# Adaptive Control of Virtualized Resources in Utility Computing Environments [*]

Pradeep Padala, Kang G. Shin
EECS, University of Michigan
Ann Arbor, MI, 48109, USA
ppadala@eecs.umich.edu

Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant
Hewlett Packard Laboratories
Palo Alto, CA 94304, USA
xiaoyun.zhu@hp.com

Kenneth Salem
University of Waterloo
Waterloo, Ontario, Canada
kmsalem@uwaterloo.ca

## ABSTRACT

Data centers are often under-utilized due to over-provisioning as well as time-varying resource demands of typical enterprise applications. One approach to increase resource utilization is to consolidate applications in a shared infrastructure using virtualization. Meeting application-level quality of service (QoS) goals becomes a challenge in a consolidated environment as application resource needs differ. Furthermore, for multi-tier applications, the amount of resources needed to achieve their QoS goals might be different at each tier and may also depend on availability of resources in other tiers. In this paper, we develop an adaptive resource control system that dynamically adjusts the resource shares to individual tiers in order to meet application-level QoS goals while achieving high resource utilization in the data center. Our control system is developed using classical control theory, and we used a black-box system modeling approach to overcome the absence of first principle models for complex enterprise applications and systems. To evaluate our controllers, we built a testbed simulating a virtual data center using Xen virtual machines. We experimented with two multi-tier applications in this virtual data center: a two-tier implementation of RUBiS, an online auction site, and a two-tier Java implementation of TPC-W. Our results indicate that the proposed control system is able to maintain high resource utilization and meets QoS goals in spite of varying resource demands from the applications.

## Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: [Modeling techniques]; D.4.8 [**OPERATING SYSTEMS**]: Performance—*Performance of Virtualized Data Center*; I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search—*Control theory*

## Keywords

Data center, server consolidation, virtualization, control theory, application QoS, resource utilization

## General Terms

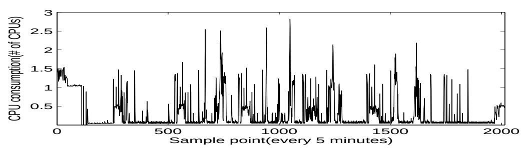
DESIGN, EXPERIMENTATION, MANAGEMENT, PERFORMANCE

[*]The work reported in this paper was supported in part by NSF under grant CCF 0444417

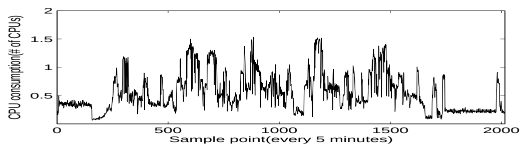
## 1. INTRODUCTION

Today's enterprise data centers are designed with a silo-oriented architecture in mind: each application has its own dedicated servers, storage and network infrastructure, and a software stack tailored for the application controls these resources as a whole. Due to the stringent requirements placed on the enterprise applications and the time-varying demands that they experience, each application silo is vastly over-provisioned to meet the application service goals. As a result, data centers are often under-utilized, while some nodes may sometimes become heavily-loaded, resulting in service-level violations due to poor application performance [3]. For example, Figures 1(a) and 1(b) show the CPU consumption of two nodes in an enterprise data center for a week. Each node has 6 CPUs, and we can see that both nodes are utilized under 10% most of the time. We also note that the maximum CPU usage is much higher than the average CPU usage. Similar problems are observed in other resources including disk, network and memory. So, if we were to provision the resources based on either the maximum or the average demand, the data center may either be grossly under-utilized or experience poor application-level QoS due to insufficient resources under peak loads.

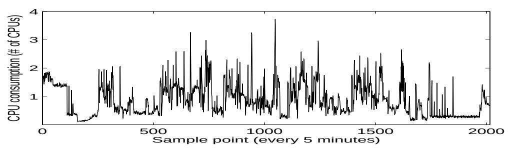
Next-generation enterprise data centers are being designed with a utility computing paradigm in mind, where all hardware resources are pooled into a common shared infrastructure and applications share these resources as their demands change over time [12]. In such a shared environment, meeting application-level QoS goals becomes a challenge as each application consumes different amount of resources. Revisiting the previous scenario of two application servers, Figure 1(c) shows the sum of the CPU consumptions from both nodes. It is evident that the combined application demand is well within the capacity of one node at any particular



(a) CPU consumption of node 1



(b) CPU consumption of node 2



(c) Sum of CPU consumptions from both nodes

**Figure 1: An example of data center server consumption**

time. If we can dynamically allocate the server capacity to these two applications as their demands change, we can easily consolidate these two nodes into one server.

Unfortunately, the complex nature of enterprise applications poses further challenges for this vision. Enterprise applications typically employ a multi-tier architecture where distinct components of a single application are placed on separate servers; for example, three-tier web applications consist of a web server tier, an application server tier, and a database server tier, spread across multiple servers. First, the resource demands placed on these separate tiers vary from one tier to another; e.g., the web tier may consume CPU and network bandwidth, whereas the database tier mainly consumes I/O bandwidth. Second, the resource demands across tiers are dependent and correlated to each other; for example, a database tier only serves connections established through the web tier. Finally, resource demands vary from one application to another; e.g., for the same number of user sessions served in the web tier, we may be seeing vastly different resource demand profiles at the database tier for different applications. As a result, dynamically adjusting resources to an application not only has to take into account the local resource demands in a node where a component of that application is hosted, but also the resource demands of all the other application components on other nodes.

In this paper, we address the problem of dynamically controlling resource allocations to individual components of complex, multi-tier enterprise applications in a shared hosting environment. We rely on control theory as the basis for modeling and designing such a feedback-driven resource control system. We develop a two-layered controller architecture that accounts for the dependencies and interactions among multiple tiers in an application stack when making resource allocation decisions. The controller is designed to adaptively adjust to varying workloads so that high resource utilization and high application-level QoS can be achieved. Our design employs a *utilization controller* that controls the resource allocation for a single application tier and an *arbiter controller* that controls the resource allocations across

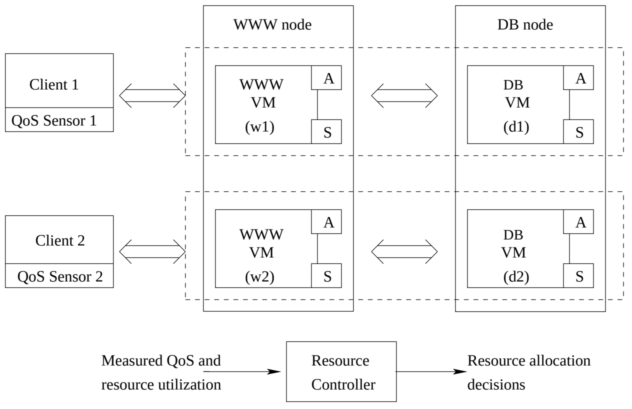


**Figure 2: A testbed of two virtualized servers hosting two multi-tier applications**

multiple application tiers and multiple application stacks sharing the same infrastructure.

To test our controllers, we have built a testbed for a virtual data center using Xen virtual machines (VMs)[5]. We encapsulate each tier of an application stack in a virtual machine and attempt to control the resource allocation at the VM level. We experimented with two multi-tier applications in our testbed: a two-tier implementation of RUBiS [2], an online auction application, and a two-tier Java implementation of TPC-W [6], an online ecommerce application. We ran experiments to test our controller under a variety of workload conditions that put the system in different scenarios, where each node hosting multiple virtual machines is either saturated or unsaturated.

Our experimental results indicate that the feedback control approach to resource allocation and our two-layered controller design are effective for managing virtualized resources in a data center such that the hosted applications achieve good application-level QoS while maintaining high resource utilization. Also, we were able to provide a certain degree of QoS differentiation between co-hosted applications when there is a bottleneck in the shared infrastructure, which is hard to do under standard OS-level scheduling.

## 2. PROBLEM STATEMENT

In this paper, we consider an architecture for a virtual data center where multiple multi-tier applications share a common pool of server resources, and each tier for each application is hosted in a virtual machine. This type of shared services environment has become of interest to many enterprises due to its potential of reducing both infrastructure and operational cost in a data center.

Figure 2 shows a testbed we built as an example of such an environment. This setup forms a small but powerful system, which allows for testing our controller in various scenarios. To avoid confusion in terminology, we use “WWW VM” and “DB VM” to refer to the two virtual machines that are used to host the web server and DB server software, respectively. We use “WWW node” and “DB node” to refer to the two physical machines that are used to host the web tier and the DB tier, respectively.

The high level goal of the resource controller is to guarantee application-level QoS as much as possible while increasing resource utilization in a utility computing environ-

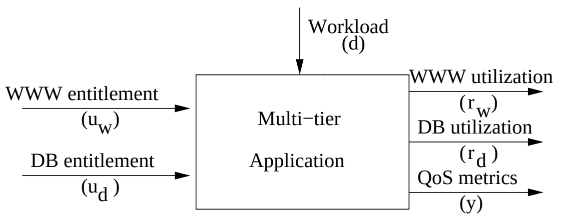


**Figure 3: An input-output model for a multi-tier application**

ment. More specifically, our controller design has the following three main objectives:

- **Guaranteed application-level QoS**: When system resources are shared by multiple multi-tier applications, it is desirable to maintain performance isolation between them and to ensure that each application can achieve its QoS goals. In this paper, this is realized by dynamically allocating virtualized resources to each virtual machine hosting an application component and always providing a safety margin below 100% utilization if possible, which generally leads to high throughput and low response time in the application.

- **High resource utilization**: It is also desirable to increase overall utilization of the shared resources so that more applications can be hosted. One way to achieve this is to maintain a high enough utilization in individual virtual machines such that there is more capacity for hosting other applications. There is a fundamental trade-off between this goal and the previous goal. It is up to the data center operators to choose an appropriate utilization level to balance these two objectives.

- **QoS differentiation during resource contention**: Whenever a bottleneck is detected in the shared resources, the controller needs to provide a certain level of QoS differentiation that gives higher priority to more critical applications. For example, one can aim to maintain a certain ratio of response times when the system is overloaded based on service level agreements of the respective applications.

## 3. SYSTEM MODELING

In control theory, an object to be controlled is typically represented as an input-output system, where the inputs are the control knobs and the outputs are the metrics being controlled. Control theory mandates that a system model that characterizes the relationship between the inputs and the outputs be specified and analyzed before a controller is designed. For traditional control systems, such models are often based on first principles. For computer systems, although there is queueing theory that allows for analysis of aggregate statistical measures of quantities such as utilization and latency, it may not be fine-grained enough for run-time control over short time scales, and its assumption about the arrival process may not be met by certain applications and systems. Therefore, most prior work on applying control theory to computer systems employs an empirical and "*black box*" approach to system modeling by varying the inputs in the operating region and observing the corresponding outputs. We adopted this approach in our work.

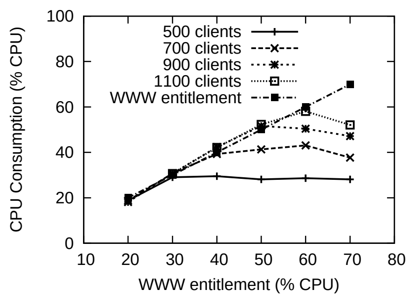
A feedback control loop requires an actuator to implement the changes indicated by the control knobs and a sensor to measure the value of the output variables. We use the CPU scheduler of the virtual machine monitor (or hypervisor) that controls the virtual machines as our actuator. The hypervisor we used provides an SEDF (Simple Earliest Deadline First) scheduler that implements weighted fair sharing of the CPU capacity between multiple virtual machines. The scheduler allocates each virtual machine a certain share of the CPU cycles in a given fixed-length time interval. Since these shares can be changed at run time, the scheduler serves as an actuator (A) in our control loop to effect allocation decisions made by the controller. The SEDF scheduler can operate in two modes: capped and non-capped. In the capped (or non-work-conserving) mode, a virtual machine cannot use more than its share of the total CPU time in any interval, even if there are idle CPU cycles available. In contrast, in the non-capped (or work-conserving) mode, a virtual machine can use extra CPU time beyond its share if other virtual machines do not need it. We use the capped mode in our implementation as it provides better performance isolation between the VMs sharing the same physical server.

The hypervisor also provides a sensor (S) to measure how many of the allocated CPU cycles are actually consumed by each virtual machine in a given period of time. This gives the resource controller information on utilization levels of individual virtual machines. In addition, we modified the RUBiS client and the TPC-W client to generate various application-level QoS metrics, including average response time and throughput in a given period.

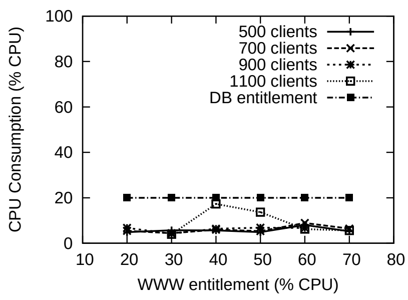
Before describing our modeling approach, we first define some terminology. We use "entitlement" ($u$) to refer to the CPU share (in percentage of total CPU capacity) allocated to a virtual machine. We use "consumption" ($v$) to refer to the percentage of total CPU capacity actually used by the virtual machine. The term "VM utilization" ($r$) is used to refer to the ratio between consumption and entitlement, i.e., $r = v/u$. For example, if a virtual machine is allocated 40% of the total CPU and only uses 20% of the total CPU on average, then its CPU entitlement is 40%, the CPU consumption is 20%, and the VM utilization is 20%/40% = 50%. Note that all of these terms are defined for a given time interval.

Figure 3 illustrates an input-output representation of the system we are controlling, a multi-tier application hosted in multiple virtual machines. The inputs to the system are the resource entitlements for the WWW VM ($u_w$) and the DB VM ($u_d$). The outputs include the utilizations of the WWW VM ($r_w$) and the DB VM ($r_d$), as well as the application-level QoS metrics ($y$) such as response time and throughput. The incoming workload ($d$) to the hosted application is viewed as a "disturbance" to the controlled system because it is not directly under control while having an impact on the system outputs. Typically as the workload varies, the input-output relationship changes accordingly, which increases the difficulty in modeling as well as controller design.

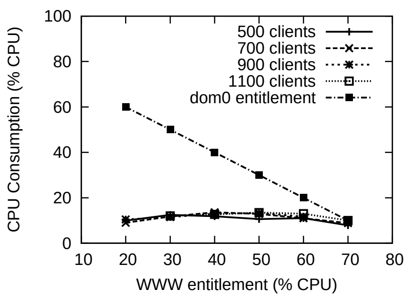
In the remainder of this section, we first describe our experimental testbed. Then we describe a set of system modeling experiments we performed to determine a model for the dynamic behaviour of our multi-tier application under



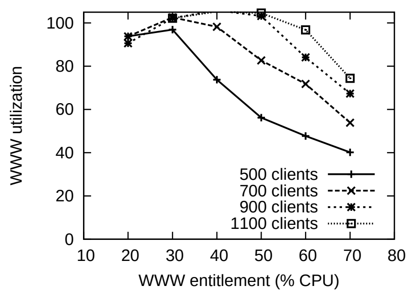
(a) WWW CPU consumption



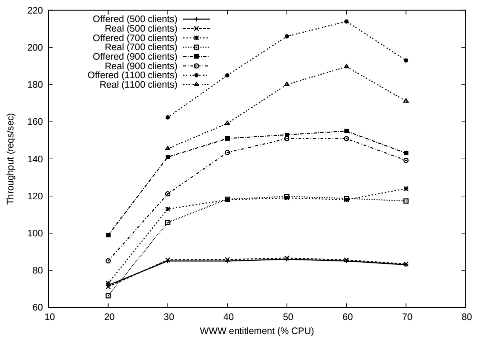
(b) DB CPU consumption



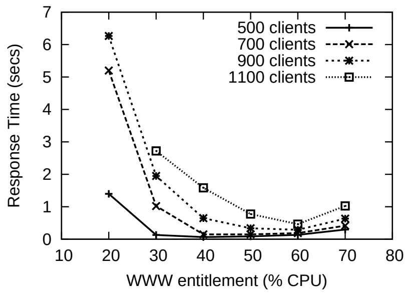
(c) dom0 CPU consumption



(d) WWW VM utilization



(e) Throughput



(f) Response time

**Figure 4: Input-output relationship in a two-tier RUBiS application for [500, 700, 900, 1100] clients**

various configurations. Our modeling experiments consists of two parts. First, we model the dynamic behavior of a single instance of our multi-tier application, and then we develop a model for the dynamic behavior of two multi-tier applications sharing the same infrastructure.

## 3.1 Experimental Testbed

Our experimental testbed consists of five HP Proliant servers, two of which are used to host two applications. Each application consists of two tiers, a web server tier and a database server tier. Apache and MySQL are used as the web server and database server, respectively, hosted inside individual virtual machines. Although the grouping of application tiers on each physical server can be arbitrary in principle, we specifically chose the design where one machine hosts two web servers and the other hosts two DB servers. This is a natural choice for many consolidated data centers for potential savings in software licensing costs.

We chose Xen as the virtualization technology and we use Xen-enabled 2.6 SMP Linux kernel in a stock Fedora 4 distribution. Each of the server nodes has two processors, 4 GB of RAM, one Gigabit Ethernet interface, and two local SCSI disks. These hardware resources are shared between the virtual machines (or domains in Xen terminology) that host the application components and the management virtual machine (which we will refer as `dom0` as in the Xen terminology). In a few testing scenarios, we restrict the virtual machines hosting the applications to share a designated CPU and direct the `dom0` to use the other CPU to isolate it from interference.

Two other nodes are used to generate client requests to the two applications, along with sensors that measure the client-perceived quality of service such as response time and throughput. The last node runs a feedback-driven resource controller that takes as inputs the measured resource utilization of each virtual machine and the application-level QoS metrics, and determines the appropriate resource allocation to all the virtual machines on a periodic basis. This setup forms a small but powerful system, which allows for testing our controller in various scenarios.

We have used two workload generators for our experiments: RUBiS [2], an online auction site benchmark, and a Java implementation of the TPC-W benchmark [6]. The RUBiS clients are configured to submit workloads of different mixes as well as workloads of time-varying intensity. We have used a workload mix called the *browsing mix* that consists primarily of static HTML page requests that are served by the web server. For more details on the workload generation using RUBiS see [2]. The TPC-W implementation also provides various workload mixes. We have used the *shopping mix*, which primarily stresses the DB server.

The use of two different workloads allows us to change overall workload characteristics by varying the intensities of the individual workloads. In particular, by increasing the relative intensity of the TPC-W workload we can increase the load on the database tier (relative to the load on the web tier), and vice versa. We are now ready to describe our system modeling experiments in this testbed.

## 3.2 Modeling single multi-tier application

In this subsection, we would first like to understand how the outputs in Figure 3 change as we vary the inputs, i.e., how the changes in the WWW/DB entitlements impact the utilization of virtual machines and the QoS metrics.

For this experiment, a single testbed node was used to host a two-tier implementation of RUBiS. A single RUBiS client with the browsing mix was used with a certain number of threads simulating many concurrent users connecting to the multi-tier application. In our experiment, we pinned the WWW VM, the DB VM, as well as dom0 to one processor. We varied the CPU entitlement for the WWW VM from 20% to 70%, in 10% increments. Since the DB consumption for the browsing mix is usually low, we did not vary the CPU entitlement for the DB VM and kept it at a constant of 20%. The remaining CPU capacity was given to dom0. For example, when $u_w = 50\%$, we have $u_d = 20\%$, and $u_{\texttt{dom0}} = 30\%$. At each setting, the application was loaded for 300 seconds while the average utilization of the three virtual machines and the average throughput and response time experienced by all the users were measured. The experiment was repeated at different workload intensities, as we varied the number of threads in the RUBiS client from 500 to 1100, in 200 increments.

Figures 4(a), 4(b), 4(c) show the CPU consumption by the WWW VM ($v_w$), the DB VM ($v_d$), and dom0 ($v_{\texttt{dom0}}$), respectively, as a function of the WWW entitlement ($u_w$). In each figure, the four different curves correspond to a workload intensity of 500, 700, 900, 1100 concurrent clients, respectively, while the straight line shows the CPU entitlement for the respective virtual machine, serving as a cap on how much CPU each virtual machine can consume. As we can see from Figure 4(a), with 500 concurrent clients, the WWW CPU consumption goes up initially as we increase $u_w$, and becomes flat after $u_w$ exceeds 30%. Figure 4(d) shows the corresponding utilization for the WWW VM, $r_w = v_w/u_w$, as a function of $u_w$. Note that the utilization exceeds 1 by at most 5% sometimes, which is within the range of actuation error and measurement noise for CPU consumption. We can see that the relationship between the virtual machine utilization and its entitlement can be approximated by the following equation:

$$r_w = \begin{cases} 100\%, & \text{if} \quad u_w <= V; \\ \frac{V}{u_w}, & \text{if} \quad u_w > V, \end{cases} \tag{1}$$

where $V$ is the maximum CPU demand for a given workload intensity. For example, for 500 concurrent clients, $V = 30\%$ approximately. This relationship is similar to what we observed for a single-tier web application as described in [30]. With a workload of 700 concurrent clients and above, the relationship remains similar, except when $u_w$ reaches 70%, the WWW consumption starts to drop. This is because when $u_w = 70\%$, dom0 is only entitled to 10% of the total CPU capacity (see Figure 4(c)), which is not enough to handle workloads with higher intensity due to higher I/O overhead. When dom0 becomes a bottleneck, the web server CPU consumption decreases accordingly. We do not see correlation between the DB CPU consumption and the DB entitlement or the workload intensity from Figure 4(b), other than the fact that the consumption is always below 20%.

Figure 4(e) shows the average offered load and achieved throughput (in requests/second) as a function of the WWW entitlement for different workload intensities. We observe that the offered load is not a constant even for a fixed workload intensity. This is because RUBiS is designed as a closed-loop client where each thread waits for a request to be completed before moving on to the next request. As a result, a varying amount of load is offered depending on how responsive the application is. For all workload intensities that were tested, an entitlement of 20% is too small for the web server, and thus the application responds slowly

| | DB node unsat. | DB node sat. |
|---|---|---|
| WWW node unsat. | WU-DU | WU-DS |
| WWW node sat. | WS-DU | WS-DS |

**Table 1: Four scenarios for two multi-tier applications**

causing the offered load to drop below its maximum. As the WWW entitlement increases, the offered load increases as well because the multi-tier system is getting more work done and responding more quickly. The offered load finally reaches a constant when the web server is getting more than its need. Similarly, as the WWW entitlement is increased, the throughput increases initially and then reaches a constant load. For a larger number of clients (700 and above), we see a similar drop at $u_w = 70\%$ in both the offered load and the throughput because dom0 starts to become a bottleneck as discussed earlier. We also see a significant loss (requests not completed) with a larger number of clients.

Figure 4(f) shows the average response time as a function of the WWW entitlement, validating our observation on the throughput-entitlement relationship. As $u_w$ increases, response time decreases initially reaching a minimum and then rises again at 70% entitlement because of dom0 not getting enough CPU. We also note that the response time is roughly inversely proportional to the throughput because of the closed-loop nature of the RUBiS client.

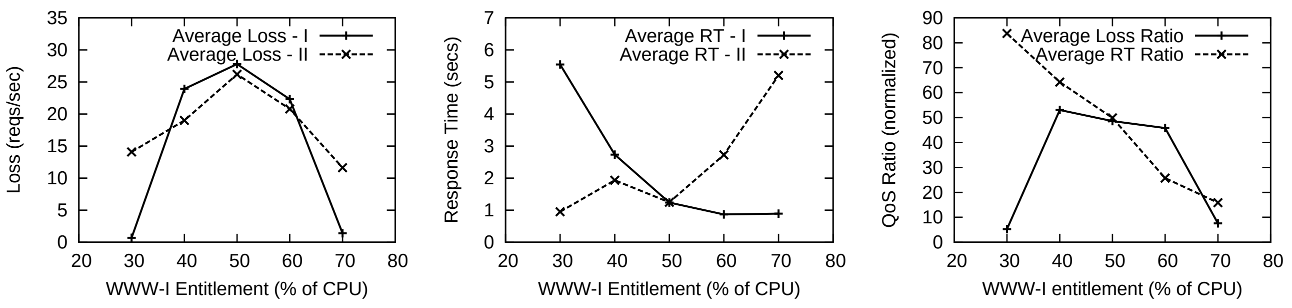
## 3.3 Modeling co-hosted multi-tier applications

Now we move on to consider a model for two multi-tier applications sharing the same infrastructure, as illustrated in Figure 2. The two applications may be driven with different workload mixes and intensities that can change over time, resulting in different and likely time-varying resource demands for the individual virtual machines hosting different application components. At any given time, either the WWW node or the DB node may become saturated, meaning the total CPU demand from both virtual machines exceeds the capacity of the node, or they may be saturated at the same time. If we use W to represent the WWW node, D to represent the DB node, S to represent saturated, and U to represent unsaturated, then the four cases in Table 1 capture all the scenarios that may occur in the shared hosting environment.

The input-output model in Figure 3 needs to be augmented to capture the inputs and outputs for both multi-tier applications. In particular, it will have four inputs ($u_{w1}$, $u_{d1}$, $u_{w2}$, $u_{d2}$) to represent the CPU entitlement for all the four virtual machines. It will also have four outputs ($r_{w1}$, $r_{d1}$, $r_{w2}$, $r_{d2}$) to represent the utilization of the four virtual machines, and two more outputs ($y_1$, $y_2$) to represent the end-to-end QoS metrics for the two applications. We also need a relative metric to enable differentiated service to the two applications when at least one of the nodes is saturated and both applications are contending for resources. Here we define a QoS differentiation metric as follows:

$$y_{ratio} = \frac{y_1}{y_1 + y_2}. \tag{2}$$

Note that we use a normalized ratio as opposed to a direct ratio to avoid numerical ill-conditioning. The QoS metric $y$ can be average response time, throughput, or loss as measured in the number of connections that are reset due to



(a) Loss (b) Response time (c) Loss ratio and RT ratio

**Figure 5: Loss ratio and response time ratio for two RUBiS applications in the WS-DU scenario**

timeouts, etc. in a given time interval.

In the following subsections, we explain the relationship between the inputs (entitlement) and the outputs (VM utilization and QoS) in all the four scenarios. We used two RUBiS instances in WU-DU, WS-DU cases and two TPC-W instances in WU-DS case, because the TPC-W clients stress the database more than the RUBiS clients.

### *3.3.1 WU-DU case*

When neither the WWW node nor the DB node is saturated, the two applications can have access to the shared resource as much as they need to. Therefore they can be viewed as two independent applications as if they each had their own dedicated nodes. In this case, the model we showed in Section 3.2 is applicable to each application and no QoS differentiation metric is necessary. As a result, we can have a controller that controls the resource entitlements for the two applications independently.

### *3.3.2 WS-DU case*

This is the scenario where the WWW node is saturated but the DB node is unsaturated. In this case, the CPU capacity of the WWW node cannot satisfy the needs of the two WWW VMs simultaneously. Arbitration is required to decide how much CPU capacity each WWW VM is entitled to based on a given QoS differentiation metric. We can either use a loss ratio or a response time ratio as defined in equation (2) as the differentiation metric.

Figures 5(a) and 5(b) show the average loss (number of connections reset per second) and average response time (seconds) for two RUBiS applications as a function of the WWW entitlement (percentage) for application 1 ($u_{w1}$). Note that $u_{w2} = 1 - u_{w1}$. Figure 5(c) shows the normalized loss ratio and response time ratio between the two applications as a function of $u_{w1}$.

As we can see from Figure 5(a), as the first WWW entitlement increases, the loss experienced by clients of both applications first increases and then decreases, resulting in a non-monotonic relationship between the loss ratio and $u_{w1}$ as evident from Figure 5(c). This is again due to the closed-loop nature of the RUBiS client where the offered load is reduced as either of the application components becomes a bottleneck.

Figure 5(b) shows a different behavior for response time, where the response time for application 1 goes up and the response time for application 2 goes down, when $u_{w1}$ is increased and $u_{w2}$ is reduced, showing a monotonic relationship between the response time and the WWW entitlement,

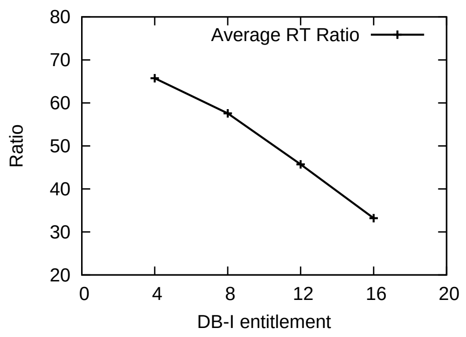


**Figure 6: Response time ratio for two TPC-W applications in the WU-DS scenario**

when the WWW VM is a bottleneck and the DB VM is not. As a result, the response ratio also shows a monotonic relationship with $u_{w1}$ as indicated in Figure 5(c). Furthermore, the relationship is close to linear. Simple linear regression shows the following relationship,

$$\Delta y_{ratio} = -1.65 \Delta u_{w1}. \tag{3}$$

This means a simple linear controller can be designed to find the right value of $u_{w1}$ (and $u_{w2}$ accordingly) to achieve a given target for the response time ratio.

### *3.3.3 WU-DS case*

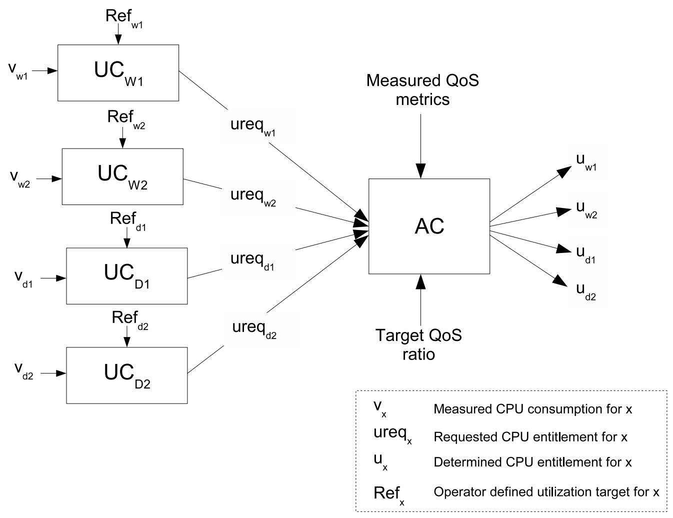
In this scenario, the WWW node is unsaturated but the DB node is saturated. We use TPC-W as the hosted application since it has a higher DB load than what RUBiS has. We use a total capacity of 40% on the DB node to force it to be saturated. (The reason why we cannot make the DB node 100% utilized is due to anomalites in the Xen SEDF scheduler.) This means, $u_{d2} = 40\% - u_{d1}$.

The relationship between the response time ratio and $u_{d1}$ is similar to the WS-DU case, as shown in Figure 6. Again the relationship can be approximated using the following linear equation:

$$\Delta y_{ratio} = -2.75 \Delta u_{d1}. \tag{4}$$

Again, a simple linear controller can be designed to manage the response time ratio between the two applications.

### *3.3.4 WS-DS case*



Ref_w1
v_w1
UC_W1
Ref_w2
v_w2
UC_W2
Ref_d1
v_d1
UC_D1
Ref_d2
v_d2
UC_D2
ureq_w1
ureq_w2
ureq_d1
ureq_d2
Measured QoS metrics
AC
Target QoS ratio
u_w1
u_w2
u_d1
u_d2
v_x Measured CPU consumption for x
ureq_x Requested CPU entitlement for x
u_x Determined CPU entitlement for x
Ref_x Operator defined utilization target for x

**Figure 7: A two-layered controller architecture**

We were unable to create this scenario in our testbed due to anomalies in the Xen SEDF scheduler. The problem is pronounced with the capping option in scheduling (which provides us with an actuator). Whenever capping is enabled, we always ran into very low CPU consumption on WWW VMs which resulted in low consumption in the DB VMs as well. After various experiments, we concluded that the problem lies in how the Xen scheduler handles capping in the context of I/O-intensive applications. We are currently investigating this further.

## 4. CONTROLLER DESIGN

In order to achieve the controller objectives outlined in Section 3.4, we designed a two-layered architecture for the controller, as illustrated in Figure 7.

The first layer consists of four independent controllers, each of which can be viewed as an "agent" for each of the four virtual machines, $w1$, $w2$, $d1$, $d2$. The role of these agents is to compute the required CPU entitlement ($ureq$) for each virtual machine such that, 1) the hosted application component gets enough resource so that it does not become a bottleneck in the multi-tier application; 2) the virtual machine maintains a relatively high resource utilization. In our design, the way to achieve these goals is to maintain a specified level of utilization in each virtual machine. Therefore, the first layer controllers are referred to as *utilization controllers* (UCs). We describe how the utilization controllers work in Section 4.1.

The second layer controller works on behalf of the shared nodes and serves as an "arbiter" that determines whether the requested CPU entitlements ($ureq$) for all of the virtual machines can be satisfied, and if not, decides the final CPU entitlement ($u$) for each VM based on a specified QoS differentiation metric, such as the response time ratio discussed earlier. It is hence referred to as the *arbiter controller* (AC) and will be described in Section 4.2.

### 4.1 Utilization controller

Resource utilization is commonly used by data center operators as a proxy for application performance because of the monotonic relationship between the two and the fact

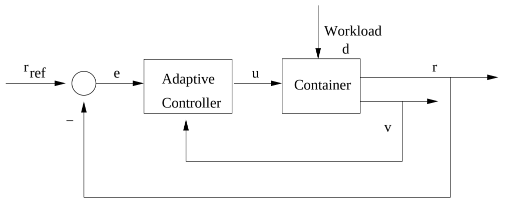


**Figure 8: Adaptive utilization controller**

that utilizaiton is easily measurable at the OS level. Take response time as an example. Simple queuing theory indicates that, when CPU is the bottleneck resource, the response time increases sharply as the CPU utilization gets close to 100%. Therefore, most operators prefer to maintain their server utilization below 100% with a certain safety margin to ensure good application performance. At the same time, utilization should be high enough in order to maintain high resource efficiency in the data center. In our design, we choose 80% as the desired utilization level for the individual VMs. This is determined by examining Figures 4(d), 4(e) and 4(f) together, which show that both the response time and the throughput are at an acceptable level when the utilization of the bottleneck tier, the WWW VM, stays below this target.

We have developed an adaptive integral controller in [30] for dynamic sizing of a virtual machine based on its consumption such that the relative utilization of the VM can be maintained in spite of the changing demand. The block diagram for the controller is shown in Figure 8. At the beginning of the control interval $k$, the controller takes as inputs the desired utilization ($r_{ref}$) and the measured consumption during the previous interval ($v(k-1)$). The controller computes the utilization of the VM ($r$) as $r(k-1) = v(k-1)/u(k-1)$ and the tracking error ($e$) as $e(k-1) = r_{ref} - r(k-1)$, and decides the resource entitlement ($u$) for the VM for the next interval.

This controller is applied in our work as the utilization controllers in the first layer of our controller architecture. For each VM, the UC calculates the required CPU entitlement ($ureq$) using the following control law:

$$ureq(k) = ureq(k-1) - K(k)e(k-1). \qquad (5)$$

The integral gain parameter $K(k)$ determines how aggressive the controller is in correcting the observed error. The value of $K(k)$ adapts automatically to the varying workload by calculating $K(k) = \lambda * v(k-1)/r_{ref}$, where $\lambda$ is a tunable constant. Compared to a standard integral controller that has a fixed $K$ value, our adaptive integral controller with a self-tuning gain makes a better tradeoff between stability and efficiency of the closed-loop system. In addition, it has been proven that this controller is globally stable if $\lambda < 1/r_{ref}$ [30].

### 4.2 Arbiter controller

The four utilization controllers submit the requested CPU entitlements, $ureq_{w1}$, $ureq_{w2}$, $ureq_{d1}$, and $ureq_{d2}$, to the arbiter controller as shown in Figure 7. There are four possible scenarios the arbiter controller needs to deal with, as shown in Table 1 in the previous section. Next, we describe the controller logic for each of the scenarios.

- **WU-DU case** ($ureq_{w1} + ureq_{w2} \leq 1$ & $ureq_{d1} + ureq_{d2} \leq 1$): In this case, all of the requested CPU entitlements can be satisfied. Therefore, the final CPU entitlements are, $u_i = ureq_i, i \in \{w1, w2, d1, d2\}$.

- **WS-DU case** ($ureq_{w1}+ureq_{w2} > 1$ & $ureq_{d1}+ureq_{d2} \leq 1$): In this case, the DB node has enough CPU capacity to satisfy both DB VMs, but the WWW node does not have sufficient capacity. Therefore, the arbiter controller grants the DB VMs their requested entitlements, i.e., $u_{d1} = ureq_{d1}$, $u_{d2} = ureq_{d2}$. At the same time, another control algorithm is needed to compute the appropriate values for $u_{w1}$ (and $u_{w2} = 1 - u_{w1}$) such that the QoS ratio $y_{ratio}$ is maintained at a specified level. Here we use a simple integral controller to perform this task. A regular integral controller implements a control law similar to the one in Eq. (5), except with a constant gain value $K$, which determines the aggressiveness of the controller in its corrective actions. We use a fixed gain instead of a variable one in this case because the relationship between $y_{ratio}$ and $u_{w1}$ is linear, as indicated by the empirical model in Eq. (3). As a result, we can show that this controller is stable if $K < 1/1.65 = 0.61$. We chose $K = 0.1$ in our implementation to provide some stability margin in face of model uncertainty and measurement noise.

- **WU-DS case** ($ureq_{w1}+ureq_{w2} \leq 1$ & $ureq_{d1}+ureq_{d2} > 1$): This case is similar to the WS-DU case, except that now it is the DB node that does not have enough capacity to satisfy both DB VMs. Therefore, we let $u_{w1} = ureq_{w1}$, $u_{w2} = ureq_{w2}$, and a similar integral controller is implemented to compute $u_{d1}$ (and $u_{d2} = 1 - u_{d1}$) to maintain the same QoS ratio. A similar analysis shows that we need to have an integral gain $K < 1/2.75 = 0.36$ for stability. Similarly, we chose $K = 0.1$ for better robustness of the controller.

- **WS-DS case** ($ureq_{w1}+ureq_{w2} > 1$ & $ureq_{d1}+ureq_{d2} > 1$): This occurs when both the WWW and DB nodes are saturated. In principle, the arbiter controller needs to compute both $u_{w1}$ and $u_{d1}$ in order to maintain the desired QoS ratio. However, we could not produce this scenario in our experiments as mentioned earlier and therefore, will not discuss it further.

The QoS ratio is the key metric that drives the arbiter controller. We have discussed the properties of both the loss ratio and the response time (RT) ratio in Section 4. In the next section, we will show experimental results using both metrics, and validate that the RT ratio is a more sensible metric for QoS differentiation between two applications.

## 5. EXPERIMENTAL RESULTS

This section presents experimental results that validate the effectiveness of our controller design in a variety of scenarios.

### 5.1 Utilization controller validation

We first need to validate the behavior of the utilization controllers to move forward with more complex experiments. The goal is to validate that when the two nodes are not saturated, 1) the utilization controllers achieve a set target utilization for the individual virtual machines; 2) both applications have QoS metrics that are satisfactory.

In this experiment, two RUBiS clients with 300 threads each were used to submit requests under a browsing mix. In the middle of the run, 300 more threads were added to the first client for a duration of 300 seconds. Initially, the two WWW VMs were each given a 50% CPU entitlement, and the two DB VMs were each given a 20% entitlement. Then the four utilization controllers adjusted the entitlement for each VM every 10 seconds, using a target utilization of 80%.

Figures 9(a) and 9(c) show the measured CPU consumption ($v$) of all the VMs, and how the entitlement ($u$) for each VM was adjusted by the utilization controller such that a roughly 20% buffer was always maintained above the consumption. It is clear that the utilization controller can automatically allocate higher CPU capacity to the first WWW VM when its user demand was increased.

Figures 9(b) and 9(d) show the resulting throughput (requests/second) and response time (seconds) for the two applications. Both applications were shown to achieve their maximum throughput and a low response time in spite of changes in resource demands throughout the run, except during the transient period.

### 5.2 Arbiter controller - WS-DU scenario

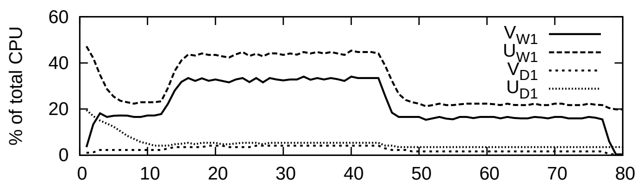
In this scenario, the WWW node is saturated, but the DB node is not saturated. Based on the two-layered controller design, the two utilization controllers for the two DB VMs are used to compute CPU entitlements on the DB node ($u_{d1}$ and $u_{d2}$), while the arbiter controller is used to determine entitlements for the two WWW VMs, where $u_{w1} + u_{w2} = 1$. We performed a set of experiments to understand the behavior of the arbiter controller with different QoS ratio metrics. Again we used two RUBiS clients under a browsing mix.

#### *5.2.1 Experiment with loss ratio*

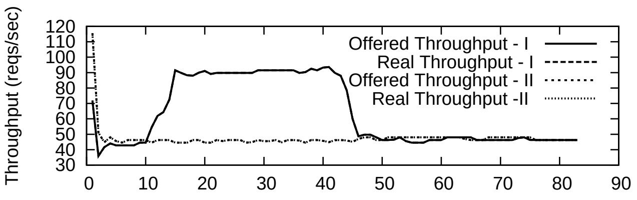
First, we drive the arbiter controller using a target for the ratio of loss seen by both clients. Loss is defined as the number of connections that are dropped after five retries. Intuitively it looks like a good metric capturing the level of performance loss when at least one component of the multi-tier application is overloaded.

In this experiment, we used 1000 threads on each client so that the total required CPU entitlement, $ureq_{w1} + ureq_{w2} > 1$, causing the WWW node to become saturated. As a result, both applications may experience some performance degradation. We set the target loss ratio to be 1:1, or the target for the normalized ratio, $y_{ratio} = \frac{y_1}{y_1+y_2}$, at 50%, where the loss ($y_i$) is normalized with respect to the offered load for application $i$. The two WWW VMs were given a CPU entitlement of 70% and 30% initially. The desirable behavior of the arbiter controller is that it should eventually (after transients) distribute the WWW node's CPU capacity equally between the two WWW VMs because the resource demands from the two hosted applications are identical.

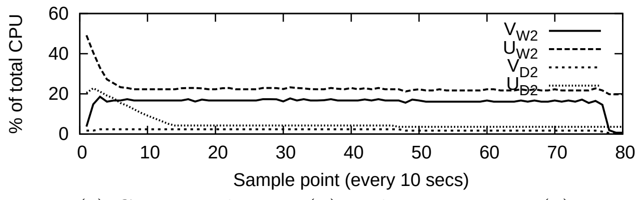
Figure 10(a) shows the measured CPU consumption ($v_{w1}$ and $v_{w2}$) and the CPU entitlement ($u_{w1}$ and $u_{w2}$) determined by the arbiter controller for the two WWW VMs. Figures 10(b) and 10(c) show the resulting throughput and response time for the two applications. Figures 10(d) and 10(e) show the the normalized loss ratio ($y_{ratio}$) and absolute loss (requests/second) seen by the two clients. As we can see, although the normalized loss ratio was maintained around 50%, as time progressed, the second WWW VM was



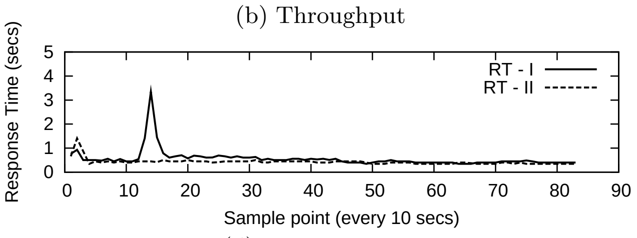
(a) CPU entitlement ($u$) and consumption ($v$)



(b) Throughput



(c) CPU entitlement ($u$) and consumption ($v$)



(d) Response time

**Figure 9: Utilization controller results in the WU-DU case**

actually receiving higher and higher CPU entitlement over time. As a result, application 2 was able to deliver much better throughput and response time compared to application 1.

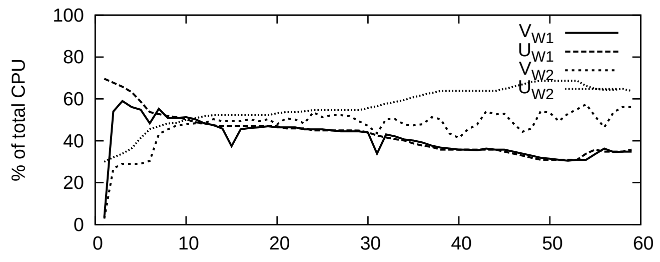
This behavior is highly undesirable. It can be explained by revisiting Figure 5(c) in Section 3. Because the relationship between the loss ratio and the entitlement is not monotonic, any given target loss ratio can either be unachievable (if too high or too low), or may have more than one equilibrium. This means that a loss ratio of 1:1 can be achieved not only when $u_{w1} = u_{w2}$, but also when $u_{w1} \neq u_{w2}$. This again is due to the closed-loop nature of the RUBiS client. A slight disturbance in the CPU entitlement can cause one client to submit less load thus getting less throughput. However, the normalized loss seen by this client may still be equal to that of the other client, resulting in the loss ratio maintained at 1:1, and the controller cannot see the erratic behavior. This problem can be fixed by using the response time ratio as the QoS differentiation metric.

### 5.2.2 *Experiments with RT ratio*

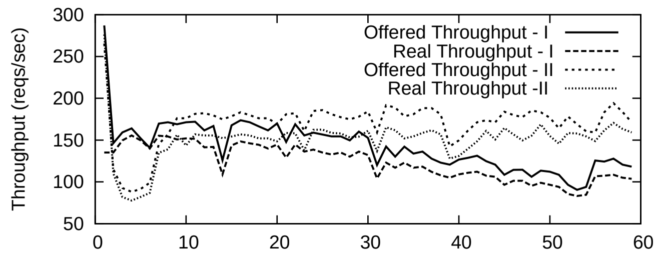
Figures 11(a), 11(b), 11(c), 11(d), 11(e) show the results using the RT ratio as the driving metric for the arbiter controller. The remaining test conditions were kept the same, and an RT ratio of 1:1 was used. Here, we see very desirable behavior where the arbiter controller grants equal entitlements to the CPU capacity between the two WWW VMs. Consequently, the two multi-tier applications achieve fairly comparable performance in terms of throughput and response time. Comparison between this result and the result from the previous experiment shows that the RT ratio is a fairer metric than the loss ratio to be used for QoS differentiation among co-hosted applications. One natural question around this subject is, why not use a throughput ratio instead? The answer is two-fold. First, for a closed-loop client, response time is on average inversely proportional to throughput when the server is overloaded. Therefore, an RT ratio implicitly determines a throughput ratio. Second, the absolute throughput is upper-bounded by the offered load. Therefore, it is not sensible to enforce an arbitrary throughput ratio between two applications if they have drastically different offered loads.

We repeated the experiment using RT ratio under different workload conditions, including workloads with different intensities and time-varying workloads, to make sure that the controller is behaving properly. Time-varying workloads can be realized by starting and stopping different numbers of threads in either of the clients over time. Figures 12(a), 12(b), 12(c), 12(d), and 12(e) show the the behavior of the controller in an experiment where both clients started with 500 threads each, and client 2 jumped to 1000 threads at the 20th control interval and dropped back to 500 threads later. The target for the normalized RT ratio is set to 70%. When both workloads were at 500 threads (before sample 20 and after sample 95), only the utilization controllers were used because neither node was saturated. Note that in Figure 12(d), the RT ratio is shown at exactly 70% for the initial period. This is not from measurement. Rather it is an indication of the fact that the arbiter controller was not used and the target RT ratio was not enforced during these two periods. The arbiter controller was only triggered when the demand from client 2 was suddenly increased and the WWW node became saturated. During this period of time, the arbiter controller was able to maintain the normalized RT ratio at the target. Since the target gives priority to application 2, it was able to achieve lower response time and higher throughput than application 1.

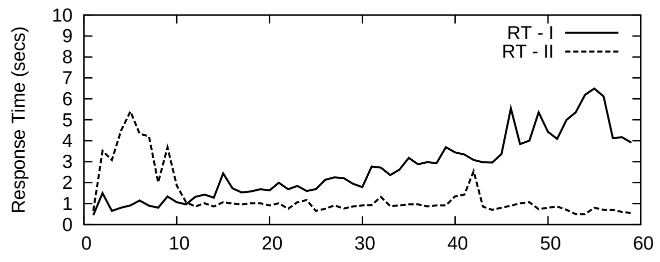
For comparison, we ran the experiment under similar time-varying workload without using a controller. The results are shown in Figures 13(a), 13(b), 13(c), 13(d), and 13(e). In this case, no resource entitlements are enforced. This is done by using the non-capped mode of the SEDF scheduler in Xen, which allows both VMs to use as much CPU as needed until the node is saturated. As we can see, the two WWW VMs consumed roughly equal amount of CPU capacity on the WWW node when both clients had 500 threads. The situation on the DB node was similar. As a result, both applications achieved comparable throughput and response time during this period of time, and the normalized RT ratio is kept around 50% on average. However, when client 2 jumped to 1000 threads at the 15th interval, the resource demand for application 2 suddenly increased. This led to an increase in CPU consumption in both WWW VM 2 and DB VM 2, and an increase in both the throughput and response time for this application. Consequently, the normalized RT ratio becomes approximately 10%, as shown in Figure 13(d).



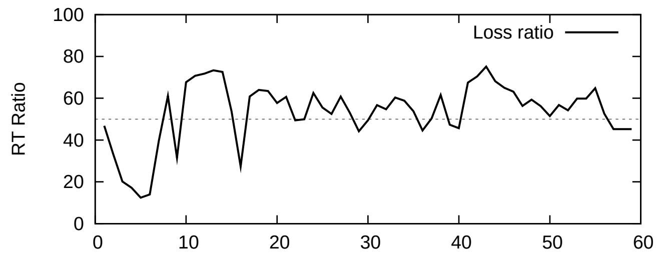
(a) CPU entitlement ($u$) and consumption ($v$)



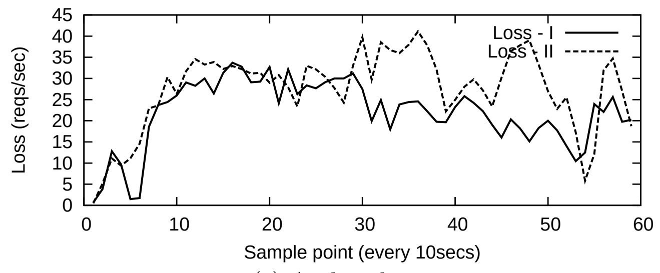
(b) Throughput



(c) Response time

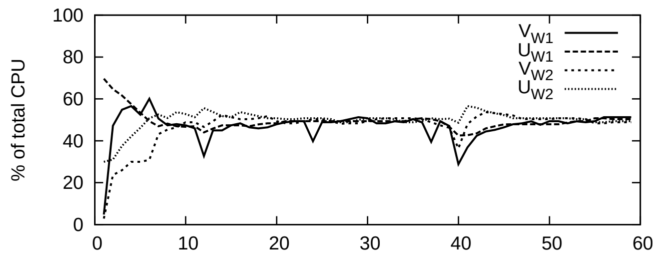


(d) Normalized loss ratio

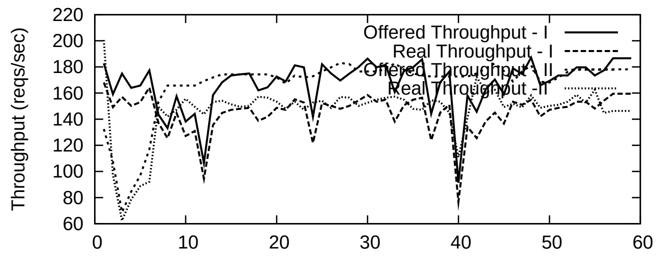


(e) Asolute loss

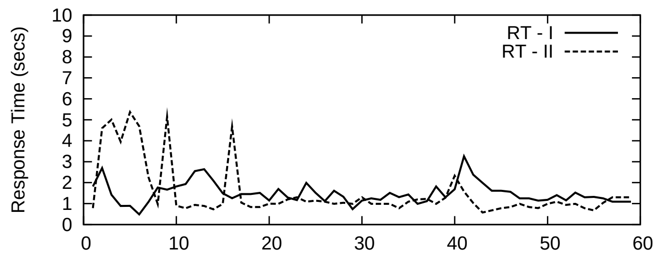
**Figure 10: Loss ratio experiment results**



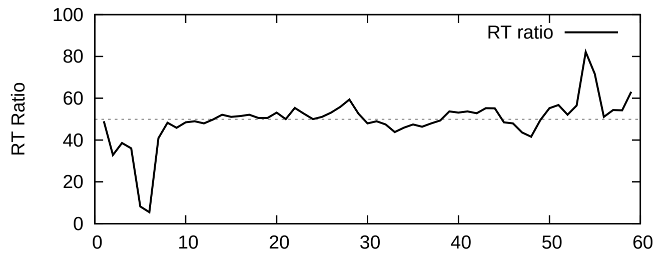
(a) CPU entitlement ($u$) and consumption ($v$)



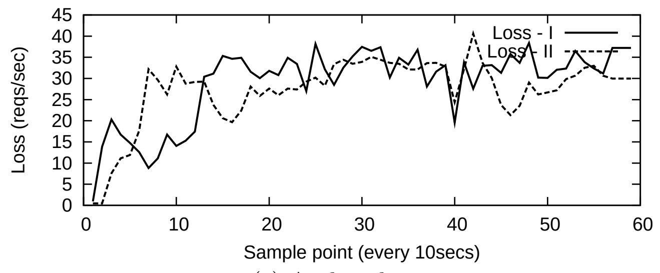
(b) Throughput



(c) Response time

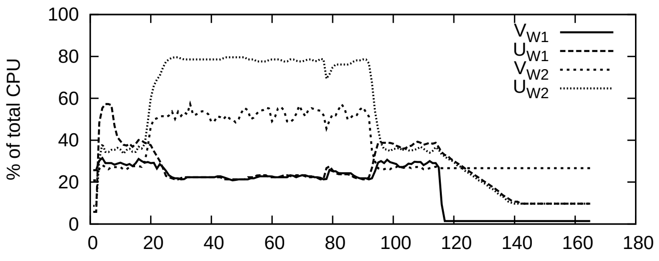


(d) Normalized RT ratio

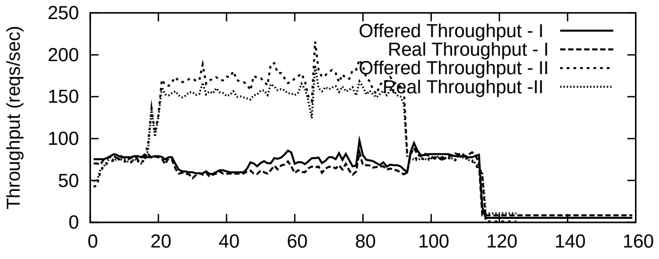


(e) Asolute loss

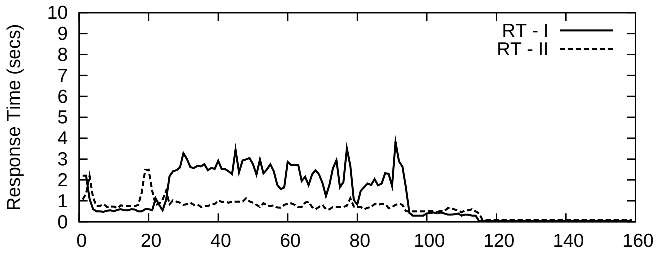
**Figure 11: RT ratio experiment results**



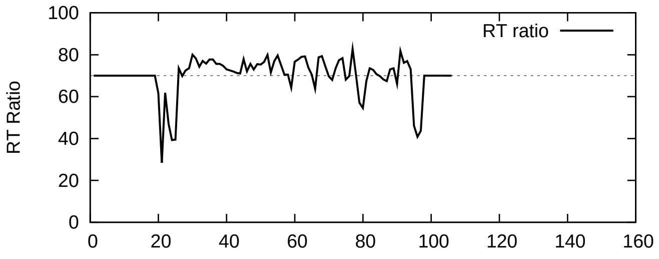
(a) CPU entitlement ($u$) and consumption ($v$)



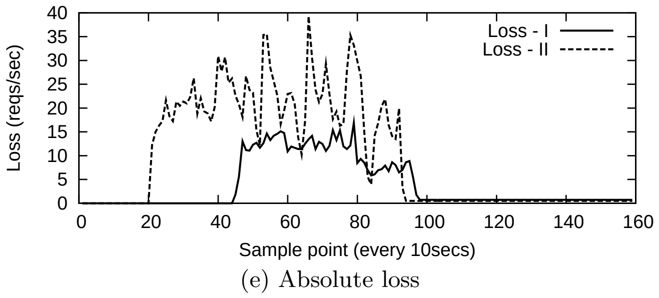
(b) Throughput



(c) Response time

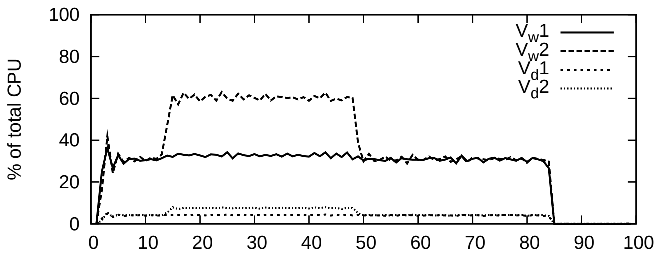


(d) Normalized RT ratio

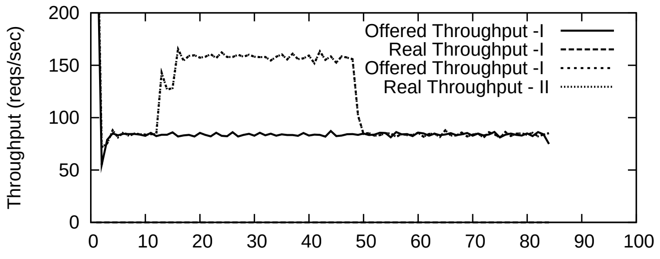


(e) Absolute loss

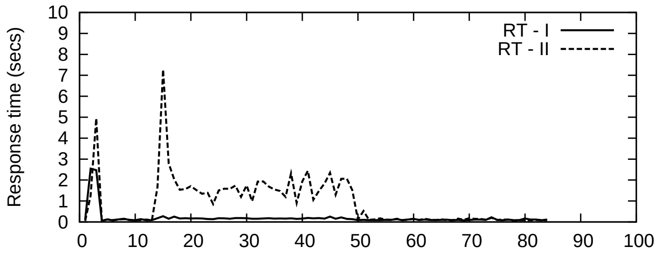
**Figure 12: RT ratio experiments - time-varying workload, with controller**



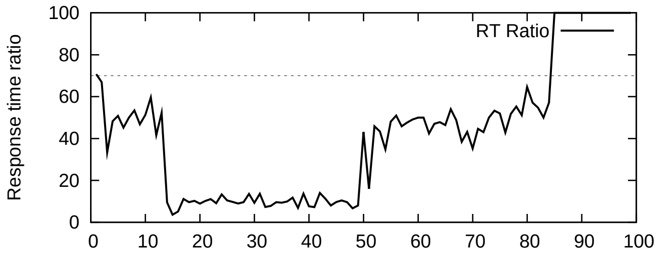
(a) CPU entitlement ($u$) and consumption ($v$)



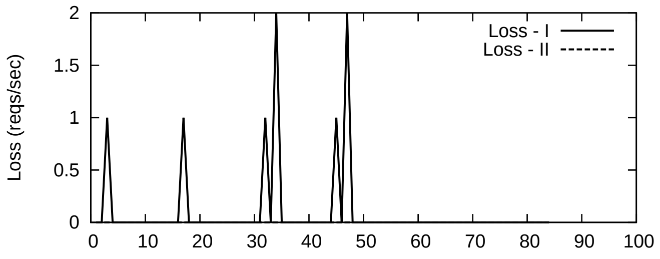
(b) Throughput



(c) Response time



(d) Normalized RT ratio



(e) Absolute loss

**Figure 13: RT ratio experiments - time-varying workload, without controller**

The exact value will be different for different combinations of workload intensities. Because we do not have control over how CPU capacity is scheduled inside the kernel, we cannot enforce a specific level of differentiation between the two applications.

## 5.3 Arbiter controller - WU-DS scenario

This is the case where the WWW node is un-saturated but the DB node is saturated. Similarly, the arbiter controller is needed to determine the CPU entitlements for the two DB VMs. We tested our controller by using two instances of the TPC-W benchmark as the two applications. A shopping mix was used to generate the client requests in order to place more stress on the DB tier. Since the total DB load does not go up to 100%, we capped the sum of $u_{d1}$ and $u_{d2}$ at 40% in order to create the WU-DS scenario for testing our controller. Again, a target RT ratio of 70% was used.

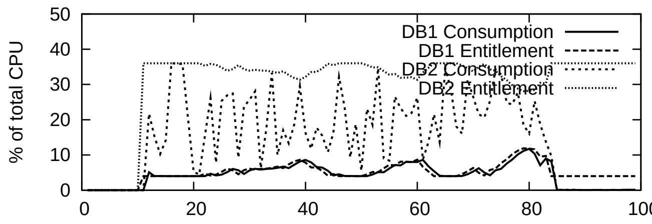
We tested the controller against different workload combinations. An example is shown in Figures 14(a), 14(b), and 14(c) where two TPC-W clients each with 300 threads were used. The arbiter controller was able to maintain the normalized RT ratio around the target, but with more oscillation than was present in the WS-DU case. This can be explained by inspecting the measured CPU consumption and response time over time. Both metrics show a great deal of variation, indicating the greater burstiness of a DB-intensive workload compared to a web-intensive workload. It is also shown in the larger scaling factor in Eq. (4) compared to Eq. (3), indicating a higher sensitivity of the response time with respect to a small change in the entitlement. In spite of the noise in the various metrics, application 2 did receive better QoS from the system in general, as driven by the RT ratio for service differentiation.

Again, for comparison, we ran the experiment under the same workload condition without using a controller. The results are shown in Figures 15(a), 15(b), and 15(c). As we can see, the CPU capacity of the DB node was shared roughly equally between the two DB VMs resulting in comparable average response time for the two applications. Figure 15(c) shows the resulting RT ratio oscillating around an average value of 50%. This result re-confirms that, without a feedback-driven resource controller, we cannot provide QoS differentiation between the two applications at a specified level when the system is overloaded.

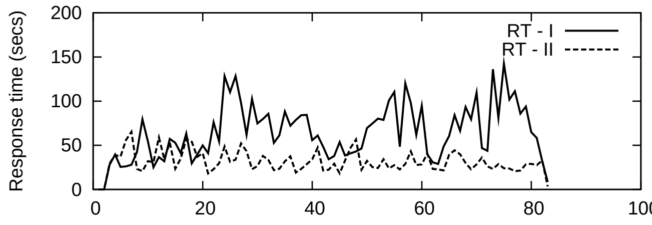
## 6. RELATED WORK

Control theory has recently been applied to computer systems for resource management and performance control [10, 14, 13, 17]. The application areas include web server performance guaranteees[1], dynamic adjustment of the cache size for multiple request classes [22], guaranteed relative delays in web servers by connection scheduling [21], CPU and memory utilization control in web servers [9], and to adjust the resource demands of virtual machines based on resource availability [31]. We focus on controlling the allocation of system resources to application components given an application's resource demand. Whereas prior work focused on one single-tier application only and required modifications to the application, we consider multiple multi-tier applications and do not require any application modifications as we only use sensors and actuators provided in the virtualization layer along with external application QoS sensors.

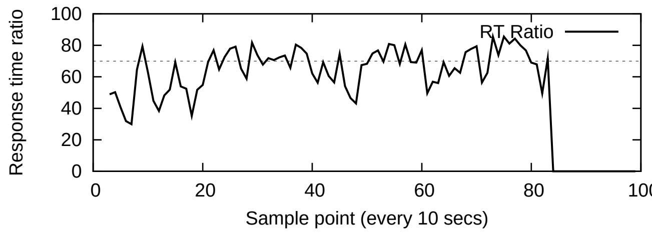
Dynamic resource management in a cluster environment



(a) CPU entitlement and consumption

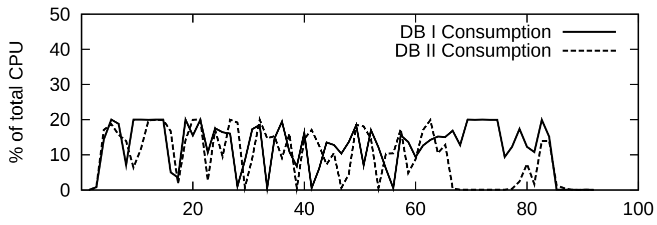


(b) Response time

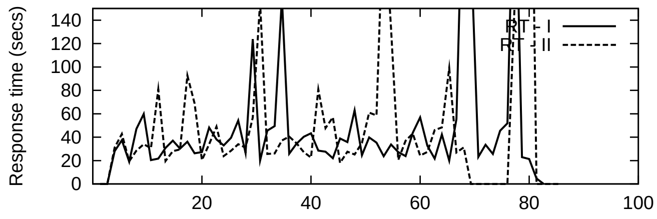


(c) Normalized RT ratio

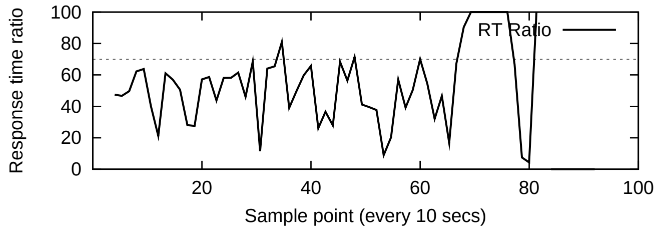
**Figure 14: Database heavy load - with controller**



(a) CPU consumption for two DB VMs



(b) Response time



(c) Normalized RT ratio

**Figure 15: Database heavy load - without controller**

has been studied with goals such as QoS awareness, performance isolation and higher resource utilization. It is formulated as an online optimization problem in [4] using periodic utilization measurements and resource allocation is implemented via request distribution. Resource provisioning for large clusters hosting multiple services is modeled as a "bidding" process in order to save energy in [8]. The active server set of each service is dynamically resized adapting to the offered load. In [25], an integrated framework is proposed combining a cluster-level load balancer and node-level class-aware scheduler to achieve both overall system efficiency and individual response time goals. In [19], resource allocation is formulated as a two-dimensional packing problem, enforced through dynamic application instance placement in response to varying resource demands. In our work, we study more fine-grained dynamic resource allocation in a virtualized server environment where application components are hosted inside individual virtual machines as opposed to individual nodes in a server cluster, and resource allocation is implemented through a fair share scheduler at the hypervisor level.

There are other efforts on dynamic resource allocation in shared data centers. In [7], time series analysis techniques are applied to predict workload parameters, and allocation involves solving a constrained nonlinear optimization problem based on estimation of resource requirements. A recent study is described in [28] for dynamic provisioning of multi-tier web applications. With the estimation of the demand in each tier, the number of servers are dynamically adjusted using a combination of predictive and reactive algorithms. In our work, the kernel scheduler in the virtual machine monitor is used as the actuator for our resource control system. The resource demand of a workload is assumed to be time-varying which may or may not be predictable. Dynamic resource allocation is done with tunable time granularity based on the measured VM utilization and application-level QoS metrics. No estimation is required for the workload demand, and the controller adapts to the changing demand from the workload automatically. Moreover, our controller can deal with resource contention between multiple applications and achieve a desired level of performance differentiation.

In our prior work, we have developed a suite of dynamic allocation techniques for virtualized servers, including adaptive control of resource allocation under overload conditions [20], nonlinear adaptive control for dealing with nonlinearity and bimodal behavior of the system [30], and nested control for a better tradeoff between resource utilization and application-level performance [32]. These approaches are suitable for applications that are hosted inside a single virtual machine. In this paper, we present a dynamic resource allocation system for multi-tier applications with individual components distributed in different virtual machines.

Traditional work on admission control to prevent computing systems from being overloaded has focused mostly on web servers. Recent work has focused on multi-tier web applications. A "gatekeeper" proxy developed in [11] accepts requests based on both online measurements of service times and offline capacity estimation for web sites with dynamic content. Control theory is applied in [16] for the design of a self-tuning admission controller for 3-tier web sites. In [18], a self-tuning adaptive controller is developed for admission control in storage systems based on online estimation of the relationship between the admitted load and the achieved performance. These admission control schemes are complementary to the dynamic allocation approach we describe in this paper, because the former shapes the resource demand into a server system whereas the latter adjusts the supply of resources for handling the demand.

Proportional share schedulers allow reserving CPU capacity for applications [15, 23, 29]. While these can enforce the desired CPU shares, our controller also dynamically adjusts these share values based on application-level QoS metrics. It is similar to the feedback controller in [26] that allocates CPU to threads based on an estimate of thread's progress, but our controller operates at a much higher layer based on end-to-end QoS metrics that span multiple tiers in a given application. Others studied resource overbooking in shared cluster environments leveraging application profiles [27] and calendar patterns (e.g., time of day, day of week) [24] to provide weak, statistical performance guarantees. These approaches require application demand profiles to be relatively stable and do not provide performance differentiation under overload situations. In contrast, our controller can cope with workload variations, even short term unanticipated changes and provides performance differentiation.

## 7. CONCLUSIONS AND FURTHER WORK

In this work, we built a testbed for a data center hosting multiple multi-tier applications using virtualization. We have developed a two-layered controller using classical control theory. The controller algorithms were designed based on input-output models inferred from empirical data using a black-box approach. The controller was tested in various scenarios including stressing the web and the database tier separately. The experimental results confirmed that our controller design achieves high utilization of the data center while meeting application-level QoS goals. Moreover, it is able to provide a specified level of QoS differentiation between applications under overload conditions, which cannot be obtained under standard OS-level scheduling that is not QoS-driven.

We still see space for improvement in controller performance, in terms of better stability and responsiveness, especially for DB intensive workloads. In this paper, only static models were used to capture the input-output relationship in the steady state, which simplifies both the modeling process and the controller design. In future work, we would like to explore the use of a dynamic model that captures more transient behavior of the system and use it as the basis for better controller design. We also want to explore the affect of VM migration on our controller.

There is a lot more work to be done in this area. We would like to extend our work to control sharing of memory, disk I/O, and network resources. These resources pose unique challenges as sharing I/O resources usually involves sharing of related buffers in the kernel as well. Currently, the virtualization technologies do a poor job of providing control of sharing and isolation of I/O resources. With the advent of I/O virtualization technologies from CPU vendors, this may change and we would like to apply our algorithms to take full advantage of the new hardware capabilities.

We also discovered various problems with Xen virtualization technology while performing this work. For example, we encountered certain anomalies with the Xen scheduler while scheduling highly-loaded DB-intensive applications. This is the main reason why we were unable to create the WS-DS scenario where both the WWW and DB nodes are saturated. We would like to investigate this further and propose changes to the scheduler such that it not only provides better accuracy but also becomes more amenable to resource control.

## 8. REFERENCES

[1] T.F. Abdelzaher, K.G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13, 2002.

[2] C. Amza, A. Ch, A.L. Cox, S. Elnikety, R. Gil, K. Rajamani, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *Proceedings of WWC-5: IEEE 5th Annual Workshop on Workload Characterization*, October 2002.

[3] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. Technical Report HPL-2002-339, Hewlett Packard Laboratories

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of the international conference on Measurement and modeling of computer systems(ACM SIGMETRICS)*, pages 90–101, 2000.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, October 2003.

[6] H.W. Cain, R. Rajwar, M. Marden, and Mikko H. Lipasti. An architectural evaluation of java TPC-W. In *HPCA*, pages 229–240, 2001.

[7] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS).*

[8] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles(SOSP)*, October 2001.

[9] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury. Mimo control of an apache web server: Modeling and controller design. In *Proceedings of American Control Conference (ACC)*, 2002.

[10] Y. Diao, J.L. Hellerstein, S. Parekh, R . Griffith, G.E. Kaiser, and D. Phung. A control theory foundation for self-managing computing systems. *IEEE journal on selected areas in communications*, 23(12):2213–2222, December 2005.

[11] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web*, 2004.

[12] S. Graupner, J. Pruyne, and S. Singhal. Making the utility data center a power station for the enterprise grid. Technical Report HPL-2003-53, Hewlett Packard Laboratories, March 2003.

[13] J. L. Hellerstein. Designing in control engineering of computing systems. In *Proceedings of American Control Conference*, 2004.

[14] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems.* IEEE Press/Wiley Interscience, 2004.

[15] M. B. Jones, D. Rosu, and M-C. Rosu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[16] A. Kamra, V. Misra, and E. Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, June 2004.

[17] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems*, pages 49–54, June 2005.

[18] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS)*, 2004.

[19] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th International Conference on World Wide Web*, pages 595–604, May 2006.

[20] X. Liu, X. Zhu, S. Singhal, and M. Arlitt. Adaptive entitlement control of resource partitions on shared servers. In *Proceedings of the 9th International Symposium on Integrated Network Management*, May 2005.

[21] C. Lu, T.F. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 2001.

[22] Y. Lu, T.F. Abdelzaher, and A. Saxena. Design, implementation, and evaluation of differentiated caching serives. *IEEE Transactions on Parallel and Distributed Systems*, 15(5), May 2004.

[23] J. Nieh and M.S. Lam. The design, implementation, and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[24] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical service assurances for applications in utility grid environments. *Performance Evaluation Journal*, 58(2-3), November 2004.

[25] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. *ACM SIGOPS Operating Systems Review*, 36(SI):225 – 238, 2002.

[26] D.C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[27] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the Fifth symposium on operating systems design and implementation (OSDI)*, pages 239 – 254, December 2002.

[28] B. Urgaonkar, P.J. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, pages 217–228, 2005.

[29] C. A. Waldspurger and W.E. Weihl. Lottery scheduling: flexible proprotional-share aresource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[30] Z. Wang, X. Zhu, and S. Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. Technical Report HPL-2005-126R1, Hewlett Packard Laboratories, Feb 2005.

[31] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In Michael Hind and Jan Vitek, editors, *VEE*, pages 2–12. ACM, 2005.

[32] X. Zhu, Z. Wang, and S. Singhal. Utility driven workload management using nested control design. In *Proceedings of American Control Conference (ACC)*, June 2006.