Using CPU Gradients for Performance-aware Energy Conservation in Multitier Systems

Shuyi Chen[1], Kaustubh R. Joshi[2], Matti A. Hiltunen[2], Richard D. Schlichting[2], and William H. Sanders[1]

[1]Coordinated Science Laboratory
University of Illinois at
Urbana-Champaign
Urbana, IL, USA
{schen38,whs}@illinois.edu

[2]AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{kaustubh,hiltunen,rick}@research.att.com

Abstract

Dynamic voltage and frequency scaling (DVFS) and virtual machine (VM) based server consolidation are techniques that hold promise for energy conservation, but can also have adverse impacts on system performance. For the responsiveness-sensitive multitier applications running in today's data centers, queuing models should ideally be used to predict the impact of CPU scaling on response time, to allow appropriate runtime trade-offs between performance and energy use. In practice, however, such models are difficult to construct and thus are often abandoned for ad-hoc solutions. In this paper, an alternative measurement-based approach that predicts the impacts without requiring detailed application knowledge is presented. The approach uses a new set of metrics, the *CPU gradients*, that can be automatically measured on a running system using lightweight and nonintrusive CPU perturbations. The practical feasibility of the approach is demonstrated using extensive experiments on multiple multitier applications, and it is shown that simple energy controllers can use gradient predictions to derive as much as 57% energy savings while still meeting response time constraints.

1. Introduction

Dynamic scaling of resources and applications to match workload demands offers a promising solution to help tackle data center power consumption, which represented 1.5% of all U.S. electrical power consumed in 2006 and is growing rapidly at an annual rate of 12% [1]. Techniques such as dynamic voltage and frequency scaling (DVFS) can scale the CPU speeds to better fit application workload and reduce energy use, while others, such as virtual-machine-based server consolidation and migration, can scale down and pack application components on fewer physical machines to allow powering down of unneeded servers.

While they enable substantial energy savings, such methods can significantly impact application performance if not applied judiciously. This is especially critical for multitier applications that power services such as e-commerce portals, search engines, social-networking sites, collaborative services, email, and enterprise management systems. Studies such as [2, 3, 4] and the experiences of large service providers [5] have repeatedly shown the importance of performance, measured using metrics such as end-to-end response time, to user satisfaction, traffic growth, and, consequently, business viability. To ensure adoption by such applications, scaling techniques will need to ensure that end-to-end responsiveness is maintained. However, current scaling techniques (e.g., DVFS) employed in popular operating systems rely only on OS-level, single-machine measurements such as CPU utilization when making decisions, and ignore the impact of scaling on end-to-end application metrics.

In this paper, we propose a new application-aware approach to resource scaling that seeks to minimize energy usage while *preserving end-to-end application performance* of multitier systems running in either non-virtualized or virtualized environments. To do so, we introduce a new set of metrics, the CPU gradient, which predicts the impacts of changes in processor frequency or VM capacity (i.e., the percentage of the CPU allocated to a given VM) at individual machines on the end-to-end response time of a multitier application. These impacts may not be obvious nor static as workloads and bottlenecks

change. However, knowledge of them enables smart scaling by making it possible to provide an application with "just enough" resources to ensure that the responsiveness constraints are met. Hosting and cloud infrastructure providers can also use such predictions to optimize allocation across multiple applications significantly reduce resource consumption and energy usage.

CPU gradients provide a graybox alternative to traditional queuing models, which face significant practical hurdles due to the design and deployment lifecycle of typical online applications. Often, there is simply not enough knowledge about the complex application or its deployment infrastructure to construct detailed models, especially when off-the-shelf components are used. Additionally, as multitier systems are increasingly deployed in outsourced data centers, in shared infrastructure clouds, and over complex networks, the application owners and the infrastructure providers are often separate entities. The former do not have enough knowledge of the infrastructure to construct good models, while the latter do not have enough knowledge of the application to do so. Finally, even when detailed information is available, manual model construction is often laborious and error-prone, and requires significant expertise, all of which discourages adoption.

CPU gradients are simple models constructed via automatic on-line measurements. They leverage general system knowledge and capture the impact of application-specific parameters and infrastructure configurations, yet require very little information, just the location of application transaction logs, templates for extracting timestamps and transaction types from the logs, and the response time requirements. For a multitier application, CPU gradients represent local point derivatives of the end-to-end response time with respect to the resource parameters, i.e., CPU frequency and VM capacity. This is done by injecting minute CPU frequency or VM capacity perturbations into the nodes of a running application and using response time measurements to estimate the derivatives. By injecting the perturbations in a pre-defined square wave pattern and using Fourier transforms to spectrally analyze the response times, we can ensure high accuracy even in noisy production environments while keeping

the injected perturbations small. Finally, using the gradient measurements and the application's response time in a baseline configuration, its response time in configurations with different CPU speeds, VM capacities, and workloads can be predicted.

The overall contribution of this paper is a runtime framework to minimize the power consumption of a system by continuously and automatically scaling CPU frequencies or VM capacities while ensuring that per-transaction application response time requirements are met, and yet requires very little application specific knowledge. We build on our prior work on Link gradients [6] in which we developed the basic idea of a gradient and the Fourier-analysis-based perturbation machinery to measure gradients. In that work, we used gradients to predict the impact of changes in network link latency on response time, a relationship that we could demonstrate was linear for many common network scenarios. However, the impacts of CPU frequency and VM capacity on response time are significantly non-linear and highly dependent on application workload. In this paper, we develop new ways to use the basic gradient measurement technique to capture the non-linearity. To capture the impact of workload, we develop novel workload shaping mechanisms and techniques to combine multiple basic gradient measurements into a composite predictor. Ultimately, we are able to predict the impacts of changes in CPU frequency/VM capacity on per-transaction response time despite operating in a shared production environment with constantly changing workload that make isolated measurement of each transaction impossible.

Two sets of experiments are used to substantiate our results. The first experiments demonstrate that the gradient models provide accurate predictions irrespective of application configuration and settings, using two multitier applications with different architectures and technologies, Java Servlet-based RUBiS and PHP-based RUBBoS. The second set of experiments demonstrate that the approach can be used to provide significant real-world energy savings while maintaining user-perceived responsiveness using traces from a production online application. The application control for these experiments is realized in

gradient-based runtime controllers for both frequency and VM capacity that balance performance against energy usage and achieve up to 50% energy savings compared to a baseline.

2. Gradient Models Overview

We begin by defining gradient-based models and review a Fourier transform-based measurement technique that we developed in the context of link gradients to accurately measure gradients in a running system.

2.1. Gradient Definition

Consider a multitier application consisting of a set of nodes represented by vector $N = (n_1, n_2, \ldots, n_m)$ and connected by a set of logical communication links represented by vector $L = (l_1, l_2, \ldots, l_r)$. Each node represents a single software component of a specific type, e.g., a web server or application server. Nodes execute using resources (e.g., physical hosts) that may be dedicated or shared. For example, several nodes may execute within separate virtual machines on the same physical host. Logical links exist between two nodes if the nodes exchange messages during system operation. Each logical link may consist of many physical network links. Together, nodes and links are called the *elements* of the application.

Each element e is associated with a vector of attributes $A^e = (a_1^e, a_2^e, \ldots, a_k^e)$ that quantify the properties of the element or the resources that may impact its performance. For example, attributes may include the fraction of the host's CPU or I/O bandwidth available to a node, or the bandwidth available for a logical link. They may also include properties of the resources themselves, e.g., the CPU speed of the host a node runs on, the link latency of a logical link, or the disk spinning speed. Each application is also associated with one or more end-to-end *progress metrics* whose values are to be predicted. Although metrics can include various properties such as throughput, rate of processing, or even power consumption, in this paper we focus on end-to-end response time

of application transactions. In doing so, we assume that the application is a transactional system whose users interact with it through a set of transactions, such as "login," "buy," and "browse," each of which utilizes a set of elements according to a transaction-specific call graph.

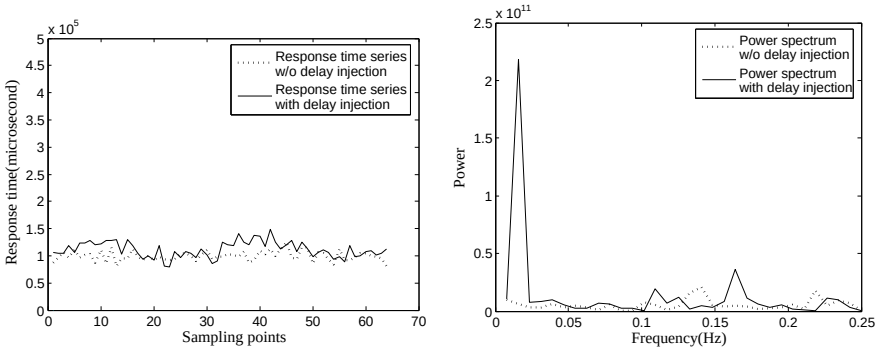
The goal of a gradient model is to quantify the relationship between element attributes and end-to-end progress metrics. Specifically, consider the values of a single type of attribute A_k, e.g., latency, for all p elements of the system in the current operating configuration c_0, i.e., $A_k(c_0) = (a_k^{e_1}(c_0), a_k^{e_2}(c_0), \ldots, a_k^{e_p}(c_0))$. We represent the relationship of the attribute to the metric M as an unknown function at the current operating point, or, $M = F\left(A_k(c_0)\right)$, while keeping other attributes unchanged in the system. Then, the question we wish to answer is, "Given the value of $M(A_k(c_0))$ at the system's current operating configuration c_0, what is its value at a different operating configuration c_1, i.e., $M(A_k(c_1))$?"

To answer this question, we take the following approach. Let the vector $\Delta A_k = A_k(c_1) - A_k(c_0) = (\Delta a_k^{e_1}, \ldots, \Delta a_k^{e_p})$ be the differential change in the attribute values between the current and the new operating configurations for attribute A_k. Assuming that the function F is differentiable and other attributes are unchanged between the old and new operating configurations, we can then use the Taylor expansion to represent the desired $M(A_k(c_1))$ as:

$$M(A_k(c_1)) = M(A_k(c_0)) + \sum_{e \in N \cup L} \left.\frac{\partial F}{\partial a_k^e}\right|_{c_0} \Delta a_k^e + O(\Delta A_k^2) \tag{1}$$

where the $O(\ldots)$ term represents the higher order derivatives and powers of the attribute values, and the derivatives $\frac{\partial F}{\partial A_k}\big|_{c_0} = (\frac{\partial F}{\partial a_k^{e_1}}\big|_{c_0}, \frac{\partial F}{\partial a_k^{e_2}}\big|_{c_0}, \ldots)$ are the gradients at the current operating configuration c_0.

If the gradients are known, this equation can be used to predict the performance of the system in the new configuration by ignoring the higher-order derivatives and powers in $O(\Delta A_k^2)$. However, doing so is justifiable only if ΔA_k is small enough to cause the higher powers to vanish, or if F is linear and thus the higher-order derivatives are zero. In practice, changes in the operating conditions could be large, making the first condition impractical. The second condition can hold true depending on the type of metric and attribute



(a) Response Time Series (b) Frequency Spectrum

Figure 1: Gradient Measurement

being considered. In [6], we discuss one type of gradient—the link gradient—for which the condition does hold. The link gradient is defined as the rate of change of the average end-to-end response time with respect to the network link latency between two servers. And we have demonstrated in [6] that the linearity assumption holds in a large enough range for this relationship. However, in general, such linearity assumptions may not hold, and non-linearity can impact the accuracy of the gradient. Although nonlinearity can always be overcome by recalculating the gradients whenever they change, it is important to minimize the need for such recalculations, not only to reduce runtime measurement overhead, but also to prove a meaningful operating range over which gradient models can make accurate predictions. We address this issue in Section 3.

The formulation of gradients also assumes that F is differentiable, which is reasonable for many multitier systems, e.g., if the response time relationships can be modeled by a Jackson queuing network. However, the assumption may be violated because of timeouts that cause discontinuous jumps in response time curves, or because of communication patterns such as parallel processing that give rise to non-differentiable functions such as min and max. In such cases, there is no recourse but to frequently recalculate the gradients dynamically. Due to these and linearity considerations, it is beneficial to have a lightweight gradient measurement technique that can be deployed at runtime in a production system and operates without much perturbation to the target system.

2.2. Spectral Measurement

The technique we use to estimate the gradient for each system element e with respect to attribute a_k^e at runtime is conceptually very simple. We inject small perturbations in the value of a_k^e, and then measure the corresponding change in the end-to-end progress metric. The ratio of the change in the metric to the change in the attribute value provides the gradient. However, the problem with such an approach is that measurements made on running systems are often very noisy, especially when resources are shared. Therefore, to get accurate estimates with tight confidence intervals, either measurements must be accumulated over long periods of time, or the perturbation must be high enough to overcome the effects of noise. Both approaches are less than ideal. Conducting measurements over long time intervals carries the risk that the system behavior might change during measurement as a result of workload changes, or other runtime adaptations. On the other hand, injecting large perturbations can be intrusive and detrimental to user experience.

To address this problem, we have developed a technique that uses the observation that while noise may be present in a system's runtime measurements, it is rarely periodic. While some sources of noise, such as garbage collection, may indeed be periodic, such sources are few, and operate at a small set of frequencies that one can easily identify by examining the metric in the frequency domain. The remainder of the noise is often spread uniformly across many different frequencies, with very little contribution at any particular frequency. Figure 2 shows a three hour trace of average response time from 2007 NLANR proxy access log [7] and its corresponding frequency spectrum. As we can see in the figure, the noise is spread across different frequencies in the spectrum. Therefore, if the perturbation is crafted by injecting a periodic signal at a frequency at which the ambient noise is low, a large portion of its energy will be concentrated on the designated frequency, one can then get significantly superior signal-to-noise ratios and more accurate measurements as we have shown in [6].

For simplicity, we use a square wave pattern for injecting perturbations.

Within a short time frame (usually several minutes), we perturb the attribute at a single element by repeatedly switching its value between its normal value and a high/low value at a frequency chosen to minimize ambient noise. This also causes a square-wave perturbation in the metric of interest. (In the extreme case, when the change in the metric has no impact on the response time, the square-wave response degenerates to a straight line.) Subsequently, we use standard Fourier transforms to compute the frequency spectrum of the time series of metric measurements made during the period of signal injection, and use that frequency spectrum to estimate the gradient using the following equation, as derived in [6].

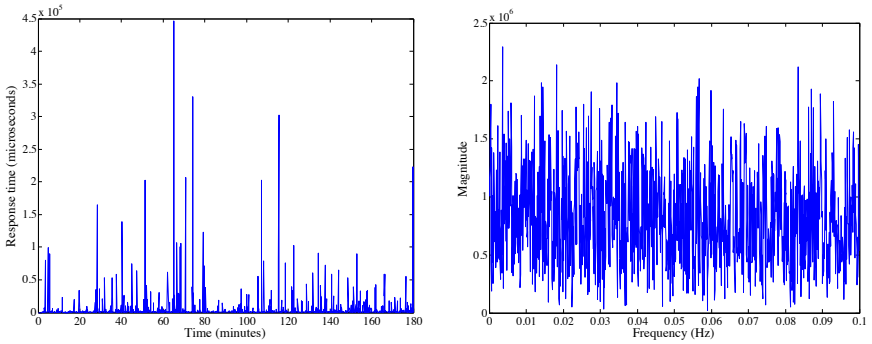
$$\frac{\partial M}{\partial A_k^e} = \frac{|\mathrm{FFT}^d(M) - \mathrm{FFT}^0(M)| \cdot sin(\frac{\pi}{2n})}{\Delta A_k^e \cdot f_d} \tag{2}$$

In this equation, n is the number of sample measurements in the metric time series, f_d represents the frequency at which the perturbation was injected, and $\mathrm{FFT}^d(M)$ and $\mathrm{FFT}^0(M)$ represent the Fast Fourier Transform (FFT) of the metric time series with and without the perturbation, respectively. Sample end-to-end response time series with and without perturbation, along with their frequency domain counterparts, are shown in Figures 1 (a) and (b). As can be seen, even a square wave that is visually difficult to discern in the time domain results in large spikes in the frequency domain. Use of this technique allows a reduction of noise and perturbation by an order of magnitude (see [6]) compared to a time-domain approach, and thus makes it possible to recalculate the gradients dynamically and cheaply while the system is running.

3. CPU Gradients

In this section, we develop techniques for allowing gradients to accurately predict the impact of CPU frequency changes and VM capacity changes on the end-to-end response time of each application transaction. In doing so, we define two new gradient metrics, which we collectively call the CPU gradients.

The first metric is the *frequency gradient*, which is defined as the rate of change of a system's mean end-to-end transactional response time with respect



(a) Time domain (b) Frequency Spectrum

Figure 2: Time and frequency domain of a real trace

to changes in the CPU frequencies f_n of individual computers. In this case, perturbations are injected by using dynamic voltage and frequency scaling (DVFS) features found in most modern microprocessors. Such gradients can be used to perform dynamic energy saving by slowing down computers to the limits allowed by their response time SLAs. Since power consumed by a CPU per unit of work increases approximately as a quadratic function of frequency (see Section 4), such a slowdown can conserve a significant amount of energy both by direct savings and because of reduced cooling requirements.

The second metric is called the *VM capacity gradient*, and assumes that the target application executes in a virtual machine environment (our implementation currently supports Xen [8]) in which some or all of the application nodes operate in separate virtual machines. Multiple VMs can be executed on a single physical host, and the virtual machine hypervisor has the ability to cap a VM's CPU to an administrator-definable fraction. Then, the VM capacity gradient is defined as the rate of change of the application's mean end-to-end transactional response time with respect to the fraction of CPU capacity allocated to each individual VM. In that case, the perturbations are injected by instructing the hypervisor to change the VM's CPU capacity in a periodic manner. The VM capacity gradient can be used to drive performance-aware server consolidation and energy conservation. Specifically, it can help determine how much each virtual machine's CPU allocation can be reduced without violating response

time SLAs, so that VMs can be packed into the fewest number of physical hosts possible.

Both the frequency gradient and VM capacity gradient present challenges for linear gradient models, because the underlying relationships between the end-to-end response time and the attributes—CPU frequency and VM capacity—are non-linear as a result of queuing effects. Furthermore, multiple factors impact the exact relationship including per-node parameters such as per-transaction service times; inter-node dependencies between nodes due to different message flows created by different transactions; and whole application effects due to dynamically changing workload and transaction type mixes, each of which stresses different parts of the system. We develop the solutions for these challenges for the frequency gradient first, and then describe how they are adapted for the VM capacity gradient.

3.1. Gradient Decomposition

To isolate the local impacts of CPU frequency from impacts due to global parameters such as inter-node message routing, we decompose the gradients into two sets of partial derivatives (or subgradients): the "system subgradient" captures the rate at which the system's end-to-end response time changes with each per-component response time, while the "machine subgradient" captures the rate at which each component's response time changes with its host's CPU frequency (or VM capacity) and the workload.

For each transaction type t, the *system subgradient* is a vector

$$\left(\partial rt_t/\partial rt_t^1, \ldots, \partial rt_t/\partial rt_t^n\right)$$

whose elements are the partial derivatives of the mean end-to-end response time of the system for transaction t with respect to the mean response time of each component, i.e., rt_t^i for that transaction. Intuitively, this subgradient is dependent on the call-graphs associated with user transactions. For example, a transaction that consists of a series of single nested calls to a web server, an application server, and a database would be expected to have a system gradient

with all values equal to one. Communication patterns such as load balancing, caching, and state replication among servers impact the system subgradient. However, the subgradient is expected to remain constant under a range of operating configurations for a given application, thus indicating a linear relationship. We have experimentally demonstrated the linearity of a similar metric: link gradients that capture the derivative of end-to-end response time with respect to link latencies between the nodes.

On the other hand, machine subgradients capture the relationship between individual component response times and host CPU frequencies/VM capacities. This relationship encapsulates most of the nonlinearity due to queuing effects, and thus we do not define the gradients directly with respect to frequency/capacity, but with respect to non-linear basis functions as described next.

3.2. Modeling Nonlinearity

To tackle the problem of nonlinearity of an unknown function F (here, the per-component response time function) with respect to the attribute A_k (here, the CPU frequency f), we recast F in terms of "basis functions" $B_k = (b_k^{e_1}, \ldots, b_k^{e_p})$ with respect to which it is approximately linear. For each element e, the basis function $b_k^e(A_k(c_1))$ is a function whose values for a configuration c_1 can be computed based solely on the values of attributes in that configuration, i.e., $A_k(c_1)$ and any constant parameters. As we show below, these basis functions can be derived using high-level knowledge of the causes of nonlinearity without the need for detailed application knowledge, and still offer good approximation of the nonlinearity. Since the value of basis functions can be computed for a new configuration, the change in basis functions between the old and new configurations, i.e., $\Delta B_k = |B_k(A_k(c_1)) - B_k(A_k(c_0))| = (\Delta b_k^{e_1}, \ldots, \Delta b_k^{e_p})$ can be used along with a gradient with respect to the basis functions, i.e., $\frac{\partial F}{\partial B_k}$, to predict the value of the progress metric in a new configuration, or $M(A_k(c_1)) \approx M(A_k(c_0)) + \frac{\partial F}{\partial B_k} \cdot \Delta B_k$.

Recognizing queuing as the primary source of nonlinearity of per-component

response time under CPU frequency changes, we use a basis function based on the mean response time relation for a single M/G/1/PS queue, i.e., $rt = \frac{s}{1-U}$, where s is the per-transaction service time and U is the utilization. Then, we utilize the fact that service time and utilization are inversely proportional to frequency f^n of node n, while utilization is also proportional to the application workload, i.e., $s \propto 1/f^n$, and

$$U^n = \langle \alpha^n \cdot \vec{w} \rangle / f^n \tag{3}$$

where U^n is the utilization at node n, $\vec{w} = (w_{t_1}, \ldots, w_{t_p})$ is the workload vector with request rates for each of the application's transactions t_1 to t_p, and α^n is a vector of node specific constants, one per each application transaction.

Therefore, we set the basis function for a node n to

$$b_f^n(f^n, \vec{w}) = \frac{1}{f^n - \langle \alpha^n \cdot \vec{w} \rangle} \tag{4}$$

and the machine subgradient for transaction t is represented as $\frac{\partial rt_t^n}{\partial b_f^n}$. Although the basis function was chosen based on knowledge that nonlinearity is caused by queuing effects and that the OS uses a time-slice-based CPU scheduler, no application-specific information, such as service times, routing matrices or communication modes (e.g., synchronous vs. asynchronous), is required. Therefore, our technique can be used with different multitier applications automatically, without any extra effort. Furthermore, even though it is based on the M/G/1/PS equation, the gradient approach's philosophy of simple, locally accurate models, that can be cheaply re-calibrated at runtime, allows the basis function to be an approximation only. Its main use is to extend the useful range of the local models and reduce the need for frequent recalibrations.

3.3. Workload Subgradient

Following the above formulation, the only remaining unknowns are the per-node, per-transaction constants α^n that relate the application workload to utilization at each host. When the system's configuration changes from c_0 to c_1, we

can use Equation 3 to write $U^n(c_1) - U^n(c_0) = \alpha^n \cdot (w(\vec{c_1}) - w(\vec{c_0}))/f^n$. Using that formulation, we see that the vector α^n is also a "workload subgradient," which represents the rate of change of node utilization at node n with respect to changes in system workload $\vec{w}$. Therefore, we use the gradient measurement machinery to measure this gradient by injecting periodic perturbations into each transaction's request rate one at a time, and observing the resulting changes in the CPU utilization at each node. Section 4 describes the techniques used for injecting workload perturbations in such a way that no additional workload beyond the application's normal workload is required, thus ensuring minimal interference in a running system.

3.4. Frequency Gradient

In practice, measuring the system and machine subgradients directly is difficult, because we cannot measure or exert direct control over the per-component response times rt_t^i. Therefore, after computing the workload subgradient and plugging it into the basis functions, we combine the system and machine subgradients using the chain rule of derivatives to form a composite *frequency gradient* that is not only easy to measure, but also capable of providing the complete relationship between end-to-end response time and per-host CPU frequencies. Based on the preceding discussion and basis functions, we define the composite predictor for the response time of transaction t as:

$$\begin{aligned} rt_t(\vec{f}(c_1), \vec{w}(c_1)) \quad &\approx \bar{rt}_t(\vec{f}(c_0), \vec{w}(c_0)) + \textstyle\sum_{i=1}^{n} \frac{\partial rt_t}{\partial rt_t^i} \frac{\partial rt_t^i}{\partial b_f^i} \\ &\quad \cdot \left[b_f^i(f^i(c_1), \vec{w}) - b_f^i(f^i(c_0), \vec{w}) \right] \end{aligned} \tag{5}$$

The vector $\left(\frac{\partial rt_t}{\partial rt_t^1} \frac{\partial rt_t^1}{\partial b_f^i}, \ldots, \frac{\partial rt_t}{\partial rt_t^n} \frac{\partial rt_t^n}{\partial b_f^n} \right)$ is the frequency gradient for transaction t, and can be measured by changing the frequency of each host machine, in a square-wave pattern, observing the changes in end-to-end response time, and using a modified version of Equation 2 with ΔA_k^e replaced by $|b_f^i(f^i(c_1), \vec{w}) - b_f^i(f^i(c_0), \vec{w})|$.

3.5. VM Capacity Gradient

In principle, VM capacity gradients are similar to frequency gradients because for a loaded system, reducing the CPU capacity allocated to a VM by fraction c is equivalent to making jobs on the virtual CPU run slower by a factor of c. We can thus use a basis function similar to the one used for the frequency gradient, but in which nodes are virtual machines rather than physical hosts, and the "frequency" $f^{(*)n}$ associated with a VM n is equal to the host frequency $f^{\text{host}(n)}$ scaled by the fractional CPU capacity c^n allocated to the VM, i.e., $f^{(*)n} = f^{\text{host}(n)} \cdot c^n$. In Xen-based environments, the hypervisor acts as a conduit for all I/O requests from each VM to the physical hardware. To deal with the additional complication due to the hypervisor, we view the hypervisor as a separate "VM" and we only compute the gradients for the VMs other than the hypervisor for the following reason.

Although the CPU gradient captures the impact of the changes in the CPU frequency/VM capacity and the workload, other factors such as I/O and memory influence an application's end-to-end response time as well. While the ultimate goal of our work is to compute gradients for all these factors and combine them into a single predictor, that is beyond the scope of this paper. Because we do not consider I/O gradients, we must also ignore the time spent by the Xen hypervisor in processing guest VM I/O requests. To compensate for this incomplete model, we first assume any errors in the response time prediction of a new configuration are due to unaccounted factors, i.e., I/O or memory operations, and re-measure the response time in this configuration to take the changes due to the unaccounted factors into account and serve as a new baseline (i.e., $\bar{rt}_t(\vec{f}(c_0), \vec{w}(c_0))$ from Equation 5). If the next prediction is still inaccurate, only then do we assume that the gradient has changed and recompute it.

3.6. Summary

The gradient model technique combines simple, locally accurate models and lightweight online measurements. As we can see in the derivation of CPU gradient, by requiring only local model to be accurate, simple models constructed

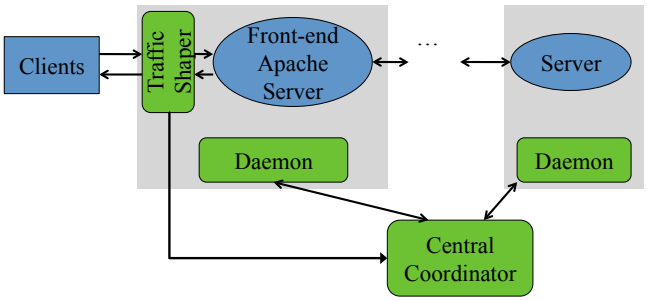


Figure 3: Gradient Computation Architecture

using high-level system knowledge, e.g., M/G/1 PS queuing model, can be used with the help of the gradient decomposition and basis function techniques. Compared to traditional modeling techniques, e.g., queuing network theory, the gradient model significantly reduces the complexity of modeling. This simple nature of the gradient technique makes it useful in practical settings in which detailed information of a system is usually unavailable. Also, the gradient measurements require only the access of the end-to-end performance metrics of the system and the ability to inject a perturbation to the element attribute in question, which makes it a gray-box technique, and it can be applied to a new system with little extra effort.

4. Architecture

In this section, we describe the gradient measurement framework, algorithm, and controllers we have developed for gradient-based energy conservation. The framework currently supports any multitier application with a web interface that can be accessed via an Apache server.

4.1. Measurement Algorithm

Computation of the frequency and VM capacity gradients first requires the computation of a workload subgradient, followed by computation of the CPU gradients. This is achieved through a distributed active monitoring framework whose architecture is shown in Figure 3. The framework consists of a central

coordinator and a set of local daemons on each machine. Each daemon is responsible for reporting the current values of the CPU frequency, VM capacity, and node utilization to the central coordinator, and changing the values of the CPU frequency and VM capacity on command from the coordinator. Local daemons are needed only on those nodes whose frequency/CPU capacity is to be managed, other nodes do not require any additional instrumentation. The central coordinator orchestrates the daemons and executes the gradient measurement algorithm. It requires the location of the web server log.

The process of measuring the workload subgradient involves injecting a square wave perturbation into the request rate for each of the application's transaction types one at a time, and collecting from all the local daemons the per-node CPU utilization time series during perturbation. Equation 2 is used to compute the workload subgradient for each transaction type and node pair. On the other hand, measuring the frequency and VM capacity gradients involves injecting square wave perturbations to the CPU frequency/VM capacity of each node one at a time, and collecting the end-to-end response time series for all the application's transactions from the web server log. Using the formulae from Section 3, gradients are computed for each transaction type and node pair after construction of the values of the basis functions.

In order to minimize intrusion, the framework conducts its measurements without injecting any additional traffic into the system, i.e., using the system's normal workload. It also measures the amount of noise present in the system's environment, and determines the smallest magnitude of perturbations that are needed to achieve a target measurement accuracy given the level of noise. To do so, the framework conducts a training phase before injecting any perturbations for gradient estimation. In this phase, it passively records the unperturbed values of the end-to-end response time for each transaction, the transaction request rates, and the node utilizations. It then analyzes these values in the frequency spectrum to automatically compute the amplitude, frequency, and length of the square wave perturbation.

The frequency with the smallest standard deviation for the target metric

(i.e., the per-node utilization in the case of the workload subgradient and end-to-end response time in the case of the CPU gradients) is chosen as the perturbation frequency in order to minimize the impact due to noise. To limit gradient estimation error to $e\%$, the perturbation square wave amplitude is chosen such that its magnitude at the perturbation frequency is greater than the $1 - e^{th}$ percentile of the value of the target metric at that frequency. Finally, the length of the perturbation is chosen by measuring the transaction request rate for the target transaction, and ensuring that the experiment is long enough to collect a fixed number of sample point bins with high probability. A detailed description of the formulae used can be found in [6].

To inject frequency perturbations, the local daemon uses the *CPUfreq* system interface enabled by the *userspace* CPU frequency scaling governor to change the frequency of the server CPU at runtime. To monitor the CPU utilization, the *sar* utility is used to collect the CPU utilization periodically. To inject VM capacity perturbations, the local daemon uses the *xm* control interface provided by Xen to set the Xen credit scheduler VM capacity, and uses the *xentop* tool to determine the domU CPU utilizations for all the virtual machines. Injecting perturbations into the transaction request rate, however, presents some unique challenges.

4.2. Traffic Shaping

Perturbing the request rate of transactions in a controllable manner and on a transaction-type-by-transaction-type basis is challenging, because we prohibit additional test traffic in a bid to keep the workloads representative of actual user requests. Instead, the measurement framework utilizes traffic shaping to convert an incoming traffic stream into one whose average rate is the same, but whose instantaneous rate varies in a square wave pattern. It does so by selectively delaying incoming requests when the square wave is "low" to create a lower-rate stream, and by releasing previously buffered packets when the square wave is "high" in order to create a higher-rate stream. This is done via a custom "request shaper" Apache module, which is deployed on the user-facing web

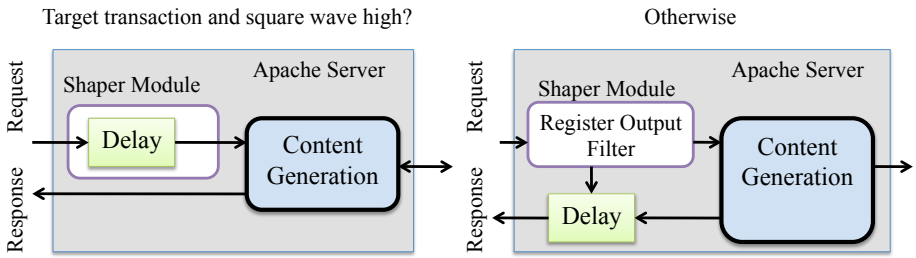


Figure 4: Custom apache module

server of the target application, as shown in Figure 3.

Using that approach, the challenge is to shape the request stream of a single transaction type without impacting transactions of other types. This is difficult because in practice, incoming transaction types are usually not uncorrelated with each other, but are produced by a sequence of interlinked user actions. Thus, delaying requests of a single transaction type delays the issue of other types of transactions as well, leads to a "bleed-over" of the square wave pattern into the request rates of other transaction types, and contaminates the impacts on the metric that we wish to measure. To solve this problem, the request shaper uniformly delays *all* user requests in the perturbation period by a uniform value equal to half the square wave period. The only difference between the target and other transactions is that when the square wave is high, the delay for the target transaction is introduced *before* the request is forwarded to the web server, while it is introduced for all other transactions *after* the request finishes processing and just before a response is sent. Thus, the square wave pattern is visible to the Apache server that processes incoming requests, but not to the end users, who experience a uniform delay across all transactions. To implement the delays, the module registers a custom output filter with the Apache interface. The algorithm for the traffic shaper is illustrated in Figure 4.

Although this technique temporarily increases the perceived response time of the system, our Fourier domain technique ensures that the delays can be kept small (usually less than the standard deviation of the response time), and all the gradients can be quickly computed within a matter of a few minutes.

4.3. Energy Controllers

Finally, we have implemented two performance-aware feedback controllers that utilize gradients to manage CPU frequencies in non-virtualized environments and VM capacity assignment and placement in virtualized environments. The controllers are instantiated with a precomputed set of gradient measurements. Their objective is to minimize the system energy consumption while ensuring that the mean end-to-end response time for each transaction type, as measured over a 5-minute rolling window, remains below a user-defined threshold T. The controllers are invoked periodically and use measured values of the mean response time since the last invocation, current CPU frequencies, VM capacities, and VM placement as feedback to produce new recommendations for CPU frequencies or VM placement that will minimize energy use in the next controller period. Next, we describe the optimization algorithms they use. Because control is orthogonal to the contributions of this paper, we have intentionally kept the algorithms simple. We are confident that better optimizers can yield even greater energy savings.

The Frequency Controller. The objective function for the frequency controller is to minimize the total average energy consumption of all the machines, which, expressed in joules/unit of work, is approximately a quadratic function of the CPU frequency, i.e., $F \propto \sum_{i=1}^{n} f_i^2$ [9]. The solution is also constrained to keep the response time of all transactions below the threshold T. Starting from a set of n highest CPU frequencies, the energy controller performs an n-dimensional gradient-descent in the direction of the steepest gradient based on the objective function. After every descent, it uses the frequency gradient model, along with the measured response times and frequencies at the time of controller invocation, to compute the response times for all transactions in the new configuration. If the response time constraints are satisfied in the new configuration, the controller keeps searching until it cannot reduce frequencies any further without a constraint violation. If the CPU frequencies are changed, the controller skips one control interval to avoid oscillations.

The VM Capacity Controller. To make an optimal decision on which machines to turn on and off, both the dynamic and idle power consumption (the latter of which often represents a significant fraction of total power) of physical hosts must be taken into account. However, machine on-off decisions have discrete power consequences while dynamic power depends on the fraction of the CPU being utilized, thus leading to a mixed discrete and continuous objective function. Instead of solving the resulting combinatoric optimization problem, the VM capacity controller ignores the idle power, and resorts to the simpler but non-optimal approach of adjusting VM capacities to minimize the total CPU capacity used by all application VMs without violating performance constraints. It then uses the first fit decreasing (FFD) algorithm [10] to bin-pack the virtual machines into the smallest number of physical machines possible. In machines with a single CPU core, we simply view each machine as a bin and do the bin-packing. In machines with multiple CPU cores, we view each CPU core as a bin and bin-pack the guest VMs into the smallest number of CPU cores.

To determine VM capacities, the controller uses the following heuristic: if the measured response time of some transaction at the time of the controller invocation is higher than the threshold T, we increase the capacity of the VM that the capacity gradients predict will result in the highest decrease in the offending transaction's response time for a unit increase in VM capacity, and repeat the process until the response time for all transactions as predicted by the capacity gradients falls below the threshold T. Conversely, when the measured response time of all transactions is smaller than the threshold, the controller uses the same heuristic to decrease the capacity of the VM which would yield the lowest increase in response time for a unit reduction in VM capacity. This process is repeated until a further reduction in VM capacity would cause a violation of the response time constraint.

To prevent oscillations in which machines repeatedly turn on and off in subsequent invocations, the controller uses hysteresis by setting a different threshold capacity for turning machines on vs. turning them off. The controller adds a new machine when the VMs cannot be bin-packed without keeping utilization

lower than 85% on the machines in use, but will turn a machine off only if the VMs can be bin-packed so that they consume less than 80% utilization on the remaining machines.

5. Evaluation

5.1. Experiment Setup

We use RUBiS, a widely used online auction site benchmark [11], and RUBBoS [12], a bulletin board benchmark modeled after online news forums such as Slashdot, for our evaluation. Specifically, we use the 3-tier servlet version of RUBiS consisting of an Apache web server, a Tomcat application server, and a MySQL database server, and the PHP version of RUBBoS, which consists of two tiers, the Apache web server hosting PHP scripts and the MySQL database server.

In the first set of experiments, we deployed an instance of RUBiS on three of our testbeds. The PM (physical machine) testbed consists of 4 machines running Ubuntu 8.04: an Intel E8400 Core 2 Duo machine and three AMD Athlon 64 3800+ machines. All of these machines are equipped with 2 GB of RAM and are connected using a 100 Mbps Ethernet switch. The VM (virtual machine) testbed consists of 4 identical machines running Xen 3.2, equipped with an Intel Pentium 1.8 GHz CPU and 2 GB of RAM, and connected using a 1Gbps Ethernet switch. The MVM (Multicore Virtual Machine) testbed consists of 3 identical machines running Xen 3.0, each of which has 16 cores and 24 GB of RAM, and the machines are connected using a 1Gbps Ethernet switch. In the PM testbed, each server of the RUBiS application runs separately on a standalone physical machine, while in the VM and MVM testbed, each server instance is configured to run in a single virtual machine with one virtual CPU on top of the Xen hypervisor. In the MVM testbed, each VM is configured to use a single core of the machine[1]. In these testbeds, the VM testbed represents an old

[1]The core can be shared by multiple VMs.

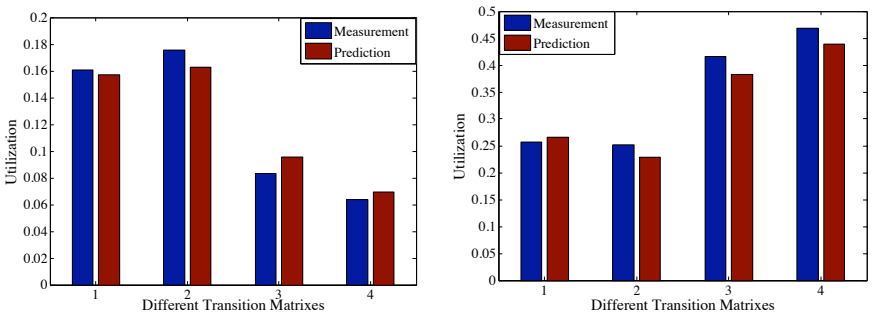
commercial server system that still exists, while the MVM testbed represents a more powerful modern multicore server system. In the following, we at times use the term "machine" to refer to either a physical machine or a virtual machine, and the meaning should be clear based on the context.

We used the standard RUBiS client emulator to generate the system workload. The browsing workload mix we used consists of 8 different transactions. In this paper, we present the results for the three most frequent transaction types: *ViewItem* (VI), *ViewUserInfo* (VU), and *SearchItemsByCategory* (SI). The *ViewItem* transaction returns information about an item, the *ViewUserInfo* transaction returns information about a user, and the *SearchItemsByCategory* transaction returns a set of items in a specific category. We treat the other 5 transaction types as a single, "synthesized" transaction type and keep its relative rate unchanged during the experiments.

5.2. Workload Gradient

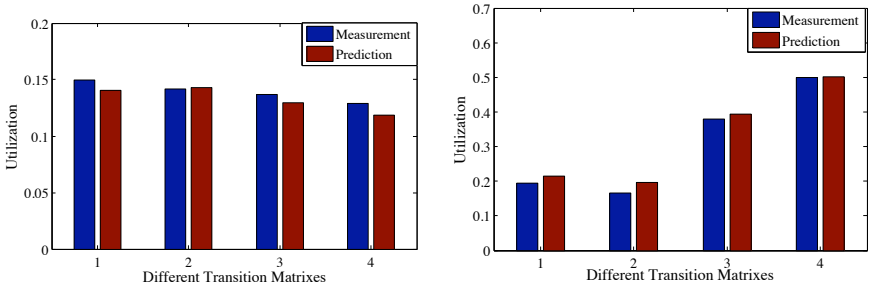
First, we present the results for the workload gradients. We used a simple RUBiS configuration, in which each tier consists of a single server, and deployed it on both testbeds. Then, we measured the workload gradients of the three transaction types for both the Tomcat and MySQL tier machines by perturbing the arrival rate of each transaction type, one at a time, using a square wave with a 4-second period. Each experiment lasted for 4 minutes. We used a custom script to measure the CPU utilization of each physical machine by reading statistics from */proc/stat* in the PM testbed and *xenmon.py* to measure the CPU utilization of each virtual machine in the VM testbed. To increase the measurement accuracy, the logging interval of the CPU utilization is set to 100 milliseconds.

To evaluate the accuracy of our workload gradient measurements, we created 4 different workload scenarios by modifying the RUBiS client transition matrix to change the relative arrival rates of the three transaction types and kept the other transaction types unchanged. Then, we used the workload gradients to predict the total CPU utilization under these scenarios. Figure 5 and 6 show



(a) Tomcat (b) MySQL

Figure 5: Predicted CPU utilization in PM testbed



(a) Tomcat (b) MySQL

Figure 6: Predicted CPU utilization in VM testbed

the predicted CPU utilization vs. measurement in the PM testbed and the VM testbed. Columns 1-4 correspond to the different workload scenarios. The results show that the workload gradient can accurately predict the total physical CPU utilization of the application in each machine.

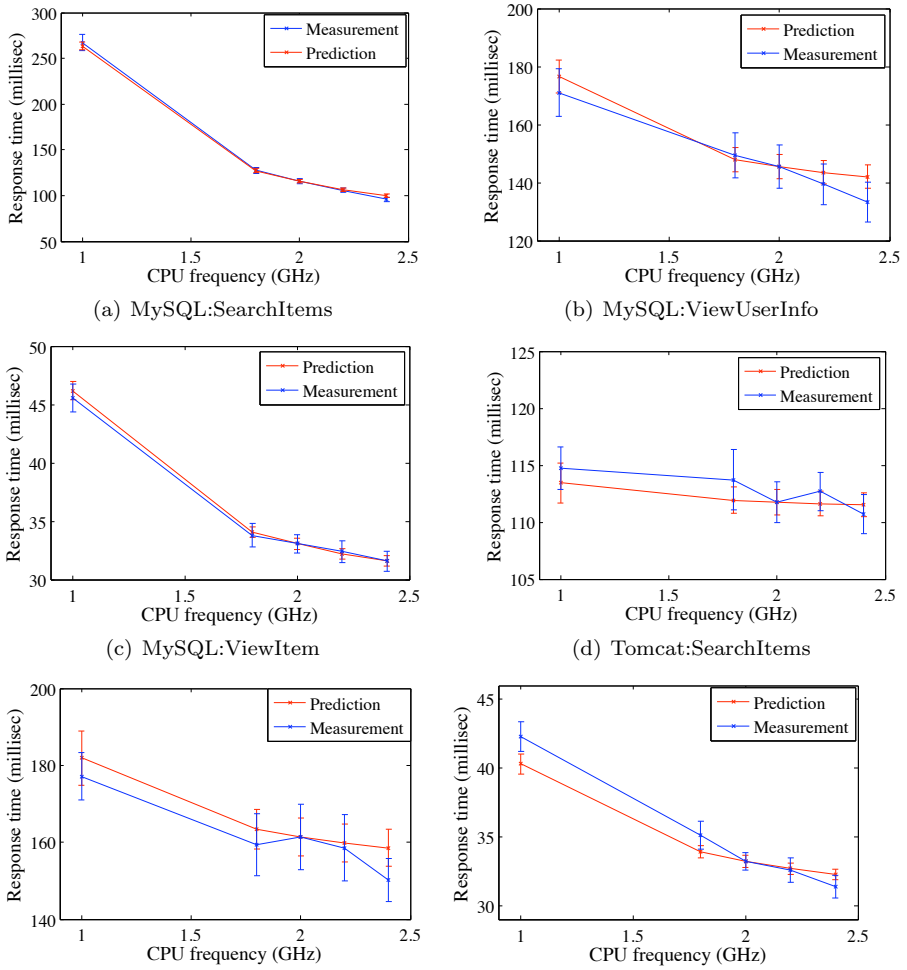
Note that computing the workload gradients requires that the transaction being measured have a high enough frequency in the existing workload. For rare transactions, our signal injection technique may not be able to inject a strong enough signal to accurately estimate the workload gradient by just leveraging existing workload. However, since those transactions are rare, their impact on the total utilization is small. Therefore, in practice, we can set a threshold for transaction arrival rate and only calculate the gradients for the transaction types that exceed this rate. Alternatively, synthetic transactions can be generated for the measurement.

5.3. CPU Gradients: Basic Configuration

We conducted a set of experiments to evaluate the predictive power of the CPU gradients using the basic configuration of RUBiS. The predictive power is evaluated in terms of the ability of the CPU gradients to predict the application's mean end-to-end response time at an operating configuration different from the one that was used to compute the gradients. In the following, we show the predictive power of the frequency gradient and VM capacity gradient.

Frequency gradient: We measured the frequency gradients for the Tomcat and MySQL tiers when both servers were running at 2.0 GHz and the workload rate remained relatively constant. For each experiment, we sampled a total of 2048 points to estimate the gradients. To evaluate the predictive power of the gradients, we varied the CPU frequency of the two servers and used equation 5 to predict the end-to-end response time of the three transactions. The predicted results are shown in Figure 7 against measurement results taken at 10-minute intervals. The error bars in the figure are the 95% confidence intervals of the estimates. The results show that, by using the basis function to approximate the nonlinearity, the frequency gradient measured at an operating configuration can be used to accurately predict the end-to-end response time as the CPU frequency changes from its lowest value (1.0 GHz) to the highest (2.4 GHz).

VM capacity gradient: We utilized the Xen credit scheduler to perturb the amount of CPU resources allocated to each VM and measure the VM capacity gradients for all three tiers. We set the base operating configuration such that the capacity allocated to a VM is 2.5 times what the VM demands on average, i.e., the virtual CPU utilization is 40%. For each experiment, we sampled a total of 2048 points when the workload remained relatively constant, and estimated the gradients. To evaluate the predictive power of the VM capacity gradients, we varied the CPU capacity allocations of the VMs running the Tomcat and MySQL tiers and used equation 5 to predict the end-to-end response times of the three transaction types considered. The predictions, along with the 95% confidence intervals, are shown in Figure 8 and Figure 9 against the measured results in both the single CPU setting and the multicore setting. The results



(a) MySQL:SearchItems (b) MySQL:ViewUserInfo

(c) MySQL:ViewItem (d) Tomcat:SearchItems

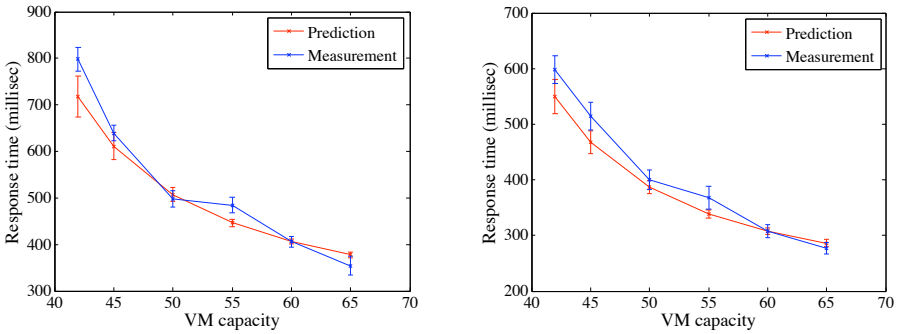
(e) Tomcat:ViewUserInfo (f) Tomcat:ViewItem

Figure 7: Prediction accuracy of the frequency gradients

show that the VM capacity gradients measured in a base configuration are able to predict the application's end-to-end response times for different transaction types within a wide operating range and that they capture the effects of queuing delay in the system to a reasonably large extent.

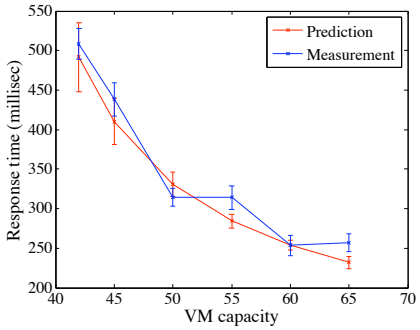
5.4. Different Communication Patterns

In the above, we have demonstrated the predictive power of the CPU gradients in a basic setup of the RUBiS application. However, real enterprise systems often have communication patterns—such as load balancing and asynchronous

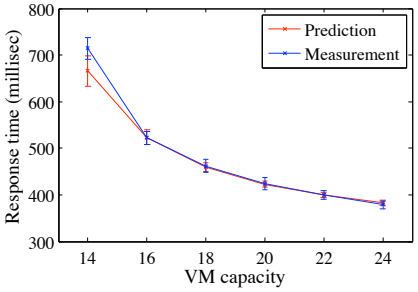


(a) MySQL:SearchItems.

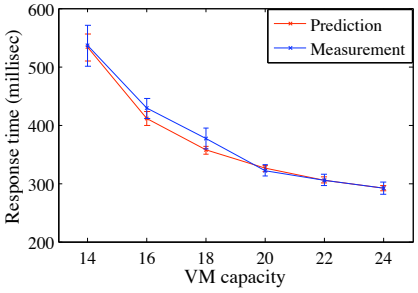
(b) MySQL:ViewUserInfo



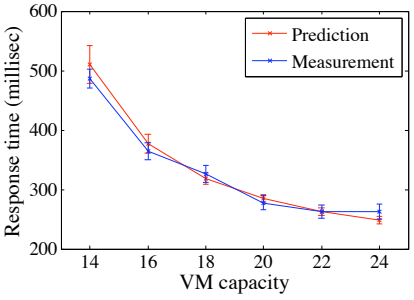
(c) MySQL:ViewItem



(d) Tomcat:SearchItems.



(e) Tomcat:ViewUserInfo

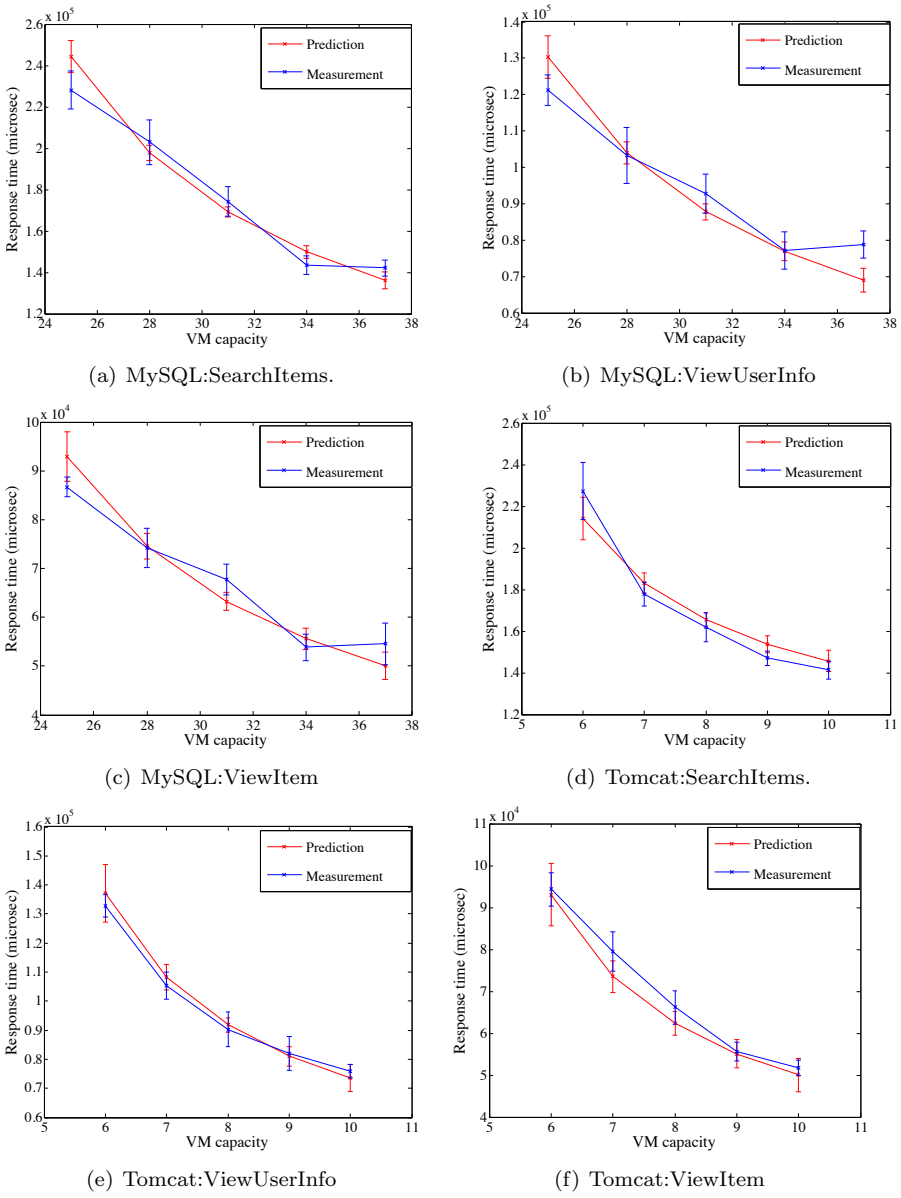


(f) Tomcat:ViewItem

Figure 8: Prediction accuracy of the VM capacity gradients in the VM testbed

communication—not captured in this basic setup. In the following, we will demonstrate the accuracy of CPU gradients under some of these communication patterns.

Load balancing: Load balancing is a technique widely used in enterprise systems, in which workload is dispatched across several instances of the same server.



(a) MySQL:SearchItems.

(b) MySQL:ViewUserInfo

(c) MySQL:ViewItem

(d) Tomcat:SearchItems.

(e) Tomcat:ViewUserInfo

(f) Tomcat:ViewItem

Figure 9: Prediction accuracy of the VM capacity gradients in the MVM testbed

In RUBiS, load balancing can be done by using the *mod_jk* Apache module. In the next experiments, we added one more Tomcat server and configured the Apache server to dispatch load equally to the two Tomcat servers. We deployed

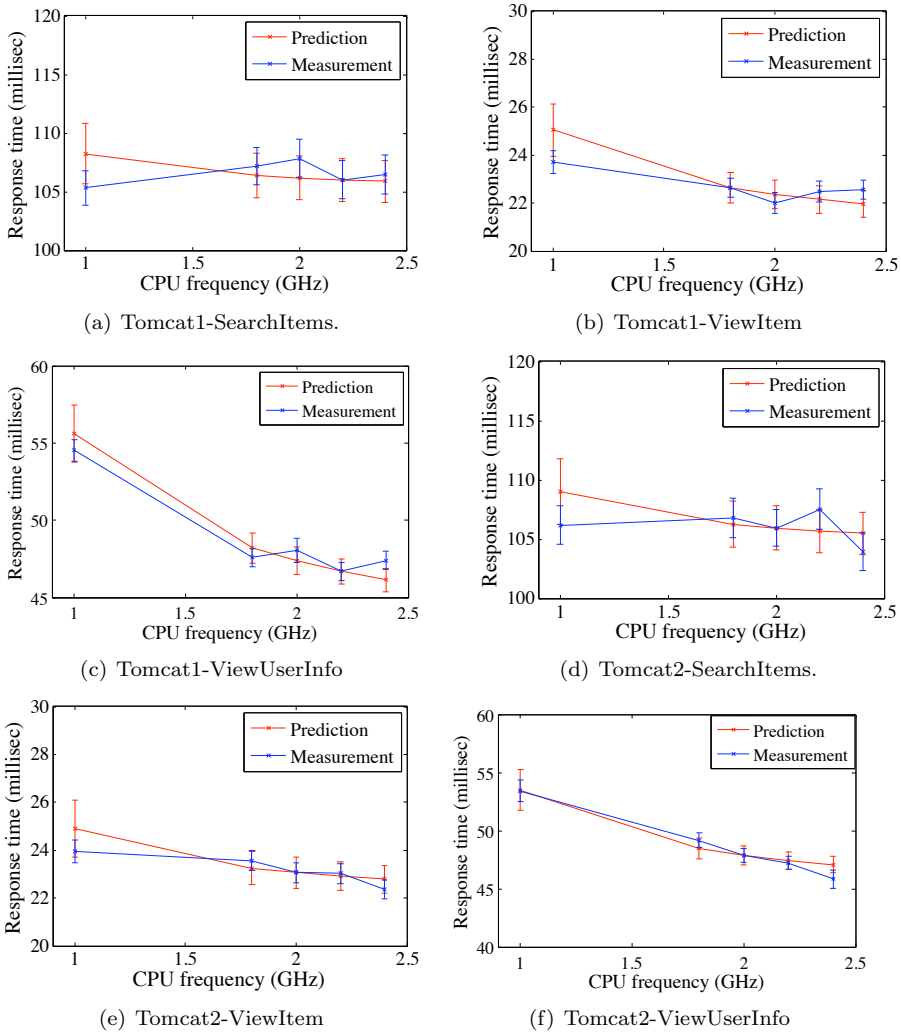
this new RUBiS configuration in both the PM and VM testbeds and computed the frequency and VM capacity gradients, respectively, for the two Tomcat servers. In the frequency gradient experiment, we measured the base response time when both servers ran at 2.0 GHz. In the VM capacity gradient experiment, we measured the base response time when the capacity allocated to a VM was 2.5 times what the VM demands on average. We then used the gradients to predict the response time of the application at different operating configurations for the three transactions based on the base configuration measurements. Figures 10, 12 and 11 show excellent agreement between the predictions and the measurements for all three transactions in all three testbeds.

Asynchronous state replication: State replication is often used in enterprise systems to ensure service availability in case of failures. We configured the two Tomcat application servers to perform asynchronous session state replication for fault tolerance. However, since the standard RUBiS servlet implementation does not support session state replication, we added replication for the *ViewUserInfo* transaction by replicating the session state of *tomcat1* in *tomcat2*. Figure 13 presents the results for frequency gradients over a range of CPU frequencies[2]. The results show that the frequency gradient is able to capture the fact that the change in the CPU frequency of the *tomcat2* server has no impact on the end-to-end response time due to the asynchronous replication.

5.5. RUBBoS Application

To further evaluate the applicability and predictive power of the CPU gradients, we deployed another multitier application, RUBBoS, on all of our testbeds. We used the standard RUBBoS client emulator to generate browse-only transactions and measured the frequency and VM capacity gradients for the three most frequent transactions: *BrowseCategory* (BC), *BrowseStoriesByCategory* (BS), and *ViewStory* (VS).

[2]The Tomcat clustering implementation requires multicast and we were unable to get the two Tomcat servers to discover each other through the Xen virtual network bridge in the VM testbed. Therefore we present only the results from the PM testbed.



(a) Tomcat1-SearchItems.

(b) Tomcat1-ViewItem

(c) Tomcat1-ViewUserInfo

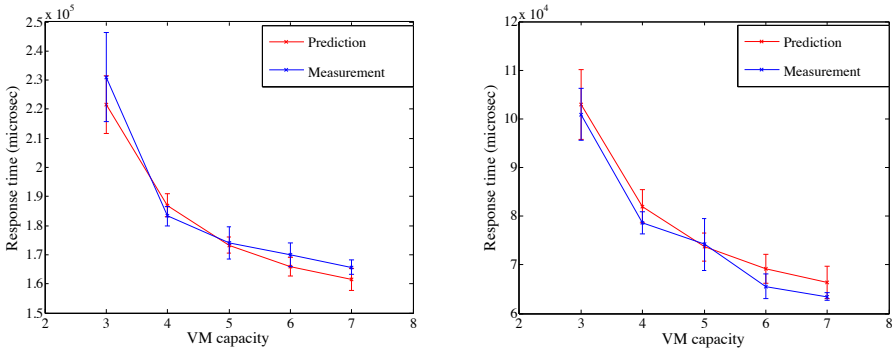
(d) Tomcat2-SearchItems.

(e) Tomcat2-ViewItem

(f) Tomcat2-ViewUserInfo

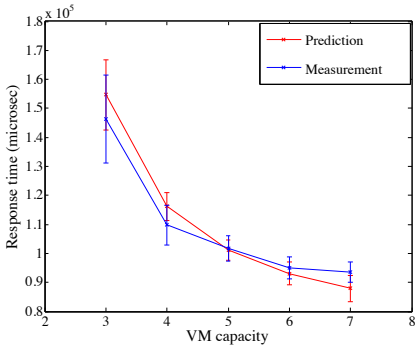
Figure 10: Load balancing (frequency gradient)

First, we showed the predictive power of the CPU gradients using the MVM testbed by changing only the CPU operating configurations of different VMs. The gradients are measured when the CPU utilization of the VM is 40% when the workload remained relatively constant. The results in Figure 16 show that the CPU gradient can predict the end-to-end transaction response time for the RUBBoS application accurately.

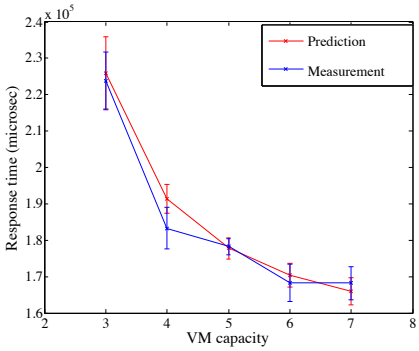


(a) Tomcat1-SearchItems.

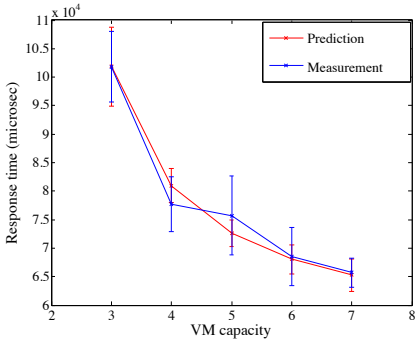
(b) Tomcat1-ViewItem



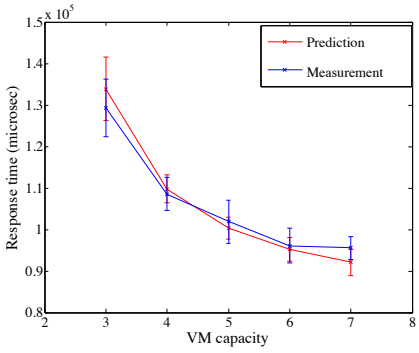
(c) Tomcat1-ViewUserInfo



(d) Tomcat2-SearchItems.



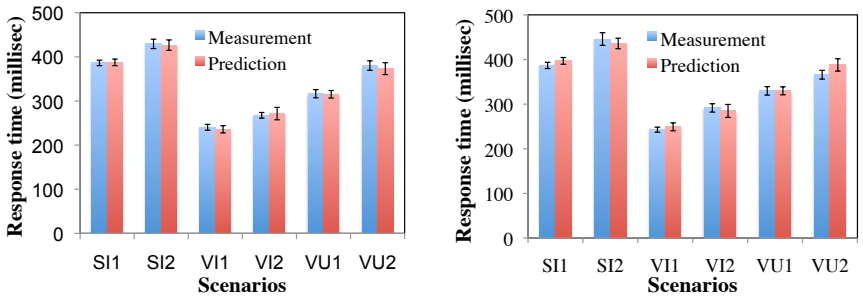
(e) Tomcat2-ViewItem



(f) Tomcat2-ViewUserInfo

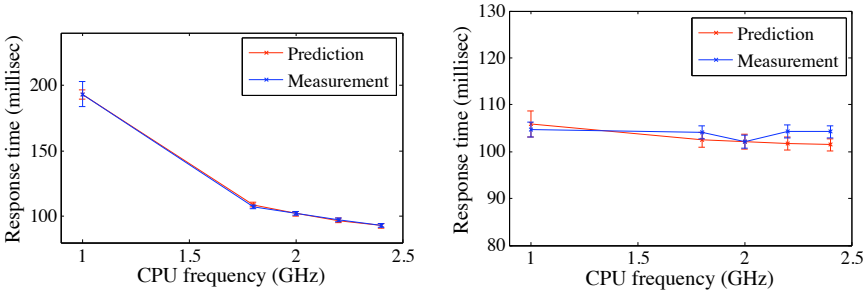
Figure 11: Load balancing in the MVM testbed (VM capacity gradient)

Then, we evaluated the predictive power of the CPU gradients by changing both the workload and the CPU operating configurations using the PM and VM testbeds. In the PM testbed experiment, we increased the workload by



(a) Tomcat1 (b) Tomcat2

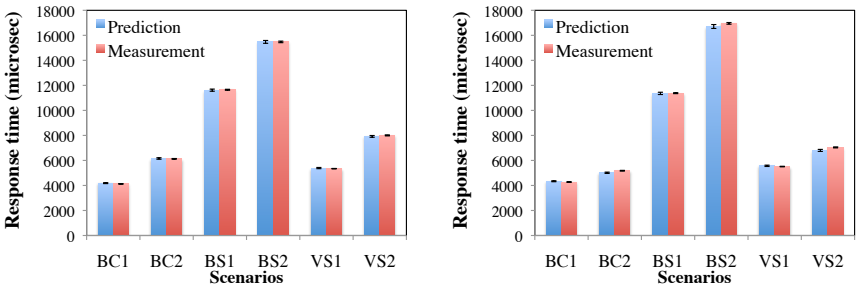
Figure 12: Load balancing in the VM testbed (VM capacity gradient)



(a) Tomcat1 (b) Tomcat2

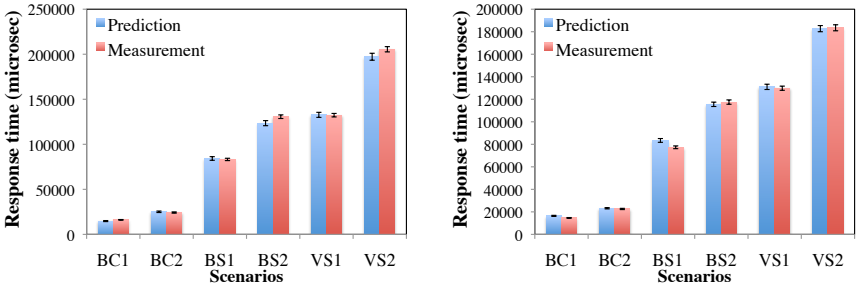
Figure 13: Asynchronous state replication

one-third of the workload at which we computed the gradients. Then, we measured the mean response time of the system when both server CPUs ran at 2.0 GHz and used equation 5 to predict the new end-to-end response time when each server CPU is changed to 1.0 GHz and 2.4 GHz independently at the new workload. Similarly, in the VM testbed experiment, we increased the workload by one-fourth from the workload at which we computed the gradients. Then, we measured the mean response time of the system when both VMs ran at 40% CPU utilization and used equation 5 to predict the new end-to-end response time as the capacity allocated to each VM changed and the system was exposed to the new workload. The predicted results of all three transactions are shown in Figures 14 and 15. The figures demonstrate that the CPU gradients measured at one workload can be used to predict the end-to-end response time at a different workload, which further strengthens the usefulness of the CPU gradient in practice.



(a) Apache (b) MySQL

Figure 14: RUBBoS prediction (frequency gradient)

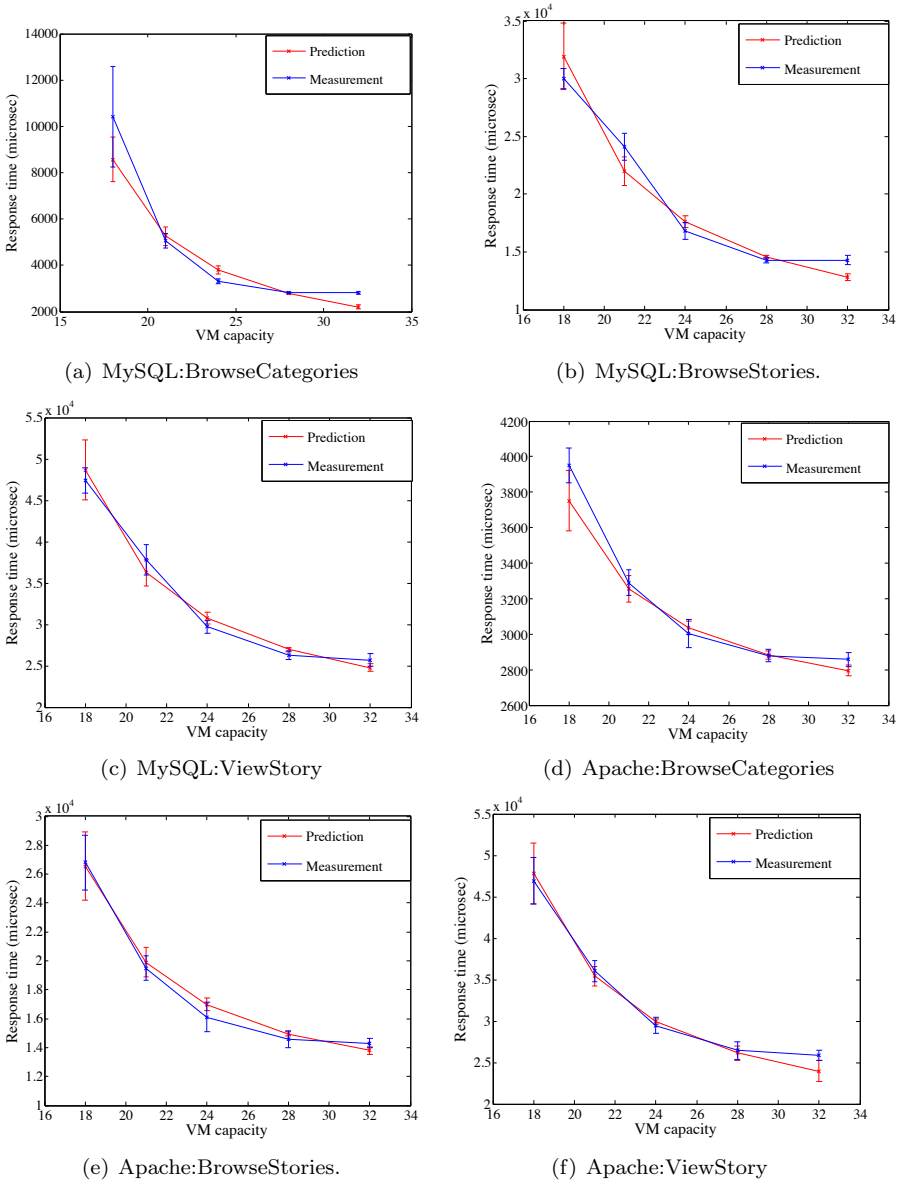


(a) Apache (b) MySQL

Figure 15: RUBBoS prediction (VM capacity gradient)

5.6. Multiple Applications

In a virtualized environment, multiple applications share the same infrastructure. Different VMs of multiple applications running on the same hardware might interfere with each other, adding challenges to our technique. To demonstrate the applicability of our technique in this scenario, we deployed RUBiS and RUBBoS together on both our VM and MVM testbed. We deployed the VM components of RUBiS and RUBBoS in the testbed using 3 arbitrary configurations and used the gradient measured above in standalone deployment to predict the end-to-end response time for each deployment configuration of the system when we decrease the VM capacities of all the VMs in the system by approximately 20% as shown in table 1. In this set of experiments, the system workload rate was intentionally reduced to 75% at which the gradients are measured, and the final results demonstrate that the gradients measured at one workload in a standalone configuration can be used to predict the response time as the workload rate changes in a shared environment.



(a) MySQL:BrowseCategories

(b) MySQL:BrowseStories.

(c) MySQL:ViewStory

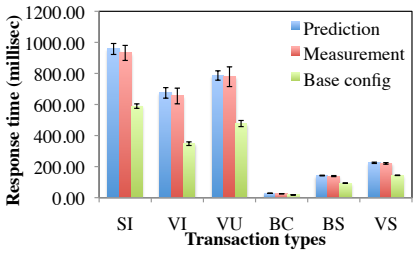
(d) Apache:BrowseCategories

(e) Apache:BrowseStories.

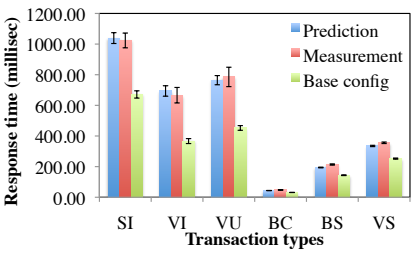
(f) Apache:ViewStory

Figure 16: Prediction accuracy of the VM capacity gradients in the MVM testbed

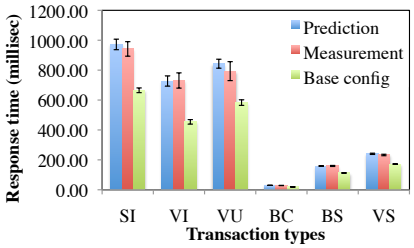
Figure 17 and Figure 18 shows the results. In each subfigures, the third column plots the measured response time for each transaction type in the base configuration in each deployment; the second column plots the measured response time after the VM capacities of all 5 VM components have changed;



(a) ConfigA



(b) ConfigB



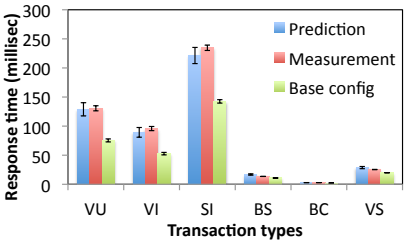
(c) ConfigC

Figure 17: Multiple applications

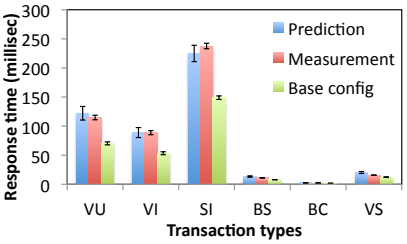
and the first column plots the predicted response time at the new configuration using the gradients from each base configuration. As the results suggested, the gradient measured at a different workload in the standalone deployment configuration can be used to accurately predict the end-to-end response time in all of our deployment configurations, in which VMs of different applications are co-located arbitrarily in a set of physical machines. However, as the new operating configuration moves further away from the base configuration, and when the system workload has changed significantly, the prediction will become less and less accurate. The results above demonstrate that the gradient can be used locally to accurately predict the response time in a large enough operating range.

Deployment	**Machine 1**	**Machine 2**	**Machine 3**
ConfigA	apache1 apache2	tomcat1	mysqld1 mysqld2
ConfigB	apache1 tomcat1 mysqld1	apache2 mysqld2	
ConfigC	apache1 tomcat1 mysqld2	apache2 mysqld1	

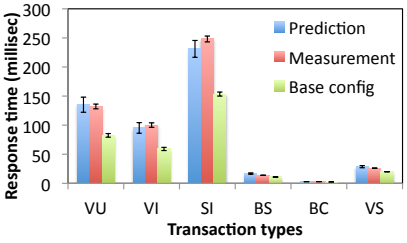
Table 1: Random deployments



(a) ConfigA



(b) ConfigB



(c) ConfigC

Figure 18: Multiple applications (multicore)

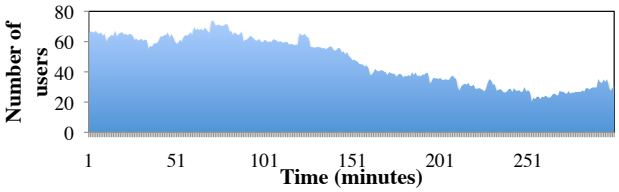


Figure 19: Workload scenario

6. Energy Conservation

In this section, we show how CPU gradients can be used for performance-aware energy conservation efforts by deploying the two energy controllers described in Section 4. We used the same 3-tier RUBiS application as the target system, but made the workload more representative of online systems by using publicly available Web traces from the 1998 World Cup site [13]. We arbitrarily chose several hours' worth of traffic and then varied the number of concurrent users in the RUBiS client emulator according to this trace, but scaled to a range that our experimental setup could handle. Figure 19 shows the number of concurrent users used in the frequency controller experiment as a function of time. For the virtual machine experiments, we used the same number of concurrent users, but increased the mean think time from one second to 7 sec-

onds to compensate for the slower testbed. The results for the controllers and other competing approaches show the mean response times for each of the three transaction types considered by the controllers, followed by the power consumption of the system over the course of the entire experiment. For simplicity, we used the same response time threshold for all transactions, but the approach can very easily support different thresholds for different transaction types. The power usage of the testbed computers was measured using a power meter. All testbeds use Watts up? Pro power meters [14] configured to report mean power consumption of the test system once every minute.

6.1. Frequency Gradient Controller

We compare the performance of the frequency gradient controller with two other commonly used techniques. The *ondemand* controller is an energy conservation controller that is a part of the Linux kernel, and comes with every standard Ubuntu distribution. The controller adjusts the local CPU frequency solely by looking at the recent CPU utilization, and aggressively reduces machine power consumption by lowering the CPU speed until the utilization goes above 80%. The *performance* controller is also a standard Linux controller, but it simply sets the CPU to its maximum frequency, thus disabling DVFS altogether. We chose a response time threshold of 150 ms by taking the average of the system's response time with all CPUs set to the minimum and maximum frequencies and at the peak system load of 75 concurrent users. Thus, it represents a responsive but attainable configuration for the resources available to the application.

The response time and power consumption results for the three controllers are shown in Figure 20. Each data point in the response time plots is an average over a 10-minute window. The thick horizontal line indicates the response time threshold. As can be seen from the response time figures, the *performance* controller easily meets the response time threshold at all times. However, the *ondemand* controller substantially exceeds the threshold for the *SearchItemsByCategory* transaction during the second half of the experiment. This is due

to the low workload in this period, which causes the utilizations to drop, and the controller to become too aggressive by setting all CPU frequencies to their lowest value. In comparison, our *gradient controller* remains substantially below the threshold, though not to the same extent as the *performance* controller.

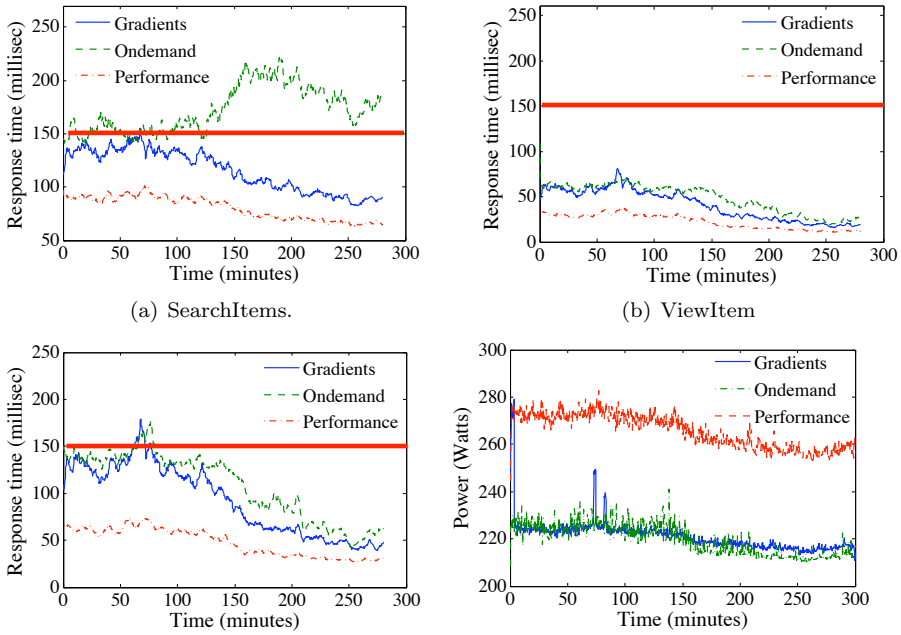
What was unexpected, however, is the fact that the response times for the *gradient* are far lower than the threshold, despite the response time optimization techniques the controller uses. The reason is the limited number of CPU frequency settings (five) in the AMD processor we used. The optimal frequency for the MySQL server would have been between 1.0 GHz and 1.8 GHz, but such an option being unavailable, the controller picked the lowest frequency that did not violate the threshold (1.8 GHz). For newer processors with a larger number of available CPU frequencies, one would expect to see the response times much closer to the threshold value.

Finally, as seen in the power consumption graph, both the *ondemand* and *gradient* controllers result in significantly reduced power consumption of approximately 18%. The *gradient* controller uses slightly more power during the second half of the experiment, because it runs the MySQL server one frequency setting higher than the *ondemand* controller does in a bid to preserve performance. However, the difference is not very high, and the results show that the gradient-based controller provides significant energy savings, similar to those of the local utilization-based controllers, while still ensuring that end-to-end responsiveness is preserved.

6.2. VM Capacity Gradient Controller

6.2.1. Single application

Next, we present the results of using the VM capacity controller described in Section 4 to generate dynamic performance-aware server consolidation decisions on the VM testbed. The consolidation was then implemented by using live migration and then shutting down unused machines to save energy. We compare the controller against two different strategies: 1) a *performance* strategy without any server consolidation in which a static deployment of a single physical



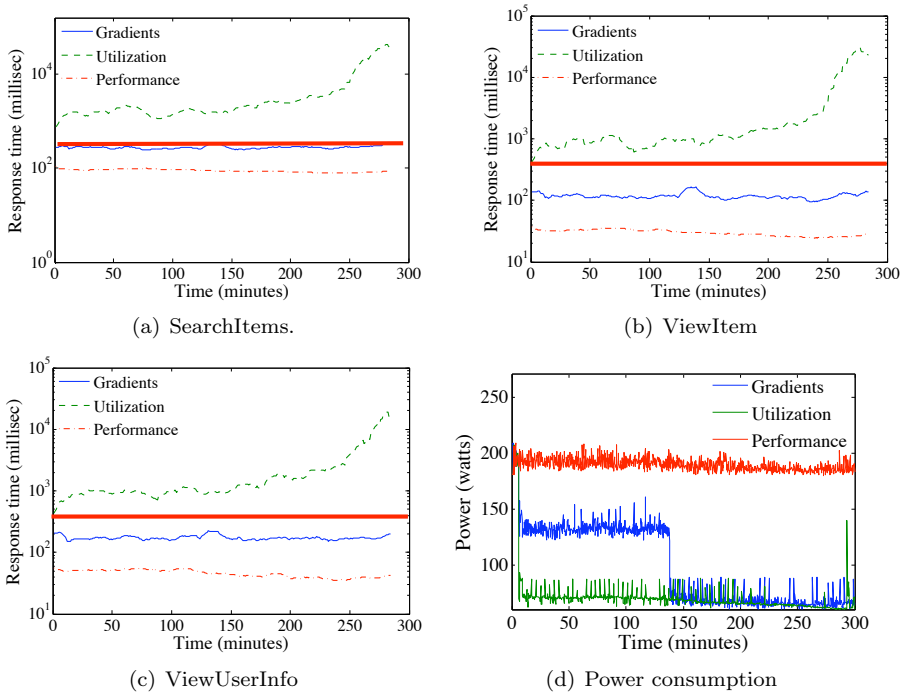
(a) SearchItems. (b) ViewItem

(c) ViewUserInfo (d) Power consumption

Figure 20: Response time and power (frequency gradient)

host per VM is used, and 2) a *utilization* strategy that scales and dynamically consolidates VMs so as to maintain constant virtual CPU utilization of 60% in each VM. To do so it uses the same FFD bin-packing algorithm described in Section 4, but does not optimize the VM capacities based on response time. For the experiments, we used a response time threshold of 300 msec derived using the same methodology used in the frequency controller experiments.

The results are shown in Figure 21. As before, each data point in the response time plot is an average over a 10-minute window. The response time is plotted on a logarithmic scale. As we can see from the result, although the *utilization* controller appears to save more energy than the *gradient* controller by consolidating all VMs into one physical machine, it causes unacceptable response time degradation that goes far beyond the threshold. Upon detailed investigation, we discovered that the large magnitude and sustained nature of this degradation was caused by initial increases in system response time that caused a degradation in system throughput and slowed the clients down. This



(a) SearchItems.

(b) ViewItem

(c) ViewUserInfo

(d) Power consumption

Figure 21: Response time and power (VM gradient)

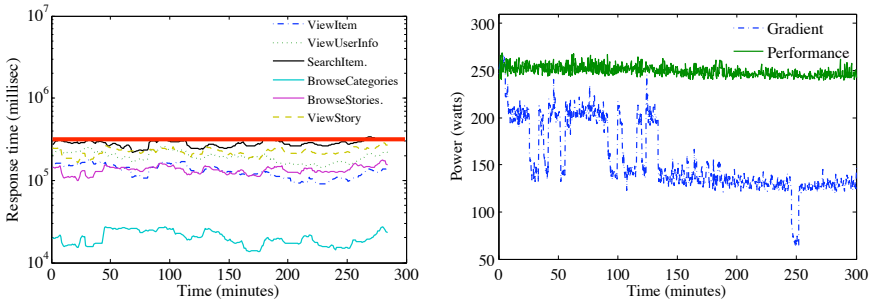
led to a decrease in workload, leading to even lower utilization and subsequently causing the controller to reduce VM capacities further. Thus, a vicious cycle was established that caused response times to spiral out of control. While new incoming users might prevent this problem from occurring to such a degree in an open system, the results reveal the dangers inherent in scaling based solely on utilization values, and thus stress the importance of performance-aware techniques.

In contrast, the *gradient* controller was able to maintain the mean response time right below the threshold for about 95% of the experiment, and, compared to the *performance* controller, still achieve 48% energy savings. Thus, it can be seen that even in virtualized environments, gradients provide the controller with the best of both worlds: energy conservation combined with performance preservation.

6.2.2. Multiple applications

In environments such as VM-based clouds and data centers, the physical infrastructure is usually shared among multiple applications at the same time. Each application has its own responsiveness requirement that needs to be met. To demonstrate the ability of our framework to tackle such shared resource scenarios, we deployed RUBiS and RUBBoS at the same time on both our VM testbed and MVM testbed. In the initial configuration, we placed both Apache servers on one machine, the Tomcat server on another machine, and the two MySQL servers separately on the remaining two machines. The same World Cup traces as before were used to produce workloads for both applications. A uniform response time threshold of 300 msec was set for all transactions. In the MVM testbed, because the 16-core machine is much more powerful than all the machines combined in the VM testbed, we programmatically disable some CPU cores in each machine and increase the workload rate so as to match the power of the MVM testbed with the workload demands. To do so, we only enable the use of 3 cores on each machine. Core 0 is dedicated for the hypervisor, while core 1 and core 2 are dedicated for the guest VMs. The controller turns off a physical machine when there is no guest VMs running on the machine.

Figure 22 shows the response time series for all the transactions of both applications and the power consumption of the system for a period of 4 hours using both the VM capacity controller and the performance strategy in the VM testbed. Due to the poor performance of the utilization strategy, it was not used as a basis for comparison. Each data point in the response time plot is an average over a 10-minute window. The response time is plotted on a logarithmic scale. As can be seen, the mean response time for all transactions of both applications stays below the threshold (indicated by the thick horizontal line) for approximately 95% of the time. In the first half of the power consumption curve in the figure, the controller repeatedly oscillates between packing all VMs into 3 machines and into 2 machines, and causes significant power spikes in the process. The spikes are due to the higher power consumption required during migration.



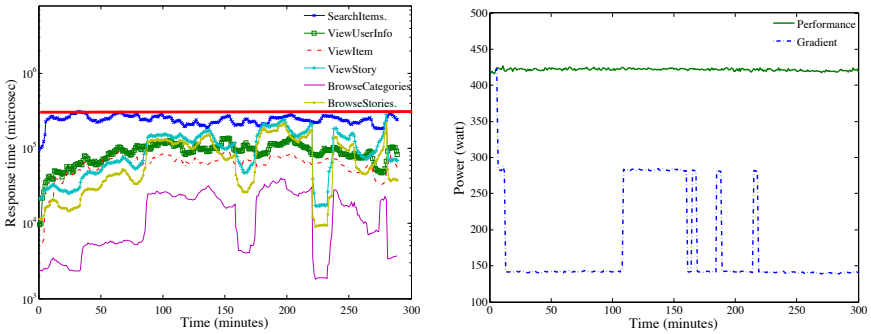
(a) Response time (b) Power consumption

Figure 22: Response time and power in the VM testbed (Multiple applications)

Nevertheless, when compared to the performance strategy, our controller allows the applications to consume 38% less energy, thus demonstrating its ability to adapt to workload changes even in a multi-application shared environment.

Figure 23 shows the response time series for all the transactions of both applications and the power consumption of the system for a period of 5 hours using both the VM capacity controller and the performance strategy in the MVM testbed. Each data point in the response time plot is an average over a 10-minute window. The response time is plotted on a logarithmic scale. As can be seen, the mean response time for all transactions of both applications stays below the threshold (indicated by the thick horizontal line) for approximately 97% of the time. As we can see, the controller adapts to higher workload during the middle of the experiment by turning on more physical machines and migrating VMs to run on them. When compared to the performance strategy, the controller allows the applications to consume 57% less energy, thus again demonstrating its ability to adapt to workload changes even in a multicore, multi-application shared environment. Note that , right now, our hardware does not allow turning on or off individual core at runtime. In the future, with a more fine-grained control over individual CPU core, even more energy saving will be possible.

Finally, more aggressive energy conservation can be achieved by combining frequency and VM capacity gradients in a VM environment. Further, with advanced control techniques that take into account migration costs, and with



(a) Response time (b) Power consumption

Figure 23: Response time and power in the MVM testbed (Multiple applications)

increased provisions for DVFS such as those present in future processors such as the Intel Nehalem, we believe that additional energy savings are possible. Nevertheless, even on current hardware, the techniques presented provide a substantial amount of energy savings with very minimal cost and effort.

7. Related Work

The general problem of performance prediction in multitier systems is a well-studied one. Queuing network formulations such as Layered Queuing Networks (LQN) [15] provide an especially appropriate formulation to model multitier systems and have been used effectively in many case studies (e.g., [16] and [17] use queuing networks and models to estimate the end to end response time of multi-tier systems). However, such models require detailed knowledge of the system transactions, their resource usage, and communication patterns. To alleviate these drawbacks, data intensive approaches including machine learning have also been proposed to construct models for black box systems as in [18] and [19]. However, such approaches can provide performance estimates only if very similar configurations have already been seen before, and thus do not provide true predictive capabilities. In contrast, by imposing restrictions on how a system's metrics evolve via basis functions formulated using high level knowledge about the behavior of distributed systems, gradients neither require detailed system knowledge nor extensive data collection. The closest related

work is an approach proposed by Stewart et. al. in [20] and [21] in which passive data collection is combined with an M/M/1 queuing model based template in order to estimate the service and waiting times at each resource of a complex multitier system. However, their techniques become increasingly inaccurate as the system load increases. In contrast, by introducing active perturbations into a running system, our approach can predict metrics in configurations that are very different from those in which measurement was performed.

The specific energy saving techniques deployed in this paper are not new. For example, dynamic voltage and frequency scaling (DVFS) has been utilized for power savings in [22, 23] or to allocate a given power budget to minimize response time [24]. To the best of our knowledge, [25] is the only work that considers end-to-end performance impact when performing DVFS for multitier applications. This work assumes a pipe-lined system, and uses a traditional M/M/1 queuing network model for performance prediction. To obtain the parameters for the model, server instrumentation along with offline profiling is used. In contrast, our work is not limited to pipelined systems as demonstrated by the Tomcat replication results, and is a blackbox approach that does not require knowledge of application topology, configuration, or instrumentation of server components.

The use of virtual machine technology, including shutting down unneeded servers, has been used for power savings in a number of research projects. While many projects introduce controllers more complex than our basic controllers (e.g., the cost of control actions is explicitly considered in [26, 27, 28]), the controllers typically base their control decisions either on measurements of response time (in a single-tier system in [26]), SLA violations for individual VMs (e.g., [28]), or CPU utilization. In contrast, since our system bases its adaptations on changes in workload, it can react before SLAs are violated or CPU utilization at some tier starts increasing. Furthermore, since the gradient models tell where and how much the resources should be adjusted, a simple control algorithm is sufficient to manage the system. Finally, separate SLA measurements for each tier are not necessary in our approach. Integrating the other advanced features

from these controllers (e.g., cost awareness and workload prediction) are part of our future work. The vManage system is one of the few ones that controls both CPU frequency and VM placement [28]. Implementing a similar controller that uses both frequency and VM capacity gradients is part of our future work.

8. Conclusions

In this paper[3], we have proposed CPU gradients, a new technique for predicting the impacts of CPU frequency and virtual machine capacity changes on the per-transaction end-to-end response times of multitier applications. Unlike traditional queuing models, CPU gradients are simple point derivative-based predictors that can be automatically constructed using runtime measurement techniques without the need for detailed knowledge of the target system. We developed runtime measurement techniques for CPU gradients, and have experimentally shown that the produced models provide accurate predictions of reality. Finally, we have also shown how CPU gradient based models can be used to construct performance aware energy controllers and have experimentally demonstrated that such controllers can save substantial amounts of energy while ensuring the responsiveness of the target system under realistic workload conditions derived from live Web traces.

References

[1] "EPA Report on Server and Data Center Energy Efficiency," Aug 2007.

[2] D. Galletta, R. Henry, S. McCoy, and P. Polak, "Web site delays: How tolerant are users?" *J. of the Assoc. for Information Sys.*, vol. 5, no. 1, pp. 1–28, 2004.

[3] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman, "Determining causes and severity of end-user frustration," *Int. Journal of Human-Computer Interaction*, vol. 17, no. 3, pp. 333–356, 2004.

[4] A. Bouch, A. Kuchinsky, and N. Bhatti, "Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service," in *Proc. CHI2000 Conf. on Human Factors in Computing Systems*, 2000, pp. 297–304.

[3] This is an extended version of our conference paper [29]

[5] D. Farber, "Google's Marissa Mayer: Speed wins," ZDNet Between the Lines, Nov. 2006.

[6] S. Chen, K. Joshi, M. Hiltunen, W. Sanders, and R. Schlichting, "Link gradients: Predicting the impact of network latency on multitier applications," in *Proc. INFOCOM*, Apr. 2009.

[7] "Web caching," 2007, http://www.web-caching.com/traces-logs.html.

[8] "Xen and the art of virtualization," in *Proc. SOSP*, 2003, pp. 164–177.

[9] K. Choi, W. Lee, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation," in *Proc. IEEE/ACM Int. conf. on Computer-aided design*, 2004, pp. 29–34.

[10] E. G. C. Jr., G. Galambos, S. Martello, and D. Vigo, *Du, D.Z., Paradalos, P.M., eds.: Handbook of Combinatorial Optimization.* Kulwer, 1998, ch. Bin Packing Approx. Algorithms: Combinatorial Analysis.

[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of EJB applications," in *Proc. OOPSLA*, 2002, pp. 246–261.

[12] ObjectWeb, "Rubbos: Bulleting borad benchmark," 2005.

[13] M. Arlitt and T. Jin, "Workload charaterization of the 1998 world cup web site," in *HP Technical Report, HPL-99-35*, 1999.

[14] EED, "Watts up? power meter," 1997, https://www.wattsupmeters.com/secure/index.php.

[15] M. Woodside, J. Neilson, D. Petriu, and S. Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed software," *IEEE Trans. on Computers*, vol. 44, no. 1, pp. 20–34, 1995.

[16] S. Bhulai, S. Sivasubramanian, R. van der Mei, and M. van Steen, "Modeling end-to-end response times in multi-tier internet applications," *Managing Traffic Performance in Converged Networks*, vol. 4516, pp. 519–532, 2007.

[17] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *Proc. ACM SIGMETRICS*, 2005, pp. 291–302.

[18] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *Proc. NSDI*, 2005, pp. 71–84.

[19] P. Bodik, C. Sutton, A. Fox, D. Patterson, and M. Jordan, "Response-time modeling for resource allocation and energy-informed slas," in *Workshop on Statistical Learning Techniques for Solving Systems Problems (MLSys)*, 2007.

[20] C. Stewart, T. Kelly, and A. Zhang, "Exploiting Nonstationarity for Performance Prediction," in *Proc. EuroSys*, Mar 2007.

[21] 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 Conf.*, June 2008.

[22] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. on Low Power Electronics and Design*, 1998, pp. 76–81.

[23] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing server energy and operational costs in hosting centers," in *Proc. ACM SIGMETRICS*, 2005, pp. 303–314.

[24] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *Proc. SIGMETRICS*, 2009.

[25] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu, "Dynamic voltage scaling in multitier web servers with end-to-end delay control," *IEEE Trans. on Computers*, vol. 56, no. 4, pp. 444–458, July 2006.

[26] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang, "Power & performance management of virtualized computing environments via lookahead control," *Cluster Comp.*, vol. 12, no. 1, pp. 1–15, 2009.

[27] 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.

[28] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan, "vManage: Loosely coupled platform and virtualization management in data centers," in *Proc. ICAC*, 2009, pp. 127–136.

[29] S. Chen, K. Joshi, M. Hiltunen, W. Sanders, and R. Schlichting, "Cpu gradients: Performance-aware energy conservation in multitier systems," in *Proc. IGCC*, Aug. 2010.