# Resource Demand Misalignment: An Important Factor to Consider for Reducing Resource Over-Provisioning in Cloud Datacenters

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

***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 and Hadoop benchmark jobs to verify the commonness 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. We then extend these algorithms to further improve the resource efficiency by considering periodical resource demand patterns that have multiple pulses in a 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 testbed experiments. The experimental results show that each improved mechanism can increase resource utilization, and reduce the number of PMs needed to satisfy tenant requests. Also, our extended refinement algorithms are effective in improving resource efficiency of the refinement algorithms.**

***Index Terms*— Resource management, cloud datacenter, load balancing, resource over-provisioning.**

## 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 [1]–[3]. Maximizing energy efficiency and resource utilization while satisfying Service Level Objective (SLO) [4] 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. Due to the largely oversubscribed nature of today's datacenters [5], 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 [6].

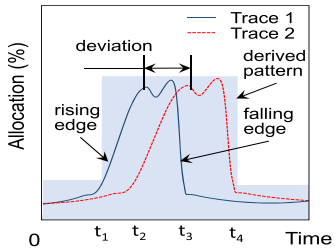



Fig. 1. Demand misalignment.

Considerable research efforts have been devoted to effective resource provisioning. Some works [7]–[11] show that the VMs (or tasks) for the same job share similar resource utilization patterns (e.g., the patterns of the two VMs in Figure 1) and use the derived pattern of VMs for each job for resource provisioning to increase resource utilization. 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 [12]. It first finds the maximum demand at each time to get the envelop, and then smoothes the envelop [9]. However, we found that the utilization curves for different VMs in the same job are 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 [9], and conducted experiments with Google Cluster trace [13] and PlanetLab trace [14]. This algorithm generates the pattern based on the 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.

Manuscript received May 31, 2017; revised February 4, 2018; accepted March 12, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Mascolo. Date of publication May 3, 2018; date of current version June 14, 2018. This work was supported in part by U.S. NSF under Grant OAC-1724845, Grant ACI-1719397, and Grant CNS-1733596, and in part by the Microsoft Research Faculty Fellowship under Grant 8300751. This paper was presented at the Proceedings of the IEEE INFOCOM 2017 [59]. *(Corresponding author: Haiying Shen.)*

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

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

Digital Object Identifier 10.1109/TNET.2018.2823642

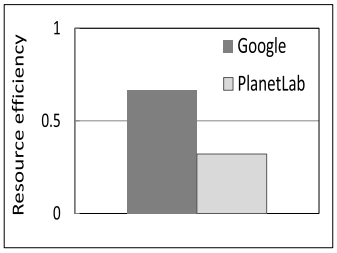



Fig. 2. Resource efficiency.

This resource utilization misalignment problem has not been revealed and studied before, so in this paper, we study the VM resource utilization from public datacenter traces and Hadoop benchmark jobs. Our study verifies the commonness of misalignments of the resource utilization. In order to improve resource utilization, 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 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) 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 during 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:

- We study the VM resource utilization from public datacenter traces and Hadoop benchmark jobs and find that different VMs running the same job exhibit similar periodical resource utilization patterns, but their resource utilization curves exhibit misalignments in time.
- To avoid overestimation in generated resource utilization pattern caused by the misalignments, we propose three algorithms to refine the resource utilization patterns.
- We then extend these algorithms to further improve the resource efficiency by considering periodical resource demand patterns that have multiple pulses in a pattern.
- We apply our three algorithms to the initial VM allocation mechanism [9] based on the predicted patterns. We conduct comprehensive trace-driven simulation and real-world testbed 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 SLO violations, and increases resource efficiency. We further compare our extended refinement algorithms with the refinement algorithms through experiments, and the experimental results confirm the effectiveness of the extended refinement algorithms in further improving the resource efficiency.

The rest of the article is organized as follows. Section II studies the VM resource utilization from public datacenter traces and Hadoop benchmark jobs to verify the commonness of the misalignment feature of the resource utilizations of the VMs of the same job. Section III presents the rationale of pattern refinement, the three refinement algorithms and the extended 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 first statistically study the VM resource utilizations from public datacenter traces, and then study the resource utilization trace from the execution of Hadoop benchmarks in a cluster consisting of twelve machines. We aim to find the answers for the following questions.

- Whether VMs running the same 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 [9] tend to generate low resource efficiency?

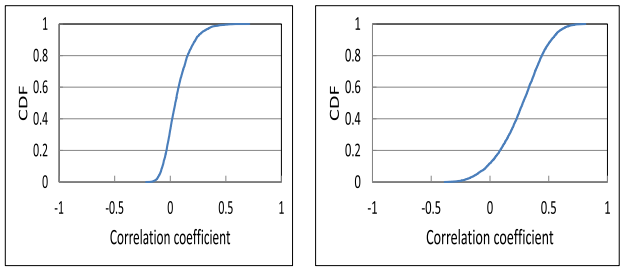
### A. Google Cluster Trace

We first analyze the resource utilization from the Google Cluster trace [13]. 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.

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

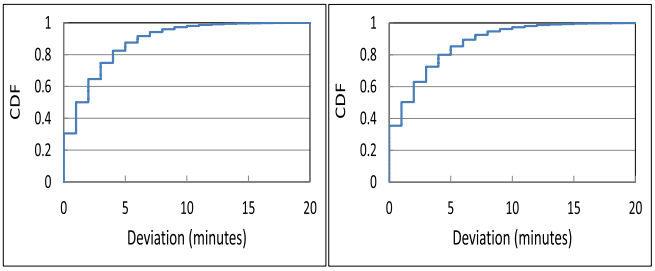
where $x_i$ and $y_i$ are the utilization values at position $i$ in the corresponding trace, $\bar{x}$ and $\bar{y}$ are the average utilizations of the corresponding trace and $n$ is the total number of positions. The correlation coefficient illustrates a quantitative measure of the correlation (i.e., 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 relationship 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, 80% of the task pairs have correlation coefficient spanning from 0 to 0.5. For memory utilization, 80% of the task pairs have correlation coefficient spanning from 0 to 0.6. 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,




(a) (b)

Fig. 3. CDF of correlation coefficient of Google trace. (a) Google CPU trace. (b) Google memory trace.




(a) (b)

Fig. 4. CDF of pulse deviation of Google trace. (a) Google CPU trace. (b) Google memory trace.

though their patterns in one seasonal period are similar as observed in previous works [7]–[11]. 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. 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 [14]. 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 from 0 to 0.2, which indicates that, compared to Google trace, the similarity between VMs running the same job in PlanetLab trace is much lower. It confirms the conjecture that there might exist resource utilization misalignments since their patterns in one seasonal period are similar as observed in previous works [7]–[11]. Figure 6 shows the CDF of the VM pairs corresponding to the pulse deviations. 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. The result confirms that there exist many pulse

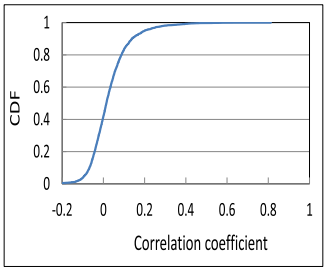



Fig. 5. CDF of correlation coefficient of PlanetLab trace.

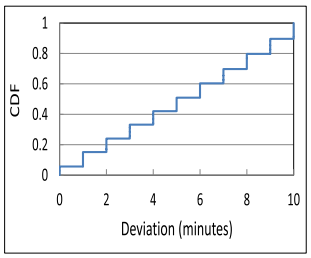



Fig. 6. CDF of pulse deviation of PlanetLab trace.

deviations in the utilization traces and the pulse deviation can be high.

### C. Hadoop Benchmarks

We then conducted MapReduce experiments on Clemson Palmetto high-performance computing (HPC) cluster [15]–[17] to measure the resource utilization traces and analyzed the traces. In the experiments, we implemented a Hadoop MapReduce framework. The framework is implemented on a cluster consists of twelve machines, each of which has two 4-core 2.3GHZ AMD Opteron 2356 processors, 16GB RAM, 193GB hard disk, and 10Gbps Myrinet interconnections [18]. We conducted representative Hadoop benchmarks [19] including *Wordcount*, *Grep*, *Terasort*, *TestDFSIO* and *PiEstimator*. Among them, *Wordcount*, *Grep* and *Terasort* are typical data-intensive applications since they need to read/write and process a large amount of data. We generated 64GB input data by *BigDataBench* [20] based on the *Wikipedia* datasets for *Wordcount*, *Grep*, *Terasort*. *TestDFSIO* write and read tests are typical I/O-intensive applications. They complete a large amount of read/write operations during the map tasks and only do some calculations like calculating the I/O rate in the reduce tasks. In the experiment, we use *TestDFSIO* write to generate 64GB data and then use *TestDFSIO* read to read the generated data. *PiEstimator* is CPU-intensive applications, which uses a statistical (quasi-Monte Carlo) method [21] to estimate the value of Π.

Figure 7 shows the CDF of task pairs corresponding to the correlation coefficient of the CPU, RX bandwidth, I/O and memory utilizations of the Hadoop benchmarks, respectively. Figure 7(a) shows the results from the CPU utilization. We see that most 80% (e.g., from 0.2 to 1) of the task pairs have correlation coefficient larger than 0.9. These results indicate that most tasks running the same application have similar CPU utilizations. We also see that all of the task pairs have correlation coefficient larger than 0.6. It means that the CPU utilizations of some tasks of the same application are not similar. Figure 7(b) shows the results from the RX bandwidth utilization. We see that the correlation coefficient spanning from 0 to 1 almost evenly. Similarly, Figure 7(c) and Figure 7(d) show the results from the I/O and memory utilizations, respectively. We see that 80% of the task pairs

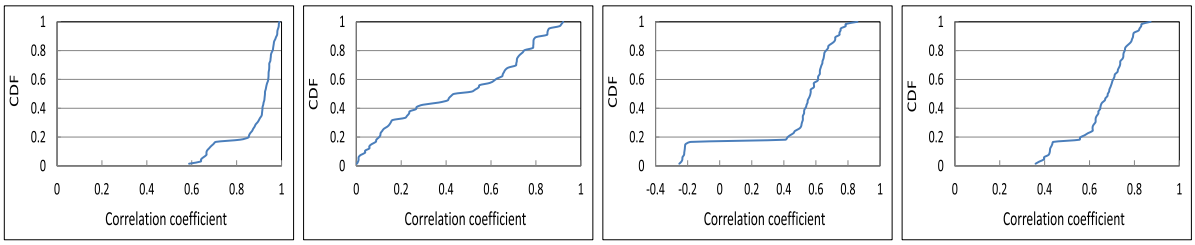



Fig. 7. CDF of correlation coefficient of Hadoop benchmarks. (a) CPU utilization. (b) RX bandwidth utilization. (c) I/O utilization. (d) Memory utilization.

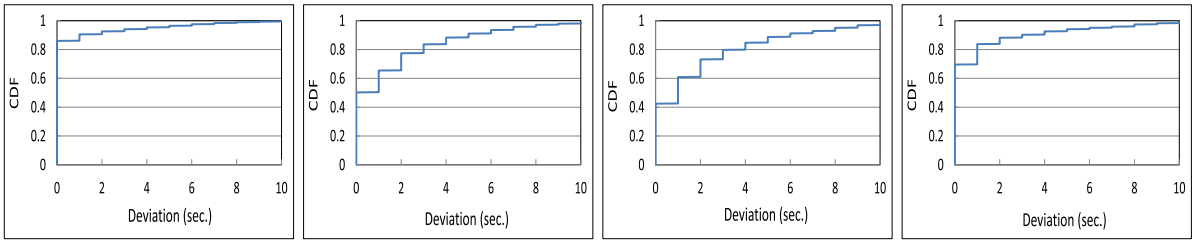



Fig. 8. CDF of pulse deviation of Hadoop benchmarks. (a) CPU utilization. (b) RX bandwidth utilization. (c) I/O utilization. (d) Memory utilization.

have correlation coefficient larger than 0.5 for I/O utilization, and 80% of the task pairs have correlation coefficient larger than 0.6 for memory utilization. They confirm that the tasks running the same application tend to have similar resource utilizations. Also, some VMs' utilizations are not similar, which may be caused by the misalignments.

Figure 8 shows the CDF of the task pairs corresponding to the pulse deviation of the CPU, RX bandwidth, I/O and memory utilizations of the Hadoop benchmarks, respectively. We see that the resource utilizations have pulse deviation spanning from 0 to 10 seconds. Figure 8(a) shows that around 15% of the task pairs have pulse deviations greater than 0, and 10% of the task pairs have pulse deviations greater than 2 seconds. It indicates that CPU utilizations of VMs are not exactly aligned even though they are running the same application. Figure 8(b), Figure 8(c) and Figure 8(d) show that the RX bandwidth utilization, I/O utilization and memory utilization have worse alignments compared to Figure 8(a). Specifically, around 50%, 59% and 30% of the task pairs have pulse deviations greater than 0, and around 36 %, 39% and 18% of the task pairs have pulse deviations greater than 2 seconds, respectively. These results confirm that utilization misalignments are common for tasks running the same application.

### D. 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 [9] as an example. We use $\mathcal{D}_i(t)$ to denote the amount of resource demand of VM $i$ among $N$ VMs at time $t$ (e.g., every one second). The algorithm first finds the maximum demand $\mathcal{E}(t)$ among the set of $\mathcal{D}_i(t)$ $(i = 1, 2, \ldots, N)$ 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).

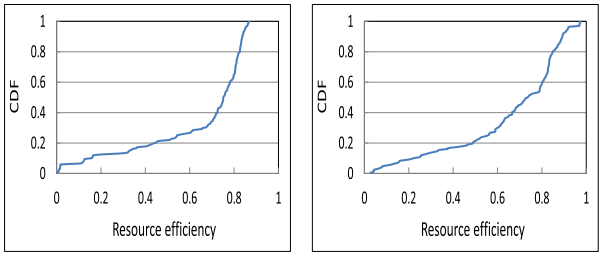
In this experiment, we used Algorithm 1 to determine the resource demand pattern and evaluated its resource efficiency. Specifically, we conducted experiments on predicting

**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** $\mathcal{P}(t)$ $(t = T_0, \ldots, T_0 + T)$

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 $\mathcal{D}(t_j)$ $(j = 1, 2, ...)$ 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 cumulative distribution figure.

Specifically, for the Google Cluster trace, PlanetLab trace and the *Grep* Hadoop benchmark, we randomly selected 100, 1000 and 100 jobs, and tested the resource efficiency of 1550, 4695 and 121 VMs, respectively. Figure 9 shows the CDF of tasks corresponding to the resource efficiency of the Google Cluster trace. Figure 9(a) and Figure 9(b) present the results of CPU utilization and memory utilization, respectively. We see that around 80% of the results have resource efficiencies




(a) (b)

Fig. 9. CDF of resource efficiency of Google trace. (a) CPU efficiency. (b) Memory efficiency.

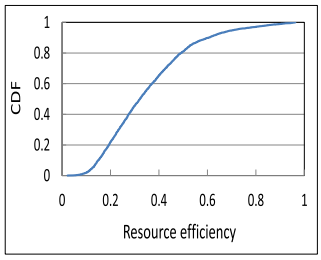



Fig. 10. CDF of resource efficiency of Planetlab trace.

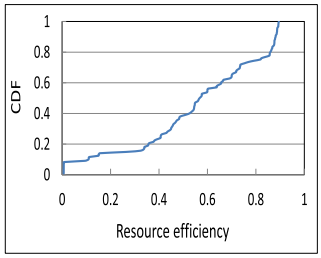



Fig. 11. CDF of resource efficiency of Hadoop bandwidth.

smaller than 0.3 for CPU, and 80% of the results have resource efficiencies smaller than 0.25 for memory. It indicates that there is a large amount of over-provisioning and the resource efficiencies of the previous predictive-based resource allocation algorithm can be further improved. Similarly, Figure 10 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 0.6, which is low. Figure 11 shows the CDF of the VMs corresponding to the resource efficiency from the CPU utilization trace of the *Grep* Hadoop benchmarks. We see that there are 40% of the VMs that have resource efficiencies lower than 0.4, which confirms the resource over-provisioning due to the reason that the generated pattern tend to have more resource demand.

## 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 further increase the resource efficiency by reducing the provided resource. 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 algorithm refines the generated pattern by varying the shape of the original pattern until it achieves the highest resource efficiency. Note that one VM can run multiple tasks since our algorithms only need a VM's resource demand, which is the sum of the resource demands of all tasks running in the VM.

### A. Elastic Resource Provisioning

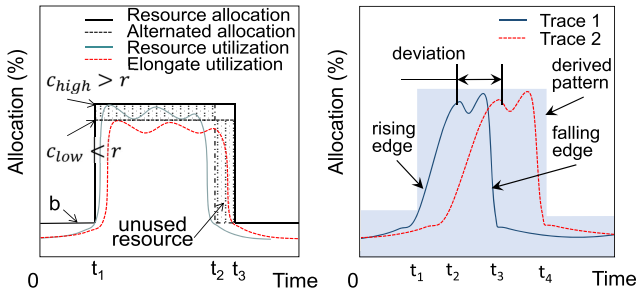
In this section, we discuss the feature of resource provisioning for time-sharing resources. This feature lays the foundation for our proposed pattern refinement algorithms. The time sharing resources have a feature that they can be flexibly provided to VMs. Take CPU resource as an example, VMs take 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) is 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 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 12(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 12(a) equal to each other, i.e.,

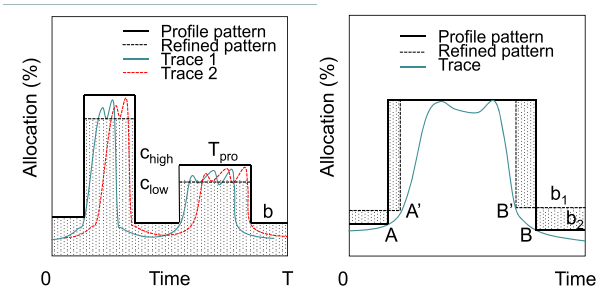
$$(c_{high} - c_{low}) \times\ T_{pro} = (c_{high} - b) \times\ (T_{pro} - T_{high}), \quad (2)$$

where $b$ is the base value of the provisioned resource which is the minimum resource amount provided to the VM. Then,




(a) (b)

Fig. 12. Resource demand misalignment. (a) Elastic resource provisioning. (b) Rising, falling edges and pulse deviation.




(a) (b)

Fig. 13. Pattern refinement. (a) Lowering cap. (b) Varying the base provision.

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 12(b)). The amount to lower the cap can be calculated based on Equation (2). For example, as shown in Figure 13(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 $(c_{high} - c_{low}) \times T_{pro} = s \times (c_{high} - b)$.

Algorithm 2 shows how to improve resource efficiency by reducing the cap of the original derived pattern. 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 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 $(c_{high} - c_{low}) \times T_{pro} = s \times (c_{high} - b)$ (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).

**Algorithm 2** Improve Resource Efficiency By Lowering Cap

1: **Input:** $\mathcal{D}_i(t)$ $(i = 1, 2, \ldots, N)$: 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** $\mathcal{P}'(t)$

*2) Reducing Pulse Width:* 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. Suppose the demand pulse of the VM comes after the beginning of the derived pulse (e.g., the rising edge of the pattern), 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 handle this problem, we propose another algorithm, which reduces the duration of each pulse of the pattern to avoid the over-provisioning.

We use Figure 14 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 14(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 14(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 the shape of the VM's original demand profile without an expansion as shown in Figure 14(b). 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,

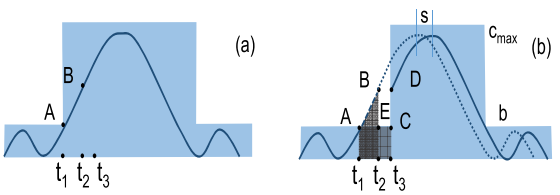



Fig. 14. Pattern refinement by (a) posting the cap provision and (b) reducing the cap width.

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** Improve Resource Efficiency by Reducing Cap Width

1: **Input:** $\mathcal{D}_i(t)$ $(i = 1, 2, \ldots, N)$: 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: **for** every pulse in $\mathcal{P}(t)$
6: Calculate maximum pulse deviation $s$ of the edges
7: $t_i \leftarrow t_1$, $Sum \leftarrow 0$
8: **while** $Sum < sb$ **do**
9: $Sum \leftarrow Sum + (\mathcal{P}(t_i) - b)$
10: $t_i \leftarrow t_i + 1$
11: $d \leftarrow t_i - t_1$
12: Update $\mathcal{P}'(t) \leftarrow b$, for $t_1 \leq t < t_1 + d$
13: **return** $\mathcal{P}'(t)$

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:

$$\int_{t_1}^{t_2} \mathcal{P}(t)dt = b(t_3 - t_1) \tag{3}$$

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

$$\int_{t_1}^{t_1+d-s} \mathcal{P}(t)dt = bd \tag{4}$$

we have

$$d = \mathcal{P}^{-1}(b) + s - t_1 \tag{5}$$

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 alternative way 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 $t_2$ that makes the area size of $t_1t_2BA$ equal to the area size of $t_1t_3CA$. We cannot easily get $\mathcal{P}^{-1}(b)$, the current method in the algorithm to find $d$ is a practical way. 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 return 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 13(b), two tentative square curve fittings with base resource $b_1$ and $b_2$, respectively, are both feasible solutions for pattern detecting. Given a resource utilization profile, the parameters that maximize the resource efficiency can be found by searching through different values of $b$.

**Algorithm 4** Improve Resource Efficiency by Varying the Base Provision

1: **Input:** $\mathcal{D}_i(t)$: 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: Determine $\mathcal{P}(t)$ based on Algorithm 1
5: Find maximum demand $c_{max}$
6: Find minimum demand $b$
7: **do**
8: $b \leftarrow b$+$\Delta$
9: Find $\mathcal{P}'_{temp}(t)$ based on $b$
10: Measure resource efficiency based on $\mathcal{E}(t_j)$ and $\mathcal{P}'_{temp}(t)$
11: if (efficiency $>$ max)
12: max $\leftarrow$ efficiency
13: $\mathcal{P}'(t) \leftarrow \mathcal{P}'_{temp}(t)$
14: **while** $(b < c_{max})$
15: **return** $\mathcal{P}'(t)$

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 13(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 13(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 13(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 a 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 $t'_j$ (or from $t'_j$ to $t_j$ for odd indices). (i.e., from points $A$ to $A'$, and from $B'$ to $B$ in Figure 13(b))

### C. Pattern Refinement Extension

*1) Pattern Refinement for Multi-Pulse Resource Demand Patterns:* Sometimes, a periodical resource demand pattern consists of multiple pulses, as shown in Figure 15, rather than a single pulse. We call such a resource demand pattern *multi-pulse resource demand pattern*. For a multi-pulse resource demand pattern, as there might be different pulse deviations for different pulses, as shown in Figure 15, simply using one single pulse deviation is insufficiently accurate to reflect the misalignment for the entire pattern. The previously proposed pattern refinement algorithms (i.e., lowering cap and reducing pulse width) only consider the deviation of one pulse. Then, if we apply the algorithms to multi-pulse resource demand patterns, the patterns of different VMs of a job must be the same, that is, the time lengths between the pulses in the patterns of the VMs must be the same. However, as the figure shows, it may not be true. Then, the performance of our proposed pattern refinement algorithms may be degraded in improving the resource efficiency. Therefore, in this section, we further propose an extended algorithm for pattern refinement to improve the resource efficiency of multi-pulse resource demand patterns. Note that in this section, unless otherwise indicated, when we mention the previously proposed pattern refinement algorithms, we mean the algorithms for lowering cap and reducing pulse width, because the situation of different deviations does not affect the algorithm for varying base provision. For a multi-pulse resource demand pattern, the algorithm for varying base provision will search the base value for the multiple pulses that maximizes the resource efficiency for the entire pattern.

In detail, we may see the cases of different misalignments as shown in Figures 16. The figure shows a series of resource demand patterns that contain two pulses. We can calculate the pulse deviation for each pulse. We denote $s_1$ as the pulse deviation for the left pulse and $s_2$ as the pulse deviation for the right pulse. We assume $s$ as the pulse deviation for the entire resource demand pattern.

- In the pattern shown in Figure 16(a), we see that there are misalignments for both pulses and $|s_1| > |s_2|$. When we apply the pattern refinement algorithms in Section III-B to the pattern, and use $s = |s_1|$ as the deviation for the entire pattern, it will result in under-provisioning for the right pulse, that is, the amount of actually provisioned resource is lower than the amount of the resource demand.
- In the pattern shown in Figure 16(b), we see that only the right pulse has misalignment, i.e., $s_1 = 0$ and $s_2 = a$. In this scenario, since $s = 0$, we do not need to apply the pattern refinement algorithms in Section III-B to the pattern. It will result in the resource over-provisioning during the right pulse time because there actually exists deviation in the right pulse.
- Generally, as shown in Figure 16(c), for a pattern that contains two pulses and the pulse deviations of the left and the right pulses satisfy $|s_1| \leq |s_2|$, when we apply the pattern refinement algorithms in Section III-B to the pattern and use $s = |s_1|$ as the deviation for the entire pattern, it will lead to resource over-provisioning during the right pulse time.

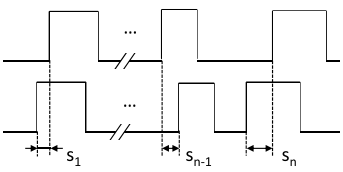



Fig. 15. An examples of multi-pulse resource demand patterns.

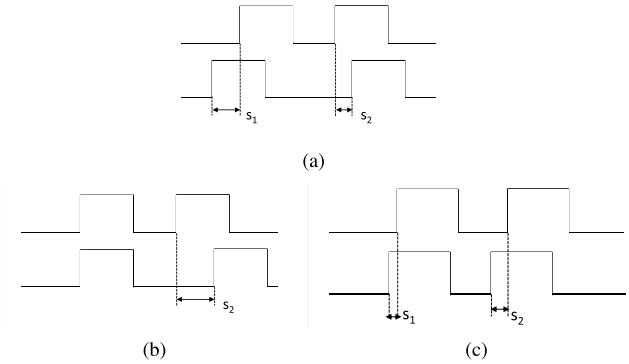



Fig. 16. Examples of multi-pulse resource demand patterns consisting of two pulses in a pattern. (a) Example 1. (b) Example 2.(c) Example 3.

In summary, in Section III-B, our algorithms greatly improve the resource efficiency of the resource demand pattern that contains a single pulse, as a single pulse deviation for these patterns correctly reflect the deviation of the entire patterns. However, as the examples shown above, for multi-pulse resource demand patterns, the entire pulse deviation $s$ may not accurately reveal the deviation of all the pulses. Then, the effectiveness of our pattern refinement algorithms is decreased if the patterns contain multiple pulses. Therefore, we need an extended version for the pattern refinement algorithms to deal with these multi-pulse cases. We present the details of the algorithm extension as follows.

Considering that the performance degradation is caused by the multiple pulses, we first need to detect the pulses that exist in a resource demand pattern. After we detect the pulses, we can divide the multi-pulse pattern into multiple single-pulse patterns. Then, for each single-pulse pattern, we apply our pattern refinement algorithm in Section III-B to improve its resource efficiency. Here, the pattern refinement algorithms include the algorithm for varying base provision. That is, this algorithm finds the base value that maximizes the resource efficiency for each pulse rather than for all pulses in the entire pattern. Finally, we can attain the entire pattern refinement

by combining all the pattern refinements for the single-pulse patterns together. This algorithm extension is straightforward – we just split the pattern into several small patterns, apply the pattern refinement algorithm to each small pattern and finally combine the results. However, the most difficult part in this extension is to detect the pulses in the pattern. Next, we will introduce a pulse detection algorithm.

*2) Pulse Detection Algorithm:* In this section, we propose an algorithm to find out the pulses in a pattern. Basically, a pulse represents a rapid change in the height from a base value to a higher value, followed by a rapid return to the base value. Hence, a pulse has two key characteristics: height and width. Note that here the width represents the time and the height represents the resource utilization.

For a resource demand pattern, we scan through the pattern to find the pulse. We set $W$ and $H$ as the thresholds for both width and height for a pulse. Further, as the pulse for resource demand pattern is not a perfect square as shown in Figure 1 because the resource demand is varying with time, we also define $\delta H$ as the threshold of height changes in a pulse. In other words, if the height change of a point in a pulse is smaller than $\delta H$, we still consider this point as in the same pulse; otherwise, we consider it as the beginning of another pulse.

Notice that we can tune the $W$ and $H$ thresholds. One may ask that our method filters out small pulses that are smaller than $W$ and $H$ in width and height. Actually even though we apply the our pattern refinement algorithms in Section III-B to such small patterns, it does not significantly improve the resource efficiency performance. This is because (i) there are not many pulses in a resource pattern lying in the small width and height ranges, and (ii) applying our pattern refinement algorithms to those small pulses does not provide significant performance improvement on resource efficiency, as the amounts of provisioned resource for these small pulse are very low. Setting $\delta H$ to a higher value can avoid detecting too many pulses in a resource demand pattern.

Algorithm 5 shows the pseudocode for the pulse detection algorithm, which returns a list of pulses in a resource demand pattern. We first check whether the current height (i.e., resource demand) of the pattern is smaller than $H$ or not. If $height < H$, then it is not a pulse in current time (Lines 5-7). If $height \geq H$, we further check whether we are currently processing a pulse or not. If not, then it means that a new pulse starts (Lines 9-11); otherwise, we need to check whether the height change is greater than $\delta H$ (Lines 12-13). If the height change is smaller than the threshold, it means that we are still processing the same pulse (Line 14); otherwise, a new pulse starts (Lines 16-17). Through this algorithm, we can detect and attain a list of the pulses in the pattern along with the pulse widths.

**Algorithm 5** Pseudocode of the Pulse Detection Algorithm

```
1: Input: the pattern from time 0 to t
     W: threshold of the minimum width of a pulse
     H: threshold of the minimum height of a pulse
     δH: threshold of the maximum height change in a pulse
2: Output: A list of the pulses in the pattern
3: RecHeight = −1, δH = 0, width = 0
4:     for i = 1 to t do
5:         if height < H //not a pulse
6:             width = 0
7:             RecHeight = −1
8:         else
9:             if RecHeight == −1 //not in a pulse, then start a new pulse
10:                width = 0
11:                RecHeight = height
12:        else
13:            if |height − RecHeight| ≤ δH //still in the same pulse
14:                width++
15:        else //not in the same pulse, then start a new pulse
16:                width = 0
17:                RecHeight = height
18:    end for
```

### D. Initial VM allocation Mechanism

This section presents a brief review of the initial VM allocation policy in [9], which places 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 [22], where the hosts are conceived as bins and the VMs as objects that need to be packed into the bins. We adopt the dimension-aware heuristic algorithm as mentioned in [9] to solve this problem, which takes advantage of cross dimensional complimentary requirements for different resources.

Algorithm 6 shows the pseudocode for the initial VM allocation policy of a VM. This policy refers to the resource demand pattern $\mathcal{P}_i(t)$ that approximately predicts the resource demands of VMs from the same tenant for the same job. The pattern can be generated by the original pattern detection algorithm (Algorithm 1) or the refinement algorithms (Algorithm 2, Algorithm 3 or Algorithm 4). Based on $\mathcal{P}_i(t)$ and the host $j$'s capacity vector $\mathcal{H}_j$, we can derive predicted fractional VM demand $\mathcal{F}_{ij}(t)$. For each candidate host, we first check whether it has enough resource for hosting the VM at each time $t = T_0, \ldots, T_0 + T$ for each resource by comparing $\mathcal{F}_{ij}(t)$ and the normalized residual resource capacity of a host $\mathcal{R}_j(t)$ (Line 5 and Lines 18-25) in order to ensure that $F_{ij}^k(t) \leq R_j^k(t)$ (i.e., the VM's demand is no more than the residual resource capacity of the host for each resource $k$) 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 $[T_0, T_0 + T]$. 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.

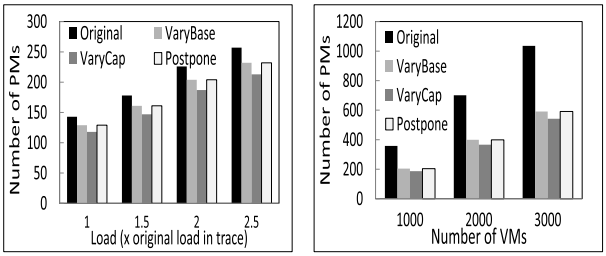
## IV. Trace-Driven Simulation

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 [9], denoted as *VaryCap*, *Postpone* and *VaryBase* (initial VM allocation using Algorithm 2, Algorithm 3 and Algorithm 4,

**Algorithm 6** Pseudocode for Initial VM Allocation

**Input:** $\mathcal{P}_i(t)$: Predicted resource demands
$\mathcal{R}_j(t)$: Residual resource capacity of candidates
**Output:** Allocated host of the VM
  $M$=Double.MAX_VALUE //initialize the distance
  **for** $j = 1$ to $m$ **do**
    **if** CheckValid($\mathcal{P}(t)$,$\mathcal{R}_j(t)$)==**false then**
      **continue**
    **else**
      **for** $k = 1$ to $d$ **do**
        $E_j^k = E_j^k + \frac{1}{T \cdot H_j^k} \int_{T_0}^{T_0+T} P^k(t)dt$
        $M_j += \{w_k(1 - E_j^k)\}^2$
      **end for**
      **if** $M_j < M$ **then**
        $M = M_j$
        AllocatedHost=host $j$
  **end for**
  **return** AllocatedHost

**function CheckValid**($\mathcal{P}(t)$,$\mathcal{R}_j(t)$):
  **for** $k = 1$ to $d$ **do**
    **for** $t = T_0$ to $T_0 + T$ **do**
      **if** $F_{ij}^k(t) > R_j^k(t)$
      **return false**
    **end for**
  **end for**
  **return true**




(a) (b)

Fig. 17. The number of PMs used from Google Cluster trace. (a) Varying workload. (b) Varying number of VMs.

respectively). We denoted the algorithm without refinement as *Original*.

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. All these methods conduct initial VM allocation.

We use the CloudSim [14] 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 [9].

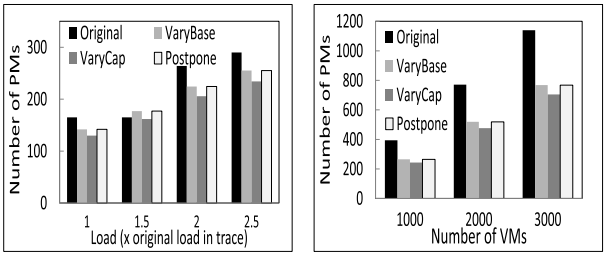
- *The number of PMs used.* This metric measures the resource efficiency of VM allocation mechanisms to host all the VMs.
- *Resource efficiency.* This metric is the ratio between the utilized and allocated amount of resource during the provision time for each VM.
- *The number of SLO violations.* This metric is the number of occurrences that a VM cannot receive the required amount of resource from its host PM.

### A. Performance With Varying Workload

We first study the performance of the three algorithms under different VM workloads using the Google Cluster trace. We varied the workload of the VMs through increasing the original workload in the trace by 1.5, 2 and 2.5 times. Figure 17(a) shows the total number of PMs used from the Google Cluster trace, which follows *VaryCap*<*Postpone*≈*VaryBase*<*Original*. *VaryBase*, *VaryCap* and *Postpone* reduce the number of PMs due to their refined VM patterns, which require relatively less resource than *Original*. *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. This figure also shows that as the workload increases, the number of PMs used increases. This is because as the actual workload increases, CompVM's predicted resource demands increase in initial VM placement. The result further confirms that the refinement algorithms reduce the amount of provisioned resource and needs much fewer PMs than the original CompVM, hence achieves higher resource efficiency. Figure 18(a) shows the total number of PMs used from the PlanetLab trace. It shows similar results as in Figure 17(a), which again confirms that the refinement algorithms reduce the number of PMs needed. The numbers from PlanetLab trace are higher than those from Google Cluster trace due to the reason that the tasks in Google Cluster trace have higher correlation coefficient, and hence the predicted patterns are more accurate.

### B. Performance With Varying Number of VMs

We then study the performance of the refinement algorithms when the number of VMs was varied from 1000 to 3000 using the Google Cluster trace and PlanetLab trace. Figure 18(a) shows the total number of PMs used to provide service for the corresponding number of VMs from Google Cluster trace. We see the result follows *VaryCap*<*Postpone*≈*VaryBase*<*Original* due to the same reasons as in Figure 17(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 the refinement algorithms in reducing the number of PMs used, thus achieving higher resource efficiency. Figure 18(b) shows the total number of PMs used from the PlanetLab trace. We see similar trend as the results from the Google Cluster trace, which confirms that the refinement algorithms are effective in reducing the number




(a) (b)

Fig. 18. The number of PMs used from PlanetLab trace. (a) Varying workload. (b) Varying number of VMs.

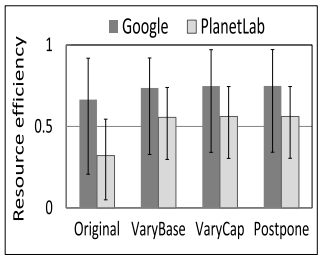



Fig. 19. Resource efficiency.

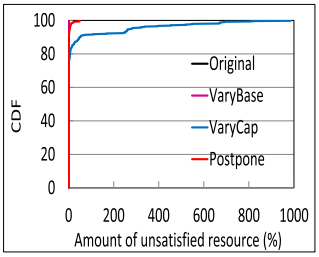



Fig. 20. Ability to satisfy SLO.

of PMs used. Compared to the Google Cluster trace, the results of PlanetLab trace have a higher number of PMs due to the similar reason mentioned before.

### C. Resource Efficiency

Figure 19 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 *Original*<*VaryBase*<*VaryCap*<*Postpone* in both traces. *VaryBase*, *VaryCap* and *Postpone* outperform *Original* 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* as they either bring down the cap to a lower level or reduce the width of the cap. *Postpone* has a similar resource efficiency as *VaryCap* because both of them reduce VM patterns based on the trace deviation.

### D. Performance with Enhanced Algorithms

In this section, we conduct experiments to study the performance of the extended pattern refinement algorithms. We implement the refinement algorithms with the pulse detection algorithm in the initial VM allocation mechanism, denoted as *VaryCap*+, *Postpone*+ and *VaryBase*+, respectively. We compare the performance of these algorithms with the original refinement algorithms without the pulse detection algorithm in

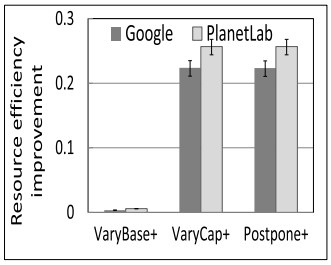
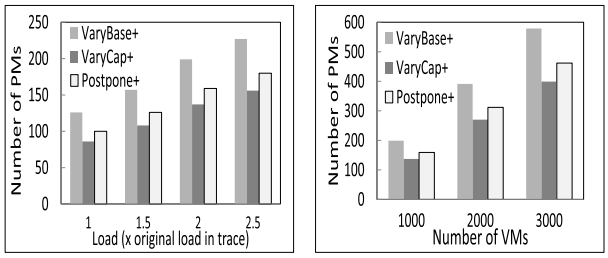



Fig. 21. Resource efficiency improvement.




(a) (b)

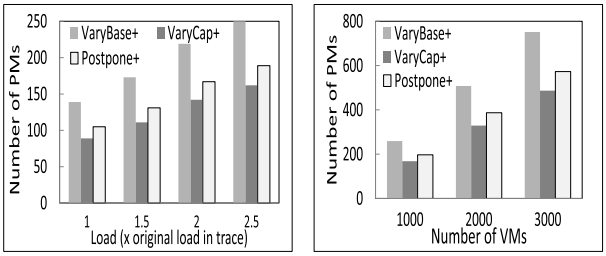
Fig. 22. The number of PMs used from Google Cluster trace. (a) Varying workload. (b) Varying number of VMs.

terms of resource efficiency improvement and the number of PMs used for hosting a certain number of VMs.

Recall that resource efficiency is defined as the ratio between the utilized and allocated amount of resource during the provision time. Figure 21 shows the resource efficiency improvement of the extended pattern refinement algorithms (*VaryCap*+, *Postpone*+ and *VaryBase*+) over the original refinement algorithms (*VaryCap*, *Postpone* and *VaryBase*), using the Google Cluster trace and PlanetLab trace, respectively. We see that overall the extended refinement algorithms improve the resource efficiency of the original pattern refinement algorithms. We also see that the resource efficiency improvement follows *VaryBase*+<*VaryCap*+≈*Postpone*+ in both traces. *VaryBase*+ has nearly zero improvement because this algorithm only varies the base value regardless of the misalignments between pulses to find the base value that maximizes resource efficiency of each pulse while *VaryBase* finds a common base value for all pulses that maximizes resource efficiency of the entire pattern.

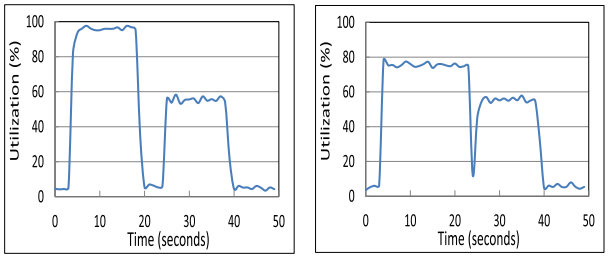
Similar to Figure 17 and Figure 18, we study the performance of the three extended algorithms under different VM workloads. Figure 22(a) shows the total number of PMs used when the workload of the VMs is varied for the Google Cluster trace. The result follows *VaryCap*+<*Postpone*+<*VaryBase*+. *VaryCap*+ and *Postpone*+ reduce the number of PMs due to their refined VM patterns, which require relatively less resource. Compared to Figure 17(a) *VaryCap*+ and *Postpone*+ further reduce the number because they more accurately calculate the deviation for each pulse in the trace, which avoids using the insufficiently accurate deviation in refining the patterns, and hence reducing the number of PMs used to host the same number of VMs. Figure 22(b) shows the total number of PMs used when the number of the VMs is varied for the Google Cluster trace. It shows similar results as Figure 22(a) due to the same reasons.

Figure 23(a) shows the total number of PMs used to provide service for the corresponding number of VMs when the workload of the VMs is varied for the PlanetLab trace. We see the result follows *VaryCap*<*Postpone*<*VaryBase* due




(a) (b)

Fig. 23. The number of PMs used from PlanetLab trace. (a) Varying workload. (b) Varying number of VMs.




(a) (b)

Fig. 24. CPU utilization before and after reducing resource provisioning. (a) CPU utilizations without resource limitation. (b) CPU utilizations with resource limitation.

to the same reasons explained before. Figure 23(b) shows the total number of PMs used to provide service for the corresponding number of VMs when the number of the VMs is varied for the PlanetLab trace. We see similar trend as the results from the Google Cluster trace, which again confirms the effectiveness of the extended refinement algorithms in reducing the number of PMs used to host the same number of VMs. The numbers of PMs used from the PlanetLab trace are higher than those from the Google Cluster trace due to the same reason mentioned in Figure 17. The experimental results confirm the effectiveness of our extended pattern refinement algorithms in further improving resource efficiency.

### *E. Performance in SLO Conformance*

Note that *VaryCap* and *Postpone* improve resource efficiency by scarifying the strict resource requirement that the demands are satisfied immediately. In order to evaluate the performance of these two algorithms, in this experiment, we define SLO violation as the failure of satisfying the resource demand within a time deadline. The time deadline was set to the same value as the maximum completion time among the similar VMs. This definition of SLO violation is reasonable since it still guarantees that the job in the VM finishes no later than the VM that finishes this job most slowly. We kept track of the amount of demands that are higher than the provisioned resource, which is determined by the patterns. If the amount of demand is smaller than the provisioned amount, the SLO is guaranteed, otherwise, the SLO is violated.

We tested 640 VMs and found that *VaryCap* and *Postpone* have 122 SLO violations and 54 SLO violations, respectively. Among the SLO violation results, we calculated the amount of resource that is needed in order to satisfy the SLO, called amount of unsatisfied resource. Figure 20 shows the CDF of the percentage of VMs with the amount of unsatisfied resource. For *VaryCap*, 99% of the VMs satisfy the SLO, and the cumulated amount of unsatisfied resources is less than 800%. For *Postpone*, 99% of the VMs satisfy the demands, and the cumulated amount of unsatisfied resources is less than 100%. The results show that *VaryCap* and *Postpone* are able to satisfy resource demands in most of the time, although they reduce the provisioned resource to achieve higher resource efficiency. *Postpone* performs better than *VaryCap* because *VaryCap* cannot guarantee that the pulse demand of a VM can still receive all of its requested resource in the provision time.

## V. Real-World Testbed Experiments

### *A. Effect of Reduced Resource Provisioning*

Recall that in Section III-A, we explained the feature of resource provisioning for time-sharing resources, which lays the foundation for our proposed pattern refinement algorithms. In this experiment, we controlled the amount of resource that is provisioned to a VM and then measured the resource utilization of this VM in order to confirm the elastic nature of resource provisioning. In order to create workload to reach high CPU utilization (e.g., 100%) and low CPU utilization to prove the elastic nature, we created a synthetic workload that generates a list of prime numbers from 0 to 99999999 in its first phase and then generates a list of prime numbers from 0 to 49999999 in its second phase. We executed this synthetic workload in a desktop machine with 2.00GHz Intel(R) Core(TM) 2 CPU and 2GB memory. We used the *cpulimit* tool [23] to curb the CPU usage of the VM by pausing the process at different intervals to keep it under the defined ceiling. We first used the batch mode of Linux *top* command to keep track of the CPU utilization at every 0.1 second, and then presented the result of the CPU utilization of every second by averaging every 10 records from the original record.

Figure 24(a) shows the CPU resource utilization of the synthetic workload with full resource provisioning without limitation. We see that the VM consumes near 100% of the CPU resource during the first half of execution (e.g., 0 second to 20 seconds), and consumes around 60% during the second half (e.g., 25 second to 40 seconds). Finally, the VM finishes the job at time 40 seconds.

Recall that our algorithm lowers the provisioned CPU resource amount while still keeping the original VM task completion time. We then used *cpulimit* to limit the CPU usage of this VM to 80%. Figure 24(b) shows the resulted CPU resource utilization. We see that the VM consumes near 80% of the CPU resource during the first half. As a result, the duration of the first phase execution is elongated (from 20 seconds to 25 seconds). Since the first phase execution finishes at 25 seconds, it does not affect the consequent second phase execution. Since the second phase requires 60% of CPU resource while it is provisioned with 80%, which is sufficient for its execution, the CPU utilization of the second phase is not affected by the resource limitation. That is, it has similar utilization as in Figure 24(a). Finally, the VM finishes the job at time 40 seconds. The CPU utilization results before and after resource limitation shown in Figure 24 confirm that the limitation of the resource provisioning leads to an elongation of the execution time (as shown in the first phase), and will not affect the completion time of the VM as long as the total amount of CPU resource allocated to the VM is sufficient for its execution.

### *B. Performance of Pattern Refinement*

In this experiment, we used workloads from the NAS Parallel Benchmark (NPB) suite [24] to run in the VMs.

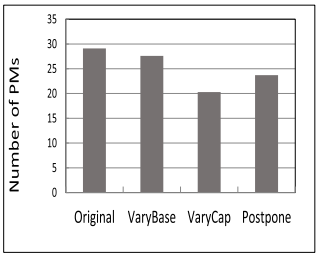



Fig. 25. The number of PMs used.

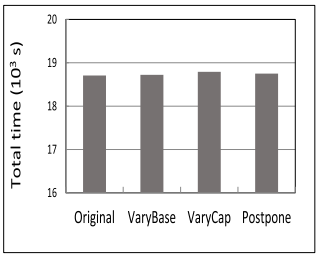



Fig. 26. Total execution time of the VMs.

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 our HPC cluster 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. In the VM placement experiment, we generated the resource utilization patterns of the programs according to the pattern refinement algorithms, and used them as the resource utilization patterns of the VMs as each VM hosts one program. Based on the generated patterns for the VMs, the VM placement algorithm is executed to determine the VM to PM mappings. Finally, we deployed the VMs in the HPC clusters and evaluated the performance.

Figure 25 shows the total number of PMs required to host the VMs. We see that the number of PMs used follows *VaryCap*<*Postpone*<*VaryBase*<*Original*. *VaryBase*, *VaryCap* and *Postpone* reduce the PMs due to their refined VM patterns, which require relative less resource than *Original*.
*VaryCap* further reduces the number because it reduces the cap value of the patterns. The results are consistent with the simulation results in Figure 17. Figure 26 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 [25]. Some of them [26]–[30] allocate physical resources to VMs only once based on static VM resource demands. For example, Srikantaiah *et al.* [30] 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, others [31]–[36] 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 [31] 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.

Some VM placement or VM migration methods predict VM workload to ensure that PMs will not be overloaded. Gmach *et al.* [37] used historical information to periodically and proactively reassign VMs to PMs for high performance. Verma *et al.* [38] presented a dynamic resource demand prediction and allocation framework in multi-tenant service clouds. In order to consider both the current and future state of resource demand and available capacity in a time period, Chen and Shen [9] 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. In other words, the generated pattern tends to have low resource efficiency when it is used to guide resource provisioning for a VM. As a result, consolidating VMs based on these patterns will result in a waste of resource.

Xu *et al.* [39] presented an overview of the previous research on managing the performance overhead of VMs under different scenarios of the IaaS cloud. To reduce the network cost in the cloud, Li *et al.* [40] proposed effective VM placement methods. Xu *et al.* [39], [41] proposed VM provisioning or migration methods that consider VM performance. Lim *et al.* [42] modelled a migration process of a VM instance and proposed a method to analyze the migration time and the performance impact on multi-resource shared systems for completing given VM assignment plan. CACEV [43] is a VM placement method for reducing cost and carbon emission. The work in [44] aims to not only satisfy the needs of current VMs but also ensure that the needs of future VMs can be satisfied. The work in [45] proposes a VM placement method that considers the peak workload characteristics of VMs. Mann [46] not only considered the VM placement in PMs, but also consider the allocation of tasks to VMs for high performance. Nejad *et al.* [47] proposed truthful greedy and optimal mechanisms so that the users do not have incentives to manipulate the system by lying about their requested bundles of VM instances and their valuations. The work in [48] proposes a performance-to-power ratio aware VM allocation in order to reduce energy consumption in clouds. The work in [49] aims to achieve better resource utilization and thermal distribution by appropriately allocating VMs. Zheng *et al.* [50] proposed a method that provides strategy-proof double auctions for multi-cloud, multi-tenant bandwidth reservation. Zhang *et al.* [51] proposed a burstiness-aware resource reservation method for server consolidation in computing clouds for high performance. Lu *et al.* [52] proposed a clique VM migration method that conducts affinity grouping of VMs for inter-cloud live migration. Unlike these works that focus on VM migration, RPRP focuses on VM resource demand prediction and misprediction correction. These VM migration methods can use the demand prediction

from RPRP in migration scheduling to improve performance. Our experimental results show the advantage of RPRP in VM migration.

Recently, some works focus on allocating network bandwidth resources to tenant VMs [8], [28], [29], [53]. Oktopus [28] provides static bandwidth reservations throughout the network. Shen and Li [53] proposed a new bandwidth sharing model to achieve minimum guarantee and proportionality in clouds, while preventing tenants from earning unfair bandwidth. Popa *et al.* [29] 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.* [8] provide 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. Some others focus on managing resource in datacenters [54]–[58]. Zhang *et al.* [54] proposed CPI$^2$ that uses cycles-per-instruction (CPI) data obtained by hardware performance counters to identify problems, select the likely perpetrators, and then optionally throttle them so that the victims can return to their expected behavior. It automatically learns normal and anomalous behaviors by aggregating data from multiple tasks in the same job. Schwarzkopf *et al.* [55] proposed a cluster scheduling architecture that uses parallelism, shared state, and optimistic concurrency control. Leverich and Kozyrakis [56] analyzed the challenges of maintaining high QoS for low-latency workloads when sharing servers with other workloads. Ghodsi *et al.* [57] proposed Constrained Max-Min Fairness (CMMF), an extension to max-min fairness that supports placement constraints, and show that it is the only policy satisfying an important property that incentivizes users to pool resources. Verma *et al.* [58] proposed a cluster manager that runs hundreds of thousands of jobs across a number of clusters. It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation.

## VII. Conclusions

In this paper, we studied the VM resource utilization from public datacenter traces and Hadoop benchmark jobs and found that different VMs running the same job exhibit similar periodical resource utilization patterns, but their resource utilization curves exhibit misalignments in time. 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 hence low resource efficiency. In order to improve resource efficiency, we proposed three VM resource utilization pattern refinement algorithms that leverage the elastic resource provisioning feature 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 further entend these refinement algorithms to enhance the resource efficiency by considering multi-pulse resource demand patterns. We then adopted these refinement algorithms in an initial VM allocation mechanism that consolidates VMs for cloud datacenters. As a result, the mechanism helps fully utilize the cloud resources, and reduce the number of PMs needed to satisfy tenant requests without compromising the SLO conformance performance. These advantages have been verified by our extensive trace-driven simulation experiments and real-world testbed experiments. The experimental results also show the effectiveness of our extended refinement algorithms in improving resource efficiency. In our future work, we will explore how to enhance the heuristic VM allocation mechanism by a dynamic programming algorithm that optimize the consolidation of VMs for high resource efficiency.

## Acknowledgements

The authors would like to thank Dr. J. Wilkes from Google for his valuable discussions and thank Dr. L. Chen for his valuable contribution.

## References

[1] J. Liu, H. Shen, and H. S. Narman, "CCRP: Customized cooperative resource provisioning for high resource utilization in clouds," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 243–252.
[2] W. Jiang, F. Liu, G. Tang, K. Wu, and H. Jin, "Virtual machine power accounting with shapley value," in *Proc. IEEE ICDCS*, Jun. 2017, pp. 1683–1693.
[3] K. PushpaLatha, R. S. Shaji, and J. P. Jayan, "A cost effective load balancing scheme for better resource utilization in cloud computing," *J. Emerg. Technol. Web Intell.*, vol. 6, no. 3, pp. 280–290, 2014.
[4] *Service Level Agreements*. Accessed: Apr. 2018. [Online]. Available: http://azure.microsoft.com/en-us/support/legal/sla/
[5] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.
[6] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. SOCC*, 2011, Art. no. 5.
[7] V. Jalaparti *et al.*, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proc. SIGCOMM*, 2015, pp. 407–420.
[8] 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. SIGCOMM*, 2012, pp. 199–210.
[9] L. Chen and H. Shen, "Consolidating complementary VMs with spatial/temporal-awareness in cloud datacenters," in *Proc. INFOCOM*, Apr./May 2014, pp. 1033–1041.
[10] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proc. IMC*, 2013, pp. 191–204.
[11] B. Palanisamy, A. Singh, L. Liu, and B. Langston, "Cura: A cost-optimized model for MapReduce in a cloud," in *Proc. IPDPS*, May 2013, pp. 1275–1286.
[12] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive elastic ReSource scaling for cloud systems," in *Proc. CNSM*, Oct. 2010, pp. 9–16.
[13] *Google Cluster Data*. Accessed: Apr. 2018. [Online]. Available: https://github.com/google/cluster-data
[14] R. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw., Pract. Exper.*, vol. 41, no. 1, pp. 23–50, 2011.
[15] Z. Li and H. Shen, "Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance," in *Proc. ICPP*, Sep. 2015, pp. 21–30.
[16] Z. Li, H. Shen, W. Ligon, and J. Denton, "An exploration of designing a hybrid scale-up/out hadoop architecture based on performance measurements," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 386–400, Feb. 2016.
[17] Z. Li and H. Shen, "Performance measurement on scale-up and scale-out hadoop with remote and local file systems," in *Proc. Cloud*, Jun./Jul. 2015, pp. 456–463.
[18] N. J. Boden *et al.*, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995.
[19] *Hadoop*. Accessed: Feb. 2015. [Online]. Available: http://devin.com/lookbusy/
[20] L. Wang *et al.*, "BigDataBench: A big data benchmark suite from Internet services," in *Proc. HPCA*, Feb. 2014, pp. 488–499.
[21] R. E. Caflisch, "Monte Carlo and quasi-Monte Carlo methods," *Acta Numer.*, vol. 7, pp. 1–49, Jan. 1998.
[22] J. Csirik, J. B. G. Frenk, M. Labbe, and S. Zhang, "On the multidimensional vector bin packing," *Acta Cybern.*, vol. 9, no. 4, pp. 361–369, 1990.

[23] *CPU Usage Limiter*. Accessed: Apr. 2018. [Online]. Available: http://cpulimit.sourceforge.net/
[24] D. H. Bailey *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *Proc. SC*, Nov. 1991, pp. 158–165.
[25] 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. IPDPS Workshops*, May 2011, pp. 890–897.
[26] U. Bellur, C. S. Rao, and S. D. M. Kumar, "Optimal placement algorithms for virtual machines," *CoRR*, vol. abs/1011.5064, Nov. 2010. [Online]. Available: http://arxiv.org/abs/1011.5064
[27] J. Xu and J. A. B. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *Proc. CPSCom*, Dec. 2010, pp. 179–188.
[28] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. SIGCOMM*, 2011, pp. 242–253.
[29] L. Popa *et al.*, "FairCloud: Sharing the network in cloud computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 187–198, 2012.
[30] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *Proc. HotPower*, 2008, pp. 1–5.
[31] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Comput. Netw.*, vol. 53, no. 17, pp. 2923–2938, 2009.
[32] M. Tarighi, S. A. Motamedi, and S. Sharifian, "A new model for virtual machine migration in virtualized cluster server based on fuzzy decision making," *CoRR*, vol. abs/1002.3329, Feb. 2010. [Online]. Available: http://arxiv.org/abs/1002.3329
[33] E. Arzuaga and D. R. Kaeli, "Quantifying load imbalance on virtualized enterprise servers," in *Proc. WOSP/SIPEW*, 2010, pp. 235–242.
[34] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proc. SC*, Nov. 2008, pp. 1–12.
[35] G. Khanna, K. Beaty, G. Kar, and A. Kochut, "Application performance management in virtualized server environments," in *Proc. NOMS*, Apr. 2009, pp. 373–381.
[36] A. Verma, P. Ahuja, and A. Neogi, "pMapper: Power and migration cost aware application placement in virtualized systems," in *Proc. Middleware*, 2008, pp. 243–264.
[37] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Resource pool management: Reactive versus proactive or let's be friends," *Comput. Netw.*, vol. 53, no. 17, pp. 2905–2922, 2009.
[38] M. Verma *et al.*, "Dynamic resource demand prediction and allocation in multi-tenant service clouds," *Concurrency Comput., Pract. Exper.*, vol. 28, no. 17, pp. 4429–4442, 2016.
[39] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proc. IEEE*, vol. 102, no. 1, pp. 11–31, Jan. 2014.
[40] X. Li, J. Wu, S. Tang, and S. Lu, "Let's stay together: Towards traffic aware virtual machine placement in data centers," in *Proc. INFOCOM*, Apr./May 2014, pp. 1842–1850.
[41] F. Xu, F. Liu, and H. Jin, "Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2470–2483, Aug. 2016.
[42] S.-H. Lim, J.-S. Huh, Y. Kim, and C. R. Das, "Migration, assignment, and scheduling of jobs in virtualized environment," in *Proc. HotCloud*, 2011, p. 2.
[43] E. Ahvar, S. Ahvar, Z. Á. Mann, N. Crespi, J. Garcia-Alfaro, and R. Glitho, "CACEV: A cost and carbon emission-efficient virtual machine placement method for green distributed clouds," in *Proc. IEEE SCC*, Jun. 2016, pp. 275–282.
[44] F. Hao, M. Kodialam, T. V. Lakshman, and S. Mukherjee, "Online allocation of virtual machines in a distributed cloud," *IEEE/ACM Trans. Netw.*, vol. 25, no. 1, pp. 238–249, Feb. 2017.
[45] W. Lin, S. Xu, J. Li, L. Xu, and Z. Peng, "Design and theoretical analysis of virtual machine placement algorithm based on peak workload characteristics," *Soft Comput.*, vol. 21, no. 5, pp. 1301–1314, 2017.
[46] Z. Á. Mann, "Interplay of virtual machine selection and virtual machine placement," in *Proc. ESOCC*, 2016, pp. 137–151.
[47] M. M. Nejad, L. Mashayekhy, and D. Grosu, "Truthful greedy mechanisms for dynamic virtual machine provisioning and allocation in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 2, pp. 594–603, Feb. 2015.
[48] X. Ruan and H. Chen, "Performance-to-power ratio aware virtual machine (VM) allocation in energy-efficient clouds," in *Proc. Cluster*, Sep. 2015, pp. 264–273.
[49] J. V. Wang, K.-Y. Fok, C.-T. Cheng, and C. K. Tse, "A stable matching-based virtual machine allocation mechanism for cloud data centers," in *Proc. IEEE Services*, Jun./Jul. 2016, pp. 103–106.
[50] Z. Zheng, Y. Gui, F. Wu, and G. Chen, "STAR: Strategy-proof double auctions for multi-cloud, multi-tenant bandwidth reservation," *IEEE Trans. Comput.*, vol. 64, no. 7, pp. 2071–2083, Jul. 2015.
[51] S. Zhang, Z. Qian, Z. Luo, J. Wu, and S. Lu, "Burstiness-aware resource reservation for server consolidation in computing clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 964–977, Apr. 2016.
[52] T. Lu, M. Stuart, K. Tang, and X. He, "Clique migration: Affinity grouping of virtual machines for inter-cloud live migration," in *Proc. NAS*, Aug. 2014, pp. 216–225.
[53] H. Shen and Z. Li, "New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants," in *Proc. INFOCOM*, Apr./May 2014, pp. 835–843.
[54] X. Zhang *et al.*, "CPI$^2$: CPU performance isolation for shared compute clusters," in *Proc. EuroSys*, 2013, pp. 379–391.
[55] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. EuroSys*, 2013, pp. 351–364.
[56] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proc. EuroSys*, 2014, Art. no. 4.
[57] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proc. EuroSys*, 2013, pp. 365–378.
[58] A. Verma *et al.*, "Large-scale cluster management at Google with borg," in *Proc. EuroSys*, 2015, p. 18.
[59] L. Chen and H. Shen, "Considering resource demand misalignments to reduce resource over-provisioning in cloud datacenters," in *Proc. INFOCOM*, May 2017, pp. 1–9.

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

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