Consolidating Complementary VMs with Spatial/Temporal-awareness in Cloud Datacenters

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

***Abstract*—In cloud datacenters, effective resource provisioning is needed to maximize energy efficiency and utilization of cloud resources while guaranteeing the Service Level Agreement (SLA) for tenants. 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. Simulation based on two real traces and real-world testbed experiments show that our initial VM allocation mechanism significantly reduces the number of PMs used, SLA violations and VM migrations of the previous resource provisioning strategies.**

I. Introduction

Cloud computing has been intensively studied recently due to its great promises [1]. Cloud providers use visualization technologies to allocate Physical Machine (PM) resources to tenant Virtual Machines (VMs) based on their resource (e.g., CPU, memory and bandwidth) requirements. 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 become critical concerns. Maximizing energy efficiency and utilization of cloud resources while satisfying Service Level Agreement (SLA) for tenants requires effective management of resource provisioning.

Previous server resource provisioning (or VM allocation) strategies can be classified into two categories: static provisioning and dynamic provisioning [2]. Static provisioning [3]–[7] allocates physical resources to VMs only once based on static VM 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 provisioning [8]–[13] has been proposed, which first consolidates VMs using a simple bin-packing heuristic and then handles the variations in VM resource requirements through live VM migrations [8]. However, VM migration generates high migration overhead and also degrades the VM performance [14]. 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-oblivious, which fails to constantly fully utilize different resources.

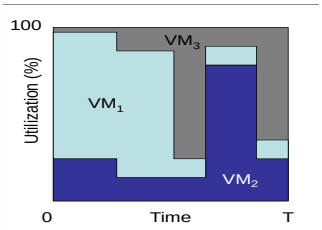


Fig. 1. Consolidating complementary VMs.

Our primary goal is to handle the aforementioned problems and design a VM allocation mechanism to further reduce the number of PMs needed for service provisioning, maximize resource utilization and reduce the number of VM migrations, while ensuring SLA guarantees. To this end, we propose an initial VM allocation mechanism that predicts the VM resource utilization patterns and 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 PM's capacity during VM lifetime period (in the temporal space). For example, a low-CPU-utilization and high-memory-utilization VM and a high-CPU-utilization and low-memory-utilization VM can be consolidated in one PM to fully utilize both of its CPU and memory resources. As shown in a simple 1-dimensional example in Figure 1, the resource utilization patterns of VM1, VM2 and VM3 are complementary to each other on the resource. Placing these three VMs together in the PM can fully utilize this resource of the PM and reduce VM migrations while still ensures the SLA guarantees.

It was indicated that when VMs are configured to run an

978-1-4799-3360-0/14/$31.00

application collaboratively, their workload pattern variations can be predicted [15]. We notice that different VMs running the same short-term application job task (e.g., MapReduce) tend to have similar resource utilization patterns, because each VM executes exactly the same source code with the same options. In long-term applications such as web services and file services, the workloads on the VMs are often driven by human requests determined by daily human activities. Therefore, these VMs exhibit periodical resource utilization patterns. Thus, based on the historical resource utilizations of VMs from a tenant, the lifetime resource utilization patterns for short-term VMs or periodical resource utilization patterns for long-term VMs requested by this tenant to run the same job can be predicted. The contribution of this paper can be summarized as follows.

- We study VMs running short-term MapReduce jobs and observe that the VMs running the same job task tend to have similar resource utilization patterns over time. We also study the PlanetLab and Google Cluster VM traces and find that different VMs running a long-term job exhibit similar periodical resource utilization patterns.
- We then design a practical algorithm to detect the resource utilization patterns from a group of VMs.
- We propose an initial VM allocation policy that consolidates complementary VMs based on the predicted VM resource demand patterns. The policy coordinates the requirements of different resources of the VMs to realize spatial/temporal-aware VM consolidation.
- We conduct comprehensive simulation based on two real traces and real-world experiments running a MapReduce job. Experimental results show that our initial VM allocation mechanism significantly reduces the number of PMs, SLA violations and VM migrations.

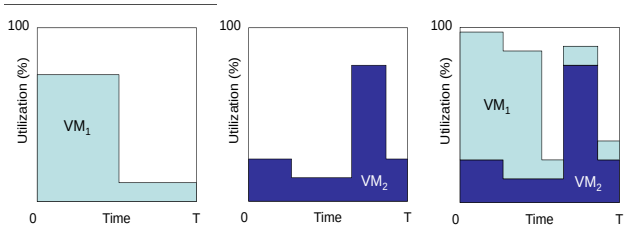
The rest of the article is organized as follows. Section II presents the details of our initial VM allocation mechanism. Section III evaluates our method in trace-driven simulation experiments. Section IV evaluates our method in real-world testbed. Section V briefly describes the related work. Finally, Section VI summarizes the paper with remarks on future work.

II. Initial VM Allocation Mechanism

A. Basic Rationale

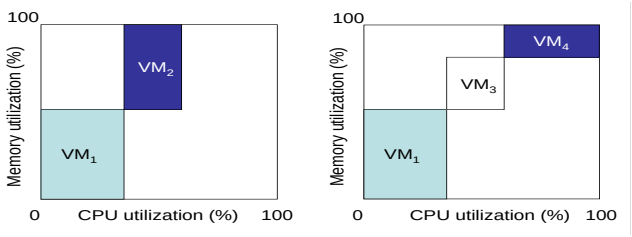
Our primary goal in designing the initial VM allocation mechanism is to minimize the number of PMs used and the number of VM migrations, and maximize resource utilization, while ensuring SLA guarantees. Figure 2 shows a simple 1-dimensional example to explain the idea of our mechanism. VM1 has a high 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 demand across every resource dimension such as CPU,



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

Fig. 2. Consolidating complementary VMs in one PM.



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

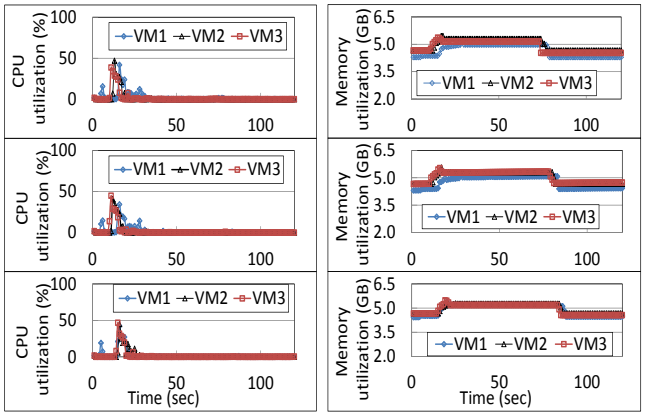
Fig. 3. Consolidating complementary VMs to fully utilize multi-dimensional resources in a PM.

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 demonstrates an example in a 2-dimensional resource space. In Figure 3(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(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.

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 II-B). Second, we discuss how to detect the patterns of a group of VMs running the same job (e.g., WordCount) (Section II-C). 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 II-D).

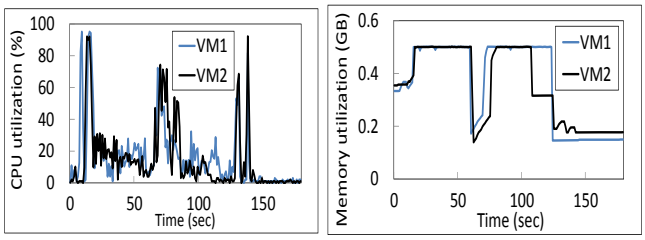
B. Profiling VM Resource Demands

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



(a) CPU utilization (b) Memory utilization

Fig. 4. VM resource utilization for *TeraSort* on three datasets.



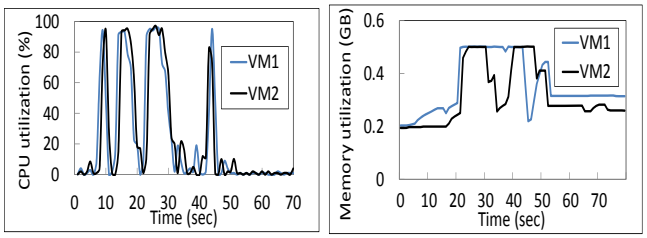
(a) CPU utilization (b) Memory utilization

Fig. 5. VM resource utilization for *TestDFSIO write*.

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.

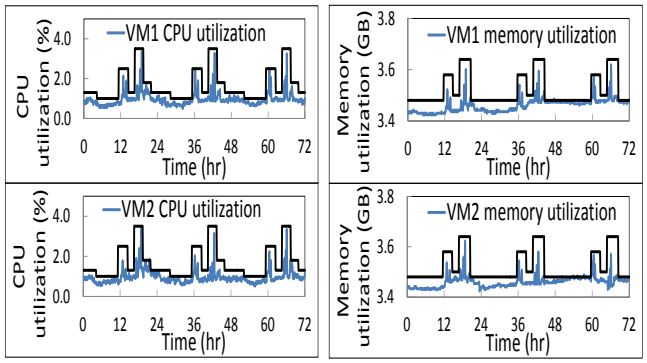
1) Utilization Patterns of VMs for Short-Term Jobs: MapReduce jobs represent an important class of applications in cloud datacenters. We profile the CPU and memory utilization patterns of typical MapReduce jobs. We conducted the profiling experiments on our cluster consisting of 15 machines (3.4GHz Intel(R) 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 4(a) and 4(b) display the resource utilization results of three VMs for different generated datasets. Figure 5 displays the resource utilizations of two VMs running *TestDFSIO write*, which generates 10 output files with each file having 0.1GB. Figure 6 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.



(a) CPU utilization (b) Memory utilization

Fig. 6. VM resource utilization for *TestDFSIO read*.



(a) CPU utilization (b) Memory utilization

Fig. 7. VM resource utilization from Google Cluster trace.

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 [16] and the PlanetLab trace [17]. 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 7(a) shows the CPU utilizations of two VMs every five minutes during three days and Figure 7(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 8 shows the CPU utilization of a randomly selected VM to show their periodical patterns.

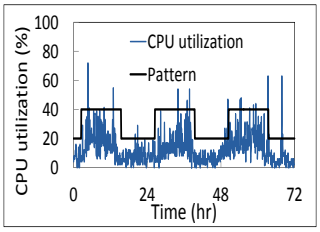
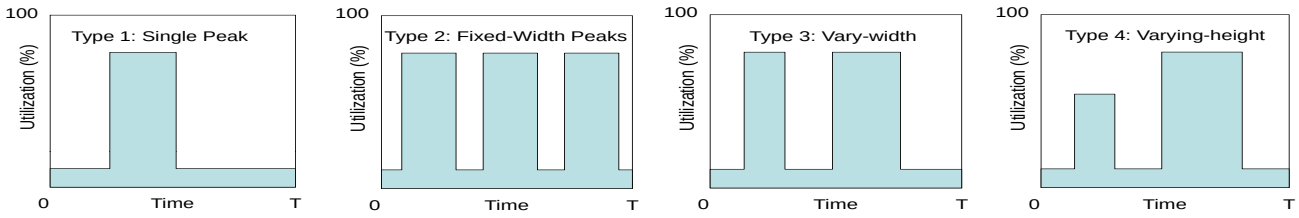


Fig. 8. VM resource utilization from PlanetLab trace.

C. VM Resource Utilization Pattern Detection

The previous section shows the existence of similar resource utilization patterns of VMs running the same job. Given



(a) Type 1: Single peak (b) Type 2: Fixed-width peaks (c) Type 3: Varying-width peaks (d) Type 4: Varying height peaks

Fig. 9. Time-varying resource utilization classification.

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 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 [18] as shown in Figure 9. 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.

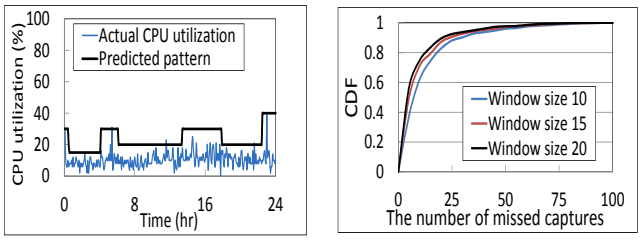
Next, we introduce how to detect the resource utilization pattern for a VM. The cloud records the resource utilizations of the VMs of a tenant. If the job on a VM is a short-term job (e.g., MapReduce job), the cloud records the entire lifetime of the job. If the job on a VM is a long-term job (e.g. Web 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.

Algorithm 1 VM resource demand pattern detection.

1: **Input:** $\mathcal{D}_i(t)$ $(i = 1, 2, ..., N)$: 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: $\mathcal{E}(t_j) \leftarrow$ LowPassFilter($\mathcal{E}(t_j)$) for each time t_j
7: */* Use sliding window to derive pattern */*
8: $\mathcal{P}(t_j) = \max_{t_j \in [t_j, t_j + Window]}\{\mathcal{E}(t_j)\}$ for each time t_j
9: */* Round the resource demand values */*
10: $\mathcal{P}(t_j) \leftarrow$ Round($\mathcal{P}(t_j)$) for each time t_j
11: **return** $\mathcal{P}(t)$ $(t = T_0, ..., T_0 + T)$

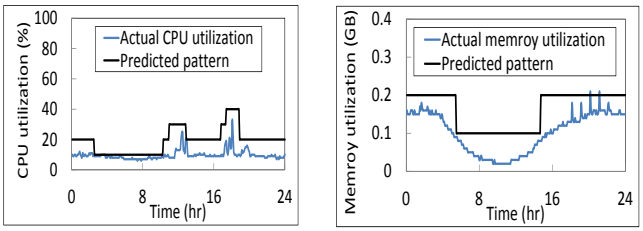
Let $\mathcal{D}_i(t)$=$(D_i^1(t), .., D_i^d(t))$ be the actual d dimension resource demands of VM i at time t. Given the resource demands of a set of VMs running the same job from a tenant, denoted by $\mathcal{D}_i(t)$ $(t$=$T_0, ..., T_0 + T,\ i$=$1, 2, ..., N)$, our pattern detection algorithm finds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

Algorithm 1 shows how to generate the resource demand



(a) Actual and predicted CPU utilizations (b) CDF of # of missed captures using the PlanetLab trace.

Fig. 10. Pattern detection using the PlanetLab trace.



(a) Actual and predicted CPU utilizations (b) Actual and predicted memory utilizations

Fig. 11. Pattern detection using the Google Clusster trace.

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 $Window$ to find the envelop of $\mathcal{E}(t)$ (Line 8). Finally, it rounds the demand values (Line 10).

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 10(a) displays the actual VM CPU utilization and the predicted pattern generated by our pattern detection algorithm using the PlanetLab trace. Figure 11(a) and Figure Figure 11(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.

When the real VM CPU request from the trace is greater than the predicted value, we say that a *missed capture* occurs. Figure 10(b) and Figure 12 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.

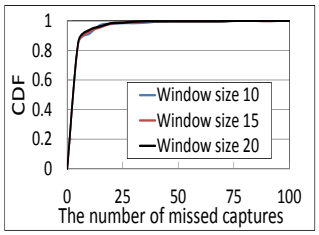


Fig. 12. CDF of # of missed captures using the Google Clusster trace.

D. Initial VM Allocation Policy

The goal of our initial VM allocation policy 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 [19], 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 [19]. 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 2 and 3 in Section II.

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

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

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

$$R_j^k(t) = 1 - U_j^k(t) = 1 - \frac{1}{H_j^k} \sum_{i=1}^{N} D_i^k(t). \quad (2)$$

Algorithm 2 Pseudocode for initial VM allocation.

```
Input: P_i(t): Predicted resource demand of requesting VM i
       R_j(t): Residual resource capacity of m host candidates
Output: Allocated host of the VM
    M=Double.MAX_VALUE //initialize the distance M
    for j = 1 to m do
        if CheckValid(P(t),R_j(t))==false then
            continue
        else
            for k = 1 to d do
                E_j^k = E_j^k + 1/(T·H_j^k) ∫_{T_0}^{T_0+T} P^k(t)dt
                M_j+ = {w_k(1 − E_j^k)}^2
            end for
            if M_j<M then
                M=M_j
                AllocatedHost=host j
    end for
    return AllocatedHost

function CheckValid(P(t),R_j(t)):
    for k = 1 to d do
        for t = T_0 to T_0 + T do
            if F_ij^k(t) > R_j^k(t) (Eq.(3)==false)
            return false
        end for
    end for
    return true
```

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 and for each resource k:

$$F_{ij}^k(t) \leq R_j^k(t), \ t = T_0, ..., T_0 + T, \ k = 1, 2..., d. \quad (3)$$

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:

$$E_j^k = \frac{1}{T \cdot H_j^k} \int_{T_0}^{T_0+T} \sum_{i=1}^{N} D_i^k(t)dt. \quad (4)$$

We use a norm-based greedy algorithm [20] 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):

$$M_j = \sum_{k=1}^{d} \{w_k(1 - E_j^k)\}^2, \quad (5)$$

where w_k is the assigned weight to resource k. 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). 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 policy. This policy 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)) 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. (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.

III. 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 [17] and Google Cluster [16]. 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 [21] and CloudScale [22], 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.

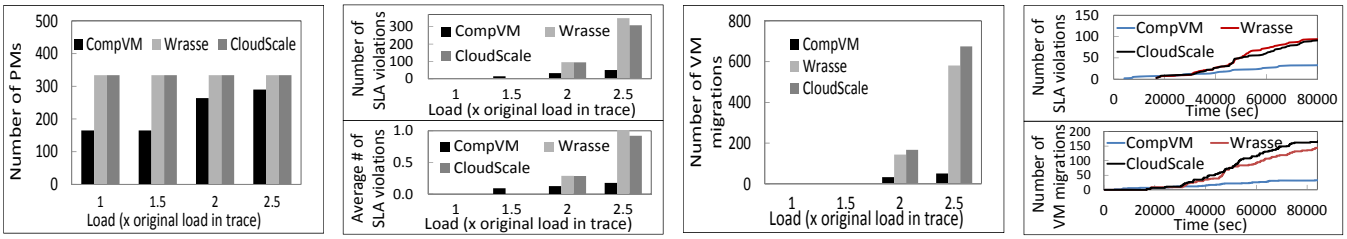
- *The number of PMs used.* This metric measures the energy efficiency of VM allocation mechanisms.
- *The number of SLA violations.* This is the number of occurrences that a VM cannot receive the required amount of resource from its host PM.
- *Average number of SLA violations.* This is the average 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 VM allocation mechanisms that required to satisfy VM demands and avoid SLA violations.

A. Performance with Varying Workload

Figure 13 and Figure 14 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.

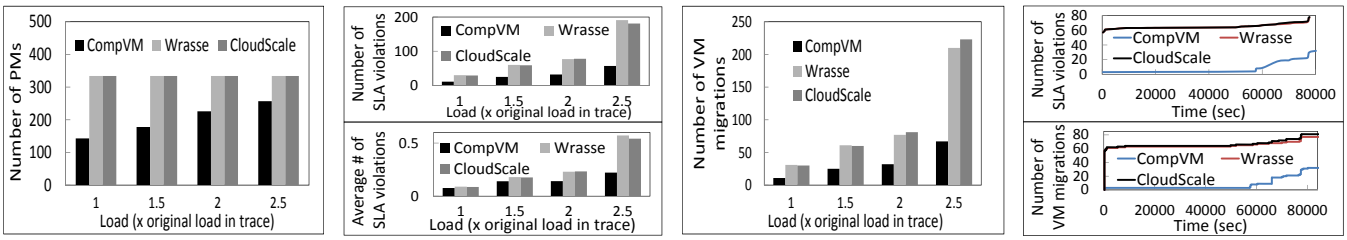
Figure 13(a) and Figure 14(a) show the total number of PMs used, which follows CompVM<CloudScale=Wrasse. CloudScale and Wrasse aim to avoid overloading each PM in initial VM placement and subsequent VM migration at each time point. This may result in some PMs that fully utilize one resource but under-utilize 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.

Figure 13(b) and Figure 14(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



(a) The number of PMs used (b) Total/average # of SLA violations (c) The number of VM migrations (d) # of SLA violations and migrations

Fig. 13. Performance under different workloads using the PlanetLab Trace.



(a) The number of PMs used (b) Total/average # of SLA violations (c) The number of VM migrations (d) # of SLA violations and migrations

Fig. 14. Performance under different workloads using the Google Cluster Trace.

CompVM's higher performance in energy efficiency and SLA guarantees.

Figure 13(c) and Figure 14(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 13(b) and Figure 14(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 13(d) and Figure 14(d) show the accumulated number of SLA violations and VM migrations over time, respectively. In Figure 13(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 14(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.

B. Performance with Varying Number of VMs

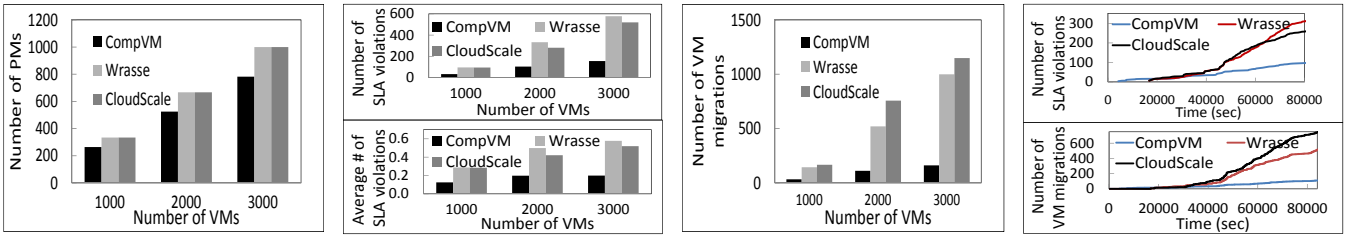
Figure 15 and Figure 16 present the performance of the three methods when the number of VMs was varied from 1000 to 3000 using the PlanetLab trace and the Google Cluster trace, respectively.

Figure 15(a) and Figure 16(a) show the total number of PMs used to provide service for the corresponding number of VMs. We see the result follows CompVM<CloudScale=Wrasse due to the same reasons as in Figure 13(a) and Figure 14(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 15(b) and Figure 16(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 13(b) and Figure 14(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.

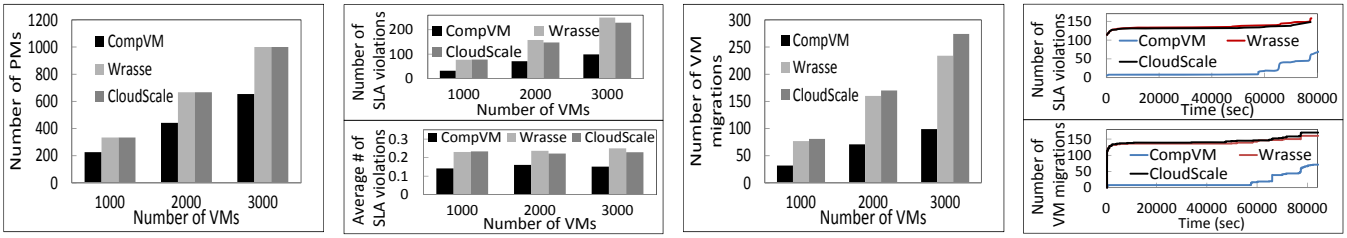
Figure 15(c) and Figure 16(c) show the total number of VM migrations in the three methods. As the number of VMs increases, the number of VM migrations increases due to the increase of SLA violations. CompVM always triggers significantly fewer VM migrations than CloudScale and Wrasse due to its much fewer SLA violations as shown in Figure 15(b) and Figure 16(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.

Figure 15(d) and Figure 16(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 13(d) and Figure 14(d) due to the same reasons.



(a) The number of PMs used (b) Total/average # of SLA violations (c) The number of VM migrations (d) # of SLA violations and migrations

Fig. 15. Performance with different number of VMs using the PlanetLab Trace.



(a) The number of PMs used (b) Total/average # of SLA violations (c) The number of VM migrations (d) # of SLA violations and migrations

Fig. 16. Performance with different number of VMs using the Google Cluster Trace.

IV. 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 [23] 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, 8.0GB virtual disk, running Debian Squeeze 6.0) in the cluster. We used publicly available workload generator *lookbusy* [24] to generate VM workloads.

Figure 17 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 CompVM is able to provide service with fewer number of PMs than Wrasse and CloudScale while ensures SLA guarantees.

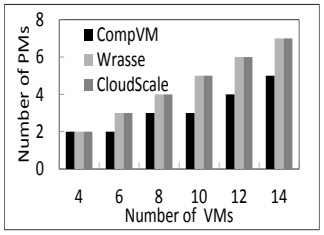
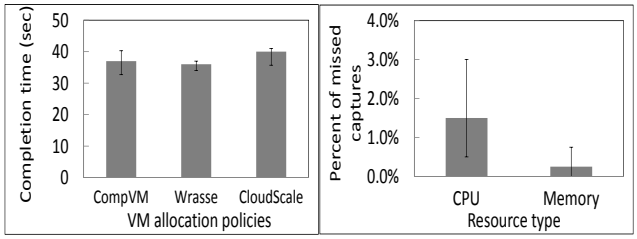


Fig. 17. # of PMs in testbed.

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 I. We see that CompVM uses fewer PMs than Wrasse and CloudScale. During the experiment, no SLA violations were detected in all three methods. Figure 18(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.

TABLE I
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



(a) Job completion time (b) % of missed captures in CompVM

Fig. 18. Performance of running WordCount job on the real-world testbed.

Figure 18(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.

V. Related Work

Recently, many static and dynamic VM allocation strategies have been proposed [2]. Static provisioning [3]–[7] allocates physical resources to VMs only once based on static VM resource demands. For example, Srikantaiah *et al.* [7] 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 [8]–[13] 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. Some works [22], [25] predict resource demands for VM migration to avoid SLA violation in the future. 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 [5], [6], [18]. Oktopus [5] provides static bandwidth reservations throughout the network. Popa *et al.* [6] 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.* [18] provides bandwidth provisioning using predicted bandwidth utilization profile. Different from these works, we focus on consolidating VMs that have demands on multi-resources rather than a single resource.

VI. Conclusions

In this paper, we propose an initial VM allocation mechanism for cloud datacenters that consolidates complementary VMs with spatial/temporal-awareness. This mechanism consolidates complementary VMs into one PM, so that in each dimension of the multi-dimensional resource space, the sum of the resource consumption of the VMs nearly reaches the capacity of the PM during the VM lifetimes. Specifically, given a requested VM, our mechanism predicts the resource demand pattern of this VM, and then finds a PM that has a remaining resource pattern complement to the VM resource demand pattern, i.e., the PM has the least residual capacity in each resource dimension big enough to hold this VM. As a result, our mechanism helps fully utilize the cloud resources, and reduce the number of PMs needed to satisfy tenant requests. It also reduces the numbers of subsequent VM migrations and SLA violations. These advantages have been verified by our extensive simulation experiments based on two real traces and real-world testbed experiments running a MapReduce job. In our future work, 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.

Acknowledgements

This research was supported in part by U.S. NSF grants IIS-1354123, CNS-1254006, CNS-1249603, CNS-1049947, CNS-0917056 and CNS-1025652, Microsoft Research Faculty Fellowship 8300751.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, B. J., and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, hype, and reality for delivering computing as the 5th utility," *FGCS*, 2009.

[2] 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.

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

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

[15] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach." in *Proc. of NOMS*, 2012.

[16] "Google cluster data," https://code.google.com/p/google clusterdata/.

[17] 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.

[18] 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.

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

[20] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing." Microsoft Research, Tech. Rep., 2010.

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

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

[23] "Xenapi," http://community.citrix.com/display/xs/Down load+SDKs.

[24] "lookbusy," http://devin.com/lookbusy/.

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