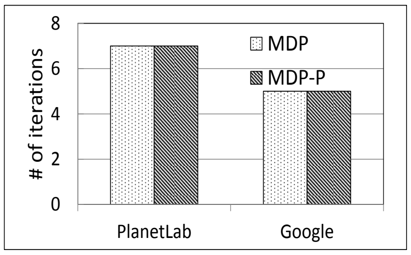
within each workload ratio. The results are consistent with Figure 5.4(d) and Figure 5.5(d) due to the same reasons. Thus, MDP and MDP* perform better then Sandpiper and CloudScale in terms of the number of VM migrations and the number of overloaded PMs in different workloads.

5.3.2 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 5.2.3. 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. (5.12), 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 6 to find the optimal policies. For each optimal policy, we applied it to CloudSim and repeated the simulation for 20 times.

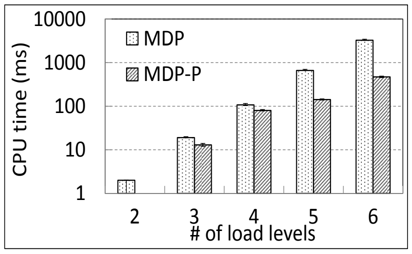


(a) Values vs. iterations.

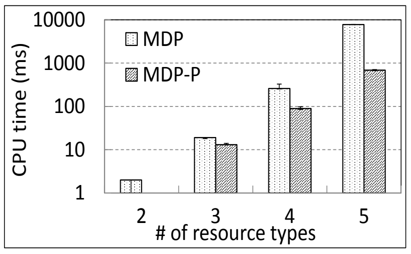


(b) The number of iterations to convergence.

Figure 5.11: Convergence speed of basic MDP and MDP with cloud profit oriented reward system.



(a) CPU time with increasing load levels.



(b) CPU time with increasing number of resource types.

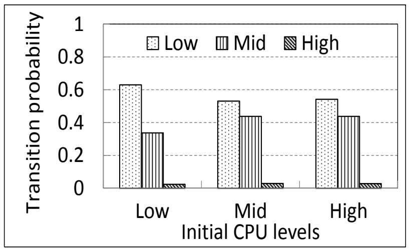
Figure 5.12: Comparison of CPU time consumption by calculating different reward systems.

5.3.2.1 The Number of VM Migrations

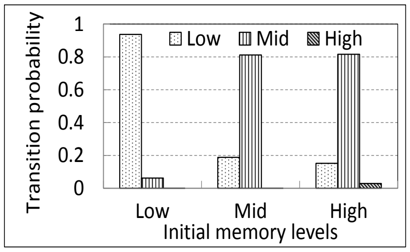
Figure 5.6 and Figure 5.7 show the experimental results with the PlanetLab trace and Google Cluster trace, respectively. Figure 5.6(a) and Figure 5.7(a) show the median, the 10th and 90th percentiles of the number of VM migrations of the four methods with different average number of VMs per PM. The number 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. (5.5) to construct the reward system, which does not explicitly reflect memory utilization of the migrating VMs. MDP-P relies on Equ. (5.18), which incorporates Equ. (5.15) and Equ. (5.16) 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.

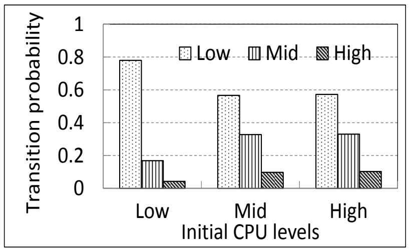
Figure 5.6(b) and Figure 5.7(b) show the median, the 10th and 90th percentiles of the number of overloaded PMs of the four methods with different average number of VMs per PM. When the average number is 5, MDP has a smaller number of overload PMs than MDP-P. The reason is that the cost for violating SLA is relatively smaller as Equ. (5.12) 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.



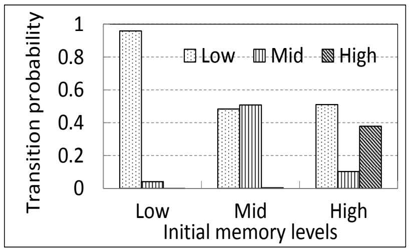
(a) CPU state transition probabilities of MDP.



(b) Memory state transition probabilities of MDP.



(c) CPU state transition probabilities of MDP-P.



(d) Memory state transition probabilities of MDP-P.

Figure 5.13: n-step transition probabilities of MDP and MDP-P (n=50).

5.3.2.2 Algorithm Convergence Time

We then test the convergence time of the value-iteration algorithm using the PlanetLab trace and Google Cluster trace. We keep track of the values for the states (i.e., Equ. (5.7) in Algorithm

6) in each iteration. Figure 5.11(a) shows the calculated values in each iteration for a PM in the CPU-medium and Mem-low state, using PlanetLab trace and Google Cluster trace. Although the calculated values converge at around 6 iterations, we still show them in the consequent iterations in order to provide a comprehensive results. In the curves, we see that the value varies at the beginning and gradually converges to a steady value after several rounds of iterations. Both MDP and MDP-P converge in around 4 iterations under Google Cluster trace, and converge in around 6 iterations under PlanetLab trace. MDP and MDP-P have the same convergence time with either Google Cluster trace or PlanetLab trace. The convergence time with different traces is different because their state transition probabilities are different. This result shows that the convergence time of MDP and MDP-P is short. Also, compared to MDP, MDP-P does not compromise the convergence time. It also indicates that the number of iterations depends on the traces, i.e., on the transition probabilities.

Figure 5.11(b) shows the number of iterations needed to reach convergence of the algorithm, using PlanetLab trace and Google Cluster trace, respectively. We also varied the parameters such as α, β, γ and average number of VMs in a PM in the reward system. Specifically, we randomly picked values in the range [1,10] for α, β, γ, and randomly picked values in the range [5,20] for the average number of VMs per PM. Our experimental results show that the number of iterations stays the same as Figure 5.11(b) under various parameter settings (we do not show the results in the figure due to space limit). The reason is that the number of iterations depends on the transition probabilities. The value (i.e., $V(s)$ in Equ. (5.7)) for each PM state is determined by the transition probabilities. This result again confirms that the convergence time of both MDP and MDP-P is short. Also, MDP-P does not compromise the convergence time of MDP.

5.3.2.3 Computation Complexity of Reward Systems

In order to evaluate the computing complexities for determining the rewards, we measure the CPU time consumption for calculating different reward systems. We use MDP and MDP-P to denote their used reward system. MDP uses the reward system introduced in Section 5.2.2 and MDP-P uses the reward system introduced in Section 5.2.3. Recall that the number of states equals L^K, where L is the number of load levels and K is the number of resource types. Since rewards are given to each state transition by taking an action, the number of states has a direct impact on the time to construct a reward system. We first set the number of resource types $K = 2$, and increased the number of load levels from 2 to 6. We repeated each test 20 times and recorded the median, the 90^{th} and 10^{th} percentiles of the results.

Figure 5.12(a) shows the CPU time in the logarithmic scale with increasing number of load levels. We see that the CPU time of both reward systems increase with the number of load levels. The reason is that the number of times for calculating rewards (i.e., $L^K \times (L^K + 1) \times L^K$) increases with the number of load levels L. MDP consumes more CPU time than MDP-P and its CPU time increases more rapidly than MDP-P, because it involves more computations. Recall that the rewards of MDP are calculated based on Equ. (5.5), and the rewards of MDP-P are calculated based on Equ. (5.23). Thus, MDP checks the state changes (i.e., states before and after the action) of each type of resources, while MDP-P only checks whether there is overloaded resource (i.e., whether the PM is overloaded). We then set the number of load levels $L = 3$, and increased the number of resource types from 2 to 5. We repeated each test 20 times and recorded the results. Figure 5.12(b) shows the CPU time in the logarithmic scale with increasing number of resource types. We see similar trend as in Figure 5.12(a) due to the same reasons. Compared to previous results in Figure 5.12(a), the CPU time of both systems increases much faster because the number of times $L^K \times (L^K + 1) \times L^K$ polynomially increases with the number of load levels L, while exponentially increases with the number of resource types K. Note that although CPU time in this experiment is higher than the CPU time to achieve load balance, which will be presented in Figure 5.15, it dose not degrade the performance of MDP because the procedure of calculating the reward system is executed offline only once.

5.3.3 Performance of the MDP with Extended Action Set

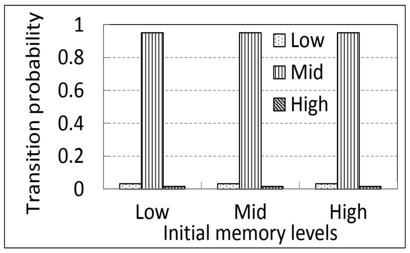
We then study the performance of the MDP with extended action set denoted by MDP-A. Similar as previous experiments, we increased the average number of VMs in a PM from 5 to 20. For each average number of VMs, we applied the optimal policies corresponding to each algorithms (i.e., MDP, MDP* and MDP-A) to CloudSim and repeated the simulation for 20 times. Figure 5.8 presents the experimental results with the Google Cluster trace. Figure 5.8(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

5.8(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).

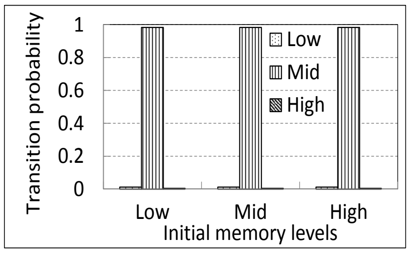
5.3.4 Comparison of Different MDP Models

5.3.4.1 *n*-Step Transition Probability

In order to investigate the capability of the MDP-based load balancing algorithms in avoiding overloaded PMs, we apply the output policy Π from Algorithm 6 to Equ. (5.10) to calculate the n-step transition probabilities of state changes (e.g., from CPU-high and memory-low to CPU-low and memory-low) and actions (e.g., migrating out a VM in VM state CPU-medium, memory-low). We compare the performances of MDP, MDP-P and MDP-A. In this experiment, we set $n = 50$. Since we conduct the MDP-based load balancing algorithm in a two-resource environment (i.e., CPU and Mem, $K = 2$) with $L = 3$ load levels for each resource, there are $L^K = 9$ PM states in total. The result of Equ. (5.10) is a 9×9 matrix indicating the probabilities of state transitions after 50 rounds. We further calculate the probabilities for state transitions with respect to each type of resources. For example, suppose if we use (H,M)→(M,L) to represent the probability of transition from state (CPU-high, Mem-medium) to state (CPU-medium, Mem-low), the probability for CPU changing from high to low can be obtained by $\frac{1}{3}\sum_x \sum_y (H,x) \rightarrow (L,y)$, where $x, y \in \{H, M, L\}$. Similarly, we can calculate the probabilities for PM memory utilization transitions. Figure 5.13 shows the n-step transition probabilities with respect to CPU and memory of MDP and MDP-P, respectively. We see that both MDP and MDP-P algorithms achieve high probabilities for resource states (of both CPU and memory) transiting from high to other states. This result confirms that both algorithms are able to eliminate resource utilization overloads with high probabilities in the long run (i.e., 50 rounds). The probabilities for CPU and memory are different because the one step transition probabilities (i.e., $P_a(s, s')$) for different resources collected from the trace are different. Figure 5.14 shows the n-step transition probabilities of MDP-A. We see that MDP-A achieves high probabilities for resource states (of both CPU and memory) transiting from any states (e.g., low, medium and high) to medium states. This is because this algorithm encourages PMs in high state to offload their workloads by migrating VMs out, and simultaneously motivates PMs in low or medium states increase their workloads by accepting migration VMs from other PMs. Compared to Figure 5.13, MDP-A has relatively higher probabilities for resource states transiting to medium states, 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.



(a) CPU state transition probabilities by MDP-A.



(b) Memory state transition probabilities by MDP-A.

Figure 5.14: n-step transition probabilities of MDP-A (n=50).

5.3.4.2 CPU Time for Load Balancing

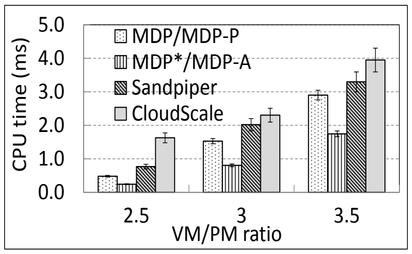
The CPU time consumption for load balancing consists of the maintenance time spent on system monitoring, the time identifying VMs to migrate, the time to determine destination PMs for VMs and the time for VM migrations. The maintenance time refers to the CPU time spent on checking whether there are overloaded 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 5.15(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

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

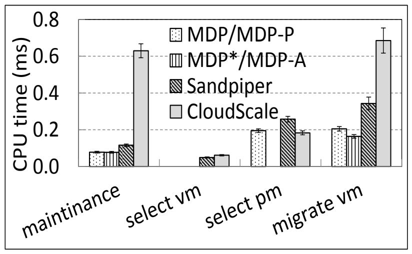
Table 5.2: Power consumption for different CPU utilizations.

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.

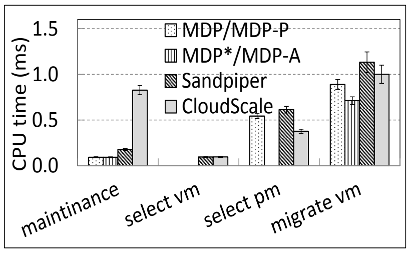
In order to give a thorough comparison between the four methods, we broke down the CPU time to different parts as shown in Figure 5.15(b), Figure 5.15(c) and Figure 5.15(d) corresponding to three VM/PM ratios. MDP and MDP* consume the least maintenance time that is used to determine whether VM migrations 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* [56] 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.



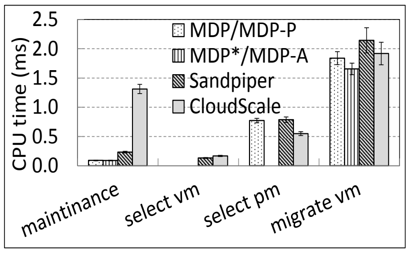
(a) Total time.



(b) CPU time breakdown (ratio=2.5).



(c) CPU time breakdown (ratio=3).



(d) CPU time breakdown (ratio=3.5).

Figure 5.15: Comparison of CPU time consumption by different methods to achieve load balance.

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.

5.3.4.3 Memory Consumption

Figure 5.16 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. In the figures, we present the results of MDP-P together with MDP, and MDP-A together with MDP*, due to the same reason mentioned before. CloudScale consumes much more memory since it needs to store a 40×40 probability transition matrix as indicated in [20] for each VM for workload prediction and it also has a higher number of VM migrations.

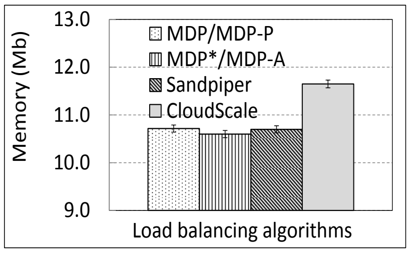


Figure 5.16: Memory consumption (ratio=3).

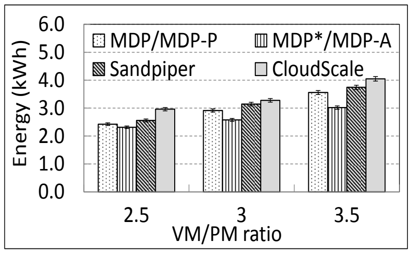


Figure 5.17: Energy consumption in algorithms.

5.3.4.4 Energy Consumption

We then compare energy consumption of the four different load balancing algorithms. Energy consumption by PMs in datacenters is mostly determined by the CPU, memory, disk storage, power supplies and cooling systems [27], and the work in [10] 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 5.2 [10]. Using this table, we calculate and compare the energy consumption of different algorithms. We ran each experiment for one hour and measured the total energy consumption of different algorithms. Figure 5.17 shows the median, the 10th and 90th percentiles of the total amount of the energy consumption among total 10 experiments. In the figures, we present the results of MDP-P together with MDP, and MDP-A together with MDP*, due to the same reason mentioned before. 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.

Chapter 6

Conclusions and Future Work

In this proposal, we propose three mechanisms to tackle the challenges in effective management of virtual resource to maximize energy efficiency and resource utilization while satisfying the SLA in cloud datacenters. Specifically, the three mechanisms are: i) initial VM allocation, ii) VM migration for load balance, and iii) proactive VM migration for long-term load balance. Accordingly, this proposal consists of three innovative components:

(1) Initial Complementary VM Consolidation. Previous resource provisioning strategies either allocate physical resources to virtual machines (VMs) based on static VM resource demands or dynamically handle the variations in VM resource requirements through live VM migrations. However, the former fail to maximize energy efficiency and resource utilization while the latter produce high migration overhead. To handle these problems, we propose an initial VM allocation mechanism that consolidates complementary VMs with spatial/temporal-awareness. Complementary VMs are the VMs whose total demand of each resource dimension (in the spatial space) nearly reaches their host's capacity during VM lifetime period (in the temporal space). Based on our observation of the existence of VM resource utilization patterns, the mechanism predicts the lifetime resource utilization patterns of short-term VMs or periodical resource utilization patterns of long-term VMs. Based on the predicted patterns, it coordinates the requirements of different resources and consolidates complementary VMs in the same physical machine (PM). This mechanism reduces the number of PMs needed to provide VM service hence increases energy efficiency and resource utilization and also reduces the number of VM migrations and SLA violations.

(2) Resource Intensity Aware VM Migration for Load Balance. The unique features of

clouds pose formidable challenges to achieving effective and efficient load balancing. First, VMs in clouds use different resources (e.g., CPU, bandwidth, memory) to serve a variety of services (e.g., high performance computing, web services, file services), resulting in different overutilized resources in different PMs. Also, the overutilized resources in a PM may vary over time due to the time-varying heterogenous service requests. Second, there is intensive network communication between VMs. However, previous load balancing methods statically assign equal or predefined weights to different resources, which leads to degraded performance in terms of speed and cost to achieve load balance. Also, they do not strive to minimize the VM communications between PMs. This proposed mechanism dynamically assigns different weights to different resources according to their usage intensity in the PM, which significantly reduces the time and cost to achieve load balance and avoids future load imbalance. It also tries to keep frequently communicating VMs in the same PM to reduce bandwidth cost, and migrate VMs to PMs with minimum VM performance degradation.

(3) Proactive VM Migration for Long-Term Load Balance. 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, 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 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.

Finally, we conducted extensive experiments to evaluate the proposed three mechanisms. i) We conducted simulation experiments based on two real traces and real-world testbed experiments to show that the initial complementary VM consolidation mechanism significantly reduces the number of PMs used, SLA violations and VM migrations of the previous resource provisioning strategies. ii) We conducted trace-driven simulation and real-world testbed experiments to show that RIAL outperforms other load balancing approaches in regards to the number of VM migrations, VM performance degradation and VM communication cost. iii) We conducted trace-driven experiments to show that the MDP-based load balancing algorithm outperforms previous reactive and proactive load balancing algorithms in terms of SLA violation, load balancing efficiency and long-term load balance maintenance.

The future work will be three folds. First, for Initial complementary VM consolidation, we will explore how to enhance the pattern detection method to catch peak bursts and how to complement VMs with peak bursts in resource consumption. Second, for resource intensity aware VM migration for load balance, we will study how to globally map migration VMs and destination PMs in the system to enhance the effectiveness and efficiency of load balancing. We will also measure the overhead of RIAL and explore methods to achieve an optimal tradeoff between overhead and effectiveness. Third, for proactive VM migration for long-term load balance, we aim to make our algorithm fully distributed to increase its scalability.

Bibliography

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

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

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4), August 2008.

[4] Amazon web service. http://aws.amazon.com/.

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

[6] Brenda S Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 6(1):49–70, 1985.

[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. of SIGCOMM*, 2011.

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

[9] U. Bellur, C. S. Rao, and M. K. SD. Optimal placement algorithms for virtual machines. *CoRR*, 2010.

[10] 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 and Computation: Practice and Experience*, 2011.

[11] 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.

[12] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proc. of IM*, pages 1059–1062, 2007.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *SPE*, 2011.

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

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

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

[17] J. Csirik, J. B. G. Frenk, M. Labbe, and S. Zhang. On the multidimensional vector bin packing. *Acta Cybern.*, 1990.

[18] Xiaobo Fan, Carla S Ellis, and Alvin R Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *Proc. of PACS*, pages 164–179. 2005.

[19] Crispin W Gardiner et al. *Handbook of stochastic methods*, volume 4. Springer Berlin, 1985.

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

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

[22] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proc. of SIGCOMM*, 2009.

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

[24] H. Hsiao, H. Su, H. Shen, and Y. Chao. Load rebalancing for distributed file systems in clouds. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2012.

[25] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Proc. of NOMS*, 2009.

[26] 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.

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

[28] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *Proc. of INFOCOM*, 2011.

[29] Yuhua Lin, Haiying Shen, and Liuhua Chen. Ecoflow: An economical and deadline-driven inter-datacenter video flow scheduling system. In *Proc. of ACM Multimedia*, pages 1059–1062, 2015.

[30] Haikun Liu, Hai Jin, Cheng-Zhong Xu, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. *Cluster computing*, 16(2):249–264, 2013.

[31] lookbusy. http://devin.com/lookbusy/.

[32] M. J. Magazine and M.-S. Chern. A note on approximation schemes for multidimensional knapsack problems. *MOR*, 1984.

[33] S. T. Maguluri, R. Srikant, and L. Ying. Stochastic models of load balancing and scheduling in cloud computing clusters. In *Proc. of INFOCOM*, 2012.

[34] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proc. of INFOCOM*, 2010.

[35] Lauri Minas and Brad Ellison. *Energy efficiency for information technology: How to reduce power consumption in servers and data centers.* Intel Press, 2009.

[36] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for vector bin packing. Technical report, Microsoft Research, 2010.

[37] K. Park and V. S. Pai. Comon: a mostly-scalable monitoring system for planetlab. *Operating Systems Review*, 2006.

[38] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual machine image distribution network for cloud data centers. In *Proc. of INFOCOM*, 2012.

[39] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: sharing the network in cloud computing. In *Proc. of SIGCOMM*, 2012.

[40] A. Rai, R. Bhagwan, and S. Guha. Generalized resource allocation for the cloud. In *Proc. of SOCC*, 2012.

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

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

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

[44] V. Shrivastava, P. Zerfos, K. Lee, H. Jamjoom, Y. Liu, and S. Banerjee. Application-aware virtual machine migration in data centers. In *Proc. INFOCOM*, 2011.

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

[46] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proc. of HotPower*, 2008.

[47] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *Proc. of WWW*, 2007.

[48] 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*, 2010.

[49] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Proc. of Middleware*, 2008.

[50] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proc. of EuroSys*, page 18, 2015.

[51] H. Viswanathan, E. K. Lee, I. Rodero, D. Pompili, M. Parashar, and M. Gamell. Energy-aware application-centric vm allocation for hpc workloads. In *Proc. of IPDPS Workshops*, 2011.

[52] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. *CoRR*, 2011.

[53] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level SLOs on shared storage systems. In *Proc. of SOCC*, page 14, 2012.

[54] H. Wang and C. L. Yoon. Multiple attributes decision making methods and applications. *Berlin:Springer*, 1981.

[55] Wei Wang, Di Niu, Baochun Li, and Ben Liang. Dynamic cloud resource reservation via cloud brokerage. In *Proc. of ICDCS*, pages 400–409, 2013.

[56] 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.

[57] 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.

[58] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 2009.

[59] Xenapi. http://community.citrix.com/display/xs/Download+SDKs.

[60] D. Xie, N. Ding, Y. C. Hu, and R. R. Kompella. The only constant is change: incorporating time-varying network reservations in data centers. In *Proc. of SIGCOMM*, 2012.

[61] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31, 2014.

[62] J. Xu and J. A B Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Proc. of CPSCom*, 2010.