# Considering Resource Demand Misalignments To Reduce Resource Over-Provisioning in Cloud Datacenters

Liuhua Chen
Department of Electrical and Computer Engineering
Clemson University, Clemson, 29634
Email: liuhuac@clemson.edu

Haiying Shen
Department of Computer Science
University of Virginia, Charlottesville, 22904
Email: hs6ms@virginia.edu

***Abstract*—Previous resource provisioning strategies in cloud datacenters allocate physical resources to virtual machines (VMs) based on the predicted resource utilization pattern of VMs. The pattern for VMs of a job is usually derived from historical utilizations of multiple VMs of the job. We observed that these utilization curves are usually misaligned in time, which would lead to resource over-prediction and hence over-provisioning. Since this resource utilization misalignment problem has not been revealed and studied before, in this paper, we study the VM resource utilization from public datacenter traces to verify the existence of the utilization misalignments. Then, to reduce resource over-provisioning, we propose three VM resource utilization pattern refinement algorithms to improve the original generated pattern by lowering the cap of the pattern, reducing cap provision duration and varying the minimum value of the pattern. These algorithms can be used in any resource provisioning strategy that considers predicted resource utilizations of VMs of a job. We then adopt these refinement algorithms in an initial VM allocation mechanism and test them in trace-driven experiments and real-world cluster experiments. The experimental results show that each improved mechanism can increase resource efficiency up to 74%, and reduce the number of PMs needed to satisfy tenant requests up to 47% while conforming the SLO requirement.**

## I. Introduction

The rapid development of cloud computing brings about the requirement of high resource utilizations in big datacenters in order to save energy consumption. Maximizing energy efficiency and resource utilization while satisfying Service Level Objective (SLO) [2] for tenants require effective management of resource provisioning. Existing works for improving resource utilization in cloud datacenter mainly focus on Virtual Machine (VM) consolidation, i.e., an approach for efficient usage of computer server resources in order to reduce the total number of servers used. Due to the largely oversubscribed nature of today's datacenters [12], resources such as CPU and bandwidth can become scarce resources shared across many tenants. When VMs with intensive resource consumptions are located in the same physical machine (PM), they compete for the scarce resources, which may lead to extended execution time and violations of SLOs [23].

Considerable research efforts have been devoted to effective resource provisioning. Some works [7], [8], [13], [15]–[18], [21], [22], [30], [32], [34] show that the VMs (or tasks) of the same job (or application) share similar resource utilization patterns (e.g., the patterns of the two VMs in Figure 1) and use the derived pattern of VMs (or tasks) for resource provisioning for each VM (or task) of the job (or application) to increase resource utilization. As the derived patterns are used for resource provisioning for both VMs and tasks, we do not specifically distinguish VMs and tasks in the following discussions. The pattern derivation of a job's VM is always conducted based on the historical resource utilization traces of many VMs of this job using techniques such as fast fourier transform [10]. It first finds the maximum demand at each time to get the envelop, and then smoothes the envelop [7]. However, we found that the utilization curves for different VMs of the same job may be misaligned in time, which would generate a pulse wider than the actual pulse in the pattern (e.g., the blue part in Figure 1). We use *pulse deviation* between VMs to measure the demand misalignment, which is defined as the time difference of the same rising or falling edges of the pulses of the VMs of a job. Using such a pattern to guide resource provisioning will lead to resource over-provisioning. For example, in Figure 1, if the actual demand is similar to trace 1, the provisioned resource from $t_3$ to $t_4$ is wasted. However, previous resource provisioning strategies neglect these resource utilization misalignments, which would lead to low resource efficiency. Here, resource efficiency is defined as the ratio between utilized and allocated amount of resource during the provision time. We also implemented a previous resource provisioning strategy in [7], and conducted experiments with Google Cluster trace [11] and PlanetLab trace [6]. This algorithm generates the pattern based on the

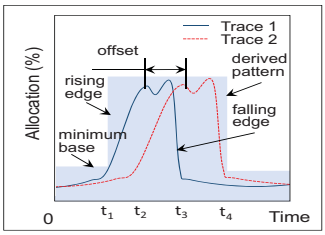



Fig. 1. Resc. emand misalignment.

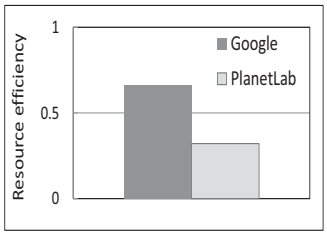



Fig. 2. Resource efficiency.

978-1-5090-5336-0/17/$31.00 ©2017 IEEE

maximum utilization among a group of similar VM resource utilization traces at each time point, and hence the misalignments of the traces tend to yield a pattern with a pulse width larger than the actual pulse width. Figure 2 shows that it can only achieve resource efficiencies of 66% and 32% for the two traces, respectively.

This resource utilization misalignment problem has not been revealed and studied before, so in this paper, we study this problem through measuring the VM resource utilization from public datacenter traces. Our study verifies the existence of misalignments of the resource utilization. In order to improve resource utilization in resource provisioning, we propose three VM resource utilization pattern refinement algorithms that improve the original generated pattern by lowering the cap of the pattern, reducing cap provision duration and varying the minimum value of the pattern, respectively. The refined utilization patterns will be used for resource provisioning.

The time sharing resources (e.g., CPU, bandwidth) have a feature that they can be elastically provided to a VM. That is, the amount of resource allocated to a VM within a short time period (e.g., 1 second) can be elastic and will not obviously affect the job completion time in the VM as long as the total amount allocated to the VM is no less than the required amount within the required time period. The first algorithm lowers the cap of a pulse in the original pattern, so that the amount of provisioned resource during the pulse period exactly equals the demanded resource amount. The second algorithm reduces the provision duration of the cap so that the actual subsequent resource utilization pattern matches the predicted pattern. As shown in Figure 1, each pattern has a minimum base value. The third algorithm finds the minimum base value that leads to the maximum resource efficiency to refine the original pattern.

The contribution of this paper can be summarized as:

- This work is the first that studies the resource demand misalignment of the VMs for the same job. We study the VM resource utilization from public datacenter traces and find that different VMs running the same job exhibit similar periodical resource utilization patterns, but their resource utilization curves exhibit misalignments in time.
- This work is the first that refines the resource utilization patterns in pattern derivation to avoid resource over-provisioning. To avoid overestimation in generated resource utilization pattern caused by the misalignments, we propose three algorithms to refine the resource utilization patterns.
- We apply our three algorithms to the predicted patterns in the VM allocation mechanism [7]. We conduct comprehensive trace-driven simulation and real-world cluster experiments to measure this mechanism with and without each of our algorithms. Experimental results show that the allocation mechanism based on the refined patterns significantly reduces the number of PMs and increases resource efficiency while conforming the SLO requirement.

Through this work, we want to show that there exist demand misalignments among VMs for the same job in some applications and environments. Therefore, if a cloud runs many jobs that have demand misalignments, which leads to significant resource over-provisioning, the cloud can use our proposed solutions to reduce resource over-provisioning. The rest of the article is organized as follows. Section II studies the VM resource utilization from public datacenter traces to verify the existence of the misalignment feature of the resource utilizations of the VMs of the same job. Section III presents the rationale of pattern refinement and three refinement algorithms. Section IV evaluates our algorithms in trace-driven simulation experiments. Section V evaluates our algorithms in a real-world testbed. Section VI presents the related work. Finally, Section VII summarizes the paper with remarks on our future work.

## II. Trace Study

In this section, we statistically study the VM resource utilizations from public datacenter traces including Google Cluster trace and PlanetLab trace. We aim to find the answers for the following questions.

- Whether VMs running the same job (or application) have similar resource utilization patterns in terms of magnitude and the timing of demand arrivals?
- Whether the resource utilization misalignments widely exist in VMs running the same job (or application)?
- Whether the patterns generated by a previous pattern detection algorithm [7] tend to generate low resource efficiency?

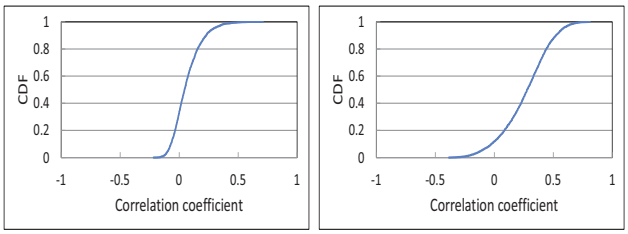
### A. Google Cluster Trace

We first analyze the resource utilization from the Google Cluster trace [11]. The Google Cluster trace records the CPU and memory resource usages on a cluster of about 11000 machines from May 2011 for 29 days. In this measurement, we randomly selected 100 jobs with 29920 tasks in total. For each job, we found all of its tasks from the trace and parsed the CPU and memory utilization of these tasks during this period. We calculated the statistical correlation coefficient (denoted by $c_r$) for each pair of the task resource utilization traces $x$ and $y$ of the same job to show their similarities.

$$c_r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \cdot \sum_{i=1}^{n}(y_i - \bar{y})^2}} \quad (1)$$

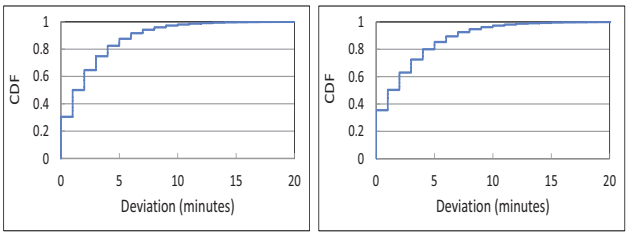
where $x_i$ and $y_i$ are the utilization values at position $i$ in the corresponding trace, $\bar{x}$ and $\bar{y}$ are the average utilizations of the corresponding trace and $n$ is the total number of positions. The correlation coefficient illustrates a quantitative measure of the correlation (i.e., statistical relationships) between the two utilization traces. 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 similarity between the two traces.

Figure 3 shows the cumulative distribution functions (CDF) of task pairs corresponding to the correlation coefficient. Figure 3(a) shows the results from the CPU utilization trace and Figure 3(b) shows the results from the memory utilization trace. For CPU utilization, 20% of the task pairs have correlation coefficient higher than 0.3. For memory utilization, 20% of the task pairs have correlation coefficient higher than 0.5.




(a) Google CPU trace. (b) Google memory trace.

Fig. 3. CDF of correlation coefficient of Google trace.




(a) Google CPU trace. (b) Google memory trace.

Fig. 4. CDF of pulse deviation of Google trace.

These results indicate that tasks running the same application may not have similar resource utilization patterns. This might be caused by the reason that the traces are misaligned in time, that is, the exact timing of rising and falling of the resource demands may not be exactly the same, though their patterns in one seasonal period are similar as observed in previous works [7], [13], [15], [18], [30]. In order to study how much these resource utilizations are misaligned, we conducted experiments to measure the pulse deviation of each pair of the resource utilizations for the same job. Figure 4 shows the CDF of the task pairs corresponding to the absolute pulse deviation. The jobs being studied have an average running time of around 100 minutes. We see that the resource utilizations have pulse deviation spanning from 0 to 10 minutes. Only 30% of the task pairs have pulse deviation 0. A majority (e.g., 70%) of the task pairs have absolute pulse deviation values greater than 5 minutes, indicating that there exist many pulse deviations in the trace and the deviation is relatively high.

### B. PlanetLab Trace

In this section, we analyze the resource utilization from the PlanetLab trace [6]. 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 experiment, we selected VM CPU utilization time series from the trace, and categorized the VMs running the same job into one group. We identified the VMs for the same job by the names of the trace file. For example, trace files with the same file name are VMs that run the same job in different places and times. Figure 5 shows the CDF of VM pairs corresponding to the correlation coefficient. We see that around 70% (e.g., from 0.2 to 0.9) of the VM pairs have correlation coefficient in the range from 0 to 0.2, which indicates that the similarity between VMs running the same job in PlanetLab trace is similar to Google trace. It confirms the conjecture that there might exist

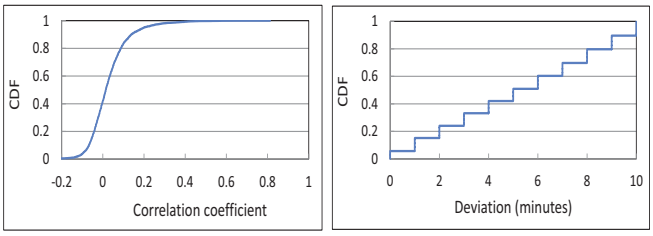



Fig. 5. CDF of correlation coefficient of PlanetLab trace.

Fig. 6. CDF of pulse deviation of PlanetLab trace.

resource utilization misalignments since their patterns in one seasonal period are similar as observed in previous works [7], [13], [15], [18], [30]. Figure 6 shows the CDF of the VM pairs corresponding to the pulse deviations. The jobs being studied have an average running time of around 100 minutes. We see that the resource utilizations have pulse deviation spanning from 0 to 10 minutes. Only 5% of the VM pairs have pulse deviation 0. This result confirms that there exist many pulse deviations in the utilization traces and the pulse deviation can be high.

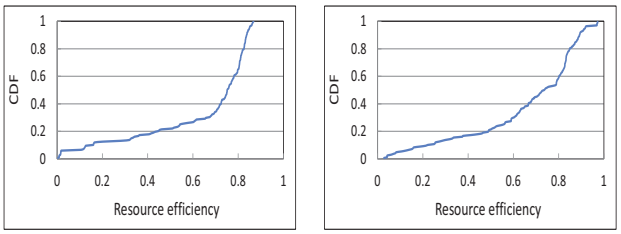
**Algorithm 1:** VM resource demand pattern detection.

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: $\mathcal{E}(t_j) \leftarrow$ 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: $\mathcal{P}(t_j) \leftarrow$ Round($\mathcal{P}(t_j)$) for each time $t_j$
11: **return** $\mathcal{P}(t)$ $(t = T_0, ..., T_0 + T)$

### C. Resource Efficiency

In the previous predictive-based resource provisioning methods, the resource demand pattern of a job's VM is derived from the demand patters of multiple VMs of this job. We take Algorithm 1 in [7] as an example. We use $\mathcal{D}_i(t)$ to denote the amount of resource demand of VM $i$ among $N$ VMs at time $t$. 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).

In this experiment, we used Algorithm 1 to determine the resource demand pattern and evaluated its resource efficiency. Specifically, we conducted experiments on predicting VM resource demand pattern based on resource utilization records of a group of VMs running the same application. We randomly selected a number of jobs, derived the CPU utilization of a VM in each job using all of its VMs and compared it with the real utilizations of each VM. The resource efficiency is calculated by dividing the amount of the provisioned resource based on the predicted pattern by the amount of real utilized resource. For example, given the demand time series $\mathcal{D}(t_j)$ $(j = 1, 2, ...)$




(a) CPU efficiency. (b) Memory efficiency.

Fig. 7. CDF of resource efficiency of Google trace.

and allocated resource time series $\mathcal{A}(t_j)$ $(j = 1, 2, ...)$, which is determined by the generated pattern of the algorithm, we need to determine the utilization time series $\mathcal{U}(t_j)$ $(j = 1, 2, ...)$, which is calculated by $\mathcal{U}(t_j) = \mathcal{D}(t_j)$ if $\mathcal{D}(t_j) < \mathcal{A}(t_j)$; $\mathcal{U}(t_j) = \mathcal{A}(t_j)$ if $\mathcal{D}(t_j) \geq \mathcal{A}(t_j)$. The resource efficiency is calculated by $\frac{\sum \mathcal{U}(t_j)}{\sum \mathcal{A}(t_j)}$. Finally, these resource efficiencies of all VMs are used to plot the CDF figure.

Specifically, for the Google Cluster trace and PlanetLab trace, we randomly selected 100 and 1000 jobs, and tested the resource efficiency of 1550 and 4695 VMs, respectively. Figure 7(a) and Figure 7(b) show the CDF of tasks corresponding to the resource efficiency of the Google Cluster trace in terms of CPU utilization and memory utilization, respectively. We see that around 80% of the tasks have resource efficiencies smaller than 0.8 for CPU utilization, and 80% of the tasks have resource efficiencies smaller than 0.7 for memory utilization. Similarly, Figure 8 shows the CDF of the VMs corresponding to the resource efficiency of the PlanetLab trace. We see that 80% of the results have resource efficiencies smaller than 0.5. The measurement results indicate that there is a large amount of resource over-provisioning and the resource efficiencies of the previous predictive-based resource allocation algorithm can be further improved.

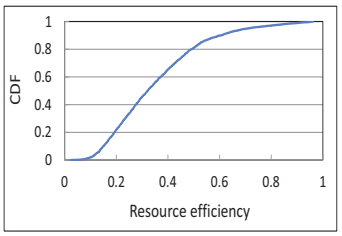
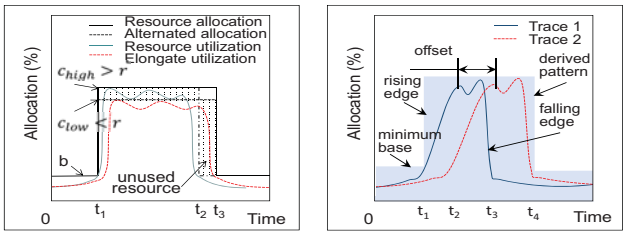



Fig. 8. CDF of resource efficiency of Planetlab trace.

## III. Pattern Refinement Algorithms

In the above section, we verified that the resource utilization patterns of multiple VMs of a job may have misalignments in time. Such misalignments may lead to resource over-provisioning and low resource efficiency. Many types of hardware resources (e.g., CPU, bandwidth and I/O resources) are shared by VMs in temporal manners, that is, VMs take turns to use the resources. This time sharing feature enables elastic resource provisioning. In Section III-A, we show that the elastic resource provisioning makes it possible for the original pattern detection algorithm to refine the pattern in order to further reduce provided resource and increase the resource efficiency. In Section III-B, we propose three refinement algorithms based on Algorithm 1. The first and second refinement algorithms leverage the elastic provisioning feature of resource to further improve the resource efficiency. The third refinement




(a) Elastic resource provisioning. (b) Rising, falling edges and pulse deviation.

Fig. 9. Resource demand misalignment.

algorithm refines the generated pattern by varying the shape of the original pattern until it achieves the highest resource efficiency. The three algorithms are independent to each other.

### A. Elastic Resource Provisioning

In this section, we discuss the feature of resource provisioning for time-sharing resources. This feature lays the foundation for our proposed pattern refinement algorithms. The time sharing resources have a feature that they can be flexibly provided to VMs. Take CPU resource as an example, VMs take turns to use the physical processing core. Suppose a VM requires 5 CPU time slots during a 3 seconds time period to complete its job. The resource provider can either schedule (1 slot, 2 slots, 2 slots) or (2 slots, 2 slots, 1 slot) for the VM in the three consecutive seconds. That means the amount of resource (e.g., CPU time slots) allocated to a VM within a short time period (e.g., 1 second) can be elastic and will not obviously affect the job completion time in the VM as long as the total amount allocated to the VM is the same (e.g., 5 slots) within its required time period (e.g., 3 seconds). The completion time of a job running in a VM is estimated by $C \times T \times I$, where $C$ is the average number of cycles per instruction, $T$ is the time per cycle, and $I$ is the number of instructions per job.

We define the fraction of CPU time (and hence the number of cycles) that a VM is allowed to use within a unit time period as its cap. Within a unit time period, as we limit the cap, the CPU time and hence the number of cycles received by the VM is decreased, resulting in an increase of the time per cycle ($T$) of the VM. As a result, the limitation of the cap leads to an elongation of the completion time. On the other hand, the completion time of a VM's job is the same as long as the total amount of time slots allocated to the VM (i.e., the number of CPU cycles) is no less than the requested amount during the required time period. These two features enable us to lower the pulse of the original pattern generated by Algorithm 1 to reduce the amount of provisioned resource to improve resource efficiency.

As shown in Figure 9(a), suppose a job requires $r$ amount of resource that can complete its work using time $T_{high}$ (from $t_1$ to $t_2$). Based on the original pattern of this job, Algorithm 1 suggests providing $c_{high}$ ($c_{high} > r$) resource for $T_{pro}$ time (from $t_1$ to $t_3$). Since $c_{high} > r$, the job will consume $r$ amount of resource and complete within time $T_{high}$ (from $t_1$ to $t_2$). In this case, the provisioned resource from $t_2$ to

$t_3$ is wasted. In order to improve resource efficiency, we can limit the provision resource amount to $c_{low}$ ($c_{low} < r$) that makes the job complete using time $T_{pro}$ (at $t_3$). Since $c_{low} < r$, the job is allowed to consume $c_{low}$ amount of resource. Due to the insufficient resource, the job will prolong the completion time and complete in time $T_{low}$ (from $t_1$ to $t_3$) when all required amount of resource is received. That is, the cumulative resource consumption $r \times T_{high} = c_{low} \times T_{pro}$ or when the sizes of the two shadow parts in Figure 9(a) equal to each other, i.e.,

$$(c_{high} - c_{low}) \times \ T_{pro} = (c_{high} - b) \times (T_{pro} - T_{high}), \quad (2)$$

where $b$ is the base value of the provisioned resource which is the minimum resource amount provided to the VM. Then, the resource efficiency is improved from $\frac{r \times T_{high}}{c_{high} \times T_{pro}}$ to $\frac{c_{low} \times T_{pro}}{c_{low} \times T_{pro}} = 1$.

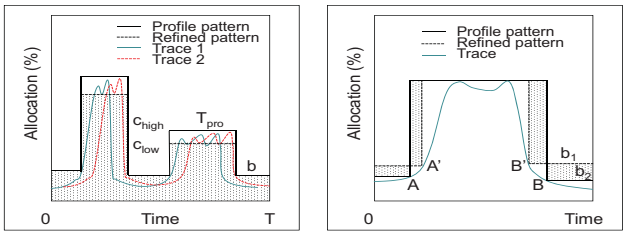
### B. Pattern Refinement Methodology

Using the elastic resource provisioning feature, we propose three pattern refinement algorithms to improve the resource efficiency of the generated patterns from Algorithm 1. The algorithms allow the cloud provider to provide resource efficiently and to potentially host more VMs in the datacenter. We present the details of each algorithm in the following.

*1) Lowering Cap:* Algorithm 1 generates the pattern for one VM of a job from historical utilizations of multiple VMs of the job. Since these VMs have pulse deviations (as confirmed in Section II), the algorithm will result in a pattern with an expansion on each pulse width, which is larger than the width of the actual demand pulse of a single VM. As a result, the resulting pattern tends to have low resource efficiency since a VM may not fully use its provided resource based on the pattern, which leads to resource over-provisioning. Inspired by the time sharing feature of resource as explained previously, we propose pattern refinement algorithms to improve resource efficiency. For every pulse in the generated pattern from Algorithm 1, we can further lower the cap to a level that saves the over-provided resource due to trace pulse deviation. The starting and ending time of the pulses in a pattern can be detected by finding the time for each pair of rising and falling edges as we explained previously (Figure 9(b)). The amount to lower the cap can be calculated based on Equation (2). For example, as shown in Figure 10(a), suppose traces 1 and 2 have pulse deviation $s$, the original pattern has width $T_{pro}$ and the cap before refinement is $c_{high}$, then we can lower the cap to $c_{low}$ so that

$$(c_{high} - c_{low}) \times \ T_{pro} = s \times \ (c_{high} - b). \quad (3)$$

Algorithm 2 shows how to refine demand pattern by reducing the cap of the original pattern derived from Algorithm 1. The algorithm first finds the envelop of the time series of resource utilizations of VMs $\mathcal{E}(t_j)$ (Line 3) and derives the resource demand pattern $\mathcal{P}(t)$ (Line 4) based on Algorithm 1. Then, it calculates the pulse deviations of each pair of the VMs based on the first rising edges as discussed in Section II (Line 5), and then selects the maximum pulse deviation (Line 7). The algorithm calculates the width of the pulse of the derived pattern $\mathcal{P}(t)$ (Line 8) by measuring the duration




(a) Pattern refinement by lowering the cap. (b) Pattern refinement by varying the base provision.

Fig. 10. Pattern refinement.

**Algorithm 2:** Demand pattern refinement by lowering cap.

1: **Input:** $\mathcal{D}_i(t)$ $(i = 1, 2, ..., N)$: Resource demands of a set of VMs
2: **Output:** $\mathcal{P}'(t)$: Refined resource demand pattern
3: Find the maximum demand at each time $\mathcal{E}(t_j)$
4: Derive resource demand pattern $\mathcal{P}(t)$ using Algorithm 1
5: Calculate the pulse deviation of the VMs
6: **for** every pulse in $\mathcal{P}(t)$
7: Calculate maximum pulse deviation $s$ of the edges
8: Measure the width of cap $T_{pro}$
9: Determine reduction of cap $c_{high} - c_{low}$
10: Update the cap of the current pulse in $\mathcal{P}'(t)$ to $c_{low}$
11: **return** $\mathcal{P}'(t)$

between the time stamps of two consequent rising and falling edges. The algorithm then determines the amount of reduction of cap $c_{high} - c_{low}$ based on Equation (3) (Line 9). Finally, it derives the refined demand pattern by lowering the value of the pulse of $\mathcal{P}(t)$ by the amount of $c_{high} - c_{low}$ (Line 10) and returns the new pattern (Line 11).

*2) Reducing Pulse Width:* Algorithm 2 requires that the pulse demand of a VM arrives at the beginning of the refined pulse (i.e., the rising edge of the pattern). If the pulse demand of a VM arrives later than the refined pattern from Algorithm 2, then the VM cannot receive all its requested resource within the provision time, i.e., the length of the pattern. If the pulse demand of the VM comes after the beginning of the refined pulse, the resource provisioned at the beginning is not fully used by the VM, hence the VM cannot receive its total requested amount of resource by the end of the pulse. To avoid this problem, we propose another algorithm, which reduces the duration of each pulse of the pattern to avoid the over-provisioning.

We use Figure 11 to demonstrate the impact of such reducing on the extension of job execution time of the VM. Given a sufficient amount of resource, a VM has a resource utilization profile as shown in Figure 11(a), where the blue area indicates the provisioned resource and the curve indicates the used resource. Suppose from time $t_1$ to $t_3$, we reduce the amount of provisioned resource from cap value $c_{max}$ to base value $b$ as shown in Figure 11(b), which results in an under-provisioning and hence a prolonged job execution time. By $t_3$, the provisioned amount of resource can only satisfy the original demands that arrive between time $t_1$ and time $t_2$. As a result, the demand profile is postponed by $t_3 - t_2$. After $t_3$, as provision increases to $c_{max}$, the demand profile $\mathcal{E}(t_j)$ follows

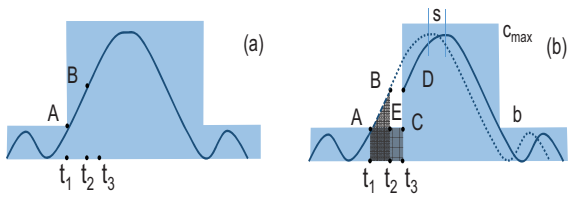



Fig. 11. Postponing the cap provision and reducing the cap width.

the shape of the VM's original demand profile without an expansion as shown in Figure 11(b) but with a delay $t_3 - t_2$. Then, the provisioned resource can be fully utilized as the falling edges of the provisioned resource and used resource are overlapped. Given enough provisioned resource (i.e., without this algorithm), the original demand profile was developing from point $B$ as indicated by the dashed curve. After reducing the provisioned resource (i.e., with this algorithm), the demand at point $B$ will not receive its requested amount of resource until point $D$. After that, the demand profile follows the shape of the original profile without any extension or deformation as indicated by the solid curve. In conclusion, the reduced provisioning results in a delay of the utilization profile.

**Algorithm 3:** Demand pattern refinement by reducing cap width.

```
1: Input: D_i(t) (i = 1, 2, ..., N): Resource demands of a set of VMs
2: Output: P'(t): Refined resource demand pattern
3:     Find the maximum demand at each time E(t_j)
4:     Derive resource demand pattern P(t) using Algorithm 1
5:     for every pulse in P(t)
6:         Calculate maximum pulse deviation s of the edges
7:         t_i ← t_1, Sum ← 0
8:         while Sum < sb do
9:             Sum ← Sum + (P(t_i) − b)
10:            t_i ← t_i + 1
11:        d ← t_i − t_1
12:        Update P'(t) ← b, for t_1 ≤ t < t_1 + d
13:    return P'(t)
```

Therefore, we can reduce the provisioned resource of a pattern by reducing the amount of resource from $c_{max}$ to $b$ in the beginning of provisions. A question is how to find the time latency to change $c_{max}$ in the original pattern to $b$ (i.e., the postponing latency). We notice that the area size of $t_1t_2BA$ and the area size of $t_1t_3CA$ are equal to each other:

$$\int_{t_1}^{t_2} \mathcal{P}(t)dt = b(t_3 - t_1) \tag{4}$$

where $\mathcal{P}(t)$ is the pattern function, and $t_3 - t_2 = s$. Suppose $d = t_3 - t_1$ is the duration that we want to reduce the resource. Considering

$$\int_{t_1}^{t_1+d-s} \mathcal{P}(t)dt = bd \tag{5}$$

we have

$$d = \mathcal{P}^{-1}(b) + s - t_1 \tag{6}$$

As it is not easy to derive $\mathcal{P}^{-1}(b)$ in the algorithm, we develop a practical approach (as described below) in Algorithm 3 to find $d$. Algorithm 3 shows this refinement algorithm to improve resource efficiency based on the above discussion. The algorithm first finds the envelop of these series $\mathcal{E}(t_j)$ (Line 3) and derives the resource demand pattern $\mathcal{P}(t)$ (Line 4) based on Algorithm 1. Next, it calculates the pulse deviations of the VMs between each other and then selects the maximum pulse deviation (Line 6). After that, it determines $d$ based on the pulse deviation $s$ (Lines 7-10). In the algorithm, we iteratively increase the value of $t_2$ from 0, and find the value that makes the area size of $t_1t_2BA$ equal to the area size of $t_1t_3CA$. Finally, the algorithm modifies $\mathcal{P}(t)$ by reducing the provisioned resource amount between time $t_1$ and $t_1 + d$ from $c_{max}$ to $b$ (Lines 11-12), and returns the new pattern $\mathcal{P}'(t)$ (Line 13).

*3) Varying Base Provision:* We refine the original pattern generated by Algorithm 1 by varying the base value $b$ of the original pattern until it achieves the highest efficiency. Different sizes of time windows leads to different base values. As shown in Figure 10(b), two tentative square curve fittings with base resource $b_1$ and $b_2$, respectively, are both feasible solutions for pattern provision. Given a resource utilization profile, the parameters that maximize the resource efficiency can be found by searching through different values of $b$.

**Algorithm 4:** Demand pattern refinement by varying the base provision.

```
1: Input: D_i(t): Resource demands of a set of VMs
2: Output: P'(t): Refined resource demand pattern
3:     Find the maximum demand at each time E(t_j)
4:     Determine P(t) based on Algorithm 1
5:     Find maximum demand c_max
6:     Find minimum demand b
7:     do
8:         b ← b+Δ
9:         Find P'_temp(t) based on b
10:        Measure resource efficiency using E(t_j) and P'_temp(t)
11:        if (efficiency > max)
12:            max ← efficiency
13:            P'(t) ← P'_temp(t)
14:    while (b < c_max)
15:    return P'(t)
```

Algorithm 4 shows the processes to improve the resource efficiency by varying $b$ of a derived pattern. Given a set of VM resource demand time series $\mathcal{D}_i(t)$ as input, the algorithm first finds the envelop of these series $\mathcal{E}(t_j)$ (Line 3) and determines the original resource demand pattern $\mathcal{P}(t)$ based on Algorithm 1 (Line 4). Then, it finds the maximum demand ($c_{max}$) of the pattern (Line 5) and the minimum base value (Line 6). The minimum base can be found by scanning the pattern $\mathcal{P}(t)$ generated by Algorithm 1. The algorithm calculates $\mathcal{P}'_{temp}(t)$ based on varying $b$ from the minimum base value (Line 9). The rationale of varying $b$ from this value is that it is the minimum value that covers all the base demands, as indicated by Algorithm 1. $\mathcal{P}'_{temp}(t)$ is the pattern after the base value is updated in $\mathcal{P}(t)$ and we will explain how to calculate $\mathcal{P}'_{temp}(t)$ later. For example, in Figure 10(b), $\mathcal{P}(t)$ represented by the solid line is changed to $\mathcal{P}'_{temp}(t)$ represented by the dotted line after base value is changed from $b_2$ to $b_1$. The algorithm then measures the resulting efficiency (Line 10). Here, the resource efficiency is calculated by $\frac{\sum \mathcal{E}(t_j)dt}{\sum \mathcal{P}'_{temp}(t)dt}$. Because we do not know the actual resource consumption, we use $\mathcal{E}(t_j)$ as the consumed resource to measure the resource efficiency for comparable comparison

to choose the pattern with the highest resource efficiency. We vary $b$ by increasing $b$ from initial value to the maximum demand $c_{max}$. The algorithm repeats this process with increasing $b$ until $b \geq c_{max}$, and finds the $b$ that leads to the maximum resource efficiency (Lines 8-14). Finally, the pattern that leads to the maximum efficiency is returned (Line 15).

We describe the process of finding $\mathcal{P}'_{temp}(t)$ based on $b$ as follows. For every new value of $b$, we find out all the time stamps $t'_j$s that have $\mathcal{E}(t'_j) = b$ (e.g., points $A'$ and $B'$ in Figure 10(b)). From the original $\mathcal{P}(t)$, we have a series of time stamps $t_j$s indicating the rising and falling of the resource provisioning (e.g., points $A$ and $B$ in Figure 10(b)). The algorithm orders these time stamps with indices starting from zero. As a result, a time stamp with an even index indicates a rising, while a time stamp with an odd index indicates falling. The new pattern $\mathcal{P}'_{temp}(t)$ is generated by changing the provision value from $c_{max}$ to $b$ in the pattern $\mathcal{P}(t)$ for every time period from $t_j$ to $t'_j$ (and from $t'_j$ to $t_j$ for odd indices) (i.e., from points $A$ to $A'$, and from $B'$ to $B$ in Figure 10(b)).

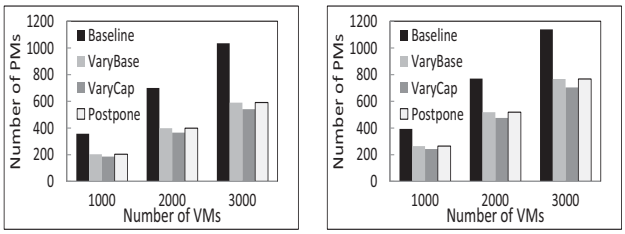
## IV. Trace-Driven Simulation

In this section, we conducted the simulation experiments to evaluate the performance of our proposed pattern refinement algorithms using the Google Cluster trace and PlanetLab trace. We implemented the proposed refinement algorithms in the initial VM allocation mechanism called CompVM [7], denoted as *VaryCap*, *Postpone* and *VaryBase* (initial VM allocation using Algorithm 2, Algorithm 3 and Algorithm 4, respectively). To the best of our knowledge, our work is the first that tries to improving resource efficiency by using refined utilization pattern in resource provisioning. Therefore, there are no state-of-the-art methods for patten refinement for comparison in the performance evaluation.

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 minutes in the pattern detection in CompVM. We compared *VaryCap*, *Postpone* and *VaryBase* with the original CompVM (denoted by *Baseline*). All these methods conduct initial VM allocation.

We used the CloudSim [6] simulator to conduct the simulation. 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. In the simulation, the pattern of each VM is predicted, and the VMs are allocated to the PMs based on their patterns and the allocation algorithm in [7]. Note that the VMs are from different users rather than a single user.

- *The number of PMs used.* It measures the resource efficiency of VM allocation mechanisms to host all VMs.




(a) Google Cluster trace. (b) PlanetLab trace.

Fig. 12. Performance with varying number of VMs.

- *Resource efficiency.* It is the ratio between the utilized and allocated amount of resource during the provision time for each VM.
- *The number of SLO violations.* It is the number of occurrences that a VM cannot receive the required amount of resource from its host PM.

### A. Performance with Varying Number of VMs

We first study the performance of the three algorithms when the number of VMs was varied from 1000 to 3000 using the Google Cluster trace. Figure 12(a) shows the total number of PMs used from the Google Cluster trace, which follows *VaryCap*<*Postpone*≈*VaryBase* <*Baseline*. *VaryBase*, *VaryCap* and *Postpone* reduce the number of PMs of *Baseline* (e.g., $\frac{Baseline-Algorithm}{Baseline}$) by 43%, 47% and 43% for Google Cluster trace. *VaryBase*, *VaryCap* and *Postpone* reduce the number of PMs due to their refined VM patterns, which require relatively less resource than *Baseline*. *VaryCap* further reduces the number because it reduces the cap value of the patterns. *Postpone* is larger than *VaryCap* due to the reason that reducing the pulse length is not as efficient as reducing the cap in providing resource for more VMs, because most of the VM patterns are characterized by a small cap with large width rather than a high cap with small width. The result confirms that the refinement algorithms reduce the amount of provisioned resource and reduces the number of PMs needed to host the VMs, hence achieve higher resource efficiency. Figure 12(b) shows the total number of PMs used from the PlanetLab trace. It shows similar results as Figure 12(a). *VaryBase*, *VaryCap* and *Postpone* reduce the number of PMs of *Baseline* by 32%, 38% and 32%, which again confirms that the refinement algorithms reduce the number of PMs. The numbers from PlanetLab trace are higher than those from Google Cluster trace because the tasks in Google Cluster trace have higher correlation coefficient, and hence the predicted patterns are more accurate.

### B. Resource Efficiency

Figure 13 shows the median, the $10^{th}$ and $90^{th}$ percentiles of resource efficiency of each VM when we applied the algorithms to Google Cluster trace and PlanetLab trace, respectively. The error bars in the figure indicate the $10^{th}$ and $90^{th}$ percentiles. We see that the resource efficiency follows *Baseline*<*VaryBase*<*VaryCap*<*Postpone* in both traces. *VaryBase*, *VaryCap* and *Postpone* improve the resource efficiency of *Baseline* (e.g., $\frac{Algorithm-Baseline}{Baseline}$) by 10%, 12% and 12%

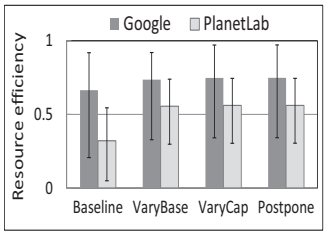



Fig. 13. Resource efficiency.

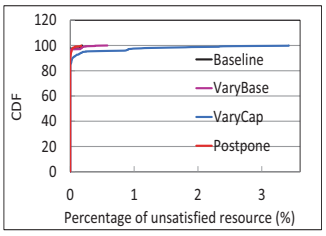



Fig. 14. Ability to satisfy SLO.

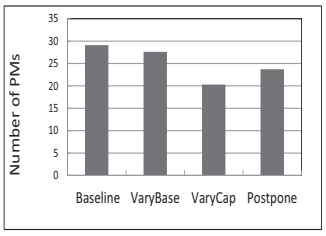



Fig. 15. The number of PM used.

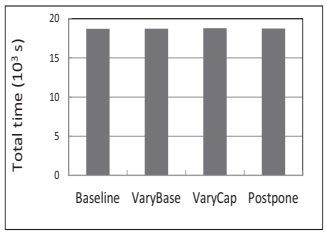



Fig. 16. Total execution time of VMs.

for Google Cluster trace, and by 73%, 74% and 74% for PlanetLab trace, respectively. *VaryBase*, *VaryCap* and *Postpone* outperform *Baseline* due to VM pattern refinements. The refinement algorithms also reduce the variations of the efficiency as indicated by the error bars. *VaryCap* and *Postpone* outperform *VaryBase* because they reduce the height or width of the cap, which makes VMs more likely to be consolidated in a PM and hence improves resource efficiency, while *VaryBase* only tunes the base value which may not greatly improve the resource efficiency. *Postpone* has a similar resource efficiency as *VaryCap* because both of them reduce VM patterns based on the trace deviation.

### C. Performance in SLO Conformance

In this experiment, we define SLO violation as the failure of satisfying the resource demand within the time deadline, which was set to the maximum completion time among the VMs of a job. We tested 640 VMs and found that *Baseline*, *VaryBase*, *VaryCap* and *Postpone* have 12, 15, 74 and 54 violations, and 99%, 98%, 89% and 91% of the VMs satisfy the demands, respectively. Figure 14 shows the CDF of the percentage of VMs with the percentage of the amount of demanded resource that cannot be satisfied by deadline. We see that even though there is a slight increase in the number of violations with each pattern refinement algorithm, the percentage of the amount of unsatisfied resource is very small. It is no more than 0.18%, 0.58%, 3.4% and 0.17% for *Baseline*, textitVaryBase, *VaryCap* and *Postpone*, respectively. To avoid these violations, we can easily add more provisioned resource. If we add more provisioned resources to avoid these violations, the resource efficiency of *VaryBase*, *VaryCap* and *Postpone* still keep the same as in Figure 13 (i.e., 10%, 12% and 12% higher than *Baseline* for the Google Cluster trace and 73%, 74% and 74% for PlanetLab trace) because the amount of the additional provisioned resource is relatively very small. We can use the method in CloudScale [23] to determine the amount of additional provisioned resource.

## V. Small Real-World Cluster Experiments

### A. Performance of Pattern Refinement

In this experiment, we used workloads from the NAS Parallel Benchmark (NPB) suite [3] to run in the VMs. The NPB suite is a small set of programs designed to help evaluate the performance of parallel supercomputers. We used the programs to emulate jobs running in the VMs. We first conducted a profiling run to collect the CPU utilization trace of each NPB programs. In the profiling run, we executed the programs on Palmetto [19], a high-performance computing (HPC) cluster (a 21,546-core 500 tera FLOPS HPC system) and recorded the CPU utilization every 0.01 seconds for every program in every machine. In this case, the programs are provisioned with sufficient CPU resource, and the collected traces are regarded as original trace. We then used the measured utilization profiles in the consequent VM placement experiments.

Figure 15 shows the total number of PMs required to host the VMs. We see that the number of PMs used follows *VaryCap*<*Postpone*<*VaryBase*<*Baseline*. *VaryBase*, *VaryCap* and *Postpone* reduce the PMs due to their refined VM patterns, which require relative less resource than *Baseline*. *VaryCap* further reduces the number because it reduces the cap value of the patterns and hence will lead to a relatively flattened pattern, which makes VMs more likely to be consolidated in a PM. The results are consistent with the simulation results in Figure 12. Figure 16 shows the total execution time of all the VMs. We see that all the algorithms perform similar with a total time around 18700 seconds. These results confirms that the pattern refinement algorithms is efficient in saving resource and hence reducing the number of the PMs while do not significantly degrade the VM performance in terms of job completion time.

## VI. Related Work

Recently, many VM allocation strategies have been proposed [28]. Some of them [4], [5], [20], [25], [31] allocate physical resources to VMs only once based on static VM resource demands. For example, Srikantaiah *et al.* [25] proposed to use Euclidean distance between VM resource demands and residual capacity as a metric for consolidation. Oktopus [4] provides static bandwidth reservations throughout the network. Yu *et al.* [33] considered the problem of scaling up a virtual network abstraction with bandwidth guarantee. Cui *et al.* [9] proposed an efficient and synergistic scheme to jointly consolidate network policies and virtual machines. However, static provisioning cannot fully utilize resources because of time-varying resource demands of VMs. To fully utilize cloud resources, others [1], [14], [24], [26], [27], [29] first consolidate VMs using a simple bin-packing heuristic and manage the resource through live VM migrations, which might result in migration overhead. For example, Sandpiper [29] 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. In order to consider both the current and future state of resource

demand and available capacity in a time period, Chen *et al.* [7] proposed an initial VM allocation mechanism that consolidates complementary VMs with spatial/temporal awareness based on the predicted lifetime resource utilization patterns of VMs. However, the pattern prediction algorithm proposed in this paper generates the pattern for one VM from historical utilizations of multiple similar VMs, but neglects the fact that these utilizations have pulse deviations. As a result, consolidating VMs based on these patterns will result in a waste of resource. Our work refines the resource utilization patterns in pattern derivation to avoid resource over-provisioning.

## VII. Conclusions

This paper is the first work that observes the resource utilization curves of different VMs running the same job exhibit misalignments in time in spite of their similar periodical patterns. Then, generating resource utilization pattern based on the traces of different VMs to guide resource provisioning to each VM will lead to resource over-provisioning and low resource efficiency. In order to improve resource efficiency, we proposed three VM resource utilization pattern refinement algorithms to improve the resource efficiency of the original generated pattern. Specifically, given a originally generated resource utilization pattern, the *VaryCap* algorithm and the *Postpone* algorithm refine the pattern by either lowering the cap of the pattern or reducing the width of the provisioning pulse; and the *VaryBase* algorithm refines the pattern by varying the base value until it achieves the highest efficiency. We then adopted these refinement algorithms in an initial VM allocation mechanism that consolidates VMs for cloud datacenters. The mechanism helps fully utilize the cloud resources, and reduce the number of PMs needed to host all VMs while conforming the SLO requirement. These advantages have been verified by our extensive trace-driven simulation experiments and real-world testbed experiments. In our future work, we will study the feasibility of our proposed algorithms to different types of traces, find the reasons for the misalignments and the scenarios that our algorithms are most suitable.

## Acknowledgements

This research was supported in part by U.S. NSF grants ACI-1661378 and CNS-1254006, and Microsoft Research Faculty Fellowship 8300751. We would like to thank Dr. John Wilkes for his valuable discussions.

## References

[1] E. Arzuaga and D. R. Kaeli. Quantifying load imbalance on virtualized enterprise servers. In *Proc. of WOSP/SIPEW*, 2010.
[2] Service Level Agreements. http://azure.microsoft.com/en-us/support/legal/sla/.
[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Proc. of SC*, pages 158–165, 1991.
[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. of SIGCOMM*, 2011.
[5] U. Bellur, C. S. Rao, and M. K. SD. Optimal placement algorithms for virtual machines. *CoRR*, 2010.
[6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.
[7] L. Chen and H. Shen. Consolidating complementary vms with spatial/temporal-awareness in cloud datacenters. In *Proc. of INFOCOM*, 2014.
[8] L. Chen, H. Shen, and S. Platt. Cache contention aware virtual machine placement and migration in cloud datacenters. In *Proc. of ICNP*, 2016.
[9] L. Cui, R. Cziva, F. P. Tso, and D. P. Pezaros. Synergistic policy and virtual machine consolidation in cloud data centers. In *Proc. of INFOCOM*, 2016.
[10] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Proc. of CNSM*, pages 9–16, 2010.
[11] Google cluster data. https://code.google.com/p/googleclusterdata/.
[12] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *Proc. of SIGCOMM*, volume 39, pages 51–62, 2009.
[13] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. of SIGCOMM*, 2015.
[14] G. Khanna, K. Beaty, and G. Kar. Application performance management in virtualized server environments. In *Proc. of NOMS*, 2009.
[15] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proc. of IMC*, 2013.
[16] Z. Li and H. Shen. Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance. In *Proc. of ICPP*, 2015.
[17] Y. Lin, H. Shen, and L. Chen. Ecoflow: An economical and deadline-driven inter-datacenter video flow scheduling system. In *Proc. of Multimedia*, 2015.
[18] B. Palanisamy, A. Singh, L. Liu, and B. Langston. Cura: A cost-optimized model for mapreduce in a cloud. *Proc. of IPDPS*, 2013.
[19] Palmetto. https://www.palmetto.clemson.edu/.
[20] 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.
[21] A. Sarker, C. Qiu, and H. Shen. A decentralized network with fast and lightweight autonomous channel selection in vehicle platoons for collision avoidance. In *Proc. of MASS*, 2016.
[22] H. Shen, L. Yu, L. Chen, and Z. Li. Goodbye to fixed bandwidth reservation: Job scheduling with elastic bandwidth reservation in clouds. In *Proc. of CloudCom*, 2016.
[23] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. of SOCC*, 2011.
[24] A. Singh, M. R. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proc. of SC*, 2008.
[25] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proc. of HotPower*, 2008.
[26] 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.
[27] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Proc. of Middleware*, 2008.
[28] 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.
[29] 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.
[30] 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.
[31] J. Xu and J. A. B. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Proc. of CPSCom*, 2010.
[32] L. Yan, K. Chen, H. Shen, and G. Liu. MobileCopy: Resisting correlated node failures to enhance data availability in dtns. In *Proc. of SECON*, 2015.
[33] L. Yu and Z. Cai. Dynamic scaling of virtual clusters with bandwidth guarantee in cloud data centers. In *Proc. of INFOCOM*, 2016.
[34] L. Yu, L. Chen, Z. Cai, H. Shen, Y. Liang, and Y. Pan. Stochastic load balancing for virtual resource management in datacenters. *TCC*, 2016.