CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems

Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu,
Department of Computer Science
North Carolina State University
{zshen5,ssubbia2}@ncsu.edu, gu@csc.ncsu.edu

John Wilkes
Google
Mountain View, CA
johnwilkes@google.com

ABSTRACT

Elastic resource scaling lets cloud systems meet application service level objectives (SLOs) with minimum resource provisioning costs. In this paper, we present CloudScale, a system that automates fine-grained elastic resource scaling for multi-tenant cloud computing infrastructures. CloudScale employs online resource demand prediction and prediction error handling to achieve adaptive resource allocation without assuming any prior knowledge about the applications running inside the cloud. CloudScale can resolve scaling conflicts between applications using migration, and integrates dynamic CPU voltage/frequency scaling to achieve energy savings with minimal effect on application SLOs. We have implemented CloudScale on top of Xen and conducted extensive experiments using a set of CPU and memory intensive applications (RUBiS, Hadoop, IBM System S). The results show that CloudScale can achieve significantly higher SLO conformance than other alternatives with low resource and energy cost. CloudScale is non-intrusive and light-weight, and imposes negligible overhead (< 2% CPU in Domain 0) to the virtualized computing cluster.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction, Monitors*; C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Measurement, Performance

Keywords

Cloud Computing, Resource Scaling, Energy-efficient Computing

1. INTRODUCTION

Most Infrastructure as a Service (IaaS) providers [1, 6] use virtualization technologies [10, 7, 3] to encapsulate applications and provide isolation among uncooperative users. However, statically partitioning the physical resource into virtual machines (VMs) according to the applications' peak demands will lead to poor resource utilization. Overbooking [39] is used to improve the overall resource utilization, and resource capping is applied to achieve performance isolation among co-located applications by guaranteeing that no application can consume more resources than those allocated to it. However, application resource demand is rarely static, varying as a result of changes in overall workload, the workload mix, and internal application phases and changes. If the resource cap is too low, the application will experience SLO violations. If the resource cap is too high, the cloud service provider has to pay for the wasted resources. To avoid both situations, multi-tenant cloud systems need an *elastic resource scaling* system to adjust the resource cap dynamically based on application resource demands.

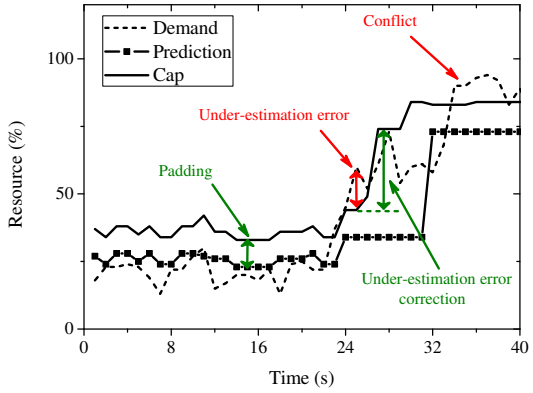


Figure 1: Problems and solutions of prediction-driven resource scaling.

In this paper we present CloudScale, a prediction-driven elastic resource scaling system for multi-tenant cloud computing. The goal of our research is to develop an automatic system that can meet the SLO requirements of the applications running inside the cloud with minimum resource and energy cost. In [24], we described our application-agnostic, light-weight online resource demand predictor and showed that it can achieve good prediction accuracy for a range of real world applications. When applying the predictor to the resource scaling system, we found that the resource scaling system needs to address a set of new problems in order to reduce SLO violations, illustrated by Figure 1. First, online resource demand prediction frequently makes over- and under-estimation errors. Over-estimations are wasteful, but can be corrected by the online resource demand prediction model after it is updated with true application resource demand data. Under-estimations are much

SOCC'11, October 27–28, 2011, Cascais, Portugal.

worse since they prevent the system from knowing the true application resource demand and may cause significant SLO violations. Second, co-located applications will conflict when the available resources are insufficient to accommodate all scale-up requirements.

CloudScale provides two complementary under-estimation error handling schemes: 1) online adaptive padding and 2) reactive error correction. Our approach is based on the observation that reactive error correction alone is often insufficient. When an under-estimation error is detected, an SLO violation has probably already happened. Moreover, there is some delay before the scaling system can figure out the right resource cap. Thus, it is worthwhile to perform proactive padding to avoid under-estimation errors.

When a scaling conflict happens, we can either reject some scale-up requirements or migrate some applications [14] out of the overloaded host. Migration is often disruptive, so if the conflict is transient, it is not cost-effective to do this. Moreover, it is often too late to trigger the migration on a conflict since the migration might take a long time to finish when the host is already overloaded. Our approach achieves *predictive migration*, which can start the migration before the conflict happens to minimize the impact of migration to both migrated and non-migrating applications. CloudScale uses conflict prediction and resolution inference to decide whether a migration should be triggered, which application(s) should be migrated, and when the migration should be triggered.

We make the following contributions in this paper:

- We introduce a set of intelligent schemes to reduce SLO violations in a prediction-driven resource scaling system.
- We show how using both adaptive padding and fast under-estimation error correction minimizes the impact of under-estimation errors with low resource waste.
- We evaluate how well predictive migration resolves scaling conflicts with minimum SLO impact.
- We demonstrate how combining resource scaling with CPU voltage and frequency scaling can save energy without affecting application SLOs.

The rest of the paper is organized as follows. Section 2 presents the system design of CloudScale. Section 3 presents the experimental results. Section 4 compares our work with related work. Section 5 discusses the limitations and future work. Finally, the paper concludes in Section 6.

2. CLOUDSCALE DESIGN

In this section, we present the design of CloudScale. First, we provide an overview of our approach. Then, we introduce the single VM scaling algorithms that can efficiently handle runtime prediction errors. Next, we describe how to resolve scaling conflicts. Finally, we describe the integrated VM resource scaling and CPU frequency and voltage scaling for energy saving.

2.1 Overview

CloudScale runs within each host in the cloud system, and is complementary to the resource allocation scheme that handles coarse-grained replicated server capacity scaling [30, 38, 9]. CloudScale is built on top of the Xen virtualization platform. Figure 2 shows the overall architecture of the CloudScale System.

CloudScale uses *libxenstat* to monitor guest VM's resource usage from domain 0. The monitored resource metrics include CPU consumption, memory allocation, network traffic, and disk I/O statics. CloudScale also uses a small memory monitoring daemon

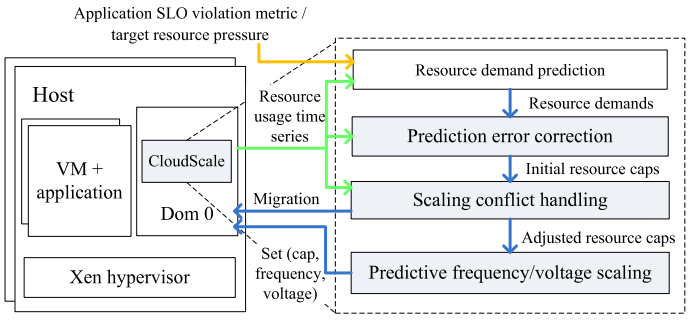


Figure 2: The CloudScale system achitecture.

within each VM to get memory usage statistics (through the */proc* interface in Linux). Measurements are taken every 1 second. We use external application SLO monitoring tools [11] to keep track of whether the application SLO is violated. CloudScale currently supports CPU and memory resource scaling. The CPU resource scaling is done by adjusting the CPU cap using the Xen credit scheduler [8] that runs in the non-work-conserving mode (i.e., a domain cannot use more than its share of CPU). The memory resource scaling is done using Xen's *setMemoryTarget* API.

The resource usage time series are fed into an online resource demand prediction model to predict the short-term resource demands. CloudScale uses the online resource demand prediction model developed in our previous work [24]. It uses a hybrid approach that employs signature-driven and state-driven prediction algorithms to achieve both high accuracy and low overhead. The model first employs a fast Fourier transform (FFT) to identify repeating patterns called signatures. If a signature is discovered, the prediction model uses it to estimate future resource demands. Otherwise, the prediction model employs a discrete-time Markov chain to predict the resource demand in the near future.

The *prediction error correction* module performs *online adaptive padding* that adds a dynamically determined cushion value to the predicted resource demand in order to avoid under-estimation errors. The *reactive error correction* component detects and corrects under-estimation errors that are not prevented by the padding scheme. The result is an initial resource cap for each application VM.

The resource usage time series are also fed into a *scaling conflict prediction* component. CloudScale decides to trigger VM migrations or resolve scaling conflicts locally in the *conflict resolution* component. The *local conflict handling* component adjusts the initial resource caps for different VMs according to their priorities when the sum of the initial resource caps exceed the capacity of the host. The *migration-based conflict handling* component decides when to trigger VM migration and which VM to migrate.

The *predictive frequency and voltage scaling* module takes the resource cap information to derive the minimum CPU frequency and voltage that can support all the VMs running on the host. The *cap adjustment* component then calculates the final resource cap values based on the ratio between the new CPU frequency and the old CPU frequency.

In the rest of this section, we will provide details about each major system module.

2.2 Prediction Error Correction

CloudScale incorporates both proactive and reactive approaches to handle under-estimation errors. In this section, we present the online adaptive padding and under-estimation correction schemes.

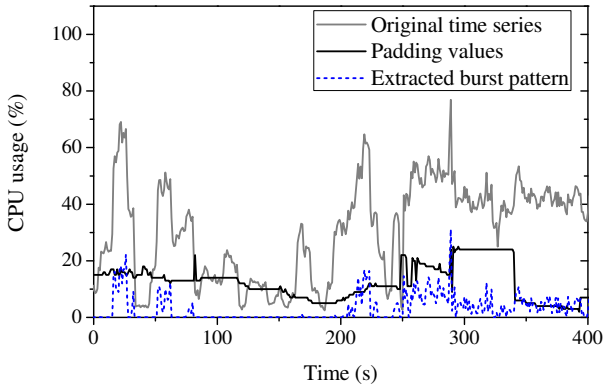


Figure 3: Padding values decided according to the extracted burst pattern.

2.2.1 Online Adaptive Padding

CloudScale uses an online adaptive padding scheme to avoid under-estimation errors by adding a small extra value to the predicted resource demand. If we pad too little, we might still encounter under-estimation errors; if we pad too much, we might have unnecessary resource waste. Our approach is based on the observation that under-estimation errors are often caused by resource usage bursts. Thus, we choose the padding value based on the recent burstiness of application resource usage and recent prediction errors.

Burst-based padding. We employ signal processing techniques to extract the *burst pattern* and use it to calculate the padding value, illustrated by Figure 3. Suppose we want to decide the padding value for time t. We examine a window of recent resource usage time series $L = \{l_{t-W_a}, ..., l_{t-1}\}$ (e.g., W_a=100). We use a fast Fourier transform (FFT) algorithm to determine the coefficients that represent the amplitude of each frequency component. We consider the top k (e.g., 80%) frequencies in the frequency spectrum as high frequencies. We then apply reverse FFT over the high frequency components to synthesize the burst pattern. Since the goal of padding is to avoid under-estimation errors, we only consider the positive values.

Next, we calculate a *burst density* metric, which is the number of positive values in the extracted burst pattern. The burst density reflects how often bursts appear in recent application resource usage. If the burst density is high (e.g., larger than 50%), CloudScale uses the maximum of all burst values as the padding value. Otherwise, CloudScale uses a smaller burst value (e.g., 80th percentile of the burst values) as the padding value to avoid over-padding. Figure 3 shows the padding values decided according to the extracted burst pattern, with W_a = 100 seconds. The padding values are higher when the application's CPU usage is bursty, and are smaller when the usage becomes stable.

Remedial padding. With padding, the scaling system can observe when the real resource demand is higher than the unpadded predicted value. We can learn from recent prediction errors to avoid under-estimation errors in the future. Let $e_1, ..., e_k$ denote a set of recent prediction errors. We calculate e_i as $x_i - x'_i$, where x_i and x'_i denote the predicted value and the observed resource demand respectively and $x_i < x'_i$. Since the goal of the remedial padding is to make up for recent resource under-provisioning, we only consider under-estimation errors, i.e. $e_i < 0$. Thus, we set $e_i = 0$ if $e_i > 0$. We then calculate a weighted moving average (WMA) of those prediction errors $WMA(e_1, ...e_k)$. The scaling system compares $|WMA(e_1, ...e_k)|$ with the padding value calculated from the burst pattern, and picks the larger one as the padding value.

2.2.2 Fast Under-estimation Correction

We wish to detect and correct under-estimation errors as soon as possible since the application will suffer SLO violation during resource under-provisioning. The key problem is that the real resource demand is unknown during under-provisioning: we only have a lower bound. Thus, we have to guess the right resource allocation.

One simple solution is to immediately raise the resource cap to the maximum possible value (i.e., all residual resources on the host). This will cause excessive resource waste, and a lot of scaling conflicts when we perform concurrent scaling for multiple collocated applications. Instead, CloudScale raises the resource cap by multiplying the current resource cap by a ratio $\alpha > 1$ until the under-estimation error is corrected. CloudScale divides time into *steps*. The length of each step is 1 second. If the resource cap for the current step is x, the resource cap after k steps will be $x \times \alpha^k$. It is possible that this scheme will cause some over-provisioning. However, since CloudScale is driven by a prediction model, the resource cap will be corrected when the prediction model catches up and learns the real demand of the application.

The value of α denotes the tradeoff between the under-estimation correction speed and the resource waste. When we use a larger α, the system raises the cap faster but may over-provision resources by a larger margin and reduce the resources that are available to other collocated applications. Thus, CloudScale dynamically decides the scale-up ratio by mapping the resource pressure and application SLO feedback to α according to the severity of the under-estimation error. The resource pressure P ($0 \leq P \leq 1$) denotes the ratio of resource usage to the resource cap. The SLO feedback is the feedback from the applications about the SLO conformance.

We define α_{min} and α_{max} as the minimum and maximum scale-up ratios we want to use in the system, which can be tuned by the cloud provider. For example, we set $\alpha_{min} = 1.2$ and $\alpha_{max} = 2$ for CPU scaling, and $\alpha_{min} = 1.1$ and $\alpha_{max} = 1.5$ for memory scaling. α_{min} is the minimum scale-up granularity we want to achieve during the under-estimation correction. α_{max} is tuned so that we can scale up to the maximum resource cap in two or three steps. We use smaller α_{min} and α_{max} for memory scaling, because the application reacts to the memory scale-up slower than CPU, and scaling up memory too quickly will cause resource waste. Let P denote the current resource pressure. We will trigger under-estimation handling when P exceeds a certain threshold P_{under} (e.g., P_{under} = 0.9). We calculate α as:

$$\alpha = \frac{P - P_{under}}{1 - P_{under}} \cdot (\alpha_{max} - \alpha_{min}) + \alpha_{min} \quad (1)$$

CloudScale currently uses a static pre-defined resource pressure threshold (90% or 75%). According to [45], the threshold actually varies with different workloads and different SLO requirements. A more intelligent way to determine the resource pressure threshold is to learn from the application workload type and the SLO feedback, which is part of our on-going work.

The mapping from application SLO feedback to α can be application specific. For example, when the SLO feedback is the request-response time, we can calculate α as:

$$\alpha = 1 + N_{vio}/N \quad (2)$$

N_{vio} denotes the number of requests that have a response time larger than the SLO violation threshold, and N denotes the total

number of requests during the previous sampling period (e.g., 1 second). For Hadoop applications, the SLO feedback can be job progress scores. We calculate α as $1+(P_{ref}-P)/P_{ref}$, where P_{ref} denotes the desired progress score derived from the target completion time of the job, and P denotes the current progress score.

When both resource pressure and SLO feedback are available, CloudScale chooses the larger one as the final α.

2.3 Scaling Conflict Handling

In this section, we describe how we handle concurrent resource scaling for multiple co-located applications. The key issue is to deal with scaling conflicts when the available resources are insufficient to accommodate all scale-up requirements on a host. We first describe how to predict the conflict. Then we introduce the local conflict handling and migration-based conflict handling schemes. Finally we describe the policy of choosing different conflict handling approaches.

2.3.1 Conflict Prediction

We can resolve a scaling conflict by either rejecting some applications' scale-up requirements or employing VM migration to mitigate the conflict. Both approaches will probably cause SLO violations, but we try to minimize these. We use a conflict prediction model to estimate 1) when the conflict will happen, 2) how serious the conflict will be, and 3) how long the conflict will last.

We leverage our resource demand prediction schemes for conflict prediction, looking further into the future. The scaling system maintains a long-term resource demand prediction model for each VM. Different with the prediction model used by the resource scaling system, which uses 1-second prediction interval, the long-term prediction model uses 10-second prediction interval in order to predict further into the future. We use W_b to denote the length of the look-ahead window of the long-term prediction model (e.g., $W_b =$ 100 seconds). Suppose a host runs K application VMs: m_1,...,m_K. Let $\{r_{i,t+1},...r_{i,t+W_b}\}$ denote predicted future resource demands on the m_i from time $t+1$ to $t+W_b$. We can then derive the total resource demand time series on the host as $\{\sum_{i=1}^{K} r_{i,t+1},...\sum_{i=1}^{K} r_{i,t+W_b}\}$. By comparing this total resource demand time series with the host resource capacity C, we can estimate when a conflict will happen (i.e., $\sum_{i=1}^{K} r_{i,t_1} > C$, t_1 denotes the conflict start time), how serious the conflict will be (i.e., the *conflict degree*: $\sum_{i=1}^{K} r_{i,t_1} - C$), and how long the conflict will last.

2.3.2 Local conflict handling

If the conflict duration is short and the conflict degree is small, we resolve the scaling conflict locally without invoking expensive migration operations. We define *SLO penalty* as the financial loss for the cloud provider when applications experience SLO violations, and we use RP_i (Resource under-provisioning Penalty) to denote the SLO penalty for the application VM m_i caused by one unit resource under-provisioning.

Using local conflict handling, we need to consider how to distribute the resource under-provisioning impact among different applications. CloudScale supports both uniform and differentiated local conflict handling. In the uniform scheme, we set the resource cap for each application in proportion to its resource demand. Suppose the predicted resource demand for the application VM m_i is r_i. We set the resource cap for m_i as $(r_i/\sum_{i=1}^{K} r_i)\cdot C$, where C denotes the total resource capacity on the host. In the differentiated scheme, CloudScale allocates resources based on application priorities or resource under-provisioning penalties (RP_i) of different applications in order to minimize the total penalty. For example, when the VMs have different priorities, CloudScale strives first to satisfy the resource requirements of high-priority applications and only share the under-provisioning impact among low priority applications. CloudScale first ranks all applications according to their priorities, and decides the resource caps of different applications based on the priority rank. If the application's resource demand can be satisfied by the residual resource, CloudScale will allocate the required resource to the application. Otherwise, CloudScale allocates the residual resources to all the remaining applications in proportion to their resource demands. We can apply a similar differentiated allocation scheme when RP_i is used to rank different applications.

We estimate the total SLO penalty for m_i based on the conflict prediction results as $\sum_{k=t_1}^{t_2} RP_i \cdot e_{i,t+k}$, where t_1 and t_2 denote the conflict start and end time, and $e_{i,t+k}$ denotes the under-estimation error at time $t+k$. We aggregate the SLO penalties of all application VMs to calculate the total resource under-provisioning penalty Q_{RP} using the local conflict handling scheme.

2.3.3 Migration-based conflict handling

If we decide to resolve the scaling conflict using VM migration, we first need to decide when to trigger the migration. We observe that Xen live migration is CPU intensive. Without proper isolation, the migration will cause significant SLO impact to both migrated and non-migrating applications on both source and destination hosts. Furthermore, without sufficient CPU, the migration will take a long time to finish, which will lead to a long service degradation time. It is often too late to trigger the migration after the conflict already happened and the host is already overloaded. To address the problem, we use *predictive migration*, which leverages the conflict prediction to trigger migration before the conflict happens. If we want to trigger migration I (e.g., $I = 70s$) before the conflict happens, the migration-based conflict handling module will check whether any conflict that needs to be resolved using migration will happen after time $t+I$, where t denotes the current time. If positive, the module will trigger the migration now at time t rather than wait until the conflict happens later after time $t+I$.

To avoid triggering unnecessary migrations for mis-predicted or transient conflicts, the migration will be triggered only if the conflict is predicted to last continuously for at least K seconds. The value of K denotes the tradeoff between correct predictions and false alarms, and can be tuned by the cloud provider. Typically we set $K = 30s$, which corresponds to three consecutive predicted conflicts using a 10-second prediction interval. As a future work, we will make K a function of the migration lead time I and the VM migration time. We may use larger K for longer migration lead time since it will be more likely to have false alarms given a longer migration lead time. For VMs that have longer migration time, we want to avoid unnecessary migrations by using a larger K for lower false alarm rate.

Next, we need to decide which application VMs should be migrated. Since modern data centers usually have high speed networks, the network cost for migration typically is not the major concern. Instead, our scheme focuses on i) migrating as few VMs as possible, and 2) minimizing SLO penalty caused by migrations. Similar to previous work [41], we consider a normalized SLO penalty metric: $Z_i = MP_i \cdot T_i/(w_1 \cdot cpu_i + w_2 \cdot mem_i)$, where MP_i (Migration Penalty) denotes the unit SLO penalty for the application VM m_i

during the migration[1]; T_i denotes the total migration time for m_i; cpu_i and mem_i denote the normalized CPU and memory utilization of the application VM m_i compared to the capacity of the host. The weights w_1 and w_2 denote the importance of the CPU resource or the memory resource in our decision-making. We can give a higher weight to the bottleneck resource that has lower availability. For example, if the host is CPU-overloaded but has plentiful memory, w_1 can be much larger than w_2 so that we will choose a VM with high CPU consumptions to release sufficient CPU resource. Intuitively, if the application has low SLO penalty during migration and high resource demands, we want to migrate this application first since the migration imposes low SLO penalty to the migrated application and can release a large amount of resources to resolve conflicts. We sort all application VMs using the normalized SLO penalty metric, and start to migrate the application VMs from the one with the smallest SLO penalty until sufficient resources are released to resolve the conflicts.

Finally, we need to decide which host the selected VM should be migrated to. CloudScale relies on a centralized controller to select the destination host for the migrated application. For example, we can use a greedy algorithm to migrate the VMs to the least loaded host that can accommodate the VM [41], or we can find a suitable host by matching the resource demand signature of the VM with the residual resource signature of the host [23].

We calculate the SLO penalty for migrating m_i as $MP_i \cdot T_i$. In our current implementation, we estimate the migration time using a linear function of average memory footprint. The function is derived from a few measurement samples using linear regression. We can then aggregate the SLO penalties of all migrated VMs to derive the total migration penalty Q_M using the migration-based conflict handling scheme.

2.3.4 Conflict Resolution Inference

CloudScale currently decides whether to trigger migration by comparing Q_{RP} and Q_M. If $Q_{RP} \geq Q_M$, CloudScale will not migrate any application VM and resolve the scaling conflict using the local conflict handling scheme. Otherwise, CloudScale migrates selected VMs out until sufficient resources are released to resolve the conflict. As a future work, we can also adopt a hybrid approach that combines both local conflict handling and migration-based conflict handling to minimize the total SLO penalty $Q_{RP} + Q_M$. We can estimate the total SLO penalty $Q_{RP} + Q_M$ of migrating different subsets of VMs, and choose the migrated subset that minimizes the total SLO penalty.

The unit SLO penalty values RP_i and MP_i are application dependent. We assume that these are provided to CloudScale by the user. Typically, batch processing applications (e.g., long running MapReduce jobs) are more tolerant of short periods of service degradation than time-sensitive interactive applications such as Web transactions.

2.4 Predictive Frequency/Voltage Scaling

CloudScale integrates VM resource scaling with dynamic voltage and frequency scaling (DVFS) to transform unused resources into energy savings without affecting application SLOs. For example, if the resource demand prediction models indicate that the total CPU resource demand on a host is 50%, we can then half the CPU frequency and double the resource caps of all application VMs. Thus, we can reduce energy consumption since the CPU runs at a slower speed but the application's SLO is unaffected. Another way of saving energy is to let the application run as fast as possible and then shutdown the machine. However, the host in the multi-tenant cloud system often runs some interactive foreground jobs that are expected to operate 24x7. Thus, we believe that slowing down the CPU is a more practical solution in this case.

Modern processors often can run at a range of frequencies and voltages and support dynamic frequency/voltage scaling. Suppose the host processor can operate at k different frequencies: $f_1 < ... < f_k$ and the current frequency is f_i. For example, the processor in our experimental testbed supports 11 different frequencies. We want to slow down the CPU based on the current resource cap information to ensure that application performance is not affected. Let C' and C denote the total CPU demand by all the application VMs and the CPU capacity of the host, respectively. We can then derive the current CPU utilization as C'/C. If the current host does not have a full utilization (i.e., $C'/C < 1$) and the current frequency f_i is not the lowest, CloudScale picks the lowest frequency f_j that meets the condition: $f_j \geq C'/C \cdot f_i$. If $f_j < f_i$, we scale down the CPU frequency to f_j. We then multiply the resource caps for all application VMs by f_i/f_j to maintain the application SLOs. To maintain the accuracy of the resource demand prediction, we also need to scale up the stored resource demand training data by f_i/f_j to match the new CPU frequency.

If the processor is not running at the highest frequency, we can increase the CPU speed to try to resolve scaling conflicts. For example, if the future CPU demand is predicted to be $C', C' > C$ and the current frequency is f_i, we will scale up the CPU frequency to the slowest one f_j among $f_{i+1}, ...f_k$ such that $f_j/f_i > C'/C$. After we set the CPU frequency to f_j, we will scale down the resource cap r_i for each application VM to $r_i \cdot (f_i/f_j)$ to match the new CPU frequency. After we reach the highest frequency, we will resort to either local conflict handling or migration-based conflict handling, as described in the previous section.

3. EXPERIMENTAL EVALUATION

We implemented CloudScale on top of the Xen virtualization platform and conducted extensive evaluation studies using the RUBiS [4] online auction benchmark (PhP version), Hadoop MapReduce systems [2, 15], and IBM System S data stream processing system [21]. This section describes our results.

3.1 Experiment setup

Most of our experiments were conducted in the NCSU's Virtual Computing Lab (VCL) [6]. Each VCL host has a dual-core Xeon 3.00GHz CPU, 4GB memory and 100Mbps network bandwidth, and runs CentOS 5.2 64bit with Xen 3.0.3. The guest VMs also run CentOS 5.2 64bit and have one virtual CPU core (the smallest scheduling unit in Xen hypervisor, similar to the task in Linux kernel).

The integrated VM scaling and DVFS experiments were conducted on the Hybrid Green Cloud Computing (HGCC) cluster in our department since VCL hosts are not equipped with power meters. Each HGCC node has a quad-core Xeon 2.53GHz processor, 8GB memory and 1Gbps network bandwidth, and runs CentOS 5.5 64 bit with Xen 3.4.3. The processor supports 11 frequency steps between 2.53 and 1.19Ghz. We used Watts Up power meters to get real time power readings from HGCC hosts. The guest VM OS is the same with the VCL host. We use Intel SpeedStep technology to perform DVFS. We run our systems on DVFS enabled Linux 2.6.18 kernel and control the CPU frequency from the host OS using the Linux CPUfreq subsystem.

In all of our experiments, we pin down Domain 0 to one core and

[1]Although Xen live migration shortens the VM downtime, the application may experience a period of high SLO violations due to the memory copy.

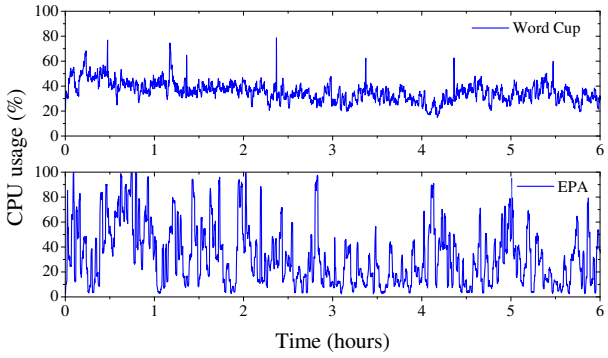


Figure 4: The real uncapped CPU demand of the RUBiS web-server under two different workloads.

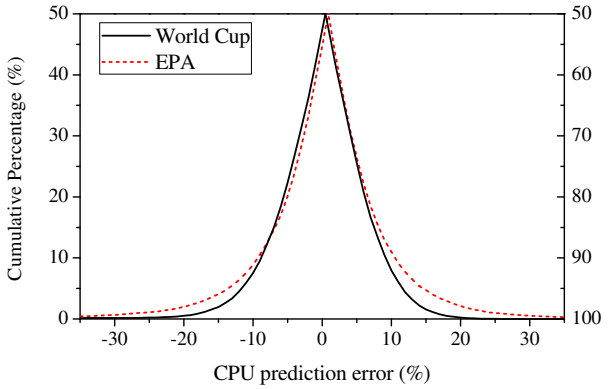


Figure 5: Folded cumulative distribution of CPU resource demand prediction errors for two different workloads. The left half is a normal CDF using the y-axis scale on the left; the right half, using the scale on the right, is the reflected upper half of the CDF.

run all guest VMs on another core. CloudScale runs within Domain 0.

CloudScale performs fine-grained monitoring by frequently sampling all resource metrics and repeatedly updates the resource demand prediction model using a number of recent resource usage samples. The resource scaling period, the sampling period, prediction model update period, and training data size are all tunable parameters. For CPU scaling, CloudScale uses a 1 second scaling and sampling period, 10 second prediction model update period, 100 recent resource usage samples as training data for the short-term resource demand prediction model, and 2000 recent resource usage samples as training data for the long-term conflict prediction model. For memory scaling, the default setup is the same as CPU scaling except that the scaling period is 10 seconds, and the training data set for short-term resource demand prediction model contains 1000 recent resource usage samples. We found that the default settings work well for all of the applications used in our experiments.

A service provider can either rely on the application itself or an external tool [11] to keep track of whether the application SLO is violated. In our experiments with RUBiS and IBM System S, we adopted the latter approach, using the workload generator to track the response time of the HTTP requests it made or the processing day of each stream data tuple. In RUBiS, the SLO violation rate is the fraction of requests that have response time larger than the pre-defined SLO threshold (200 ms) during each experiment run.

Scheme	Prediction Error Correction	Online Adaptive Padding
Correction	Resource pressure	none
Dynamic padding	none	Dynamic
Padding-X%	none	Constant X%
CloudScale RP	Resource pressure	Dynamic
CloudScale RP + SLO	Resource pressure + SLO feedback	Dynamic

Table 1: Configurations of different schemes.

In System S, the SLO violation rate is the fraction of data tuples that have processing delay larger than the pre-defined SLO threshold (20 ms). In Hadoop experiments, we used the progress score provided by the Hadoop API as the SLO feedback, and transform the target job completion time into the desired progress score. The SLO violation rate is sampled every second in RUBiS and System S. In Hadoop, since calling the Hadoop API to get the progress score will take a long time (> 1 minute) to return, we try to get the updated progress score as fast as possible by calling the API again immediately after getting the returned value.

To evaluate CloudScale under workloads with realistic time variations, we used per-minute workload intensity observed in real-world Web traces [5] to modulate the request rate of the RUBiS benchmark and the input data rate to the System S stream processing system. We constructed two workload time series: 1) the request rate observed in each minute of the six-hour World Cup 98 web server trace starting at 1998-05-05:00.00; and 2) the request rate observed in each minute of the six-hour EPA web server trace starting at 1995-08-29:23.53.

For comparison, we also implemented a set of alternative schemes and several variations of the CloudScale system, summarized in Table 1: 1) *Correction*: the scaling system performs resource pressure triggered prediction error correction only; 2) *Dynamic padding*: the scaling system performs dynamic padding only; 3) *Padding-x%*: the scaling system performs a constant percentage padding by adding x% predicted value; 4) *CloudScale RP*: the scaling system performs both dynamic padding and scaling error correction, and the scaling error correction is only triggered by resource pressure; and 5) *CloudScale RP+SLO*: the scaling system performs both dynamic padding and scaling error correction, and the scaling error correction is triggered by both resource pressure and SLO feedback.

3.2 Results

Figure 4 shows the *real CPU demand* (the CPU usage achieved with no resource caps) for the RUBiS Web server under two test workload patterns. Both workloads have fluctuating CPU demands. We focus on CPU resource scaling in RUBiS experiments since it appears to be the bottleneck in this application.

Figure 5 shows the accuracy of online resource demand prediction using folded cumulative distributions. The results show that the resource demand prediction makes less than 5% significant under-estimation or over-estimation errors (i.e., $|e| > 10\%$) for the World Cup workload and about 10% significant under-estimation or over-estimation errors for the EPA workload.

We conducted experiments for both single VM and multiple VMs. For single VM case, the VM hosts a RUBiS web server. For the multi-VM case, there were two VMs running on the same physical host, each hosting a RUBiS Web server driven by the World Cup workload and the EPA workload respectively. The database servers run on different physical hosts. During the multi-VM scaling experiment, all the VMs have equal priority.

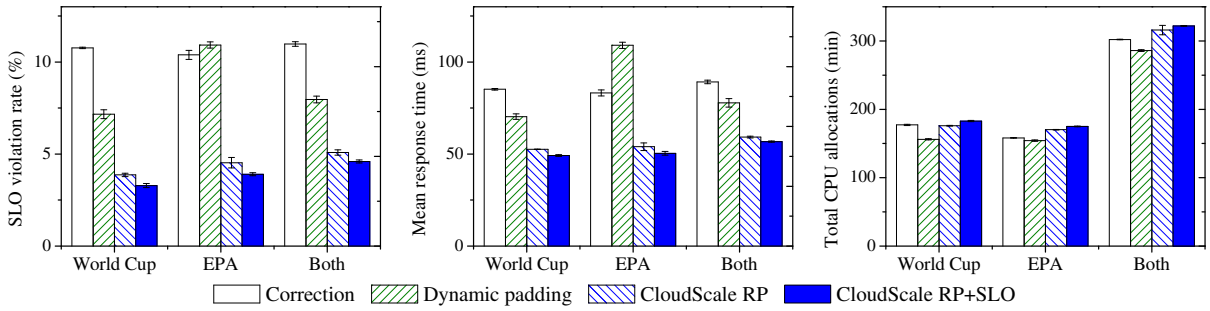


Figure 6: Performance comparison for different prediction-based scaling algorithms, maintaining 90% resource pressure. The left figure shows the mean SLO violation rate of the RUBiS system under the World Cup 98 workload, the EPA workload, and both workloads. The middle figure shows the mean request-response time of RUBiS system under different workloads. The right figure shows the total CPU allocation to the application VMs under different workloads.

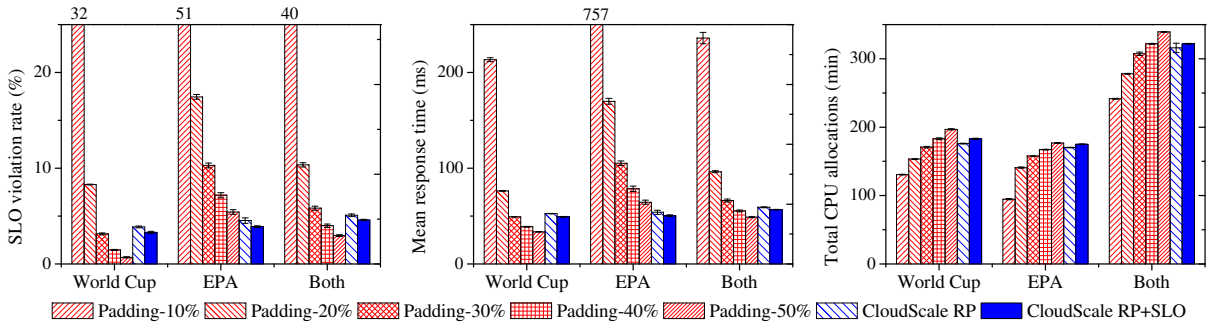


Figure 7: Performance comparison for CloudScale and constant padding algorithms, maintaining 90% resource pressure. The left figure shows the mean SLO violation rate of the RUBiS system under the World Cup 98 workload, the EPA workload, and both workloads. The middle figure shows the mean request-response time of RUBiS system under different workloads. The right figure shows the total CPU allocation to the application VMs under different workloads.

Figure 6 and Figure 7 show the performance and total CPU allocations of different scaling schemes. In these experiments, the resource pressure threshold to trigger under-estimation error correction is set at 90% and the SLO triggering threshold is set at 5% requests experience SLO violation (i.e., response time $>$ 200ms). The total CPU allocation is calculated based on the resource cap set by different schemes. Each experiment is repeated three times and we report both mean and standard deviations.

Figure 6 shows that CloudScale can achieve lower SLO violation rate and smaller response time than other schemes. Both reactive correction and dynamic padding when used alone can partially alleviate the problem. But dynamic padding works better for the World Cup workload, while the reactive correction works better for the EPA workload, because the EPA workload shows more fluctuations and reactive correction becomes more important. We also observe that runtime SLO feedback is helpful, but not vital, for CloudScale to reduce SLO violations. CloudScale also achieves better performance than other schemes in the multi-VM concurrent scaling case.

Figure 7 shows the performance comparison between CloudScale with different constant padding schemes. The results show that if we pad too little (e.g., padding-10%), we have high SLO violations and if we pad too much (e.g., padding-50%), we have low SLO violation rate but at high resource cost. When the resource allocation reaches a certain threshold, more padding does not reduce the response time and SLO violation rate much. More importantly, it is hard to decide how much to pad in advance if we use constant padding schemes. In contrast, CloudScale does not need to specify the padding percentage, and is able to adjust the padding during runtime automatically.

We learned from our experiments that the prediction-only scheme performs poorly without under-estimation correction and padding, which is not shown in the figures. The reason is that the prediction model cannot get the real resource demand when under-estimation error happens, and can only learn the distorted demand values. Since the application cannot consume more resources than the resource cap, once the resource cap is pushed down, it will not be raised anymore. In this case, the scaling system will predict and allocate less and less resources, and the application will suffer from severe resource under-provisioning.

Figure 8 and Figure 9 shows the results of the same set of experiments but with the resource pressure threshold set at 75%. The SLO violation rates are consistently lower for all algorithms when compared to the 90% resource pressure threshold cases. Note that CloudScale RP+SLO achieves much better performance in this case. That is because CloudScale RP+SLO considers both resource pressure and SLO feedback. Moreover, since the SLO triggering threshold is set at 5%, and we derive the scale-up ratio α using equation 2, α derived from SLO feedback is typically very small and the resource pressure feedback becomes the dominant factor in triggering scaling error corrections. We can see that CloudScale still achieves the best application performance with low resource cost. The benefit of CloudScale is more significant for EPA trace, achieving much lower SLO violations and smaller response time than the generous constant padding scheme "padding-50%" but with a similar resource cost. The reason is that the EPA trace is very bursty

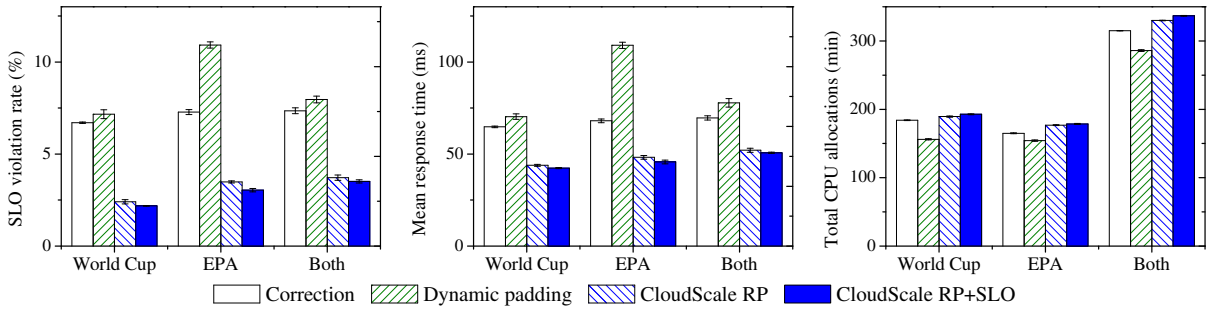


Figure 8: Performance comparison for different prediction-based scaling algorithms, maintaining 75% resource pressure. The left figure shows the mean SLO violation rate of the RUBiS system under the World Cup 98 workload, the EPA workload, and both workloads. The middle figure shows the mean request-response time of RUBiS system under different workloads. The right figure shows the total CPU allocation to the application VMs under different workloads.

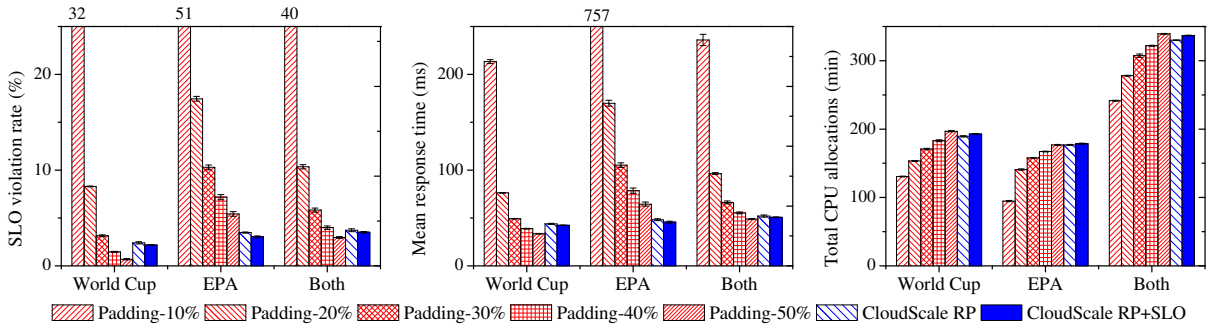


Figure 9: Performance comparison for CloudScale and constant padding algorithms, maintaining 75% resource pressure. The left figure shows the mean SLO violation rate of the RUBiS system under the World Cup 98 workload, the EPA workload, and both workloads. The middle figure shows the mean request-response time of RUBiS system under different workloads. The right figure shows the total CPU allocation to the application VMs under different workloads.

and intelligent handling of under-estimation plays a critical role in this case.

Figure 10 shows the energy saving effectiveness of the predictive CPU scaling. We repeated the same RUBiS experiments as the CloudScale RP+SLO scheme on the HGCC cluster, and set the resource pressure threshold as 90%. We ran each experiment for 6 hours and measured the total energy consumption for both CloudScale without DVFS and with DVFS. Without DVFS, the CPU runs at the maximum frequency. The idle energy consumption is measured when the host is idle and stays at its lowest power state. The workload energy consumption is derived by subtracting the idle energy consumption from the total energy consumption. The results show that CloudScale with DVFS enabled (CloudScale DVFS) can save 8-10% total energy consumption, and 39-71% workload energy consumption with little impact to the application performance and SLO conformance. We also observe that compared to the workload energy consumptions, the idle energy consumptions are dominating in all cases. This is because all HGCC hosts are powerful quad-core machines and each experiment run only uses two cores: one core for the application VM and one core for Domain 0. We believe that CloudScale DVFS can achieve higher total energy saving when more cores are utilized.

We now evaluate our conflict handling schemes. We run two RUBiS web server VMs on the same host, and maintain a 75% resource pressure with dynamic padding. The local conflict handling uses the uniform handling policy that treats the two VMs uniformly. The memory size of VM1 is 1GB, and the memory size of VM2 is 2GB, so the migration time of VM2 is much longer than VM1. We sample the SLO violation rate every second for both VMs, and calculate the total time that the application experiences different SLO violation rates. Figure 11(a) shows the SLO violation time under the local conflict handling scheme. We can see that when conflict happens, both VMs suffer from high SLO violation rates for a long period of time. The total time that both VMs experience SLO violations adds up to 351 seconds. We then evaluate the migration-based conflict handling schemes. We first test with the reactive migration scheme where the scaling system triggers the live migration to migrate VM2 out after it detects a sustained scaling conflict using the algorithm proposed in [41]. In Figure 11(b), we can see that the migration reduces the SLO violation time significantly. The total time of having SLO violations becomes 92 seconds. However, the live migration still takes a long time to finish when the system is overloaded and both non-migrating and migrated VMs experience significant SLO violations during the migration. We then enable the CloudScale's VM selection algorithm that selects the migrated VM based on the normalized SLO penalty metric. In this experiment, we used equal weights for CPU and memory. We set $RP_1 = RP_2 = 1$ and $MP_1 = MP_2 = 8$ (RP_1 and RP_2 denote the unit resource under-provisioning penalties for VM1 and VM2 respectively; MP_1 and MP_2 denote the unit migration penalty for VM1 and VM2 respectively). We tried different ratios between MP and RP, and find that setting $MP/RP = 8$ provides a reasonable tradeoff between local conflict resolving and migration. The migration time is estimated using the regression-derived function shown by Figure 14. In this

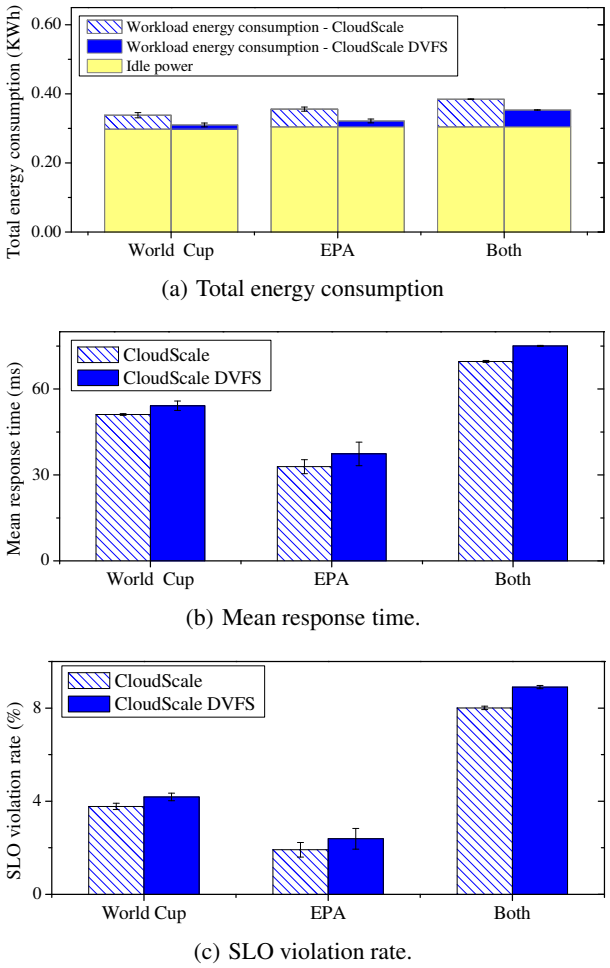


Figure 10: Effect on energy consumption and application performance of RUBiS application using predictive frequency/voltage scaling (maintaining 90% resource pressure).

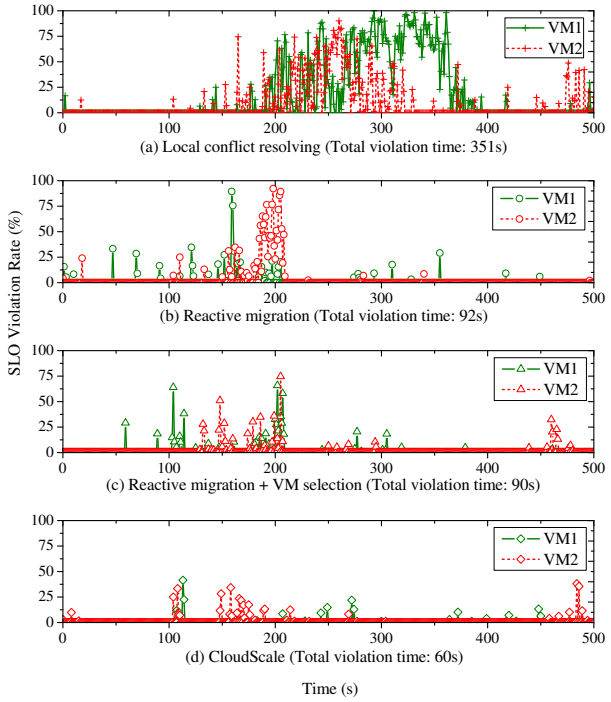


Figure 11: Time series of SLO violation rate for different scaling conflict resolving schemes.

case, the system selects VM1 instead of VM2 to migrate out. From Figure 11(c) we can see that although the total SLO violation duration is similar to the previous case (90 seconds), the SLO violation rates are much smaller.

We then repeat the same experiment using CloudScale's conflict handling scheme. The predictive migration triggers the migration of VM1 about 70 seconds before the conflict happens. From Figure 11(d) we can see that the live migration can finish in a short period of time and both VMs experience shorter SLO violation time. The total SLO violation duration is reduced to 60 seconds.

Figure 12 shows the cumulative percentage of continuous SLO violation duration under different SLO violation rates. When resolving the conflict locally, more than 10% of the violation durations are longer than 5 seconds. Using reactive migration, less than 5% of the SLO violation durations are more than 5 seconds, but the application can still experience up to 19 seconds of continuous SLO violations. After enabling our VM selection algorithm, the maximum continuous SLO violation time becomes 4 seconds. In contrast, when using CloudScale's conflict handling scheme, the continuous SLO violation durations are always less than 2 seconds, and 90% of the continuous SLO violation durations are only 1 second.

Figure 13 shows the cumulative percentage of SLO violation rates using different scaling conflict schemes. When using CloudScale's conflict handling scheme, the application does not experience any SLO violation for 94% of the time, and the SLO violation rate is less than 20% for 99% of the time. All of the other schemes incur higher SLO violation rates than CloudScale.

To measure the accuracy of our conflict prediction algorithms, we used six hours of CPU demand traces for two RUBiS web servers used in previous experiments for different scaling schemes. We first mark the start time of all significant conflicts, then use our conflict prediction algorithms to predict the conflict within different time windows. By comparing predicted conflicts with true conflicts, we calculate the number of true positive predictions (N_{tp}): the conflicts that are predicted correctly; the number of false-negative predictions (N_{fn}): the conflicts that were not predicted; the number of false-positive predictions (N_{fp}): the predicted conflicts that did not happen; and the number of true-negative predictions (N_{tn}): the non-conflicts that are predicted correctly. The true positive rate A_T and false alarm rate A_F are defined in a standard way as $A_T = N_{\text{tp}}/(N_{\text{tp}}+N_{\text{fn}})$; $A_F = N_{\text{fp}}/(N_{\text{fp}}+N_{\text{tn}})$. Figure 15 shows the true positive and false positive rate of our conflict prediction algorithms under different migration lead time requirements. As expected, the prediction accuracy decreases with a longer migration lead time (i.e., triggering migration earlier). However, since predictive migration is triggered before conflict happens, the SLO impact of false alarms is small. We plan to further improve the accuracy of the conflict prediction algorithm in our future work.

We then apply the scaling system to a Hadoop MapReduce application. Unlike RUBiS, the Hadoop application is memory intensive, so we focus on memory scaling in this set of experiments. We used the Word Count and Grep MapReduce sample applications. One VM holds all the map tasks while another VM holds all the reduce tasks. The number of slots for map tasks or reduce tasks is set to 2 on both nodes. Figure 16 shows the real memory

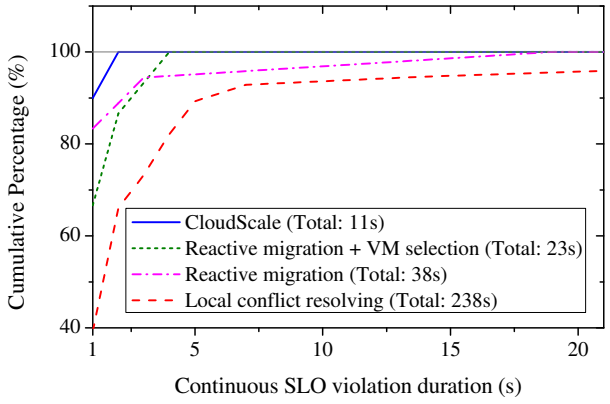


Figure 12: CDF of continuous SLO violation duration (when SLO violation rate is larger than 20%) comparison among different scaling conflict resolving schemes for the RUBiS system.

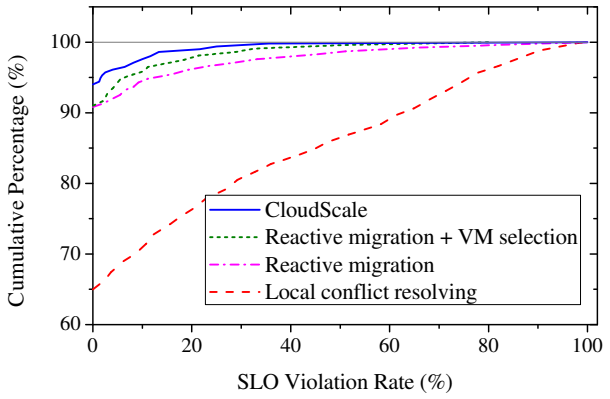


Figure 13: CDF of SLO violation rate comparison among different scaling conflict resolving schemes for the RUBiS system.

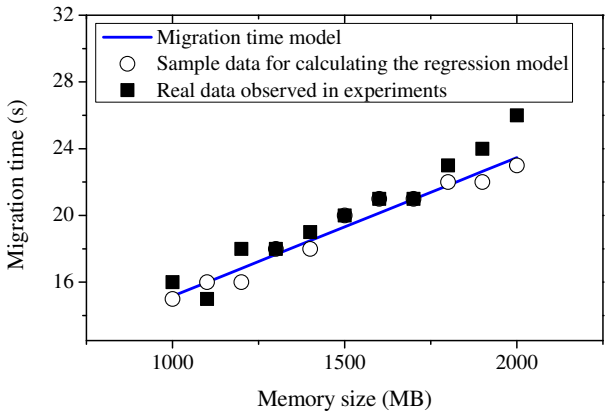


Figure 14: Migration time estimation model. The model is a linear regression of the sample data we got by measuring the migration time of the VMs with different memory sizes. We also put the real data that we observed in the experiments to show the accuracy of the model.

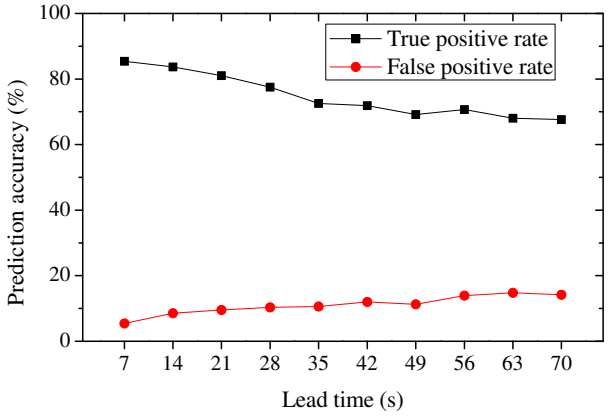


Figure 15: Conflict prediction accuracy under different migration lead time requirements.

demand (the memory footprint achieved with no memory cap) of the VM hosting map tasks. We can see that the memory footprint of the VM fluctuates a lot. The peak memory footprint of word count is 591MB, and the peak memory footprint of grep is 579MB. Without scaling, we have to perform memory allocation based on the maximum memory footprint, which will cause memory waste. Moreover, it is hard to get the maximum memory footprint in advance. We apply predictive memory scaling on Hadoop. Figure 17 shows the prediction accuracy of memory scaling. As with CPU, CloudScale can achieve good prediction accuracy for memory.

Figure 18 shows the job completion time and average memory caps of different scaling schemes. Since the progress score of map tasks is more accurate than that of reduce tasks, we only performed memory scaling on the map VM and the reduce VM is always given sufficient memory. The resource pressure threshold is set as 90%. We observe that CloudScale can achieve the shortest job completion time with low memory cap. We use "mean" to denote the static memory allocation scheme that allocates a fixed amount of memory based on the average memory footprint in the real memory demand trace. The results show that this static scheme works poorly, which significantly increases the job completion time: the system spends a long time on swapping memory pages when more memory is needed. In contrast, CloudScale does not know the exact memory demand in advance, but still achieves better performance. The effect of progress score feedback is not significant. As mentioned before, calling the Hadoop API to get the progress score will take a long time (10 to 60 seconds) to return, especially when the node is busy with memory page swapping. When the application is suffering from resource under-provisioning, CloudScale cannot trigger under-estimation handling in time because it cannot get the SLO feedback immediately.

We now evaluate CloudScale on IBM System S, a production data stream processing system. We run one of the sample applications provided by System S. It is a tax calculation application that takes commodity records including commodity name, seller, price, quantity and state as the input stream tuples and calculates the final price for each tuple based on the tax rates of different states. We used the sample data provided by the System S. To emulate dynamic data arrivals, we used the World Cup and EPA workload to regulate the input data rate. The average input rate is about 1 million data tuples per second. The application consists of 7 distributed processing elements (PEs). We run each PE within one VM. All VMs are deployed on different hosts. Since CloudScale focuses on resource scaling within single host, we perform CPU scaling on one of the PEs and always give sufficient resources to the other PEs. We set the resource pressure to 90%, and measure the

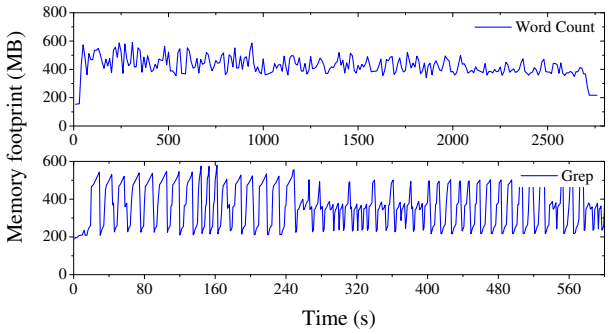


Figure 16: The memory footprint trace of the Map tasks of Hadoop MapReduce applications.

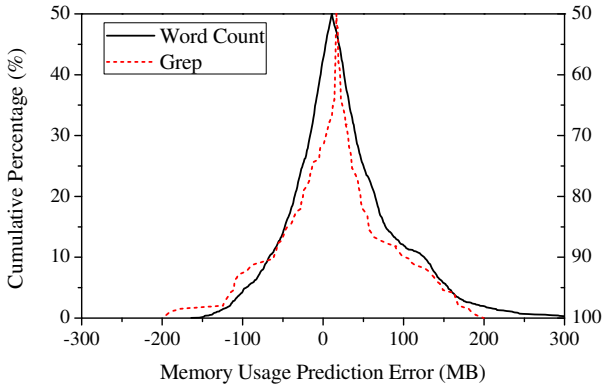


Figure 17: Folded cumulative distribution of memory resource demand prediction errors for two different Hadoop MapReduce applications. The left half is a normal CDF using the y-axis scale on the left; the right half, using the scale on the right, is the reflected upper half of the CDF.

per-tuple processing delay. Figure 19 shows the performance and average CPU cap achieved by different scaling algorithms. The results show that CloudScale can achieve much lower processing delay than other prediction-based scaling schemes with low CPU caps.

We now evaluate the overhead of the CloudScale system. Table 2 shows the CPU overhead of all the key operations in CloudScale. The results show that CloudScale is light-weight for all operations. In our experiment environment, running CloudScale only consumes about 2% CPU resource in Domain 0. Thus, we believe that CloudScale is suitable for large-scale cloud systems.

4. RELATED WORK

Existing production cloud system scaling techniques such as Amazon Auto Scaling [1] is not fully automatic, which depend on the user to define the conditions for scaling up or down resources. However, it is often difficult for the user to figure out the proper scaling conditions, especially when the application will be executed on a third-party virtualized cloud computing infrastructure.

Several projects [30, 38, 9] studied coarse-grained capacity scaling scheme by dynamically adding or releasing server nodes in a particular system tier. The tier scaling schemes focus on determining how many server hosts are needed in each tier using queueing theory [38], machine learning [9], or control theory [30], and how to rebalance workload among replicated server instances. In comparison, our work focuses on fine-grained VM-level resource scaling, which can be used on each server node to adaptively adjust resource allocation to different VMs for reducing resource and energy cost. Our scaling scheme is complementary to the host-level tier scaling scheme.

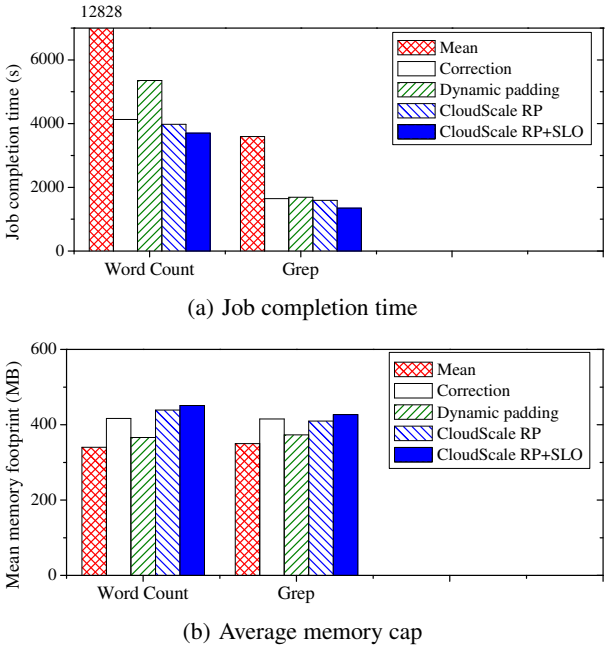


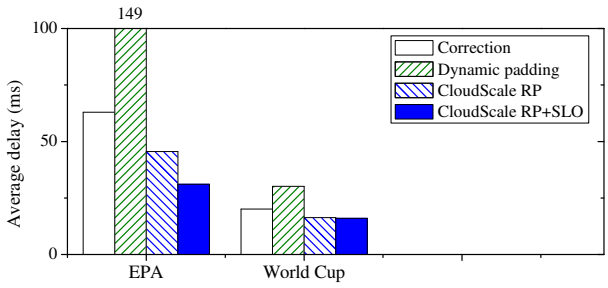
Figure 18: Performance of memory scaling on two different Hadoop MapReduce applications (maintaining 90% resource pressure).

Operations	CPU cost
Model training (100 samples)	69.7 ± 0.3 ms
Prediction	0.1 ± 0.0 ms
Dynamic padding (100 samples)	1.3 ± 0.1 ms
CPU resource scaling	4.0 ± 0.1 ms
Memory resource scaling	9.4 ± 0.3 ms

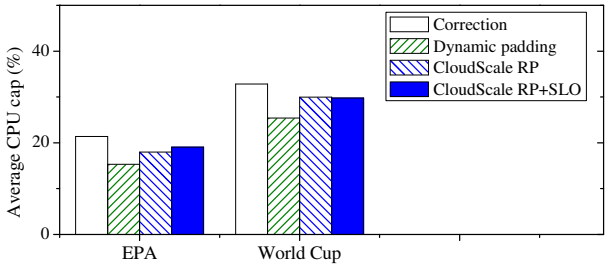
Table 2: Mean and standard deviation of CPU execution costs for all core operations in CloudScale, averaged over 300 operations on Xeon 3.0GHz CPU.

Previous work [44, 29, 31, 36, 32] has extensively studied applying control theory to achieve adaptive fine-grained resource allocations based on SLO conformance feedback. However, those approaches often have parameters that need to be specified or tuned offline, and need some time to converge to the optimal (or near-optimal) decisions. In contrast, CloudScale does not require any offline tuning and can achieve elastic resource allocation without assuming any prior knowledge about applications.

Our work is closely related to trace-driven resource allocation schemes. Rolia et al. [33] proposed a dynamic resource allocation scheme by multiplying estimated resource usage with a burst factor that is derived offline based on different QoS levels. In contrast, our scheme performs online burst pattern extraction and uses the burst pattern to dynamically decide the padding value. Chandra et al. [12] proposed workload prediction using auto-regression and histogram based methods. Gmach et al. [22] used a Fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. Our previous system PRESS [24] pro-



(a) Average response time



(b) Average CPU cap

Figure 19: Performance of CPU scaling on IBM System S (maintaining 90% resource pressure).

vides a hybrid resource demand prediction scheme that can achieve both high accuracy and low overhead. In contrast, this work focuses on efficiently handling prediction errors and concurrent scaling conflicts to achieve elastic resource scaling for multi-tenant cloud systems.

Previous work has proposed model-driven resource allocation schemes. Those approaches use statistical learning methods [34, 37, 20, 35] or queueing theory [17] to build models that allow the system to predict the impact of different resource allocation policies on the application performance. However, those models need to be built with non-trivial overhead and calibrated in advance. Moreover, the resource allocation system needs to assume certain prior knowledge about the application and the running platform (e.g., input data size, cache size, processor speed), which often is impractical in the cloud system. In contrast, CloudScale is completely application and platform agnostic, which makes it more suitable for cloud computing infrastructures that often host third-party applications. Other work has used offline or online profiling [39, 40, 43, 25] to experimentally derive application resource requirements using benchmark or real application workloads. However, profiling needs extra machines and may take a long time to derive resource requirements. Our experiments using the perfect prediction scheme also show that profiling often does not work well for fine-grained resource control since a tiny shift in time will cause significant performance impact.

Virtual machine migration [14] has been widely used for dynamic resource provisioning. Sandpiper [41] is a system that automates the task of monitoring and detecting hotspots, determining a new mapping of physical to virtual resources, and initiating necessary migrations. It uses both black-box and gray-box approach to detect hotspots and determine resource provisioning. The migration is triggered in Sandpiper when a certain metric exceeds some threshold for a sustained time and the next predicted value also exceeds the threshold. In comparison, our work leverages live VM migrations to resolve significant scaling conflicts. Our scheme leverages long-term prediction to trigger migration *before* conflict happens and makes migration decisions based on the impact of migration to application SLOs. Entropy [27] is a resource manager for homogeneous clusters, which performs dynamic consolidation based on constraint programming and takes migration overhead into account. Entropy assumes that the resource demand is known in advance. In contrast, our work focuses on addressing the challenge in predicting the resource demand and the impact of migration.

Although initial work [19, 42] on power saving focused on mobile devices, it has become increasingly important to consider energy saving while managing large-scale hosting centers. Muse [13] is one pioneering work that integrates energy management into comprehensive resource management. It proposed an economic approach to adaptive resource provisioning and an on-power capacity scaling system that can adaptively turn on/off some hosts based on the workload needs. ACES [26] is an automatic controller for energy-aware server provisioning that provisions servers to meet workload demand while minimizing the energy, maintenance and reliability cost. ACES tries to balance the tradeoff between energy savings and reliability impact due to on-off cycles. It uses regression analysis to predict workload demand in the near future, and has a model to quantify the reliability impact in terms of its dollar cost. ACES focuses on using low power states (off, sleep, hibernate) instead of DVFS. In comparison, our work focuses on integrating fine-grained resource scaling and DVFS to achieve energy saving. Our approach is complementary to Muse and ACES system, which can be particularly useful for multi-tenant cloud systems when host shutdown is not an option. DVFS has been shown to be effective for reducing power consumption of large-scale computer systems [28]. Previous work (e.g., [16]) focuses on OS level task characterization and uses learning algorithms to estimate the best suited voltage and frequency setting. Fan et al. [18] used simulation to calculate the potential of power and energy saving in large scale systems using power management techniques based on DVFS. It considers triggering DVFS according to different CPU utilization thresholds. In comparison, our scheme integrates DVFS with VM scaling and leverages predicted resource caps to derive the proper frequency/voltage setting.

5. FUTURE WORK

Although demonstrated efficient in experiments, CloudScale has several limitations which we plan to address in our future work.

CloudScale currently uses a pre-defined resource pressure threshold. Although the resource pressure maintenance and SLO feedback handling can work together, CloudScale currently does not adjust resource pressure threshold dynamically according to the workload type or SLO feedback. However, different types of applications might need varying resource pressure thresholds for triggering the under-estimation handling. For example, an interactive application typically needs to avoid high resource pressure for maintaining sufficient resources to serve any requests as soon as they arrive. In contrast, for batch jobs, we can afford to maintain a tight resource pressure to achieve high resource utilization without significant SLO violations. To make CloudScale more intelligent, we can automatically tune the resource pressure threshold based on some general knowledge about the application (e.g. interactive v.s. batch jobs) or coarse-grained SLO feedback.

CloudScale can scale on different metrics independently, but does not coordinate the scaling operations on them. It is a non-trivial research task to efficiently handle potential interference between different resource scaling operations. The problem is that adjusting the allocation of one resource type can affect the usage of another

type of resource, which might introduce more dynamics into the system and cause more prediction errors. To address the problem, we plan to investigate multi-metric prediction model that can predict multiple metrics together and scale them concurrently.

Similar to multi-metric scaling, it is also challenging to handle multi-tier application scaling, in which different tiers have inter-dependency and scaling on one tier can affect the others. We can integrate CloudScale with host-level scaling techniques [30, 38, 9] to handle multi-tier application scaling efficiently by predicting the resource demand of different tiers at the same time and coordinating the scaling operation on different hosts.

CloudScale performs long-term conflict prediction by extracting the repeating pattern in the resource usage trace. When the repeating pattern is not found, CloudScale relies on multi-step Markov prediction algorithms for long-term predictions. However, multi-step Markov prediction has limited prediction accuracy since the correlation between the resource prediction model and the actual resource demand becomes weaker as we look further into the future. We are investigating other long-term prediction models to better handle the case when no periodic pattern is found in the training data.

CloudScale currently works in the capping mode, which isolates co-located applications by ensuring that the application cannot consume more resources than those allocated to it. Xen credit scheduler also supports a weight mode: assigning each VM a weight which indicates the relative CPU share of the VM. CloudScale can be easily extended to support weight mode by adjusting the weight of the VMs dynamically based on the resource demand prediction. In contrast to the capping mode, VMs can consume residual CPU resources out of their shares in weight mode. However, when resource contention happens, weight mode cannot provide performance isolation, and it is impossible to know the real demand of the collocated VMs since their resource usages are affected by each other. As a future work, we will leverage our conflict prediction to dynamically switch between capping mode and weight mode. When there is no conflict, CloudScale can work in weight mode to improve the resource utilization. When there are conflicts, CloudScale can work in capping mode to ensure performance isolation.

6. CONCLUSION

In this paper, we presented *CloudScale*, an automatic elastic resource scaling system for multi-tenant cloud computing infrastructures. CloudScale consists of three key components: 1) combining online resource demand prediction and efficient prediction error handling to meet application SLOs with minimum resource cost; 2) supporting multi-VM concurrent scaling with conflict prediction and predicted migration to resolve scaling conflicts with minimum SLO impact; and 3) integrating VM resource scaling with dynamic voltage and frequency scaling (DVFS) to save energy without affecting application SLOs. We have implemented CloudScale on top of the Xen virtualization platform and conducted extensive experiments using the RUBiS benchmark driven by real Web server traces, Hadoop MapReduce systems, and a commercial stream processing system. The experimental results show that CloudScale can achieve much better SLO conformance than other alternative schemes with low resource cost. CloudScale can resolve scaling conflicts with up to 83% less SLO violation time than other schemes. CloudScale can save 8-10% total energy consumption, and 39-71% workload energy consumption with little impact to the application performance and SLO conformance. CloudScale is light-weight and application-agnostic, which makes it suitable for large-scale cloud systems.

7. ACKNOWLEDGEMENT

This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF, ARO, or U.S. Government. The authors thank the anonymous reviewers for their insightful comments.

8. REFERENCES

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] Apache Hadoop System. http://hadoop.apache.org/core/.

[3] KVM (Kernel-based Virtual Machine). http://www.linux-kvm.org/page/Main_Page.

[4] RUBiS Online Auction System. http://rubis.ow2.org/.

[5] The IRCache Project. http://www.ircache.net/.

[6] Virtual Computing Lab. http://vcl.ncsu.edu/.

[7] VMware Virtualization Technology. http://www.vmware.com/.

[8] Xen Credit Scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler.

[9] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In *Proc. CIDR*, 2009.

[10] P. Barham and et al. Xen and the art of virtualization. In *Proc. SOSP*, 2003.

[11] D. Breitgand, M. B.-Yehuda, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. NAP: a building block for remediating performance bottlenecks via black box network analysis. In *Proc. ICAC*, 2009.

[12] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proc. IWQoS*, 2004.

[13] J. Chase, D. Anderson, P. N. Thakar, and A. M. Vahdat. Managing energy and server resources in hosting centers. In *Proc. SOSP*, 2001.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI*, 2005.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. Dec. 2004.

[16] G. Dhiman and T. S. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proc. ISLPED*, 2007.

[17] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-based resource provisioning in a web service utility. In *Proc. USITS*, 2003.

[18] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. ISCA*, 2007.

[19] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. SOSP*, 1999.

[20] A. Ganapathi, H. Kuno, and et al. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. ICDE*, 2009.

[21] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. *Proc. SIGMOD*, 2008.

[22] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity

management and demand prediction for next generation data centers. In *Proc. ICWS*, 2007.

[23] Z. Gong and X. Gu. PAC: Pattern-driven Application Consolidation for Efficient Cloud Computing. In *Proc. MASCOTS*, 2010.

[24] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for Cloud Systems. In *Proc. CNSM*, 2010.

[25] S. Govindan, J. Choi, and et al. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proc. Eurosys*, 2009.

[26] B. Guenter, N. Jain, and C. Williams. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *Proc. INFOCOM*, 2011.

[27] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proc. VEE*, 2009.

[28] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Proc. SC*, 2005.

[29] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *Proc. ICAC*, 2009.

[30] H. Lim, S. Babu, and J. Chase. Automated control for elastic storage. In *Proc. ICAC*, 2010.

[31] P. Padala and et al. Adaptive control of virtualized resources in utility computing environments. In *Proc. Eurosys*, 2007.

[32] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proc. Eurosys*, 2009.

[33] J. Rolia, L. Cherkasova, M. Arlitt, and V. Machiraju. Supporting application QoS in shared resource pools. *Communications of the ACM*, 2006.

[34] P. Shivam, S. Babu, and J. Chase. Learning application models for utility resource planning. In *Proc. USITS*, 2003.

[35] P. Shivam, S. Babu, and J. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *Proc. VLDB*, 2006.

[36] S.S.Parekh, N.Gandhi, J.L.Hellerstein, D.M.Tilbury, T. Jayram, and J. P. Bigus. Using control theory to achieve service level objectives in performance management. In *Real Time Systems*, 2002.

[37] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A dollar from 15 cents: cross-platform management for internet services. In *Proc. USENIX Annual Technical Conference*, 2008.

[38] B. Urgaonkar, M. S. G. Pacifici, P. J. Shenoy, and A. N. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. SIGMETRICS*, 2005.

[39] B. Urgaonkar, P. Shenoy, and et al. Resource overbooking and application profiling in shared hosting platforms. In *Proc. OSDI*, 2002.

[40] T. Wood, L. Cherkasova, and et al. Profiling and modeling resource usage of virtualized applications. In *Proc. Middleware*, 2008.

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

[42] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proc. SOSP*, 2003.

[43] W. Zheng, R. Bianchini, and et al. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. USENIX Annual Technical Conference*, 2009.

[44] X. Zhu and et al. 1000 Islands: integrated capacity and workload management for the next generation data center. In *Proc. ICAC*, June 2008.

[45] X. Zhu, Z. Wang, and S. Singhal. Utility-driven workload management using nested control design. In *Proc. American Control Conference*, 2006.